ក្រសួងអប់រំ យុវជន និងកីឡា

វិទ្យាស្ថានជាតិអប់រំ

សៀវភៅឯកទេសៈ

មូលដ្ឋានគ្រឹះនៃភាសា JAVASCRIPT (ES2024)

FUNDAMENTAL OF JAVASCRIPT (ES2024)

ឈ្មោះ សារឿន គឹមស្រ៊ាន

ដើម្បីបំពេញលក្ខខណ្ឌបញ្ចប់ការបណ្តុះបណ្តាល
បរិញ្ញាបត្រជាន់ខ្ពស់អប់រំ វិជ្ជាជីវៈគ្រូបង្រៀនកម្រិតឧត្តម
(បរិញ្ញាបត្រ+២)

ជំនាន់ទី២ ឯកទេសៈ ព័ត៌មានវិទ្យា

ឆ្នាំ២០២៣ - ២០២៥

ក្រសួងអប់រំ យុវជន និងកីឡា

Ministry of Education, Youth and Sport

វិទ្យាស្ថានជាតិអប់រំ

National Institute of Education

សៀវភៅនានុភត្តន៍

មូលដ្ឋានគ្រឹះនៃភាសា JavaScript (ES2024)

FUNDAMENTAL OF JAVASCRIPT (ES2024)

ឈ្មោះ សារឿន គឹមស្រ៊ាន់

ដើម្បីបំពេញលក្ខខណ្ឌបញ្ចប់ការបណ្តុះបណ្តាល
បរិញ្ញាបត្រជាន់ខ្ពស់អប់រំ វិជ្ជាជីវៈគ្រូបង្រៀនកម្រិតឧត្តម
(បរិញ្ញាបត្រ + ២)

ជំនាន់ទី២ ឯកទេស ព័ត៌មានវិទ្យា

ឆ្នាំ២០២៣ - ២០២៥

គ្រូណែនាំគោល ៖ លឹម គឹមឃុន
គ្រូណែនាំរង ៖ រុយ សីរីសោភា

សេចក្តីអះអាងរបស់បេក្ខជន

ខ្ញុំបាទ សូមបញ្ជាក់ថា សៀវភៅដែលមានចំណងជើងថា « **មូលដ្ឋានគ្រឹះនៃភាសា JAVASCRIPT(ES2024)** » សម្រាប់បំពេញលក្ខខណ្ឌបញ្ចប់ការបណ្តុះបណ្តាលបរិញ្ញាបត្រជាន់ខ្ពស់អប់រំវិជ្ជាជីវៈគ្រូបង្រៀនកម្រិតឧត្តម(បរិញ្ញាបត្រ+២) នៅវិទ្យាស្ថានជាតិអប់រំ ពិតជាស្នាដៃរបស់ខ្ញុំបាទ ប្រាកដមែន។ ស្នាដៃនេះមិនបានប្រើប្រាស់ក្នុងការបំពេញលក្ខខណ្ឌសិក្សា ដើម្បីទទួលសញ្ញាបត្រនៅវិទ្យាស្ថានសាកលវិទ្យាល័យ ឬគ្រឹះស្ថានណាមួយឡើយ។ ព្រមជាមួយគ្នា ក្នុងស្នាដៃនេះពុំមានសេចក្តីដកស្រង់ ឬខ្លឹមសារណាមួយត្រូវបានប្រើប្រាស់ក្នុងសៀវភៅនេះ ដោយគ្មានការអនុញ្ញាតពីអ្នកនិពន្ធ ឬគ្មានការចុះក្នុងបញ្ជីឯកសារយោងឡើយ។ ខ្ញុំបាទសូមបញ្ជាក់ថា សៀវភៅនេះ ពិតជាត្រូវបានសរសេរដោយខ្ញុំបាទពិតប្រាកដមែន។

ថ្ងៃចន្ទ ៣កើត ខែស្រាពណ៍ ឆ្នាំម្សាញ់ សប្តស័ក ព.ស. ២៥៦៩
រាជធានីភ្នំពេញ ថ្ងៃទី២៨ ខែកក្កដា ឆ្នាំ២០២៥

ហត្ថលេខា

សារឿន គឹមស្រ៊ាន់

អារម្ភកថា

សៀវភៅ មូលដ្ឋានគ្រឹះនៃភាសា JAVASCRIPT (ES2024) ត្រូវបានរៀបចំយ៉ាងល្អិតល្អន់ ដើម្បីផ្តល់នូវមូលដ្ឋានគ្រឹះដ៏រឹងមាំសម្រាប់ការសិស្សាស្រាវជ្រាវ និងធ្វើជាម្ចាស់JavaScript ដែលជាភាសាសរសេរកម្មវិធីមួយដែលត្រូវបានប្រើប្រាស់យ៉ាងទូលំទូលាយបំផុតក្នុងការអភិវឌ្ឍន៍គេហទំព័រទំនើប។ សៀវភៅនេះគ្របដណ្តប់លើគោលគំនិត និងលក្ខណៈពិសេសសំខាន់ៗដែលត្រូវបានណែនាំនៅក្នុង ECMAScript 2024 ដែលជួយអ្នកសិក្សាឱ្យទាន់សម័យជាមួយនឹងភាពជឿនលឿនចុងក្រោយបំផុត ។

ទន្ទឹមនឹងនេះ ខ្លឹមសារត្រូវបានរៀបចំឡើងដើម្បីបញ្ចូលគ្នានូវចំណេះដឹងទ្រឹស្តីជាមួយនឹងការអនុវត្តជាក់ស្តែង។ ជំពូកនីមួយៗណែនាំអំពីគោលគំនិតនៃការសរសេរកម្មវិធីស្មុគ ដូចជាអថេរ ប្រភេទទិន្នន័យ មុខងារ Class Optional chaining និង Modern JavaScript រួបមន្ត។ មានឧទាហរណ៍ និងលំហាត់សរសេរ Code ជាក់ស្តែងដែលត្រូវបានរួមបញ្ចូល ដើម្បីពង្រឹងទិដ្ឋភាពទ្រឹស្តី ជានានូវបទពិសោធន៍សិក្សាដោយដៃសមរម្យសម្រាប់អ្នកចាប់ផ្តើមដំបូង និងអ្នកអភិវឌ្ឍន៍ដែលមានបទពិសោធន៍។

សៀវភៅនេះបម្រើជាធនធានសិក្សាដ៏ទូលំទូលាយសម្រាប់សិស្សានុសិស្ស អ្នកអប់រំ អ្នកអភិវឌ្ឍន៍ និងអ្នកសិក្សាដោយខ្លួនឯងដែលស្វែងរកការកសាងមូលដ្ឋានគ្រឹះដ៏រឹងមាំនៅក្នុងកម្មវិធី JavaScript ។ មេរៀនត្រូវបានរៀបចំឡើងដើម្បីឱ្យមានភាពជឿនលឿន ណែនាំអ្នកអានមួយជំហានម្តងៗតាមរយៈលក្ខណៈស្មុគរបស់ភាសា ខណៈពេលដែលសង្កត់ធ្ងន់លើការអនុវត្ត និងស្តង់ដារសរសេរ Code ។

បន្ថែមពីលើការពន្យល់តាមទ្រឹស្តី សៀវភៅនេះផ្តល់នូវលំហាត់សរសេរ Code បញ្ហាប្រឈម និងឧទាហរណ៍អន្តរកម្ម ដែលផ្តល់គោលគំនិតដល់អ្នកអានឱ្យអនុវត្តចំណេះដឹងរបស់ពួកគេដោយផ្ទាល់។ តាមរយៈគោលការណ៍ការដែលបានរៀបរាប់នៅក្នុងសៀវភៅនេះ អ្នកអាននឹងត្រូវបានយល់ច្បាស់អំពីមេរៀន ដើម្បីអភិវឌ្ឍកម្មវិធីបណ្តាញអន្តរកម្ម បង្កើនជំនាញសរសេរ Code និងរួមចំណែកដល់គម្រោងកម្មវិធីទំនើបប្រកបដោយប្រសិទ្ធភាព។

ខ្ញុំសូមថ្លែងអំណរគុណយ៉ាងស្មោះស្ម័គ្រចំពោះសិស្សានុសិស្ស និស្សិត អ្នកអប់រំ និងអ្នកជំនាញទាំងអស់ដែលបានរួមចំណែកផ្តល់មតិកែលម្អរាល់ចំណុចខ្វះខាត ដើម្បីធ្វើឱ្យសៀវភៅនេះក្លាយជាឧបករណ៍សិក្សាដ៏មានតម្លៃ។ សូមថ្លែងអំណរគុណយ៉ាងជ្រាលជ្រៅចំពោះសហគមន៍ JavaScript សម្រាប់ការបង្កើតលក្ខណៈថ្មីៗជាបន្តបន្ទាប់ និងស្មារតីសហការគ្នា ដែលជំរុញឱ្យមានការអភិវឌ្ឍន៍ធនធានបែបនេះ។

ខ្ញុំសង្ឃឹមថាសៀវភៅនេះនឹងបម្រើជាដៃគូដែលអាចទុកចិត្តបានក្នុងដំណើររបស់អ្នកទៅកាន់ជំនាញ JavaScript (ES2024) និងលើកទឹកចិត្តអ្នកឱ្យស្វែងយល់ពីលទ្ធភាពគ្មានទីបញ្ចប់នៃការអភិវឌ្ឍន៍គេហទំព័រដោយទំនុកចិត្ត និងចង់ដឹងចង់ឃើញ។

ឧទ្ទិសស្នាដៃ

កូនសូមធ្វើការប្រណិបត្តិន៍ឧទ្ទិសនូវអំណរអរគុណយ៉ាងជ្រាលជ្រៅចំពោះលោកឪពុក និងអ្នកម្តាយ ដែលបានផ្តល់កំណើតឱ្យរូបកូន និងសតិបញ្ញាភ្លឺថ្លា ឈ្លាសវៃ ដែលកូនមាន ព្រមទាំងការបីបាច់ថែរក្សាកន្លងមក ជាពិសេសជាងនេះទៅទៀត បានផ្តល់កំណើតដល់ក្រុមគ្រួសារឱ្យមានបងស្រីប្រុស ដែលមានចិត្តសប្បុរសល្អ មេត្តា ករុណា និងអំណត់អត់ធ្មត់ជាទីបំផុត។

ព្រមទាំងលោកយាយ លោកតា អ៊ុំប្រុស អ៊ុំស្រី លោកពូ អ្នកមីង បងប្អូនញាតិមិត្តទាំងអស់ដែលបានជួយបីបាច់ថែរក្សា ផ្គត់ផ្គង់ ទំនុកបម្រុង និងការអប់រំអស់ពីកម្លាំងកាយចិត្ត និងសម្ភារគ្រប់បែបយ៉ាង ព្រមទាំងផ្តល់នូវការទូន្មានប្រៀនប្រដៅ ដោយផ្តល់នូវដំបូន្មានល្អៗប្រកបដោយព្រហ្មវិហារធម៌គ្រប់បែបយ៉ាង ក្នុងការជួយជំរុញដំណើរការសិក្សារបស់ខ្ញុំបាទប្រព្រឹត្តិទៅបានយ៉ាងល្អប្រសើរ ទីបំផុតខ្ញុំបាទបានបញ្ចប់ការសិក្សា និងក្លាយជាបញ្ញាវន្តមួយរូបប្រកបដោយភាពជោគជ័យជាស្ថាពរ។

សូមគោរពថ្លែងអំណរគុណយ៉ាងជ្រាលជ្រៅបំផុតចំពោះ៖

ឯកឧត្តម បណ្ឌិត សៀង សុវណ្ណា នាយកវិទ្យាស្ថានជាតិអប់រំ ដែលបានផ្តល់ការអនុញ្ញាត និងលើកទឹកចិត្តនៅក្នុងការរៀបចំសរសេរសៀវភៅមួយក្បាលនេះ ដើម្បីចែករំលែកចំណេះដឹងដល់និស្សិត បញ្ញាវន្តខ្មែរ អ្នកបណ្តុះបណ្តាល និងអ្នកអានទាំងអស់ ជាឯកសារយោងសម្រាប់ស្វ័យសិក្សាស្វែងយល់ពីចំណេះដឹងស្តីពី **មូលដ្ឋានគ្រឹះនៃ** JAVASCRIPT (ES2024) ជាខេមរភាសាប្រកបដោយភាពងាយស្រួល។

សូមគោរពថ្លែងអំណរគុណដល់លោកនាយករង លោក លោកស្រី ដែលជាសាស្ត្រចារ្យ និងគណៈកម្មការគ្រប់គ្រងវគ្គបណ្តុះបណ្តាលនេះទាំងអស់ ដែលបានយកចិត្តទុកដាក់ក្នុងការបណ្តុះបណ្តាលគរុនិស្សិតទាំងអស់រហូតបានជោគជ័យ។ ខ្ញុំបាទក៏មិនអាចបំភ្លេចបាននូវការដឹងគុណជូនចំពោះលោក **ឈឹម គឹមឃុន** ជាគ្រូណែនាំគោល និងលោក **រុយ សិរីសោភា** ជាគ្រូណែនាំរង បានចំណាយពេលវេលាដ៏មានតម្លៃជួយបង្ហាត់បង្ហាញលើផ្នែកបច្ចេកទេស វិធីសាស្ត្រនៃការរៀបចំ និងផ្តល់ជាធាតុចូលសំខាន់ៗសម្រាប់សំណេរស្រាវជ្រាវ ព្រមទាំងលើកទឹកចិត្តដល់រូបខ្ញុំបាទ។

ជាទីបញ្ចប់ខ្ញុំបាទសូមគោរពជូនពរចំពោះឯកឧត្តម លោក លោកស្រី សាស្ត្រចារ្យ លោកគ្រូ អ្នកគ្រូ អ្នកនាងកញ្ញា និងមិត្តរួមជំនាន់ទាំងអស់សូមមានសុខភាពល្អ កម្លាំងកាយមាំមួន ជោគជ័យគ្រប់ការកិច្ច និងសូមទទួលបាននូវពុទ្ធពរទាំងបួនប្រការគឺ អាយុ វណ្ណៈ សុខៈ និងពលៈ កុំបីឃ្លៀងឃ្លាតឡើយ។

អំពីអ្នកនិពន្ធ

លោក សារឿន គឹមស្រាន់ គឺជាអ្នកជំនាញបច្ចេកវិទ្យាព័ត៌មានដែលមានបទពិសោធន៍ជាង ២ ឆ្នាំក្នុងវិស័យ IT និងជាអ្នកអភិវឌ្ឍកម្មវិធី។ លោកមានទស្សនវិស័យច្បាស់លាស់ និងបំណងចង់ចែករំលែកចំណេះដឹងដល់សហគមន៍។ លោកបានចាប់ផ្តើមពីការបង្រៀនអ្នកសិក្សាជំបូង រហូតដល់ការបង្រៀននៅក្នុងវគ្គសិក្សាជំនាញក្នុងវិស័យបច្ចេកវិទ្យា។

លោក សារឿន គឹមស្រាន់ បានផ្តោតលើការសិក្សា និងអភិវឌ្ឍបច្ចេកវិទ្យាថ្មីៗ ជាពិសេសភាសា CODE JAVASCRIPT ដែលជាភាសាមូលដ្ឋានសម្រាប់ការបង្កើតគេហទំព័រ កម្មវិធីតាមប្រព័ន្ធ API វេបសេយ និងកម្មវិធីប្រព័ន្ធគ្រប់គ្រងទិន្នន័យ។ ការសរសេរសៀវភៅ "មូលដ្ឋានគ្រឹះនៃ JAVASCRIPT (ES2024)"គឺជាកិច្ចការដ៏មានន័យសម្រាប់លោក ដោយវាបង្ហាញពីចំណេះដឹង និងបទពិសោធន៍ដែលលោកបានស្នាដៃក្នុងរយៈពេលយូរ។ សៀវភៅនេះត្រូវបានរៀបចំជាការណែនាំល្អសម្រាប់អ្នកថ្មី និងអ្នកចង់ពង្រឹងជំនាញ JAVASCRIPT ជាមូលដ្ឋាន។ លោកបានធ្វើឱ្យការរៀនសូត្រភាសា JAVASCRIPT ក្លាយជាដំណើរងាយស្រួល និងមានប្រសិទ្ធភាព តាមរយៈការប្រើឧទាហរណ៍ជាក់ស្តែងលំហាត់បាន និងការដោះស្រាយបញ្ហាដោយប្រកបដោយប្រយោជន៍។

លោក សារឿន គឹមស្រាន់ មិនត្រឹមតែជាអ្នកបង្រៀន និងអ្នកនិពន្ធប៉ុណ្ណោះទេ គាត់គឺជាអ្នកជំរុញឱកាសសិក្សារបស់អ្នកដទៃ។ គាត់ជឿថា ចំណេះដឹងគឺជាអាវុធសំខាន់ក្នុងការបង្កើតអនាគតដ៏ច្នៃប្រឌិត។ សៀវភៅរបស់គាត់មិនត្រឹមតែជាការណែនាំអំពី JAVASCRIPT ប៉ុណ្ណោះទេ ប៉ុន្តែវាក៏ជាឧបករណ៍ដែលចូលរួមក្នុងការសាងសង់វិស័យ IT ក្នុងស្រុកផងដែរ។

សៀវភៅ "**មូលដ្ឋានគ្រឹះនៃ** JAVASCRIPT (ES2024)" គឺជាគំរូល្អសម្រាប់អ្នកចាប់ផ្តើម និងអ្នកអភិវឌ្ឍកម្មវិធីដែលចង់បន្តស្វែងយល់ និងស្វែងរកការងារថ្មីៗ។ សង្ឃឹមថាសៀវភៅនេះនឹងក្លាយជាជំនួយសំខាន់សម្រាប់អ្នករៀនសូត្រ និងជាឧបករណ៍មានតម្លៃក្នុងការកសាងជំនាញ និងអាជីពអនាគតរបស់ពួកគេ។

ថ្ងៃចន្ទ ៤រោច ខែ អាសាឍ ឆ្នាំម្សាញ់ សប្តស័ក ព.ស. ២៥៦៩
រាជធានីភ្នំពេញ ថ្ងៃទី ១៤ ខែកក្កដា ឆ្នាំ២០២៥

ហត្ថលេខា

សារឿន គឹមស្រាន់

មាតិកា

បញ្ជីតារាង

បញ្ជីរូបភាព

អក្សរកាត់

អក្សរកាត់	អក្សរពេញ
អយក.	ក្រសួងអប់រំ យុវជន និងកីឡា
វជអ.	វិទ្យាស្ថានជាតិអប់រំ
HTML	Hypertext Markup Language
CSS	Cascading Style Sheets
JS	JavaScript
ECMA	European Computer Manufacturer's Association
CDN	Content Delivery Network
REPL	Read-Eval-Print Loop
DOM	Document Object Model
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
JWT	JavaScript Web Token
API	Application Programming Interface
UI	User-Interface
OOP	Object-Oriented Programming
SQL	Structured Query Language
URL	Uniform Resource Locator

ជំពូកទី ១៖ សេចក្តីផ្តើម

១.១ ឈ្មោះបញ្ហានៃការស្រាវជ្រាវ

ក្នុងសតវត្សទី ២១ ប្រទេសនានាលើពិភពលោកកំពុងវិវឌ្ឍទៅរកភាពរីកចម្រើនយ៉ាងខ្លាំងក្នុងវិស័យបច្ចេកវិទ្យា និងការអភិវឌ្ឍឌីជីថល។ ស្ថាប័ន អង្គការ និងក្រុមហ៊ុនទាំងឡាយ តែងតែមានតម្រូវការប្រើប្រាស់បច្ចេកវិទ្យា ដើម្បីជួយសម្រួលដល់ការងារ ការគ្រប់គ្រង និងដោះស្រាយបញ្ហាជាច្រើនដែលកើតមានក្នុងប្រព័ន្ធការងាររបស់ពួកគេ។ ប្រទេសកម្ពុជាក៏កំពុងអភិវឌ្ឍន៍លើវិស័យនេះ ដោយទទួលស្គាល់ថាវាជាចំណុចសំខាន់សម្រាប់ជំរុញសេដ្ឋកិច្ច និងប្រព័ន្ធអប់រំឌីជីថលឱ្យកាន់តែមានគុណភាព។

ក្នុងការអភិវឌ្ឍន៍នេះ ភាសាកូដ JavaScript គឺជាភាសាបង្កើតគេហទំព័រដែលមានការពេញនិយមខ្លាំងនៅលើពិភពលោក។ វាត្រូវបានប្រើសំខាន់ៗសម្រាប់ការសរសេរកម្មវិធី Client-Side ក្នុង Browser ហើយក៏អាចដំណើរការលើផ្នែក Server-Side តាមរយៈ Node.js ផងដែរ។ JavaScript គឺជាភាសាដែលអាចសរសេររួមជាមួយ HTML និង CSS ដើម្បីបង្កើតគេហទំព័រដែលមានលក្ខណៈអន្តរកម្ម (Dynamic) និងអាចភ្ជាប់ទៅប្រព័ន្ធទិន្នន័យ តាម API ឬប្រព័ន្ធសុវត្ថិភាពផ្សេងៗ។

ការអភិវឌ្ឍន៍ JavaScript មិនឈប់ត្រឹមតែ រូបមន្ត ធម្មតាទេ។ ជាមួយនឹងការអាប់ដេត ECMAScript 2024 ភាសានេះមានលក្ខណៈថ្មីៗ ដូចជា Optional Chaining, Nullish Coalescing Operator និង Private Class Fields ដែលជួយឱ្យអ្នកអភិវឌ្ឍន៍ អាចសរសេរកូដបានមានសមត្ថភាព ប្រសិទ្ធភាព និងកាត់បន្ថយកំហុស។ លើសពីនេះ Node.js ក៏ធ្វើឱ្យ JavaScript អាចដំណើរការក្នុង Server-Side ដែលអាចបង្កើត API និងប្រព័ន្ធ backend មានប្រសិទ្ធភាព។

ក្នុងបរិបទនេះ សៀវភៅ Fundamental of JavaScript (ES2024) ត្រូវបានស្រាវជ្រាវ និងចងក្រងឡើង ដើម្បីផ្តល់ចំណេះដឹងគ្រឹះ និងមូលដ្ឋានសំខាន់ៗ សម្រាប់អ្នកចាប់ផ្តើម សិស្ស អ្នកបង្រៀន និងអ្នកអភិវឌ្ឍកម្មវិធី។ សៀវភៅនេះ មានគោលបំណងបង្រៀនពីចំណុចមូលដ្ឋានដូចជា រូបមន្ត កូដ អថេរ ប្រភេទទិន្នន័យ ប្លុកកូដ ប្រតិបត្តិការ ការគ្រប់គ្រងលំដាប់បញ្ជា និងលក្ខណៈថ្មីៗ ក្នុង ECMAScript 2024។ ជាមួយនេះ សិស្សនឹងយល់ច្បាស់ពី Debugging, Testing និងការភ្ជាប់ប្រព័ន្ធដែលមានតួនាទីសំខាន់សម្រាប់ការអភិវឌ្ឍកម្មវិធី ឲ្យមានគុណភាព និងងាយស្រួលថែទាំ។

ដូច្នេះ ការស្រាវជ្រាវនេះ ជារបៀបបង្កើតប្រព័ន្ធសិក្សាដែលសមរម្យ ស្របតាមតម្រូវការពិភពលោក និងស្ថាប័ននៅកម្ពុជា។ វាជួយសិក្សាឲ្យបានងាយស្រួល ស្រាល និងច្បាស់លាស់ យ៉ាងពិសេសសម្រាប់អ្នកដែលចាប់ផ្តើមដោយគ្មានដែនកំណត់អាយុ និងចំណេះដឹងដំបូង។

១.២ បញ្ហាស្រាវជ្រាវ

ក្នុងការសិក្សា និងប្រើប្រាស់ភាសា JavaScriptសិស្ស និងអ្នកចាប់ផ្តើមតែងតែប្រទះប្រទាំងបញ្ហា៖

- មិនយល់ច្បាស់ពី រូបមន្ត ថ្មីៗ ក្នុង ECMAScript 2024 ដូចជា Optional Chaining, Nullish Coalescing Operator, Private Fields។
- ខ្វះឯកសារភាសាខ្មែរដែលរៀបចំអោយងាយស្រួលសិក្សា។
- មិនទាន់ចេះគ្រប់គ្រង Tools ដូចជា VS Code, Debugger, Git Version Control។
- មិនទាន់មានគំរូគម្រោង (Project) ងាយស្រួលសាកល្បង។
- អនុវត្តកូដតែ Client-Side ប៉ុណ្ណោះ មិនទាន់ប្រើ Node.js ឬ Frameworks ផ្សេងទៀត។

បញ្ហាទាំងនេះ បង្ហាញថាការស្រាវជ្រាវ និងចងក្រងសៀវភៅសិក្សា JavaScript (ES2024) មានសារៈសំខាន់ក្នុងការដោះស្រាយចំណុចខ្វះខាត និងជួយសម្រួលការសិក្សា។

១.៣ ចំណោទបញ្ហា

- តើគួរចាប់ផ្តើមសិក្សា JavaScript គួរចាប់ផ្តើមពីកន្លែងណា?
- តើ រូបមន្ត ថ្មីៗ និងលក្ខណៈថ្មីៗរបស់ ECMAScript 2024 អាចជួយសម្រួលការសរសេរកូដយ៉ាងដូចម្តេច?
- តើការយល់ដឹងលើ debugging, testing, និង integration មានសារៈសំខាន់យ៉ាងណា?

១.៤ គោលបំណងនៃប្រធានបទស្រាវជ្រាវ

យោងតាមបញ្ហាខាងលើនេះ អ្នកស្រាវជ្រាវបានជ្រើសប្រធានបទ "Fundamental of JavaScript (ES2024)" ដើម្បីដោះស្រាយបញ្ហា និងបំពេញតម្រូវការសិស្ស អ្នកបង្រៀន និងអ្នកអភិវឌ្ឍ។ គោលបំណងរួមមាន៖

- ដើម្បីបង្កើតសៀវភៅ "Fundamental of JavaScript (ES2024)" ជាភាសាខ្មែរ ងាយស្រួលសិក្សា។
- ផ្តល់ឧទាហរណ៍កូដ និងលំហាត់សម្រាប់អ្នកចាប់ផ្តើម។
- ណែនាំពីលក្ខណៈថ្មីៗក្នុង ECMAScript ដូចជា Class Fields និង Optional Chaining។
- ជួយសិស្ស អ្នកអភិវឌ្ឍ និងអ្នកបង្រៀនប្រើប្រាស់ JavaScript ដោយមានប្រសិទ្ធភាព។
- ជំរុញឱកាសការងារនៅវិស័យ IT ក្នុងប្រទេសកម្ពុជា។
- បង្កើតសៀវភៅសិក្សាដែលមិនកំណត់អាយុ និងចំនេះដឹងជំបូង។

- ជាការផ្តត់ផ្តង់មូលដ្ឋានសម្រាប់ការបង្រៀន និងរៀនក្នុងសាលារៀន និងសហគមន៍ Coding។
- ជាជំនួយសម្រាប់ការអនុវត្តគម្រោង JavaScript Client-Side និង Server-Side។

១.៥ សារៈសំខាន់នៃការស្រាវជ្រាវ

ការស្រាវជ្រាវ និងចងក្រងសៀវភៅនេះ មានសារៈសំខាន់ណាស់ ចំពោះអ្នកសិក្សា និងអ្នកអភិវឌ្ឍន៍ ប្រកបដោយអត្ថប្រយោជន៍ដូចជា៖

- យល់ច្បាស់ពី រូបមន្ត និង Standard Coding ក្នុង ECMAScript 2024។
- អាចប្រើប្រាស់លក្ខណៈថ្មីៗ ដូចជា Class Fields, Optional Chaining និង Nullish Coalescing
- ចេះគ្រប់គ្រង Debug និង Version Control ដោយប្រើ Git។
- អាចសរសេរកូដ Client-Side និង Server-Side ដោយប្រើ Node.js និង Frameworks ទំនើប
- ជួយបង្កើនឱកាសក្នុងការងារក្នុងវិស័យ IT និងឧស្សាហកម្ម Tech នៅកម្ពុជា។
- កាត់បន្ថយភាពលំបាកក្នុងការស្វែងរកឯកសារ និងគំរូគម្រោង JavaScript។
- ផ្តល់មូលដ្ឋានដល់ការសិក្សាបន្ត និងស្រាវជ្រាវបច្ចេកវិទ្យាថ្មីៗ។

៦ ទិសភាព និងដែនកំណត់នៃភាសា JavaScript

- វិសាភាព៖
  - ប្រើប្រាស់បានលើ Client-Side និង Server-Side។
  - បង្ហាញពីចំណុចសំខាន់ៗនៃ Javascript ដូចជា រូបមន្ត, Variables, Functions, និង Loops
  - អាចបង្កើតវេបសាយ គេហទំព័រ Single Page App, Mobile App និង API។
  - អាចប្រើលក្ខណៈពិសេសថ្មីដែលបានណែនាំនៅក្នុង ES2024 បានដូចជា Optional chaining and Private methods ។
  - គាំទ្រ Frameworks ដូចជា React, Angular, Vue និង Express.js។
  - សរសេរកូដមានរបៀបរៀបរយ និងងាយស្រួលក្នុងការថែទាំ
  - ឧទាហរណ៍ជាក់ស្តែង ដើម្បីបង្ហាញពីការអនុវត្ត។ដែនកំណត់៖
  - ពឹងផ្អែកលើ Browser និង Interpreter សំខាន់ៗ។
  - ការគ្រប់គ្រង រូបមន្ត និង Debugging យ៉ាងម៉ត់ចត់។

១.៧ ទិសភាពនៃការសិក្សា

- អាចសិក្សារបស់ខ្លួនតាមសៀវភៅ និងឯកសារយោង។
- អាចចូលរួមវគ្គសិក្សាអនឡាញ។
- អាចសិក្សាក្នុងថ្នាក់ក្រុម ឬ Coding Bootcamp។
- អាចសិក្សាជាមួយគម្រោង Open Source។
- អាចសិក្សាដោយប្រើគម្រោងប្រើប្រាស់ Node.js, React និង Express។
- សិក្សាបានទាំង Coding បុរាណ និង Modern JavaScript (ES2024)។

១.៨ ដែនកំណត់នៃការប្រើប្រាស់

- តម្រូវឱ្យមានកុំព្យូទ័រ និងការតម្លើង Tools។
- ត្រូវប្រើប្រាស់ Browser និង Runtime Environment ដែលគាំទ្រ។
- តម្រូវឱ្យសិក្សា Debugging និង Testing ច្បាស់។
- តម្រូវឱ្យមានឧបករណ៍ Version Control ដូចជា Git។
- ត្រូវរៀន Coding Standards និង Best Practices។
- មិនគ្របដណ្តប់លើភាសាផ្សេងទៀត លើកលែងតែការចូលរួម API និង Interoperability។

ជំពូកទី ២៖ មូលដ្ឋានភាសា JAVASCRIPT

មេរៀនទី ១៖ សញ្ញាណដំបូងរបស់ភាសា JavaScript

១.១ និយមន័យ

JavaScript ជាភាសាកម្មវិធីមួយដែលមានលក្ខណៈ dynamic, high-level និង interpreted ដែលមានតួនាទីសំខាន់ក្នុងការអភិវឌ្ឍន៍គេហទំព័រផ្នែក client-side និងសំរាប់សកម្មភាពដែលពាក់ព័ន្ធនឹង browser environment។ ភាសានេះត្រូវបានបង្កើតឡើងដើម្បីឱ្យគេហទំព័របានក្លាយជាកម្មវិធីអន្តរកម្ម ដែលអាចផ្លាស់ប្តូរក្នុងពេលជាក់ស្តែងដោយឆ្លើយតបនឹងសកម្មភាពរបស់អ្នកប្រើប្រាស់។ JavaScript បានក្លាយជាភាសាមានសក្តានុពលយ៉ាងខ្លាំង មិនត្រឹមតែ client-side តែប៉ុណ្ណោះទេ គឺក៏អាចប្រើប្រាស់នៅផ្នែក server-side តាមរយៈ Node.js, កម្មវិធី (mobile apps), ឧបករណ៍ IoT, និងកំណែចុងក្រោយនៃការអភិវឌ្ឍន៍។

JavaScript ត្រូវបាននិពន្ធអាស្រ័យលើស្តង់ដារ **ECMAScript** ដែលកំណត់សំណុំច្បាប់និងលក្ខណៈរបស់ភាសានេះ។ JavaScript ត្រូវបានធ្វើឱ្យកាន់តែមានប្រសិទ្ធភាពជាមួយការផ្តល់ជូននូវ features ថ្មីៗក្នុងកំណែ ES2024 ដែលជាផ្នែកមួយនៃការផ្តល់លទ្ធភាពអភិវឌ្ឍន៍កាន់តែទូលំទូលាយសម្រាប់អ្នកសរសេរកម្មវិធី។

១.២ ប្រវត្តិនៃ JavaScript

JavaScript ត្រូវបានបង្កើតឡើងក្នុងឆ្នាំ 1995 ដោយលោក Brendan Eich នៅពេលធ្វើការនៅ Netscape Communications ។ ដំបូងឡើយ វាត្រូវបានគេហៅថា "Mocha" បន្ទាប់មក "LiveScript" ហើយចុងក្រោយគឺ "JavaScript" ដើម្បីបង្កើនប្រជាប្រិយភាពរបស់ Java នៅពេលនោះ។ ទោះបីជាឈ្មោះរបស់វាក៏ដោយ JavaScript មិនទាក់ទងនឹង Java ទេ។ ឈ្មោះគឺជាយុទ្ធសាស្ត្រទីផ្សារតែប៉ុន្នោះ។

- នៅក្នុង Netscape Navigator 2.0 betas (ខែកញ្ញា 1995) វាត្រូវបានគេហៅថា LiveScript
- នៅក្នុង Netscape Navigator 2.0 beta 3 (ខែធ្នូ ឆ្នាំ 1995) វាបានទទួលឈ្មោះចុងក្រោយរបស់វា គឺ JavaScript

១.៣ ការវិវត្តន៍នៃ JavaScript

ភាសានេះបានក្លាយជាមូលដ្ឋានគ្រឹះនៃការអភិវឌ្ឍន៍គេហទំព័រយ៉ាងឆាប់រហ័ស ព្រោះវាអនុញ្ញាតឱ្យកម្មវិធីរុករកតាមអ៊ីនធឺណិតអាចដោះស្រាយកិច្ចការដូចជា សុពលភាពទម្រង់ និង dynamic content updates ដោយមិនចាំបាច់មានអន្តរកម្មជាមួយម៉ាស៊ីនមេ។ ប៉ុន្មានឆ្នាំមកនេះ JavaScript បានវិវត្តន៍យ៉ាងខ្លាំង ដោយ ECMAScript បម្រើជាការបញ្ជាក់ផ្លូវការរបស់វា។

ការវិវត្តន៍នៃ JavaScript អាចត្រូវបានតាមដានតាមរយៈព្រឹត្តិការណ៍សំខាន់ៗជាច្រើន៖

- ECMAScript 3 (1999)៖ បានណែនាំមុខងារស្នូលដូចជាកន្សោមធម្មតា ការសាកល្បងចាប់កំហុស និងមុខងាររៀបចំខ្សែអក្សរកាន់តែប្រសើរ។
- ECMAScript 5 (2009)៖ បានបន្ថែមលក្ខណៈសំខាន់ៗដូចជា របៀបតឹងរ៉ឹង ការគាំទ្រ JSON និង array methods such as forEach, map, and filter.
- ECMAScript 6 (ES6, 2015)៖ កំណែសម្គាល់ដែលណែនាំអថេរដែលមាន block-scoped (let, const), arrow functions, promises, classes, and modules.
- ES2024៖ ការបោះពុម្ពចុងក្រោយបំផុតនៃ ECMAScript ផ្តោតលើការកែលម្អការអនុវត្ត សម្រួលរូបមន្ត និងបន្ថែមសមត្ថភាពថ្មីសម្រាប់អ្នកអភិវឌ្ឍន៍។

JavaScript បានរីកចម្រើនទៅជាភាសាដ៏មានអានុភាពដែលលាតសន្ធឹងលើសពី Browser ដោយការកើនឡើងនៃ Node.js សម្រាប់ការអភិវឌ្ឍន៍ផ្នែកខាងម៉ាស៊ីនមេ និង frameworks ផ្សេងៗដូចជា React, Vue និង Angular សម្រាប់បង្កើត front-end applications ។

១.៤ ហេតុអ្វីត្រូវរៀន JavaScript?

JavaScript គឺជាភាសាសំខាន់មួយសម្រាប់អ្នកអភិវឌ្ឍន៍ ដោយសារភាពទូលំទូលាយ និងភាពអាចប្រើប្រាស់បានរបស់វា។ នេះជាហេតុផលសំខាន់ៗមួយចំនួនដើម្បីរៀន JavaScript៖

- Essential for Web Development ៖ JavaScript គឺជាភាសាសរសេរកម្មវិធីតែមួយគត់ដែលត្រូវបានគាំទ្រ modern web browsers ទាំងអស់។ ប្រសិនបើអ្នកចង់បង្កើតគេហទំព័រអន្តរកម្ម dynamic JavaScript គឺមិនអាចខ្វះបាន។
- High Demand ៖ JavaScript គឺជាភាសាមួយក្នុងចំណោមភាសាដែលប្រើយ៉ាងទូលំទូលាយបំផុតនៅទូទាំងពិភពលោក ហើយអ្នកបង្កើត JavaScript មានតម្រូវការខ្ពស់នៅទូទាំងឧស្សាហកម្មផ្សេងៗ។ ការកើនឡើងនៃ full-stack JavaScript frameworks ដូចជា Node.js បានពង្រឹងជំហររបស់ខ្លួនបន្ថែមទៀតនៅក្នុងការអភិវឌ្ឍន៍ទាំង client-side and server-side development ។
- Community and Ecosystem ៖ JavaScript មានសហគមន៍ដ៏សកម្មដ៏ធំមួយនៃអ្នកអភិវឌ្ឍន៍ដែលរួមចំណែកដល់ Ecosystem ដែលមានការរីកចម្រើនឥតឈប់ឈរនៃ libraries និង frameworks ។ frameworks ដែរពេញនិយមដូចជា React, Vue និង Angular ធ្វើឱ្យវាកាន់តែងាយស្រួលក្នុងការបង្កើតកម្មវិធីស្មុគស្មាញ។

- Wide Application ៖ លើសពីគេហទំព័រ JavaScript ត្រូវបានប្រើនៅក្នុងកម្មវិធីផ្នែកខាងម៉ាស៊ីនមេ ការអភិវឌ្ឍន៍កម្មវិធីទូរស័ព្ទ (ឧ. React Native) desktop applications (e.g., Electron) និងសូម្បីតែការអភិវឌ្ឍន៍ហ្គេម។
- ការរៀន JavaScript ផ្តល់មិនត្រឹមតែជំនាញបច្ចេកទេសប៉ុណ្ណោះទេ ប៉ុន្តែថែមទាំងការយល់ដឹងជាមូលដ្ឋាននៃគោលគំនិតនៃការសរសេរកម្មវិធីសំខាន់ៗដែលអាចអនុវត្តបានចំពោះ languages and frameworks ផ្សេងទៀត។

១.៥ ការដំឡើង JavaScript

ដើម្បីចាប់ផ្តើមជាមួយ JavaScript យើងត្រូវដំឡើងខាងក្រោមត្រូវបានណែនាំ៖

- Browser ៖ ដោយសារ JavaScript ដំណើរការពីដើមនៅក្នុង web browsers អ្វីដែលអ្នកត្រូវធ្វើដើម្បីចាប់ផ្តើមសាកស្បងគឺ modern browser ដូចជា Google Chrome, Mozilla Firefox ឬ Microsoft Edge ។ browser ភាគច្រើនរួមបញ្ចូលឧបករណ៍អ្នកអភិវឌ្ឍន៍ដែលអនុញ្ញាតឱ្យអ្នកសរសេរ និង debug JavaScript ដោយផ្ទាល់។
- Text Editor៖ ខណៈពេលដែលអ្នកអាចសរសេរ JavaScript នៅក្នុង Text Editor ណាមួយការប្រើកម្មវិធីកែ Code ដូចជា Visual Studio Code ឬ Sublime Text នឹងធ្វើឱ្យការអភិវឌ្ឍន៍កាន់តែមានប្រសិទ្ធភាពជាមួយនឹងលក្ខណៈពិសេសដូចជា ការបន្លិចរូបមន្ត ការបំពេញដោយស្វ័យប្រវត្តិ និងការរកឃើញកំហុស។
- Node.js៖ ប្រសិនបើអ្នកចង់ដំណើរការ JavaScript នៅលើម៉ាស៊ីនមេ ឬបង្កើតគម្រោងស្មុគស្មាញបន្ថែមទៀត សូមដំឡើង Node.js ដែលជា runtime environment អនុញ្ញាតឱ្យ JavaScript ដំណើរការនៅខាងក្រៅ browser ។ វាក៏ភ្ជាប់មកជាមួយ npm (Node Package Manager) ដែលជាឧបករណ៍សម្រាប់គ្រប់គ្រង JavaScript libraries and dependencies
- Version Control ៖ វាត្រូវបានណែនាំឱ្យប្រើឧបករណ៍ Version Control ដូចជា Git ដើម្បីតាមដានការផ្លាស់ប្តូរនៅក្នុង Code របស់អ្នក និងសហការជាមួយអ្នកដទៃ។ Platforms like GitHub ធ្វើឱ្យវាងាយស្រួលក្នុងការរៀបចំ host your projects online

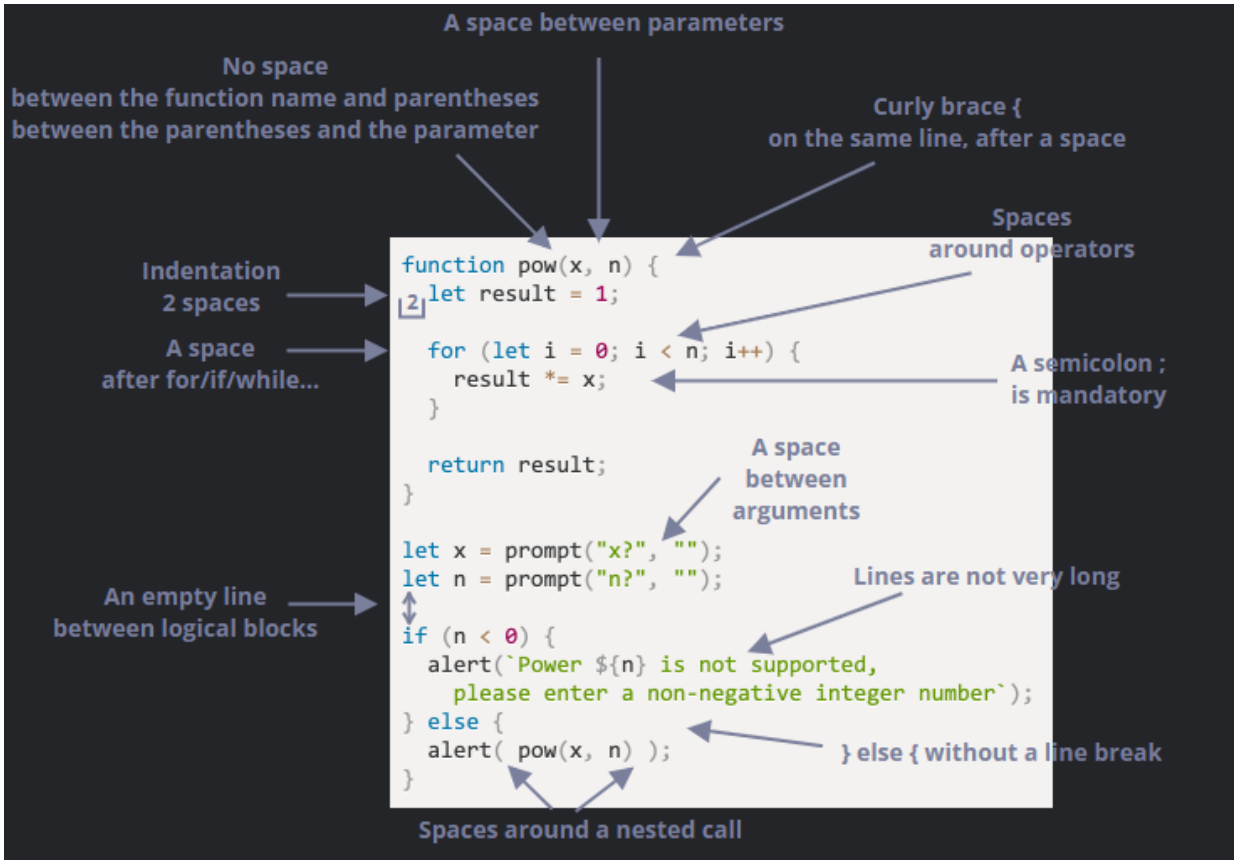
មេរៀនទី ២៖ ភាគថ្នាប់, បញ្ហាញ្ញលទ្ធផលនិង Comments

២.១ រចនាប័ទ្មសរសេរ Code

គ្រប់កូដរបស់យើងត្រូវតែស្អាត និងងាយស្រួលអានតាមដែលអាចធ្វើទៅបាន។ នោះគឺជាសិល្បៈនៃការសរសេរកម្មវិធី ដើម្បីអោយកិច្ចការដែរមានការស្មុគស្មាញមានការងាយស្រួលក្នុងការមើលនិងថែទាំ ហើយសរសេរកូដ វាមានតាមរបៀបដែលត្រឹមត្រូវ និងអាចអានបានដោយអ្នកសរសេរ ឬអ្នកអាន។ រចនាប័ទ្មកូដល្អជួយយើងយ៉ាងខ្លាំងក្នុងរឿងនោះ។

រូបមន្ត៖

នេះគឺជារូបដែលមានរចនាប័ទ្មកូដច្បាប់លាស់សម្រាប់ណែនាំមួយចំនួន (សូមមើលខាងក្រោមសម្រាប់ព័ត៌មានលម្អិតបន្ថែម)៖



រូបភាព 1៖ Coding Style

២.១.១ ដង្កៀបអង្កាញ់ (Curly Braces)

នៅក្នុងគម្រោង JavaScript ភាគច្រើន ដង្កៀបអង្កាញ់ត្រូវបានសរសេរក្នុងរចនាប័ទ្ម "អេហ្ស៊ីប" ជាមួយនឹងដង្កៀបបើកនៅលើបន្ទាត់ដូចគ្នានឹងពាក្យគន្លឹះដែលត្រូវគ្នា មិនមែននៅលើបន្ទាត់ថ្មីទេ។ វាក៏គួរតែមានចន្លោះនៅពីមុខតង្កៀបបើកដូចនេះ៖

```
if (condition) {
  // do this
  // ...and that
  // ...and that
}
```

ការសរសេរក្នុងបន្ទាត់តែមួយ ដូចជា if (condition) doSomething( ), គឺជាករណីគែមដ៏សំខាន់។ តើយើងគួរប្រើដង្កៀបដែរឬទេ?

ខាងក្រោមនេះជាបំរែបំរួលកំណត់ចំណាំ ដូច្នេះអ្នកអាចវិនិច្ឆ័យលទ្ធភាពអានរបស់វាសម្រាប់ខ្លួនអ្នក៖

😠 ពេលខ្លះអ្នកចាប់ផ្តើមធ្វើវា។ អាក្រក់! ដង្កៀបកោងមិនចាំបាច់ទេ៖

```
if (n < 0) {alert(`Power ${n} is not supported`);}
```

😠 បំបែកទៅជាបន្ទាត់ជាច់ដោយឡែកដោយគ្មានដង្កៀប។ កុំធ្វើបែបនោះ ងាយស្រួលបង្កើតកំហុសនៅពេលបន្ថែមបន្ទាត់ថ្មី៖

```
if (n < 0)
  alert(`Power ${n} is not supported`);
```

😒 បន្ទាត់មួយដោយគ្មានដង្កៀប អាចទទួលយកបានប្រសិនបើវាខ្លី៖

```
if (n < 0) alert(`Power ${n} is not supported`);
```

😀ជម្រើសល្អបំផុត៖

```
if (n < 0) {
  alert(`Power ${n} is not supported`);
```

```
}
```

សម្រាប់លេខ Code ខ្លីៗ បន្ទាត់មួយត្រូវបានអនុញ្ញាត ឧ if (cond) return null។ ប៉ុន្តែប្លុក Code (បំរែបំរួលចុងក្រោយ) ជាធម្មតាអាចអានបានច្រើនជាង។

២.១.៤ សញ្ញាក្បៀស (Semicolons)

សញ្ញាក្បៀសគួរតែមានវត្តមានបន្ទាប់ពី statement នីមួយៗ ទោះបីជាវាអាចត្រូវបានរំលងក៏ដោយ។ នៅក្នុង JavaScript មានករណីជាច្រើនដែលការបំបែកបន្ទាត់មិនត្រូវបានបកស្រាយថាជាសញ្ញាក្បៀស ដែលទុកឱ្យ Code ងាយរងកំហុស។ មើលបន្ថែមទៀតអំពី Code structure។

ប្រសិនបើអ្នកជាអ្នកសរសេរកម្មវិធី JavaScript ដែលមានបទពិសោធន៍ អ្នកអាចជ្រើសរើសរចនាប័ទ្ម Code គ្មានលេខដូច StandardJS ។ បើមិនដូច្នេះទេ វាជាការល្អបំផុតក្នុងការប្រើសញ្ញាក្បៀសដើម្បីជៀសវាងការភ្លាក់ដែលអាចកើតមាន។ អ្នកអភិវឌ្ឍន៍ភាគច្រើនដាក់សញ្ញាក្បៀស។

២.១.២ ប្រភេទបន្ទាត់

គ្មាននរណាម្នាក់ចូលចិត្តអាន Code ផ្តេកវែងនោះទេ។ វាជាការអនុវត្តដ៏ល្អបំផុតដើម្បីបំបែកពួកគេ ឧទាហរណ៍៖

```
// backtick quotes ` allow to split the string into multiple lines
let str = `
  ECMA International's TC39 is a group of JavaScript developers,
  implementers, academics, and more, collaborating with the
community
  to maintain and evolve the definition of JavaScript.
`;
```

ហើយសម្រាប់ if statements៖

```
if (
  id === 123 &&
  moonPhase === 'Waning Gibbous' &&
  zodiacSign === 'Libra'
) {
  letTheSorceryBegin();
}
```

ប្រវែងបន្ទាត់អតិបរមាគួរតែត្រូវបានយល់ព្រមនៅកម្រិតក្រុម។ ជាធម្មតាវាមាន 80 ឬ 120 តួអក្សរ។

២.១.៣ ការចូលបន្ទាត់

ការចូលបន្ទាត់នៅក្នុង Javascript មានពីរប្រភេទ៖

ក. ការចូលបន្ទាត់ផ្តេក៖ ដកឃ្លា ២ ឬ ៤។ ការចូលបន្ទាត់ផ្តេកត្រូវបានបង្កើតឡើងដោយប្រើចន្លោះ 2 ឬ 4 ឬនិមិត្តសញ្ញាផ្ទាំងផ្តេក (កូនសោ Tab)។ អត្ថប្រយោជន៍មួយនៃចន្លោះនៅលើផ្ទាំងគឺថាដកឃ្លាអនុញ្ញាតឱ្យមានការកំណត់រចនាសម្ព័ន្ធចូលបន្ទាត់ដែលអាចបត់បែនបានជាងនិមិត្តសញ្ញាផ្ទាំង។

ជាឧទាហរណ៍ យើងអាចតម្រឹមប៉ារ៉ាម៉ែត្រជាមួយនឹងតង្កៀបបើកដូចនេះ៖

```
show(parameters,
  aligned, // 5 spaces padding at the left
  one,
  after,
  another
) {
// ...
}
```

ខ. ការចូលបន្ទាត់បញ្ឈរ៖ បន្ទាត់ទទេសម្រាប់បំបែក Code ទៅជាប្លុកឡូជីខល។ សូម្បីតែមុខងារតែមួយ ជារឿយៗអាចបែងចែកទៅជាប្លុកឡូជីខល។ ក្នុងឧទាហរណ៍ខាងក្រោម ការចាប់ផ្តើមនៃអថេរ រង្វិលជុំចម្បង និងការត្រឡប់លទ្ធផលត្រូវបានបំបែកជាបញ្ឈរ៖

```
function pow(x, n) {
  let result = 1;
  //             <--
  for (let i = 0; i < n; i++) {
    result *= x;
  }
  //             <--
  return result;
```

```
}
```

បញ្ចូលបន្ទាត់ថ្មីបន្ថែម ដែលវាជួយធ្វើឱ្យ Code កាន់តែអាចអានបាន។ វាមិនគួរមានច្រើនជាងប្រាំបួនបន្ទាត់នៃ Code ដោយគ្មានការចូលបន្ទាត់បញ្ជារទេ។

២.១.៥ Nesting Levels

ព្យាយាមជៀសវាងការក្នុងការសរសេរ Code ច្រើនកម្រិតពិបាកពេក។ ជាឧទាហរណ៍ នៅក្នុងរង្វិលជុំ ជួនកាលវាជាការល្អក្នុងការប្រើ continue ការណែនាំ ដើម្បីជៀសវាងការដាក់ nestingបន្ថែម។ ជាឧទាហរណ៍ ជំនួសឱ្យការបន្ថែម ifលក្ខខណ្ឌដែលមានមូលដ្ឋានដូចនេះ៖

```
for (let i = 0; i < 10; i++) {
  if (cond) {
    ... // <- one more nesting level
  }
}
```

យើងអាចសរសេរ៖

```
for (let i = 0; i < 10; i++) {
  if (!cond) continue;
  ...  // <- no extra nesting level
}
```

រឿងស្រដៀងគ្នានេះអាចត្រូវបានធ្វើជាមួយ if/elseនិង return។ ឧទាហរណ៍ ពីរខាងក្រោមគឺដូចគ្នាបេះបិទ។

ជម្រើសទី 1៖

```
function pow(x, n) {
  if (n < 0) {
    alert("Negative 'n' not supported");
  } else {
    let result = 1;
    for (let i = 0; i < n; i++) {
      result *= x;
```

```
    }


    return result;
  }
}
```

ជម្រើសទី 2៖

```
function pow(x, n) {
  if (n < 0) {
    alert("Negative 'n' not supported");
    return;
  }
  let result = 1;
  for (let i = 0; i < n; i++) {
    result *= x;
  }
  return result;
}
```

ទីពីរគឺអាចអានបានច្រើនជាងមុន ពីព្រោះ "ករណីពិសេស" n < 0ត្រូវបានដោះស្រាយទាន់ពេល។ នៅពេលដែលការត្រួតពិនិត្យរួចរាល់ យើងអាចបន្តទៅកាន់លំហូរ Code "មេ" ដោយមិនចាំបាច់មានការដាក់បន្ថែម។

២.១.៦ ទីតាំងអនុគមន៍

ប្រសិនបើអ្នកកំពុងសរសេរមុខងារ "helper" ជាច្រើន និង Code ដែលប្រើពួកវា មានវិធីបីយ៉ាងក្នុងការរៀបចំមុខងារ។

- ប្រកាសមុខងារ ខាងលើ Code ដែលប្រើពួកវា៖

```
// function declarations
function createElement() {
  ...
```

```
}
function setHandler(elem) {
  ...
}
function walkAround() {
  ...
}
// the code which uses them
let elem = createElement();
setHandler(elem);
walkAround();
```

- Code ដំបូងបន្ទាប់មកមុខងារ

```
// the code which uses the functions
let elem = createElement();
setHandler(elem);
walkAround();
// --- helper functions ---
function createElement() {
  ...
}
function setHandler(elem) {
  ...
}
function walkAround() {
  ...
}
```

- លាយបញ្ចូលគ្នា៖ មុខងារមួយត្រូវបានប្រកាសកន្លែងដែលវាត្រូវបានប្រើដំបូង។ ភាគច្រើននៃពេលវេលា second variant ត្រូវបានគេពេញចិត្ត។ នោះគឺដោយសារតែនៅពេលដែលការអាន Code ដំបូងយើងចង់ដឹង ថាវាធ្វើដូចម្តេច ។ ប្រសិនបើលេខ Code ទៅមុន នោះវាច្បាស់តាំងពីដំបូង។ បន្ទាប់មក ប្រហែលជាយើងនឹងមិនចាំបាច់អានមុខងារទាំងអស់នោះទេ ជាពិសេសប្រសិនបើឈ្មោះរបស់ពួកគេពិពណ៌នាអំពីអ្វីដែលពួកគេធ្វើ។

២.១.៧ ការណែនាំអំពីរចនាប័ទ្ម (Style Guides)

ការណែនាំអំពីរចនាប័ទ្មមានច្បាប់ទូទៅអំពី "របៀបសរសេរ" Code ឧ. សម្រង់មួយណាដែលត្រូវប្រើ, ចំនួនដកឃ្លាដែលត្រូវចូលបន្ទាត់, ប្រវែងបន្ទាត់អតិបរមា។ល។ របស់តូចៗជាច្រើន។ នៅពេលដែលសមាជិកទាំងអស់នៃក្រុមប្រើប្រាស់ការណែនាំអំពីរចនាប័ទ្មដូចគ្នា Code មើលទៅមានលក្ខណៈឯកសណ្ឋាន ដោយមិនគិតពីសមាជិកក្រុមណាដែលសរសេរវានោះទេ។

ជាការពិតណាស់ ក្រុមមួយតែងតែអាចសរសេរការណែនាំអំពីរចនាប័ទ្មផ្ទាល់ខ្លួនរបស់ពួកគេប៉ុន្តែជាធម្មតាវាមិនចាំបាច់ទេ។ មានការណែនាំដែលមានស្រាប់ជាច្រើនដើម្បីជ្រើសរើស។
ជម្រើសពេញនិយមមួយចំនួន៖

- ការណែនាំអំពីរចនាប័ទ្ម JavaScript របស់ Google
- ការណែនាំអំពីរចនាប័ទ្ម JavaScript របស់ Airbnb
- Idiomatic.JS
- StandardJS
- (បូករួមទាំងច្រើនទៀត)

ប្រសិនបើអ្នកជាអ្នកអភិវឌ្ឍន៍ថ្មីថ្មោង សូមចាប់ផ្តើមជាមួយសន្លឹកនៅដើមដំពូកនេះ។ បន្ទាប់មកអ្នកអាចរកមើលការណែនាំអំពីរចនាប័ទ្មផ្សេងទៀត ដើម្បីជ្រើសរើសគំនិតបន្ថែមទៀត ហើយសម្រេចចិត្តថាតើមួយណាដែលអ្នកចូលចិត្តជាងគេ។

២.១.៨ ការពិនិត្យស្វ័យប្រវត្តិ (Linters Automated Linters)

Linters គឺជាឧបករណ៍ដែលអាចពិនិត្យ Style Code របស់អ្នកដោយស្វ័យប្រវត្តិ និងធ្វើការកែលម្អការផ្តល់យោបល់។ រឿងដ៏អស្ចារ្យអំពីពួកវាគឺថា ការត្រួតពិនិត្យរចនាប័ទ្មក៏អាចរកឃើញកំហុសមួយចំនួនផងដែរ ដូចជាការវាយអក្សរនៅក្នុងអថេរ ឬឈ្មោះមុខងារ។ ដោយសារតែលក្ខណៈពិសេសនេះ ការប្រើ linter ត្រូវបានណែនាំ បើទោះបីជាអ្នកមិនចង់នៅជាប់នឹង "រចនាប័ទ្ម Code " ជាក់លាក់មួយក៏ដោយ។

នេះគឺជាឧបករណ៍ linting ល្បីមួយចំនួន៖

- JSLint - មួយនៃស្រទាប់ទីមួយ។
- JSHint - ការកំណត់ច្រើនជាង JSLint ។
- ESLint - ប្រហែលជាថ្មីបំផុត។

Link ភាគច្រើនត្រូវបានរួមបញ្ចូលជាមួយកម្មវិធីនិពន្ធដ៏ពេញនិយមជាច្រើន៖ គ្រាន់តែបើកកម្មវិធីជំនួយនៅក្នុងកម្មវិធីនិពន្ធ និងកំណត់រចនាសម្ព័ន្ធរចនាប័ទ្ម។

ឧទាហរណ៍សម្រាប់ ESLint អ្នកគួរធ្វើដូចខាងក្រោម:

1. ដំឡើង Node.js ។
2. ដំឡើង ESLint ដោយប្រើពាក្យបញ្ជា npm install -g eslint (npm គឺជាកម្មវិធីដំឡើងកញ្ចប់ JavaScript) ។
3. បង្កើតឯកសារកំណត់រចនាសម្ព័ន្ធដែលមានឈ្មោះ .eslintrcនៅក្នុង root នៃគម្រោង JavaScript របស់អ្នក (នៅក្នុងថតដែលមានឯកសារទាំងអស់របស់អ្នក)។
4. ដំឡើង/បើកកម្មវិធីជំនួយសម្រាប់កម្មវិធីនិពន្ធរបស់អ្នកដែលរួមបញ្ចូលជាមួយ ESLint ។ អ្នកកែសម្រួលភាគច្រើនមានមួយ។

   នេះជាឧទាហរណ៍នៃ .eslintrcឯកសារ៖

```
{
  "extends": "eslint:recommended",
  "env": {
    "browser": true,
    "node": true,
    "es6": true
  },
  "rules": {
    "no-console": 0,
    "indent": 2
  }}
```

នៅទីនេះការណែនាំ "extends" បង្ហាញថាការកំណត់រចនាសម្ព័ន្ធគឺផ្អែកលើ "eslint:recommended" នៃការកំណត់។ វាក៏អាចទាញយកសំណុំច្បាប់រចនាប័ទ្មពីគេហទំព័រ និង ជំនួសវិញ។ សូមមើល https://eslint.org/docs/user-guide/getting-started សម្រាប់ព័ត៌មានលម្អិតបន្ថែមអំពីការដំឡើង។

២.២ ការឆ្លាប់ JavaScript

២.២.១ JavaScript ជាភាសាស្គ្រីបលំនាំដើម

JavaScript គឺជាភាសា Default Scripting នៅក្នុង HTML ។ អ្វីដែលអ្នកអាចធ្វើបានជាមួយ JavaScript:

- បន្ថែម និងលុបខ្លឹមសារចេញពីទំព័របណ្តាញ។
- ការចូលប្រើ និងផ្លាស់ប្តូរគុណលក្ខណៈធាតុ រួមទាំងប្រភព និងClass។
- បញ្ចូលការសម្គាល់ទៅក្នុងគេហទំព័រដោយប្រើមុខងារ innerHTML ។
- ផ្លាស់ប្តូរ Style attribute ។

២.២.២ ការបន្ថែម JavaScript ទៅ HTML

សារៈសំខាន់ចម្បងនៃ JavaScript គឺនៅក្នុងផ្នែកនៃការបង្កើតគេហទំព័រ។ JavaScript ត្រូវបានបញ្ចូលទៅក្នុង HTML នៃគេហទំព័រ។ មានវិធីសាស្ត្រពីរក្នុងការបន្ថែម JavaScript ទៅ HTML

- បង្កើត Code ប្រភព JavaScript នៅក្នុងឯកសារដាច់ដោយឡែកមួយ ហើយបន្ទាប់មកបង្កើតនៅក្នុង Code HTML តំណភ្ជាប់ទៅកាន់ JavaScript ។
- បង្កើតតំបន់មួយនៅក្នុង Code HMTL សម្រាប់ការបង្កប់ JavaScript

២.២.៣ ការប្រើប្រាស់ធាតុ <script>

សម្រាប់ការបញ្ចូល Code JavaScript ទៅក្នុង HTML Code JavaScript ត្រូវបានបញ្ចូលទៅក្នុងធាតុ <script> រវាង <script> និង </script>។ យើងអាចបញ្ចូល scripts ទៅក្នុងធាតុ <head> ឬ <body>។ ជម្រើសទាំងពីរមានគុណសម្បត្តិផ្ទាល់ខ្លួន:

Scripts ខាងក្រៅភាគច្រើនដែលត្រូវបានបញ្ចូលទៅក្នុងគេហទំព័រ ឧទាហរណ៍ Bootstrap, JQuery និងផ្សេងទៀតត្រូវបានបញ្ចូលទៅក្នុងធាតុ <head>។ នៅក្នុងករណីនៃការបញ្ចូល scriptsផ្ទាល់ខ្លួនទៅក្នុង <head> យើងអាចមានទិដ្ឋភាពទូទៅនៃ scripts ទាំងអស់នៅក្នុងធាតុមួយ។ សព្វថ្ងៃនេះគឺជារឿងធម្មតាក្នុងការប្រើការបញ្ចូល Code JavaScipt នៅបញ្ចប់កូដ <body> ហេតុផលគឺការខិតខំប្រឹងប្រែងសម្រាប់ការផ្ទុកគេហទំព័ររលឿនជាងមុន។ ចាប់តាំងពីការចងក្រង scripts ធ្វើឱ្យថយចុះការ

បង្ហាញទំព័រ វាជាការប្រសើរជាងក្នុងការផ្ទុកមាតិកានៃទំព័រសម្រាប់អ្នកប្រើប្រាស់ និងផ្ទុក scripts ជាបន្តបន្ទាប់។

២.២.៤ ធាតុ HTML សម្រាប់ការបញ្ចូល JavaScript

តើធាតុ HTML មួយណាអាចផ្ទុកកម្មវិធីពី JavaScript?

- <|javascript>
- <|img>
- <|code>
- <|script>
- <|p>

២.២.៥ កូដ JavaScript ក្នុង <body>

នៅក្នុង Code HTML បញ្ចូលស្លាកទៅក្នុងធាតុ <body> សម្រាប់ Code Javascript ។ ឧទាហរណ៍៖

```
<!doctype html>
<html>
  <head>
    <title>First JavaScript</title>
  </head>
    <body>
//JavaScript code
    </body>
</html>
```

២.២.៦ ការភ្ជាប់ JavaScript ពីឯកសារផ្សេង (External JavaScript files)

ជម្រើសបន្ទាប់សម្រាប់ការសរសេរ Code JavaScript គឺនៅក្នុងឯកសារដាច់ដោយឡែក (saved with suffix.js) បន្ទាប់មកដើម្បីបង្កើតតំណនៅក្នុង Code HTML សម្រាប់ឯកសារ ហើយបើកវាដោយប្រើ Browser ។ ការដាក់ Scripts ទៅក្នុងឯកសារខាងក្រៅបែងចែក Code HTML ពី JavaScript និងសម្រួលដល់ការអាន និងការកែសម្រួលបន្ថែមនៃ Scripts ។

ដូចគ្នានឹងការបញ្ចូល Javascript ទៅក្នុងទំព័រដែរ ក្នុងករណីឯកសារខាងក្រៅ យើងអាចបញ្ចូលជាតុ <script> ទៅក្នុង element <head> ឬ <body>។ លក្ខណៈសំខាន់មួយគឺ src ដែលយើងបញ្ជាក់កន្លែងដែលឯកសារត្រូវបានរក្សាទុក និងរបៀបដែលវាត្រូវបានដាក់ឈ្មោះ។

ឧទាហរណ៍៖

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

ប្រសិនបើឯកសារដែលមាន JavaScript មាននៅលើអាសយដ្ឋានគេហទំព័រ វាអាចផ្ទុកឯកសារពីគេហទំព័រដោយបញ្ជាក់អាសយដ្ឋាន URL ត្រឹមត្រូវទៅក្នុង Attribute src។

ឧទាហរណ៍៖

```
<script src="https://www.my_page.com/js/my.js"></script>
```

២.២.៧ ការបញ្ចូលតំណ JavaScript ដោយប្រើ src

តើប៉ារ៉ាម៉ែត្រមួយណាដែលត្រូវប្រើសម្រាប់បញ្ចូលតំណ Javascript ក្នុងជាតុ <script>?

- src
- link
- script
- js
- href

២.២.៨ ការភ្ជាប់ JavaScript ទៅនឹង HTML

នៅក្នុង Code HTML បញ្ចូលតំណភ្ជាប់ទៅក្នុងជាតុ <body> នៅលើ Javascript ដែលរក្សាទុកក្នុងឯកសារ test.js។

ឧទាហរណ៍៖

```
<!doctype html>
<html>
  <head>
    <title>First JavaScript</title>
  </head>
  <body>
    <script src=" "></script>
```

```
  </body>
</html>
```

២.២.៩ តំណភ្ជាប់ URL JavaScript ខាងក្រៅ External JavaScript URL linking

នៅក្នុង Code HTML បញ្ចូលតំណភ្ជាប់ទៅក្នុងធាតុ <body> នៅលើ Javascript ដែលរក្សាទុកក្នុងអាសយដ្ឋាន url http://www.mypage.eu/js/test.js

ឧទាហរណ៍៖

```
<!doctype html>
<html>
  <head>
    <title>First JavaScript</title>
  </head>
  <body>
    <script   ="http://www.mypage.eu/js/test.js"></script>
  </body>
</html>
```

២.៣ លទ្ធផល និងយោបល់ (Results and comments)

២.៣.១ សរៈសំខាន់នៃ comments

ជំហានដំបូងជាមួយភាសាសរសេរ Code JavaScript គឺដើម្បីបន្ថែម Comments ។ Comments ត្រូវបានប្រើប្រាស់ជាទូទៅដោយអ្នកសរសេរកម្មវិធី ដែលជាផ្នែកមួយនៃ Code ទោះបីជាកុំព្យូទ័រ (JavaScript interpreter) មិនអើពើនឹងពួកគេ។ គោលបំណងរបស់ពួកគេគឺដើម្បីរក្សាទុក Comments និងកំណត់ចំណាំសម្រាប់អ្នកសរសេរកម្មវិធី។

ដោយប្រើ Comments Code កាន់តែមានប្រសិទ្ធភាព នៅពេលអ្នកត្រឡប់ទៅសរសេរវាឡើងវិញ។ ការសរសេរកម្មវិធីគឺនៅក្នុងគម្រោងធំជាងដែលជាសកម្មភាពសហការ ហើយនោះជាមូលហេតុដែលការ Comments នៅក្នុង Code ប្រភពគឺជាកាតព្វកិច្ចសម្រាប់ក្រុមធំ។ Comments អាចពន្យល់ពីគំនិតរបស់អ្នកបង្កើតកម្មវិធី ការណែនាំអំពីការបម្រុងទុកសម្រាប់ Developers ផ្សេងទៀតនៅក្នុង Code ឬបន្ថែមកំណត់ចំណាំសំខាន់ៗផ្សេងទៀត។

២.៣.២ ប្រភេទនៃ comments

Comments ក្នុង JavaScript មានពីរប្រភេទ៖

ក. A one-lined code៖ វាត្រូវបានកំណត់ដោយតួអក្សរ // ដែលនៅពីក្រោយដែលជា Comment text ឧទាហរណ៍៖

```
// my comment starting at the beginning of the line
```

ចុងបញ្ចប់នៃ One-lined comment មិនត្រូវបានកំណត់ទេ ចុងបញ្ចប់នៃបន្ទាត់គឺមានន័យថាជាការបញ្ចប់នៃ Comment។

```
function my_f(input) // my second comment
```

ខ. A multiline comment៖ ជាធម្មតាវារួមបញ្ចូលផ្នែកធំនៃអត្ថបទ ប្រហែលជាសម្រាប់បន្ទាត់ច្រើន។ ប្រភេទនៃ Comments នេះត្រូវបានបញ្ចូលរវាងតួអក្សរ /* និង */ ។

ឧទាហរណ៍៖

```
/*
This whole text will be
understood by the computer as a
comment
*/
```

យើងថែមទាំងអាចប្រើ Commenting inside ប្រភេទនេះនៅក្នុងបន្ទាត់នៃ Code ។ ឧទាហរណ៍៖

```
function my_f(/* can be even zero*/ input) //my own function
```

២.៣.៣ Comment តែមួយបន្ទាត់ (One-line comment)

បញ្ចូលទៅក្នុង Code Javascript, one-liner code comment with a text៖ I am excited about Javascript.

ឧទាហរណ៍៖

```
<!doctype html>
<html>
  <head>
```

```
    <title>First JavaScript</title>
  </head>
  <body>
    <script>
              I am excited about Javascript
    </script>
  </body>
</html>
```

២.៣.៤ Comment ច្រើនបន្ទាត់ (Multi-line comment)

បញ្ចូលទៅក្នុង Code Javascript, multiline comment of 3 lines ដែលចាប់ផ្តើមដោយ " Lecture 1 " និងបញ្ចប់ដោយ " I am excited, what is next. " ។

ឧទាហរណ៍៖

```
<!doctype html>
<html>
  <head>
    <title>First JavaScript</title>
  </head>
  <body>
    <script>
            Lecture 2:
            Comments
            I am excited, what is next.
    </script>
  </body>
</html>
```

២.៣.៥ ការប្រើប្រាស់ Comment

បញ្ចូល tags ត្រឹមត្រូវសម្រាប់ Comment ទៅក្នុង Code Javascript៖

1. សម្គាល់អត្ថបទដែលចាប់ផ្តើមដោយ " If you were " និងបញ្ចប់ដោយ " Then they would exclaim៖" ចូលទៅក្នុង Multiline comment ។

2. បញ្ចូលប្រយោគ " Oh, what a pretty house that is!" ទៅក្នុង One-liner comment ។

ឧទាហរណ៍៖

```
<!doctype html>
<html>
  <head>
    <title>First JavaScript</title>
  </head>
  <body>
    <script>
        If you were to say to the grown-ups:
        "I saw a beautiful house made of rosy brick, with geraniums
in the windows and doves on the roof,
  " they would not be able to get any idea of that house at all.
        You would have to say to them:
        "I saw a house that cost $20,000."
        Then they would exclaim:
           "Oh, what a pretty house that is!"
    </script>
  </body>
</html>
```

២.២.៦ ការប្រើអនុគមន៍ alert( )

មុខងារដំបូងដែលយើងនឹងប្រើគឺ alert( )។ មុខងារបង្ហាញការជូនដំណឹងជាមួយសារ (text) និង Button OK ។ ជារឿយៗវាត្រូវបានគេប្រើជាសារព្រមានសម្រាប់អ្នកប្រើប្រាស់ web application ។

វាមិនត្រូវបានណែនាំឱ្យប្រើមុខងារនេះញឹកញាប់ពេកទេ ព្រោះវារារាំងអ្នកប្រើប្រាស់ចូលទៅកាន់ផ្នែកខ្លះនៃទំព័របណ្តាញ ខណៈដែល window មានសារនឹងមិនត្រូវបានបិទ។

នៅក្នុងវគ្គសិក្សារបស់យើង យើងនឹងប្រើមុខងារសម្រាប់តែការគ្រប់គ្រងភាពត្រឹមត្រូវនៃ Code របស់យើងប៉ុណ្ណោះ។ ដូច្នេះ Scripts របស់យើងភាគច្រើននឹងត្រូវបានបញ្ចប់ដោយការជូនដំណឹងមុខងារ ( ) សម្រាប់លទ្ធផលចុងក្រោយ។ ផ្នែកសំខាន់នៃការជូនដំណឹងមុខងារ alert( ) គឺជាប៉ារ៉ាម៉ែត្រនៃមុខងារមួយ។ វាត្រូវបានប្រើដើម្បីដោះស្រាយតម្លៃបញ្ចូលទៅក្នុងអនុគមន៍។ ប៉ារ៉ាម៉ែត្របញ្ចូលនឹងជាអត្ថបទដែលយើងនឹងប្រើក្នុង window ដែលបង្ហាញដល់អ្នកប្រើប្រាស់

ឧទាហរណ៍៖

```
alert("Hello world!");
```

ស្រដៀងគ្នាទៅនឹងភាសាសរសេរ Code ផ្សេងទៀត សូម្បីតែនៅក្នុង JavaScript វាជាការសំខាន់ក្នុងការបញ្ចប់បន្ទាត់ ឬមុខងារមួយ (នៅចុងបញ្ចប់នៃបន្ទាត់) ដោយបញ្ចូលសញ្ញាក្បៀស ; (or line feed) ។ វាបំបែកពាក្យបញ្ជាបុគ្គលពីគ្នាទៅវិញទៅមក។

តាមទ្រឹស្តី យើងអាចសរសេរ Code Javascript ទាំងមូលទៅជាបន្ទាត់តែមួយ។ សញ្ញាសម្គាល់មានសារៈសំខាន់សម្រាប់អ្នកបកប្រែដើម្បីកំណត់អត្តសញ្ញាណចុងបញ្ចប់នៃពាក្យបញ្ជា។ ការបែងចែក Code ទៅជាបន្ទាត់តែមួយគឺសម្រាប់តែគម្លាភាព និងលទ្ធភាពនៃការអាន Source code ប៉ុណ្ណោះ។

២.៣.៧ ការបង្ហាញសារជាមួយនឹង alert( )

សារភាគច្រើនជាអត្ថបទ។ ប្រសិនបើយើងចង់ធ្វើការជាមួយអត្ថបទនៅក្នុង JavaScript (យើងនឹងប្រើលេខផងដែរ) វាចាំបាច់ក្នុងការបញ្ចូលអត្ថបទទៅក្នុងសម្រង់។

ពេលខ្លះ apostrophes ត្រូវបានប្រើ គោលបំណងរបស់ពួកគេនឹងត្រូវបានរៀបរាប់នៅពេលក្រោយ។

បញ្ចូល Function alert ទៅក្នុង Code JavaScript ហើយបង្ហាញសារ " First message from programmer!" ដល់អ្នកប្រើប្រាស់។ កុំភ្លេចសញ្ញាក្បៀសនៅចុងបញ្ចប់នៃបន្ទាត់!

ឧទាហរណ៍៖

```
<!doctype html>
<html>
```

```
  <head>
    <title>First JavaScript</title>
  </head>
  <body>
    <script>
        ("First message from programmer!")
    </script>
  </body>
</html>
```

២.៣.៨ ការបង្ហាញសារជាច្រើនជាមួយនឹង alert( )

បញ្ចូល Function សម្រាប់បង្ហាញសារពីរទៅក្នុង Code JavaScipt ។ សារទីមួយនឹងរួមបញ្ចូលអត្ថបទ "Welcome to my web page" និងទីពីរ "Feel free to look around" ។ កុំភ្លេចសញ្ញាក្បៀសនៅចុងបញ្ចប់នៃបន្ទាត់!

ឧទាហរណ៍៖

```
<!doctype html>
<html>
  <head>
    <title>First JavaScript</title>
  </head>
  <body>
    <script>
        ("Welcome to my web page")
        ("Feel free to look around")
    </script>
  </body>
</html>
```

២.៤ លំហាត់

1. ពន្យល់ពីអត្ថបទ "JavaScript ជាភាសាស្ត្រីបលំនាំដើម"។
2. តើអ្វីខ្លះដែលអាចធ្វើបានជាមួយ JavaScript នៅលើគេហទំព័រ?
3. តើអ្វីជាការប្រើប្រាស់នៃ <script> element?
4. តើប្រយោជន៍នៃការបែងចែក JavaScript ទៅក្នុង external files មានអ្វីខ្លះ?
5. តើ comment ជាប្រភេទអ្វីខ្លះនៅក្នុង JavaScript?
6. អាចបញ្ជាក់ពីភាពខុសគ្នារវាង var, let និង const បានទេ?
7. តើអ្វីជា Data Types នៅក្នុង JavaScript?
8. ពន្យល់ពីមុខងារ typeof ក្នុង JavaScript។
9. តើអ្វីជាការកំណត់ Event Handling នៅក្នុង JavaScript?
10. តើបែបបទណាដែលអាចបន្ថែម JavaScript នៅក្នុង HTML?
11. សរសេរ Code HTML ដែលមាន JavaScript ដើម្បីបង្ហាញសារ "Welcome to JavaScript Learning!" ដោយប្រើ alert( ) function។
12. បង្កើត external JavaScript file (example.js) ដែលមាន alert message "Hello from external file!" ហើយភ្ជាប់វាទៅក្នុង HTML។
13. សរសេរ Code JavaScript ដើម្បីបង្ហាញ ២ alert messages៖
    - "Welcome to my website"
    - "Enjoy your stay!"
14. សរសេរ Code JavaScript ដែលប្រើ both single-line និង multi-line comments។
15. សរសេរ Code JavaScript ដើម្បីបង្ហាញ "Hello, JavaScript!" នៅលើ console។
16. បង្កើត script ដែលប្រើ let ដើម្បីប្រកាសអថេរ name ហើយបង្ហាញវាតាម console។
17. សរសេរ Code JavaScript ដើម្បីគណនា ផលនៃ ២ និង ៣ ហើយបង្ហាញលទ្ធផលតាម console

មេរៀនទី ៣៖ អថេរ និង ប្រភេទទន្ទិន្ន័យ

JavaScript គឺជា Dynamically typed language ដែលមានន័យថាប្រភេទអថេរត្រូវបានកំណត់ក្នុងអំឡុង Runtime ជាជាង Compile-time។ ការយល់ដឹងពីរបៀបប្រកាសអថេរ និងគ្រប់គ្រងប្រភេទទិន្នន័យផ្សេងៗគ្នា គឺមានសារៈសំខាន់ណាស់ក្នុងការសរសេរ Code JavaScript ប្រកបដោយប្រសិទ្ធភាពនិងគ្មានកំហុស។ នៅក្នុងមេរៀននេះ យើងនឹងពិភាក្សាអំពីរបៀបប្រកាសអថេរដោយប្រើ let និង const ហើយស្វែងយល់ទាំង Primitive and Reference types ដែលត្រូវបានប្រើនៅក្នុង JavaScript ។

៣.១ ការប្រកាសអថេរ let និង const

JavaScript ផ្តល់នូវវិធីបីយ៉ាងក្នុងការប្រកាសអថេរ៖ var, let, និង const ។ នៅក្នុង modern JavaScript, letនិង const ត្រូវបានគេពេញនិយម ដោយសារតែវាអនុញ្ញាតឱ្យអ្នកអភិវឌ្ឍ Code កំណត់អថេរដោយមានគោលដៅច្បាស់លាស់ជាង var, ដូចជាអថេរដែលត្រូវតែផ្លាស់ប្តូរ (ប្រើ let) និងអថេរមិនត្រូវផ្លាស់ប្តូរបន្ទាប់ពីកំណត់ (ប្រើ const), ធ្វើឱ្យ Code កាន់តែងាយយល់ និងត្រឹមត្រូវ។

៣.១.១ let

ពាក្យគន្លឹះ Let អនុញ្ញាតឱ្យអ្នកប្រកាសអថេរដែលអាចផ្លាស់ប្តូរបាន មានន័យថាតម្លៃរបស់ពួកគេអាចត្រូវបានកំណត់ឡើងវិញនៅពេលក្រោយ។ ទោះយ៉ាងណាក៏ដោយ let គឺជា block-scoped ដែលមានន័យថាពួកវាមាននៅក្នុងប្លុកដែលពួកគេត្រូវបានកំណត់តែប៉ុណ្ណោះ(ជាធម្មតានៅក្នុង curly braces {})។

ក. Property ចម្បងនៃ let៖

- Block-Scoped៖ អថេរដែលប្រកាសជាមួយ let គឺជាការប្រកាសក្នុង block-scoped។ មានន័យថា វាអាចប្រើបានតែនៅក្នុងប្លុក (block) ដែលវាត្រូវបានប្រកាស (ដូចជា {} ក្នុង loops, conditions, functions, etc.)។ ក្រៅប្លុកនោះ អថេរនឹងមិនអាចប្រើបានទេ។

ឧទាហរណ៍៖

```
if (true) {
    let x = 10;
    console.log(x); // 10
}
console.log(x); // ReferenceError: x is not defined
```

- Hoisting but Temporal Dead Zone (TDZ)៖ អថេរដែលប្រកាសជាមួយ let ត្រូវបាន hoisted ដូចជាការប្រកាសផ្សេងៗទៀត ប៉ុន្តែវានឹងស្ថិតនៅក្នុង Temporal Dead Zone (TDZ) រហូតដល់វាត្រូវបានប្រកាសជាក់ស្តែង។ នេះមានន័យថា អ្នកមិនអាចប្រើអថេរ let មុនពេលប្រកាសវានៅក្នុង Code បានទេ។

ឧទាហរណ៍៖

```
console.log(y); // ReferenceError: Cannot access 'y'
before initialization
let y = 5;
```

- Reassignment Allowed៖ អថេរដែលប្រកាសជាមួយ let អាចត្រូវបានផ្លាស់ប្តូរតម្លៃជារៀងរាល់ពេលដែលអ្នកចង់បាន។ វាខុសពី const ដែលមិនអនុញ្ញាតឲ្យកំណត់តម្លៃឡើងវិញ។

ឧទាហរណ៍៖

```
let z = 20;
z = 30; // Allowed
console.log(z); // 30
```

- No Re-declaration in the Same Scope៖ អថេរដែលប្រកាសជាមួយ let មិនអាចត្រូវបានប្រកាសឡើងវិញនៅក្នុងទីកន្លែងសកម្មភាពដូចគ្នានោះទេ។ វាជាការការពារប្រឆាំងនឹងកំហុសដែលអាចកើតមាននៅពេលប្រកាសអថេរដដែលជាច្រើនដង។

ឧទាហរណ៍៖

```
let a = 10;
let a = 20; // រំលឹកError: Identifier 'a' has already been declared
```

៣.១.២ const

ពាក្យគន្លឹះ const នៅក្នុង JavaScript ត្រូវបានប្រើដើម្បីប្រកាសអថេរមួយដែលមានតម្លៃ ថេរ (constant) ហើយមិនអាចត្រូវបានប្តូរតម្លៃម្តងទៀត។ នេះមានន័យថា ពេលដែលអ្នកប្រកាសអថេរ const ហើយផ្តល់តម្លៃមួយ វាមិនអាចត្រូវបានកំណត់តម្លៃថ្មីឡើងវិញទៀតទេ។ ក៏ប៉ុន្តែ, អថេរដែលប្រកាសជាមួយ const ដូចជា objects ឬ arrays ហើយបើទោះបីវាមិនអាចត្រូវបានកំណត់ឡើងវិញ, តែខ្លឹមសារនៃ object ឬ array នោះក៏អាចត្រូវបានផ្លាស់ប្តូរបាន។

ក. Property ចម្បងរបស់ const៖

- ទម្រង់ថេរ (Immutable Reference)៖ អថេរដែលប្រកាសជាមួយ const មិនអាចកំណត់តម្លៃថ្មីឡើងវិញបាន។

ឧទាហរណ៍៖

```
const a = 10;
a = 20; // Error: Assignment to constant variable.
```

- Block-Scoped៖ ដូចជា let, អថេរដែលប្រកាសជាមួយ const គឺមាន block-scoped, មានន័យថាវាមានសកម្មភាពតែនៅក្នុងប្លុកដែលវាត្រូវបានប្រកាស ហើយមិនអាចប្រើបាននៅក្រៅប្លុកនោះទេ។

ឧទាហរណ៍៖

```
if (true) {
    const name = "Alice";
    console.log(name); // "Alice"
}
console.log(name); // ReferenceError: name is not defined
```

- Must Be Initialized៖ អថេរដែលប្រកាសជាមួយ const ត្រូវបានទាមទារឱ្យត្រូវបានកំណត់តម្លៃនៅពេលប្រកាសជាដំបូង ហើយមិនអាចឲ្យកំណត់បាននៅពេលក្រោយ។

ឧទាហរណ៍៖

```
const x; // រូបមន្តError: Missing initializer in const
declaration
x = 5;
```

- Mutability of Objects and Arrays៖ ទោះបី const មិនអនុញ្ញាតឲ្យកំណត់តម្លៃម្តងទៀត, តែក្នុងករណីនៃ objects និង arrays, អ្នកអាចផ្លាស់ប្តូរខ្លឹមសារដូចជា properties ឬ elements ក្នុងនោះ

ឧទាហរណ៍៖

```
const person = { name: "Alice", age: 25 };

person.age = 26; // Allowed, as the object itself is mutable.

console.log(person.age); // 26

const arr = [1, 2, 3];

arr.push(4); // Allowed

console.log(arr); // [1, 2, 3, 4]
```

៣.២ ប្រភេទទិន្នន័យបឋម

៣.២.១ វិធីសាស្ត្រនៃទិន្នន័យបឋម (Methods of primitives)

JavaScript អនុញ្ញាតឱ្យយើងធ្វើការជាមួយប្រភេទទិន្នន័យបឋម (ខ្សែអក្សរ លេខ។ល។) ដូចជាObject។ ពួកគេក៏ផ្តល់នូវវិធីសាស្ត្រដើម្បីហៅដូច្នេះដែរ។ យើងនឹងសិក្សាវាក្នុងពេលឆាប់ៗនេះ ប៉ុន្តែដំបូងយើងនឹងឃើញពីរបៀបដែលវាដំណើរការ ព្រោះជាការពិតណាស់ ប្រភេទទិន្នន័យបឋមមិនមែនជា Object (ហើយនៅទីនេះយើងនឹងធ្វើឱ្យវាកាន់តែច្បាស់)។

សូមក្រឡេកមើលភាពខុសគ្នាសំខាន់ៗរវាងប្រភេទទិន្នន័យបឋម និងObject។

- ប្រភេទទិន្នន័យបឋម

  គឺជាតម្លៃនៃប្រភេទទិន្នន័យបឋម។ មាន ៧ ប្រភេទ ៖ string, number, bigint, boolean, symbolនិង .nullundefined

- ប្រភេទObject

  មានសមត្ថភាពរក្សាទុកតម្លៃច្រើនជាលក្ខណសម្បត្តិ។ អាចត្រូវបានបង្កើតជាមួយ {}ឧទាហរណ៍ ៖ {name៖ "John", age៖ 30}. មានប្រភេទObjectផ្សេងទៀតនៅក្នុង JavaScript៖ functions, for example, are objects.

Object ល្អបំផុតមួយគឺយើងអាចរក្សាទុកមុខងារមួយជាលក្ខណសម្បត្តិរបស់វា។

```
let john = {
  name: "John",
  sayHi: function() {
```

```
    alert("Hi buddy!");
  }
};
john.sayHi(); // Hi buddy!
```

ដូច្នេះនៅទីនេះយើងបានបង្កើត Object មួយ john ជាមួយនឹងវិធីសាស្ត្រ sayHi។ Object ដែលភ្ជាប់មកជាមួយជាច្រើនមានរួចហើយ ដូចជា Object ដែលដំណើរការជាមួយកាលបរិច្ឆេទ កំហុស ធាតុ HTML ជាដើម។ ពួកវាមានលក្ខណសម្បត្តិ និងវិធីសាស្ត្រផ្សេងៗគ្នា។ ប៉ុន្តែលក្ខណៈពិសេសទាំងនេះមកជាមួយតម្លៃ! Object គឺ "Heavier " ជាង Primitive។ ពួកគេត្រូវការធនធានបន្ថែម ដើម្បីគាំទ្រគ្រឿងចក្រខាងក្នុង។

៣.២.២ និទ្ទេសវត្ថុជា Object (A primitive as an object)

នេះជាភាពចម្លែកដែលអ្នកសរសេរ JavaScript ប្រឈមមុខ៖

- មានរឿងជាច្រើនដែលចង់ធ្វើជាមួយប្រភេទទិន្នន័យបឋម ដូចជាខ្សែអក្សរ ឬលេខ។ វាជាការល្អណាស់ក្នុងការចូលប្រើពួកវាដោយប្រើវិធីសាស្ត្រ។
- ប្រភេទទិន្នន័យបឋមត្រូវតែលឿន និងស្រាលតាមដែលអាចធ្វើទៅបាន។

ដំណោះស្រាយមើលទៅហាក់បីដូចជាឆ្គងបន្តិច ប៉ុន្តែវាគឺ៖

1. ប្រភេទទិន្នន័យបឋមនៅតែជាប្រភេទទិន្នន័យបឋម។ តម្លៃតែមួយតាមការចង់បាន។
2. ភាសាអនុញ្ញាតឱ្យចូលប្រើវិធីសាស្ត្រ និងProperty នៃខ្សែអក្សរ លេខ ប៊ូលីន និងនិមិត្តសញ្ញា។
3. ដើម្បីឱ្យវាដំណើរការ " Object wrapper " ពិសេសដែលផ្តល់មុខងារបន្ថែមត្រូវបានបង្កើតឡើង ហើយបន្ទាប់មកត្រូវបានបំផ្លាញ។

- " Object wrapper " មានភាពខុសប្លែកគ្នាសម្រាប់ប្រភេទទិន្នន័យបឋមនីមួយៗ ហើយត្រូវបានគេហៅថា ៖ String, Number, Boolean, Symbol និង BigInt. ដូច្នេះពួកគេផ្តល់សំណុំវិធីសាស្ត្រផ្សេងៗគ្នា។

  ឧទាហរណ៍៖ មានstring method str.toUpperCase( ) ដែលត្រឡប់អក្សរធំ str។

នេះជារបៀបដែលវាដំណើរការ៖

```
let str = "Hello";
alert( str.toUpperCase() ); // HELLO
```

សាមញ្ញទេ ? នេះជាអ្វីដែលពិតជាកើតឡើងនៅក្នុង str.toUpperCase( )៖

1. ខ្សែអក្សរ strគឺជាប្រភេទទិន្នន័យបឋម។ ដូច្នេះក្នុងពេលចូលប្រើប្រាស់លក្ខណៈសម្បត្តិរបស់វា Object ពិសេសមួយត្រូវបានបង្កើតឡើងដែលដឹងពីតម្លៃនៃខ្សែអក្សរ ហើយមានវិធីសាស្ត្រដែលមានប្រយោជន៍ ដូចជា toUpperCase( ).
2. វិធីសាស្ត្រនោះរត់និងត្រឡប់ខ្សែអក្សរថ្មី (បង្ហាញដោយ alert) ។
3. Object ពិសេសត្រូវបានបំផ្លាញដោយបន្សល់ទុក strតែប្រភេទទិន្នន័យបឋម។

ដូច្នេះ primitives អាចផ្តល់នូវវិធីសាស្ត្រ ប៉ុន្តែពួកគេនៅតែមានទម្ងន់ស្រាល។

ម៉ាស៊ីន JavaScript ធ្វើឱ្យដំណើរការនេះមានប្រសិទ្ធភាពខ្ពស់។ វាថែមទាំងអាចរំលងការបង្កើតObject បន្ថែមទៀតផង។ ប៉ុន្តែវានៅតែត្រូវប្រកាន់ខ្ជាប់នឹងការបញ្ជាក់និងប្រព្រឹត្តដូចជាវាបង្កើតមួយ។

លេខមានវិធីសាស្ត្ររបស់វាផ្ទាល់ ឧទាហរណ៍ toFixed(n) បង្គត់លេខទៅភាពជាក់លាក់ដែលបានផ្តល់ឱ្យ

```
let n = 1.23456;

alert( n.toFixed(2) ); // 1.23
```

៣.២.៣ ខ្សែអក្សរ (String)

ប្រភេទ String តំណាងឲ្យអក្សរតួ (characters) ឬអក្សរច្រើនដែលស្ថិតក្នុងស៊ូតាក្រដាស (sequence of characters) ហើយត្រូវបានដាក់ក្នុង "" ឬ "។

ឧទាហរណ៍៖

```
let name = "Alice";

let greeting = 'Hello, world!';
```

៣.២.៤ លេខ (Number)

ប្រភេទលេខតំណាងឲ្យចំនួនគត់ទាំងអស់ជាប្រភេទវិជ្ជមាន, អវិជ្ជមាន (integer) និងទសភាគ (floating-point) នៅក្នុង JavaScript។

ឧទាហរណ៍៖

```
let age = 25;

let price = 99.99;
```

៣.២.៥ តម្លៃពិត/មិនពិត (Boolean)

ប្រភេទ Boolean មានតម្លៃចំនួន 2 គឺ true និង false ត្រូវបានប្រើសម្រាប់តំណាងឲ្យត្រឹមត្រូវ (truthy) ឬកំហុស (falsy) ។

ឧទាហរណ៍៖

```
let isOnline = true;

let isAdult = false;
```

៣.២.៦ មិនបានកំណត់ (Undefined)

Undefined ត្រូវបានកំណត់អំពីអថេរដែលត្រូវបានប្រកាសតែមិនទាន់បានផ្តល់តម្លៃ។ អថេរនឹងទទួលតម្លៃ undefined បើវាមិនបានកំណត់តម្លៃនៅពេលប្រកាស។

ឧទាហរណ៍៖

```
let x;

console.log(x); // Output: undefined
```

៣.២.៧ គ្មានតម្លៃ (Null)

Null ជាការកំណត់តម្លៃដោយចេតនាឲ្យអថេរមានតម្លៃទទេ (empty value) ។ វាប្រៀបបាននឹងគ្មានតម្លៃត្រូវបានបង្កប់នៅក្នុងអថេរមួយ។

ឧទាហរណ៍៖

```
let person = null;
```

៣.២.៨ សញ្ញា (Symbol)

សញ្ញាត្រូវបានបន្ថែមនៅក្នុង ECMAScript 2015 (ES6) ហើយវាតំណាងឲ្យឯកត្តា (unique) ដែលមិនស្ទួន។ Symbol ត្រូវបានប្រើសម្រាប់ការបង្កើតការផ្លាស់ប្តូរដែលមានឈ្មោះតែមួយឬប្រើប្រាស់តម្លៃត្រូវតែមួយមិនអាចធៀបបាន។

ឧទាហរណ៍៖

```
let symbol1 = Symbol('description');

let symbol2 = Symbol('description');

console.log(symbol1 === symbol2); // false, because each symbol is
unique
```

៣.២.៩ ចំនួនគត់ធំ (Bigint)

ចំនួនគត់ធំត្រូវបានបន្ថែមនៅក្នុង ECMAScript 2020 ដើម្បីដំណើរការលេខធំៗជាងលេខធម្មតា (Number) ដែល JavaScript អាចគ្រប់គ្រងបាន។ វាត្រូវបានប្រើសម្រាប់តម្លៃដែលធំជាង 2^53-1 ឬតូចជាង -(2^53-1) ។

ឧទាហរណ៍៖

```
let bigIntNumber = 9007199254740991n; // Adding 'n' makes it a BigInt
```

៣.៣ ប្រភេទទិន្នន័យដែលមិនមែនជាបឋម (Non-Primitive Data Types)

Object Data Types នៅក្នុង JavaScript គឺជាប្រភេទទិន្នន័យដែលអាចផ្ទុកតម្លៃជាច្រើនដោយប្រើតំណាងឲ្យគ្រឿងផ្សំផ្សេងៗ។ វាអាចផ្ទុកទិន្នន័យនិងមុខងារ (functions) និងគេហៅថា objects។ ប្រភេទទិន្នន័យនេះមានProperty គឺ non-primitive និង mutable, មានន័យថា អ្នកអាចបន្ថែម ប្តូរ ឬ លុបតម្លៃនៅក្នុង object បាន។

៣.៣.១ Objects

Objects តំណាងឲ្យទិន្នន័យដែលមានគ្រឿងផ្សំជាច្រើន (properties) ដែលស្ថិតក្នុងការតំណាងឲ្យគ្រឿងផ្សំដែលមានអត្ថន័យផ្ទាល់ខ្លួន។ ការប្រើប្រាស់មធ្យោបាយ {} ដើម្បីបង្កើត object និងការប្រើប្រាស់ការចូលដំណើរការ និងកែប្រែដោយdot notation ឬ bracket notation ។

ឧទាហរណ៍៖

```
// Creating an object
let person = {
    name: "Alice",
    age: 30,
    greet: function() {
        console.log("Hello!");
    }
};
// Accessing properties
console.log(person.name); // Output: Alice
```

```
console.log(person['age']); // Output: 30

// Calling a method

person.greet(); // Output: Hello!
```

៣.៣.២ អារេ (Array)

អារេគឺជា object ដែលបានផ្ទុកបញ្ជីនៃទិន្នន័យក្នុងលំដាប់ជាក់លាក់ (ordered collection of items) ។ប្រើសញ្ញាអក្សរអ្នកនឹងកំណត់ឲ្យបានល្អសម្រាប់ការផ្ទុកទិន្នន័យជាបញ្ជី។

ឧទាហរណ៍៖

```
let numbers = [1, 2, 3, 4, 5];

// Accessing elements

console.log(numbers[0]); // Output: 1

// Modifying elements

numbers[2] = 10;

console.log(numbers); // Output: [1, 2, 10, 4, 5]

// Adding elements

numbers.push(6);

console.log(numbers); // Output: [1, 2, 10, 4, 5, 6]
```

៣.៣.៣ អនុគមន៍ (Functions)

អនុគមន៍ គឺជា object ដែលអាចត្រូវបានប្រើដើម្បីផ្តល់នូវអនុគមន៍ដើម្បីអនុវត្តន៍ការងារតាមលំដាប់សំរាប់ការកំណត់មួយ។

ឧទាហរណ៍៖

```
function add(a, b) {
    return a + b;
}

// Calling the function

console.log(add(5, 3)); // Output: 8
```

៣.៣.៤ Classes

Classes គឺជា object ដែលមានការប្រើប្រាស់ដើម្បីបង្កើត objects ជាមួយនឹងលក្ខណៈនិងមុខងារដូចគ្នា។ Classes ត្រូវបានបង្កើតតាមរយៈពាក្យគន្លឹះ class ហើយបានទទួលប្រើប្រាស់នូវ constructor ដើម្បីបង្កើត objects និង method ក្នុងវា។

ឧទាហរណ៍៖

```
class Person {
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }
    greet() {
        console.log(`Hello, my name is ${this.name}`);
    }
}
let person1 = new Person("Alice", 30);
person1.greet(); // Output: Hello, my name is Alice
```

៣.៤ លំហាត់

1. តើភាពខុសគ្នារវាង let និង const នៅក្នុង JavaScript ដូចម្តេចខ្លះ? ចូលលើកឧទាហរណ៍។
2. ពន្យល់ពីគោលគំនិតនៃ Block scope និងរបៀបដែលវាអនុវត្តចំពោះ let និង const ។
3. ហេតុអ្វីបានជាត្រូវបានណែនាំឱ្យប្រើ const តាមលំនាំដើមនៅពេលប្រកាសអថេរ?
4. តើនឹងមានអ្វីកើតឡើង ប្រសិនបើអ្នកព្យាយាមកំណត់អថេរ const ឡើងវិញ? ចូលលើកឧទាហរណ៍។
5. តើភាពខុសគ្នារវាង null និង undefined នៅក្នុង JavaScript ដូចម្តេចខ្លះ? ចូលលើកឧទាហរណ៍។
6. តើ Symbol នៅក្នុង JavaScript គឺជាអ្វី? តើវាខុសពីប្រភេទទិន្នន័យបឋមផ្សេងទៀតយ៉ាងដូចម្តេច?

7. តើភាពខុសគ្នារវាង Array and an object ក្នុង JavaScript ដូចម្តេចខ្លះ??
8. ចូលពន្យល់ពីរបៀបដែល Array នៅក្នុង JavaScript អាចផ្ទុកធាតុនៃប្រភេទទិន្នន័យផ្សេងៗគ្នា។ ចូលលើកឧទាហរណ៍។
9. តើមានអ្វីកើតឡើងនៅពេលដែលអ្នកព្យាយាមចូលប្រើ Index នៅក្នុង Array ដែលមិនមាន? ចូលសរសេរឧទាហរណ៍ដើម្បីបង្ហាញ។
10. ចូលប្រកាស Object តំណាងឱ្យសៀវភៅដែលមាន Properties ដូចជាចំណងជើង អ្នកនិពន្ធ និងឆ្នាំបោះពុម្ព។ Print តម្លៃរបស់អ្នកនិពន្ធដោយប្រើសញ្ញាចុច. និងសញ្ញាតង្កៀប[]។

មេរៀនទី ៤៖ ប្រមាណវិធី (Operators)

នៅក្នុង JavaScript មានប្រមាណវិធីសម្រាប់អនុវត្តប្រតិបត្តិការផ្សេងៗគ្នាលើអថេរ(Variables) និងទិន្នន័យ(Data)។ ការយល់ដឹងពីរបៀបប្រើពួកវាឱ្យបានត្រឹមត្រូវគឺមានសារៈសំខាន់សម្រាប់ការសរសេរ Code ដ៏ល្អ, ប្រសិទ្ធភាព និងអាចរក្សាបាន។ មេរៀននេះស្វែងយល់អំពីប្រភេទ operators ផ្សេងៗ ដូចជា ប្រមាណវិធីនព្វន្ធ, ប្រមាណវិធីប្រៀបធៀប, ប្រមាណវិធីតក្កវិជ្ជា, ប្រមាណវិធីផ្តល់តម្លៃ និងច្រើនទៀត។ យើងក៏នឹងពិភាក្សាអំពី Operator precedence and associativity ផងដែរ ដែលដើរតួយ៉ាងសំខាន់ក្នុងរបៀបដែលត្រូវបានវាយតម្លៃកន្សោម។

៤.១ ប្រមាណវិធីនព្វន្ធ (Arithmetic Operators)

Arithmetic Operators អនុញ្ញាតឱ្យអ្នកអនុវត្តការគណនាគណិតវិទ្យានៅក្នុងកម្មវិធីរបស់អ្នក។ ប្រភេទនៃ arithmetic operators ខាងក្រោម៖

ឧទាហរណ៍៖

```
let x = 10;
let y = 5;
let result = x + y; // result is 15
```

តារាង 1៖ Arithmetic Operators

និមិត្តសញ្ញា	អត្ថន័យ	ឧទាហរណ៍	លទ្ធផល
+	បូក	10+2	12
-	ដក	10-5	5
*	គុណ	5 * 5	25
/	ចែក	10/2	8
%	ចែកយកសំណល់	5%3	2
* *	ស្វ័យគុណ២	52	25
++	បូកបន្ថែមម្តងមួយ	5++	6

--	ដកចេញម្តងមួយ	5--	4

៤.២ ប្រមាណវិធីប្រៀបធៀប (Comparison Operators)

Comparison Operators គឺជា Operator ដែលអនុញ្ញាតឱ្យអ្នកប្រៀបធៀបតម្លៃពីរ។ វាល់ការប្រៀបតែងតែបោះតម្លៃ Boolean (true or false) ដោយផ្អែកលើថាតើការប្រៀបធៀបត្រឹមត្រូវឬអត់។ ចូលពិនិត្យលក្ខខណ្ឌដូចខាងក្រោម៖

ឧទាហរណ៍៖

```
let result = (5 == '5'); // true (due to type coercion)
```

តារាង 2៖ Comparison Operators

និមិត្តសញ្ញា	**អត្ថន័យ**	**ឧទាហរណ៍**	**លទ្ធផល**
= =	ស្មើ (equal)	5= =3	False
= = =	ស្មើទាំងតម្លៃ និងប្រភេទ ទិន្នន័យដូច	x === "5"	False
!=	ខុសពី	5!=3	True
!==	ប្រៀបធៀបទាំងតម្លៃ និង ប្រភេទ	5!== '5'	True
<	តូចជាង	5<3	False
<=	តូចជាងឬស្មើ	5<=3	False
>	ធំជាង	5>3	True
>=	ធំជាងឬស្មើ	5>=3	True

?	ប្រើសម្រាប់សិក្សាលក្ខខណ្ឌ ដូច if/else	condition ? <expression if true> ៖ <expression if false>	

៤.៣ ប្រមាណវិធីតក្កវិជ្ជា (Logical Operators)

ជាឈ្នាប់មួយសម្រាប់តក្កវិជ្ជាដែលមានតម្លៃតែ True and False ប៉ុណ្ណោះ។ យើងដឹងថា ឈ្នាប់នឹង (&&) ពិតលុះត្រាតែលក្ខខណ្ឌទាំងសងខាងពិតទាំងពីរ ចំណែកឯឈ្នាប់ឬ (||) ពិតក្នុងករណីដែលលក្ខខណ្ឌណាមួយពិត។ ឈ្នាប់មិន គឺបញ្ច្រាស់លក្ខខណ្ឌ បើលក្ខខណ្ឌពិត (True) វានឹងក្លាយជាមិនពិត (False)។ប្រភេទនៃ Logical operators ៖

ឧទាហរណ៍៖

```
let result = (5 > 2 && 10 < 15); // true
```

តារាង 3៖ Logical Operators

និមិត្តសញ្ញា	អត្ថន័យ	ឧទាហរណ៍	លទ្ធផល
&&	ឈ្នាប់នឹង	(5>3) && (5!=3)	True
||	ឈ្នាប់ឬ	(5<3) || (5=3)	False
!	ឈ្នាប់មិន	!(5<3)	True

៤.៤ ប្រមាណវិធីផ្តល់តម្លៃ (Assignment Operators

ប្រមាណវិធីដោយផ្តល់តម្លៃក្នុង JavaScript គឺជាឧបករណ៍ដែលប្រើសម្រាប់ផ្តល់តម្លៃទៅអថេរ (variables) ឬកំណត់តម្លៃសម្រាប់អថេរនោះ។ ឧបករណ៍ទាំងនេះមិនត្រឹមតែផ្តល់តម្លៃនោះទេ ប៉ុន្តែអាចធ្វើសកម្មភាពគណិតវិទ្យាផ្សេងៗទៀតៗប្រភេទនៃ Assignment Operators៖

ឧទាហរណ៍៖

```
let x = 10;
x += 5; // x = x + 5;
```

```
console.log(x); // output: 15
```

តារាង 4៖ Assignment Operators

និមិត្តសញ្ញា	អត្ថន័យ	ឧទាហរណ៍	លទ្ធផល
=	ផ្តល់តម្លៃមួយទៅអថេរ	x = 10	10
+=	បន្ថែមតម្លៃទៅអថេរដែលមានស្រាប់	x += 5	15
-=	ដកតម្លៃពីអថេរដែលមានស្រាប់	x -= 5	10
*=	គុណតម្លៃថ្មីជាមួយអថេរដែលមានស្រាប់	x *= 2	20
/=	ចែកតម្លៃថ្មីដោយអថេរដែលមានស្រាប់	x /= 2	10
%=	ផ្តល់តម្លៃសំណល់ក្រោយការចែក	x %= 3	1
**=	បង្កើតស្វ័យគុណ២ទៅអថេរដែលមានស្រាប់	x **= 2	1

៤.៥ ប្រមាណវិធីត្រីភាគ (Ternary Operator)

ប្រមាណវិធីត្រីភាគក្នុង JavaScript គឺជាវិធីសម្រាប់សរសេរលក្ខខណ្ឌ (condition) ដោយសង្ខេបជាការសម្រេចចិត្តក្នុងមួយជួរ។ វាជាឧបករណ៍ដែលអនុញ្ញាតឱ្យអ្នកពិនិត្យលក្ខខណ្ឌ និងបញ្ជូនតម្លៃដាក់អថេរឬដាក់សកម្មភាពផ្សេងៗ ដោយផ្អែកលើលទ្ធផលនៃការពិនិត្យលក្ខខណ្ឌនោះ។

រូបមន្ត៖

```
condition? expr1 : expr2
```

- condition៖ លក្ខខណ្ឌដែលត្រូវត្រួតពិនិត្យ (ត្រូវបានវាយជាតម្លៃ boolean true ឬ false)។
- expr1៖ តម្លៃ ឬសកម្មភាពដែលនឹងត្រូវអនុវត្ត បើលក្ខខណ្ឌមានតម្លៃជា true។
- expr2៖ តម្លៃ ឬសកម្មភាពដែលនឹងត្រូវអនុវត្ត បើលក្ខខណ្ឌមានតម្លៃជា false។

Ternary Operator គឺជាជំនួយដើម្បីប្តូរពាក្យ if...else ធម្មតាទៅជាទម្រង់ Code ខ្លី។

ឧទាហរណ៍៖

```
let age = 18;
let status = (age >= 18) ? 'adult' : 'minor';
// status is 'adult'
```

៤.៦ ប្រមាណវិធី Bitwise

Operators Bitwise ចាត់ទុកប្រតិបត្តិកររបស់ពួកគេជាលំដាប់នៃ 32 bits (binary digits) ហើយដំណើរការនៅកម្រិតនៃ bits នីមួយៗ។ទោះបីជា Operators Bitwise កម្រត្រូវបានប្រើក្នុងការសរសេរកម្មវិធី JavaScript ប្រចាំថ្ងៃក៏ដោយ ពួកវាអាចមានប្រយោជន៍ក្នុងស្ថានភាពមួយចំនួនដែលពាក់ព័ន្ធនឹងការគណនាកម្រិតទាប។ប្រភេទនៃ Bitwise Operators៖

ឧទាហរណ៍៖

```
let a = 5;  // 0101 in binary
let b = 3;  // 0011 in binary
console.log(a & b);  // 0001 -> 1
console.log(a | b);  // 0111 -> 7
console.log(a ^ b);  // 0110 -> 6
console.log(~a);     // 1010 -> -6 (because of two's complement)
console.log(a << 1); // 1010 -> 10
console.log(a >> 1); // 0010 -> 2
```

តារាង 5៖ Bitwise Operators

និមិត្តសញ្ញា	ឧទាហរណ៍	លទ្ធផល
&	5 & 3	1
|	5 | 3	7
^	5 ^ 3	6
~	~5	-6

<<	5 << 1	10
>>	5 >> 1	2
>>>	5 >>> 1	2

៤.៧ លំដាប់ និងទំនាក់ទំនង (Precedence and Associativity)

៤.៧.១ ប្រមាណវិធី Precedence

Operator Precedence និយាយអំពីលំដាប់នៃការដំណើរការ operators ក្នុងពេលដែលច្រើនប្រើនៅក្នុង Code មួយ។ ប្រតិបត្តិការដែលមាន precedence ខ្ពស់នឹងត្រូវបានគណនាមុនបំផុត។

ឧទាហរណ៍៖

```
let result = 3 + 4 * 2; // result is 11, not 14
```

លំដាប់ Precedence ខ្លះក្នុង JavaScript៖

- ( ) (Parentheses) – Precedence ខ្ពស់បំផុត
- * * * * (Exponentiation)
- * * * * * (Multiplication), / (Division), % (Modulus)
- + (Addition), - (Subtraction)
- <, > (Comparison)
- ==, ===, !=, !== (Equality)
- && (Logical AND), || (Logical OR)
- = (Assignment) – Precedence ទាបបំផុត

៤.៧.២ ទំនាក់ទំនងប្រមាណវិធី (Operator Associativity)

ទំនាក់ទំនងប្រមាណវិធីបានកំណត់លំដាប់នៃការគណនាក្នុងករណីប្រតិបត្តិការដែលមាន precedence ដូចគ្នា។ វានិយាយថាប្រតិបត្តិការនោះត្រូវបានគណនាដោយចាប់ផ្តើមពីខាង ឆ្វេងទៅស្តាំ ឬ ស្តាំទៅឆ្វេង។

ក. Left-to-Right Associativity៖ ប្រតិបត្តិការត្រូវបានគណនាជាលំដាប់ពីឆ្វេងទៅស្តាំ។ ប្រតិបត្តិការទូទៅជាងគេមាន associativity មួយនេះ។

ឧទាហរណ៍៖

```
let result = 10 - 5 - 1; // Output: 4
```

ខ. Right-to-Left Associativity៖ ប្រតិបត្តិការត្រូវបានគណនាពីស្តាំទៅឆ្វេង។ ប្រតិបត្តិការដូចជា assignment operators និង exponentiation មាន associativity មួយនេះ។

ឧទាហរណ៍៖

```
let x = y = 5; // y gets 5, then x gets 5
```

Associativity ចំនួនខ្លះ៖

- Left-to-right៖ +, -, *, /, %, <, >, ==, ===, &&
- Right-to-left៖ =, +=, -=, **, **=, !, ~

គ. Parentheses៖ Parentheses (()) មាន precedence ខ្ពស់បំផុត ហើយអាចប្រើដើម្បីបញ្ជូនលំដាប់ precedence ការគណនាដោយដាក់លាក់។

ឧទាហរណ៍៖

```
let result = (5 + 3) * 2; // Output: 16
```

៤.៨ លំហាត់

1. តើ === និង == ផ្សេងគ្នាយ៉ាងដូចម្តេច? ចូលលើកឧទាហរណ៍ និងពន្យល់ពីលទ្ធផល។
2. តើ Bitwise AND (&) និង Logical AND (&&) មានភាពខុសគ្នាយ៉ាងណា? ចូលលើកឧទាហរណ៍នៃការប្រើប្រាស់ Bitwise AND និង Logical AND ដើម្បីបង្ហាញភាពខុសគ្នា។
3. ចូរពន្យល់នូវ Operator Precedence និង Associativity។ ចូលលើកឧទាហរណ៍ពីប្រយោគដែលលុបចោល Associativity ដូចជា 5 + 3 * 2 - 1។
4. តើ Symbolic Operators គឺជាអ្វី?
   - ចូលលើកឧទាហរណ៍នៃការប្រើ Symbolic Operators ដូចជា +, -, *, និង **។
   - ចូលបង្ហាញនៃលទ្ធផល 2 ** 3 ** 2?
5. តើ Logical NOT (!) ធ្វើការជាអ្វី? ចូលសរសេរ Code ដែលបង្ហាញលទ្ធផលនៃ !true, !false, និង !!0
6. ចូលប្រកាសអថេរពីរគឺ a = 8 និង b = 4 ។ អនុវត្ត operations ខាងក្រោម ហើយកត់ត្រាលទ្ធផល

- Addition
- Subtraction
- Multiplication
- Division
- Modulus
- Exponentiation

7. ចូលសរសេរកម្មវិធីដែល៖
   - ប្រកាស x = 15 និង y = '15' ។
   - ប្រើ loose equality (==) និង strict equality (===) ដើម្បីប្រៀបធៀបតម្លៃ។
   - ប្រៀបធៀបការប្រើធំជាង (>), តិចជាង (<), និងមិនស្មើគ្នា (!=) ប្រតិបត្តិករ។
8. ចូលសរសេរកម្មវិធីដើម្បីពិនិត្យមើលថាតើអ្នកណាអាចបោះឆ្នោតបាន៖
   - ប្រកាសអាយុអថេរ ហើយកំណត់តម្លៃពីការបញ្ចូលរបស់អ្នកប្រើ (ប្រើ prompt( ))។
   - ប្រើ logical operators ដើម្បីពិនិត្យមើលថាតើបុគ្គលនោះមានសិទ្ធិបោះឆ្នោតឬអត់ (អាយុ ≥ 18) និងជាពលរដ្ឋ (isCitizen = ពិត)។
9. ចូលបង្កើតកម្មវិធីមួយដែលប្រកាសចំនួនអថេរ = 10 ។ អនុវត្តប្រតិបត្តិការខាងក្រោមដោយប្រើ assignment operators៖
   - បន្ថែម 5 ដើម្បីរាប់។
   - គុណនឹង 2 ។
   - ដក 4 ចេញពីការរាប់។
   - ចែកចំនួនដោយ 3 ។
   - អនុវត្ត modulus ជាមួយ 3 ។
10. សរសេរកម្មវិធីដែល៖
    - ស្នើឱ្យអ្នកប្រើប្រាស់បញ្ចូលលេខដោយប្រើ prompt( )។
    - ប្រើ ternary operator ដើម្បីពិនិត្យមើលថាតើលេខវិជ្ជមាន អវិជ្ជមាន ឬសូន្យ។
11. ចូលសរសេរកម្មវិធីដែលប្រកាសអថេរពីរ a = 5 និង b = 3 ។អនុវត្ត 0perator ដូចខាងក្រោម:
    - Bitwise AND (&)
    - Bitwise OR (|)
    - Bitwise XOR (^)

- Left Shift (<<)
- Right Shift (>>)

12. ទស្សន៍ទាយលទ្ធផលនៃកន្សោមខាងក្រោម រួចសរសេរ Code ដើម្បីផ្ទៀងផ្ទាត់៖
    - let result = 3 + 4 * 5 - 2;
    - let result = (3 + 4) * (5 - 2);
    - let result = 10 / 2 + 5 * 3 - 1;
    - let result = 2 ** 3 ** 2;
13. ចូលបង្កើតម៉ាស៊ីនគណនា BMI សាមញ្ញដោយប្រើ JavaScript៖
    - សួរអ្នកប្រើប្រាស់សម្រាប់ទម្ងន់ (គិតជាគីឡូក្រាម) និងកម្ពស់ (គិតជាម៉ែត្រ) ដោយប្រើប្រអប់បញ្ចូល ( ) ។
    - គណនា BMI ដោយប្រើរូបមន្ត៖ BMI = weight / (height * height)
    - ប្រើ ternary operator ដើម្បីកំណត់ថាតើមនុស្សនោះមានទម្ងន់ក្រោម ធម្មតា លើសទម្ងន់ ឬធាត់។
14. ចូលសរសេរកម្មវិធីដែលអ្នកប្រើប្រាស់បញ្ចូលលេខពីរ និងលក្ខខណ្ឌឡូជីខល (AND, OR, NOT)។ កម្មវិធីគួរតែត្រឡប់លទ្ធផលដោយផ្អែកលើការបញ្ចូល។
    - អ្នកប្រើបញ្ចូលលេខពីរ និងសញ្ញាប្រមាណវិធី (+, -, *, /, %, **) ។
    - ប្រើ switch statement ដើម្បីអនុវត្តប្រតិបត្តិការដែលបានជ្រើសរើស។

មេរៀនទី ៥៖ Control Flow statements

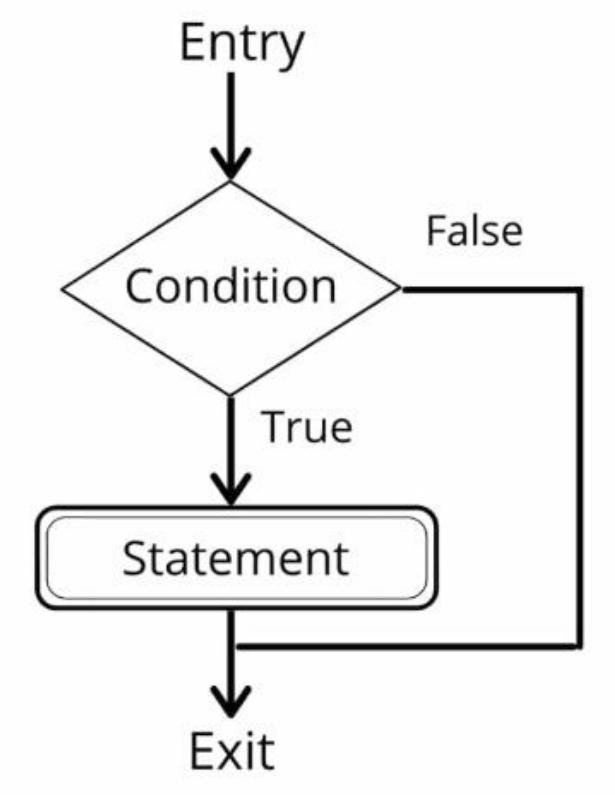
Control flow statements មានសារៈសំខាន់ក្នុងការសរសេរកម្មវិធី ដោយសារវាជួយគ្រប់គ្រងផ្លូវប្រតិបត្តិនៃ Code ដោយផ្អែកលើលក្ខខណ្ឌជាក់លាក់ ឬប្រតិបត្តិ Code ម្តងហើយម្តងទៀត។ មេរៀននេះគ្របដណ្តប់សំខាន់ទៅលើ control flow statements in JavaScript, រួមទាំង conditional statements and loops។

៥.១ Conditional Statements

Conditional statements អនុញ្ញាតឱ្យកម្មវិធីរបស់អ្នកធ្វើការសម្រេចចិត្តដោយផ្អែកលើលក្ខខណ្ឌជាក់លាក់។នៅក្នុងJavaScript, the primary conditional statements គឺif, elseif, else, និង switch។

៥.១.១ if Statement

if statement ជា Statement ដែលគេប្រើសម្រាប់ត្រួតពិនិត្យមើលលក្ខខណ្ឌពិត ឬមិនពិត។



រូបភាព 2៖ If Statement

រូបមន្ត៖

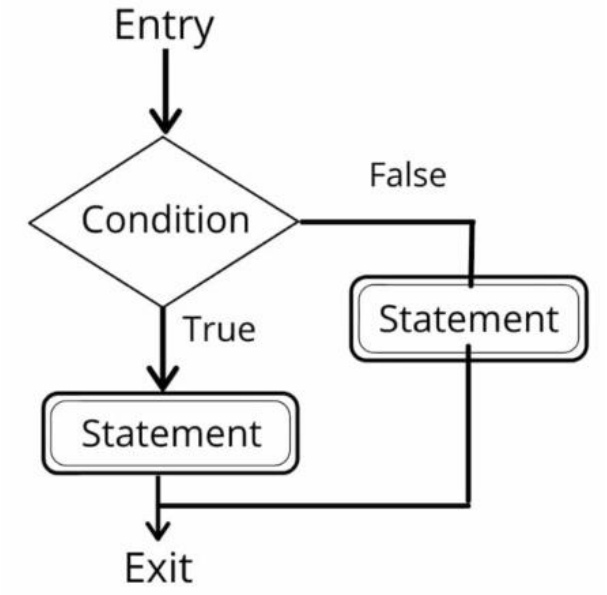
```
if (condition) {
    // code to execute if condition is true
}
```

ឧទាហរណ៍៖

```
let age = 20;
if (age >= 18) {
    console.log("You are an adult.");
}
```

៥.១.២ else if and else Statements

The else if statement អនុញ្ញាតឱ្យអ្នកពិនិត្យមើលលក្ខខណ្ឌជាច្រើនតាមលំដាប់លំដោយ។ ចំពោះ else statement ប្រតិបត្តិប្រសិនបើគ្មានលក្ខខណ្ឌមុនណាមួយត្រូវបានបំពេញ។



រូបភាព 3៖ else if and else Statements

រូបមន្ត៖

```
if (condition1) {
    // code to execute if condition1 is true
} else if (condition2) {
    // code to execute if condition2 is true
```

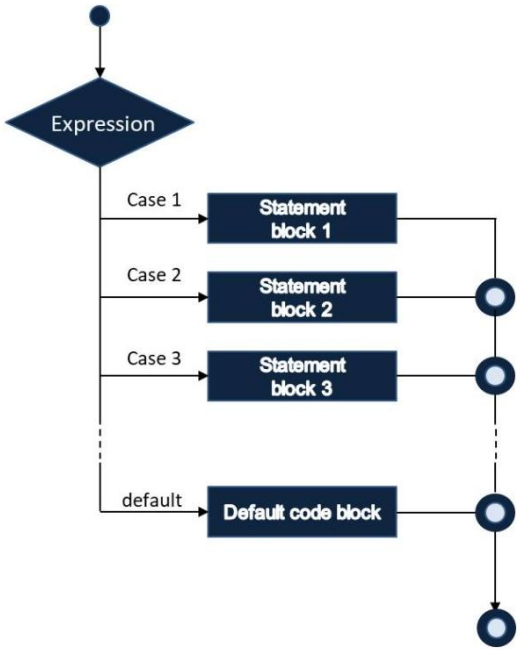
```
} else {
    // code to execute if none of the above conditions are true
}
```

ឧទាហរណ៍៖

```
let score = 85;
if (score >= 90) {
    console.log("Grade: A");
} else if (score >= 80) {
    console.log("Grade: B");
} else {
    console.log("Grade: C");
}
```

៥.១.៣ switch Statement

យើងអាចប្រើប្រាស់នូវ Switch Statement ដើម្បីជំនួសឱ្យការប្រើប្រាស់នូវ IF ឬ If …else ឬ if …else if Statement បាន។ ដោយយើងគ្រាន់តែវាយតម្លៃលើ Expression មួយ ហើយលោតទៅកាន់ Statement ដែលគ្មាន Label ជាមួយ Case Clause ដែលត្រូវគ្នា នឹងតម្លៃរបស់ Expression នោះ។ បើគ្មាន Case ណាមួយត្រូវគ្នាទេ នោះ switch statement នឹងលោតទៅកាន់ Statement ដែលមាន Label ដោយ Default។



រូបភាព 4៖ switch Statement

រូបមន្ត៖

```
switch (expression) {
    case value1:
        // code to execute if expression equals value1
        break;
    case value2:
        // code to execute if expression equals value2
        break;
    default:
        // code to execute if expression does not match any case
}
```

ឧទាហរណ៍៖

```
let day = 3;

switch (day) {
```

```
    case 1:
        console.log("Monday");
        break;
    case 2:
        console.log("Tuesday");
        break;
    case 3:
        console.log("Wednesday");
        break;
    default:
        console.log("Other day");
}
```

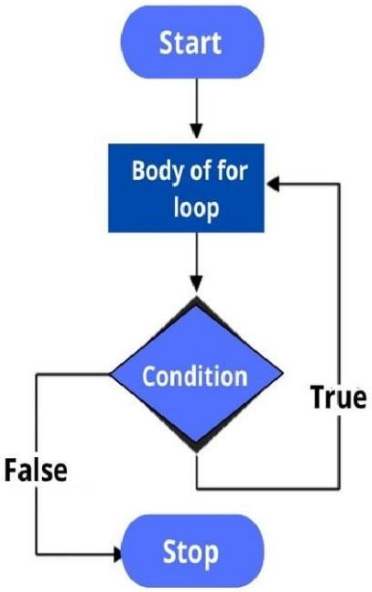
៥.២ រង្វិលដុំ (Loops)

Loops ត្រូវបានគេប្រើនូវពេលដែលលោកចង់ឱ្យ Statement ណាមួយដំណើរការដដែលៗរហូតម្តងហើយម្តងទៀត ហើយឱ្យវាឈប់នៅពេលជួបលក្ខខណ្ឌណាមួយ។

Loops មានដូចជា៖ for, while, do...while, for...in, and for...of ។

៥.២.១ for Loop

For Loop Statement ជា Loop Statement ដែលដំណើរការម្តងហើយម្តងទៀតរហូតដល់អស់លក្ខខណ្ឌមានន័យថា បើលក្ខណ្ឌនៅតែពិត វានៅតែដំណើរការ។ វាមានបីផ្នែកគឺ៖ initialization, condition, and iteration។



រូបភាព 5៖ for Loop

រូបមន្ត៖

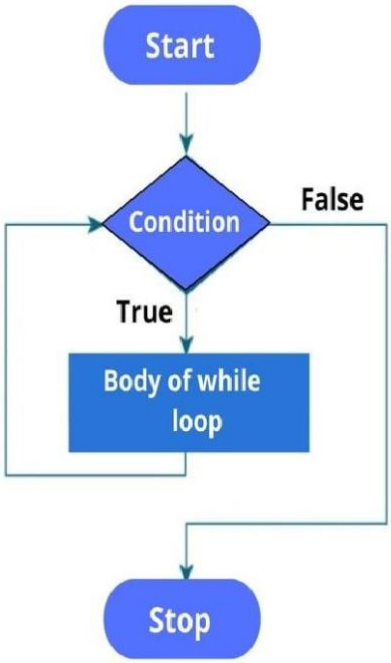
```
for (initialization; condition; iteration) {
    // code to execute in each loop iteration}
```

ឧទាហរណ៍៖

```
for (let i = 0; i < 5; i++) {
    console.log(i);
}
```

៥.២.២ while Loop

ជា Pre-test loop ដែលមានន័យថា វាធ្វើការពិនិត្យលើលក្ខខណ្ឌមុននឹង Execute code វាធ្វើការ Execute Statement ម្តងហើយម្តងទៀត កាលណា Condition មានតម្លៃពិត (True)។



រូបភាព 6៖ while Loop

រូបមន្ត៖

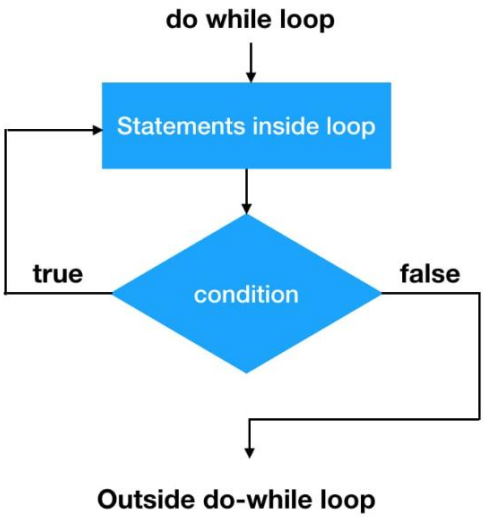
```
while (condition) {
    // code to execute as long as condition is true
}
```

ឧទាហរណ៍៖

```
let count = 0;

while (count < 5) {
    console.log(count);
    count++;
}
```

៥.២.៣ do...while Loop

do...while loop គឺស្រដៀងនឹង while loop ប៉ុន្តែជានាថាប្លុក Code នឹងប្រតិបត្តិយ៉ាងហោចណាស់ម្តងមុនពេលពិនិត្យលក្ខខណ្ឌ។



រូបភាព 7៖ do...while Loop

រូបមន្ត៖

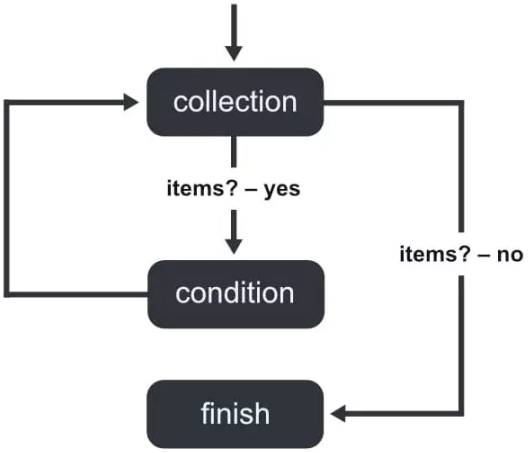
```
do {
    // code to execute
} while (condition);
```

ឧទាហរណ៍៖

```
let num = 0;
do {
    console.log(num);
    num++;
} while (num < 5);
```

៥.២.៤ for...in Loop

for...in loop ធ្វើម្តងទៀតលើ properties ដែលអាចរាប់បាននៃ object មួយ។ វាមានប្រយោជន៍សម្រាប់ការធ្វើម្តងទៀត object properties ។



រូបភាព 8៖ for...in Loop

រូបមន្ត៖

```
for (let key in object) {
    // code to execute for each key in the object
}
```

ឧទាហរណ៍៖

```
let person = { name: "John", age: 30, city: "New York" };


for (let key in person) {
    console.log(key + ": " + person[key]);
}
```

៥.២.៥ for...of Loop

for... of loop ត្រូវបានប្រើសម្រាប់ធ្វើម្តងទៀតលើ object ដែលអាចធ្វើឡើងវិញបានដូចជា arrays and strings។

វាផ្តល់នូវវិធីសាមញ្ញមួយក្នុងការធ្វើឡើងវិញតាមរយៈ values of an iterable។

រូបមន្ត៖

```
for (let value of iterable) {
    // code to execute for each value in the iterable
}
```

ឧទាហរណ៍៖

```
let numbers = [1, 2, 3, 4, 5];


for (let number of numbers) {
    console.log(number);
}
```

៥.៣ លំហាត់

1. ចូលសរសេរកម្មវិធីដែលជំរុញឱ្យអ្នកប្រើប្រាស់បញ្ចូលអាយុរបស់ពួកគេ។ ប្រើ if...else statemen ដើម្បី Print ថាតើពួកគេមានសិទ្ធិបោះឆ្នោតដែរឬទេ (អាយុចាប់ពី 18 ឆ្នាំឡើងទៅ)
2. ចូលបង្កើតកម្មវិធីដែលយកពិន្ទុមកបញ្ចូល ហើយ Print និទ្ទេសដែលត្រូវគ្នាដោយប្រើ if...else if...else statement៖
    - A for scores 90 and above
    - B for scores 80 to 89
    - C for scores 70 to 79
    - D for scores 60 to 69
    - F for scores below 60
3. ចូលសរសេរកម្មវិធីដោយប្រើ switch statement ដែលយកលេខ (1-7) ជាការបញ្ចូល និង Print ថ្ងៃដែលត្រូវគ្នានៃសប្តាហ៍ (1 សម្រាប់ថ្ងៃច័ន្ទ 2 សម្រាប់ថ្ងៃអង្គារ។ល។)។
4. ចូលសរសេរ for loop ដែល Print លេខគូទាំងអស់រវាង 1 និង 20 ។
5. ចូលបង្កើតកម្មវិធីដែលប្រើ for loop ដើម្បី Print តារាងគុណសម្រាប់លេខដែលបានបញ្ចូលដោយអ្នកប្រើប្រាស់។
6. សរសេរ while loopមួយដែលគណនាផលបូកនៃលេខពី 1 ដល់ 100 ហើយ Prints the result
7. បង្កើតហ្គេមទាយលេខ ដែលកម្មវិធីបង្កើតលេខចៃដន្យរវាង 1 និង 10។ ប្រើ do...while loop ដើម្បីអោយ អ្នកប្រើប្រាស់ឱ្យទាយលេខម្តងហើយម្តងទៀត រហូតដល់ពួកគេទទួលបានត្រឹមត្រូវ
8. បង្កើត Object តំណាងឱ្យមនុស្សដែលមាន Properties ដូចជាឈ្មោះ អាយុ និងទីក្រុង។ ប្រើ for...in loop ដើម្បី Prints property នីមួយៗ និងតម្លៃរបស់វា។

9. បង្កើតArrayនៃផ្លែឈើដែលអ្នកចូលចិត្ត ហើយប្រើ for...of loop ដើម្បី Prints ឈ្មោះផ្លែឈើនីមួយៗ
10. សរសេរកម្មវិធីដែល Prints លេខពី 1 ដល់ 100៖
    - For multiples of 3, print "Fizz" instead of the number.
    - For multiples of 5, print "Buzz".
    - For numbers that are multiples of both 3 and 5, print "FizzBuzz".
11. សរសេរកម្មវិធីដែលក្លែងធ្វើប្រព័ន្ធភ្លើងចរាចរណ៍។ ជាស់តឿនអ្នកប្រើប្រាស់សម្រាប់ពណ៌ (ក្រហម លឿង ឬបៃតង) ហើយបោះពុម្ពសកម្មភាព៖
    - "ឈប់" សម្រាប់ពណ៌ក្រហម
    - "បន្តយល្បឿន" សម្រាប់ពណ៌លឿង
    - "ទៅ" សម្រាប់ពណ៌បៃតង
    - ប្រើ if...else if...else statement សម្រាប់កិច្ចការនេះ។
12. បង្កើតកម្មវិធីដែលពិនិត្យមើលថាតើឆ្នាំដែលបានបញ្ចូលដោយអ្នកប្រើប្រាស់គឺជាឆ្នាំបង្គ្រប់។ ឆ្នាំបង្គ្រប់កើតឡើងរៀងរាល់ 4 ឆ្នាំម្តង លើកលែងតែឆ្នាំដែលបែងចែកដោយ 100 ប៉ុន្តែមិនបែងចែកដោយ 400 ។ Use an if...else statement for the logic.
13. សរសេរកម្មវិធីឱ្យអ្នកប្រើប្រាស់បញ្ចូលពាក្យសម្ងាត់ ហើយប្រើ if...else statement ដើម្បីពិនិត្យមើលថាតើពាក្យសម្ងាត់ត្រូវនឹងលក្ខខណ្ឌទាំងនេះដែរឬទេ៖
    - At least 8 characters
    - Contains a number
    - Contains an uppercase letter
    - Print "Strong password" ប្រសិនបើលក្ខខណ្ឌទាំងអស់ត្រូវបានបំពេញ; បើមិនដូច្នោះទៅprint "Weak password".

សេចក្តីសន្និដ្ឋានជំពូកទី ២៖ មូលដ្ឋានភាសា JAVASCRIPT

ជំពូកនេះគ្របដណ្តប់លើសញ្ញាណដំបូង អថេរ ប្រភេទទិន្នន័យ ប្រមាណវិធី និង Control Flow Statements។

មេរៀនទី ១៖ សញ្ញាណដំបូងរបស់ភាសា JavaScript

មេរៀននេះណែនាំអំពី JavaScript ដែលជាភាសាកម្មវិធីដែលមានលក្ខណៈ dynamic, high-level និង interpreted។ JavaScript មានតួនាទីសំខាន់ក្នុងការបង្កើតគេហទំព័រអន្តរកម្ម (interactive) និងដំណើរការនៅផ្នែក client-side (នៅក្នុង browser)។

- ប្រវត្តិ និងការវិវត្តន៍

JavaScript ត្រូវបានបង្កើតឡើងនៅឆ្នាំ 1995 ដោយលោក Brendan Eich នៅ Netscape Communications។ ភាសានេះវិវត្តន៍តាមស្តង់ដារ ECMAScript (ES) ជាផ្លូវការ។ កំណែសំខាន់ៗរួមមាន ES6 (2015) ដែលបានណែនាំ let, const, arrow functions, classes, និង modules។

- ហេតុផលដែលត្រូវរៀន

JavaScript គឺជាភាសាតែមួយគត់ដែលត្រូវបានគាំទ្រដោយ browsers ទាំងអស់ ហើយមានតម្រូវការខ្ពស់នៅក្នុងទីផ្សារការងារ (ជាពិសេស full-stack តាមរយៈ Node.js)។ វាមានសហគមន៍ដ៏ធំ និងប្រព័ន្ធអេកូឡូស៊ីនៃ libraries និង frameworks ពេញនិយម (ដូចជា React, Vue, Angular)។

- ការដំឡើង JavaScript

ដើម្បីចាប់ផ្តើម អ្នកត្រូវការ Browser ទំនើប (ដូចជា Chrome) ជាមួយនឹង Developer Tools, Text Editor (ដូចជា Visual Studio Code), Node.js ដើម្បីដំណើរការនៅខាងក្រៅ browser, និង Git/GitHub សម្រាប់ការគ្រប់គ្រងកំណែ។

មេរៀនទី ២: កាតភ្ជាប់, បង្ហាញលទ្ធផល និង Comments

មេរៀននេះផ្តោតលើរបៀបភ្ជាប់ JavaScript ទៅ HTML, ការសរសេរកូដប្រកបដោយគុណភាព និងការប្រើប្រាស់ comments។

- រចនាប័ទ្មសរសេរកូដ (Coding Style)

ការសរសេរកូដដែលងាយស្រួលក្នុងការអានគឺសារៈសំខាន់ណាស់។ គោលការណ៍សំខាន់ៗរួមមាន៖

  - ដង្កៀបអង្កាញ់ (Curly Braces): គួរប្រើ Egyptian style (ដង្កៀបបើក { នៅលើបន្ទាត់ជាមួយពាក្យគន្លឹះ)។
  - សញ្ញាក្បៀស (Semicolons): ត្រូវមាននៅចុងបញ្ចប់នៃ statement នីមួយៗដើម្បីការពារកំហុស។

- ការចូលបន្ទាត់ (Indentation): ប្រើ spaces (២ ឬ ៤) ដើម្បីរៀបចំកូដ និងបន្ទាត់ទទេដើម្បីបំបែកប្លុកឡូជីខល។
- Linters (ដូចជា ESLint): ត្រូវបានណែនាំដើម្បីជួយពិនិត្យរចនាប័ទ្មដោយស្វ័យប្រវត្តិ។

- ការភ្ជាប់ JavaScript (Attaching JavaScript)

  មានពីរវិធី៖

  - ការបង្កប់កូដក្នុង HTML: ដាក់កូដផ្ទាល់រវាងស្លាក <script> និង </script> (ពេញនិយមដាក់នៅចុង <body>)។
  - ការភ្ជាប់ពីឯកសារខាងក្រៅ: ប្រើ attribute src នៅក្នុងស្លាក <script> ដើម្បីភ្ជាប់ឯកសារ .js ដាច់ដោយឡែក។
- លទ្ធផល និងយោបល់ (Output and Comments)
  - Comments: ត្រូវបានប្រើសម្រាប់កំណត់ចំណាំ ឬការពន្យល់ (// សម្រាប់បន្ទាត់មួយ និង /* ... */ សម្រាប់ច្រើនបន្ទាត់)។
  - អនុគមន៍ alert(): ប្រើដើម្បីបង្ហាញសារជូនដំណឹងទៅអ្នកប្រើប្រាស់ (ឧទាហរណ៍: alert("Hello world!");) ប៉ុន្តែវារារាំងសកម្មភាពរបស់អ្នកប្រើប្រាស់នៅលើទំព័រ។

មេរៀនទី ៣: អថេរ និង ប្រភេទទិន្នន័យ

មេរៀននេះពន្យល់អំពី អថេរ (Variables) និង ប្រភេទទិន្នន័យ (Data Types) នៅក្នុង JavaScript ដែលជាភាសា Dynamically typed។

- ការប្រកាសអថេរ

  នៅក្នុង JavaScript ទំនើប គេប្រើ let និង const ជំនួសឱ្យ var៖
  - let (អាចប្តូរតម្លៃបាន): អនុញ្ញាតឱ្យផ្លាស់ប្តូរតម្លៃនៅពេលក្រោយ ហើយជា Block-Scoped
  - const (តម្លៃថេរ): តម្លៃមិនអាចកំណត់ឡើងវិញបានទេ ហើយត្រូវតែកំណត់តម្លៃនៅពេលប្រកាស។ វាជា Block-Scoped
- ប្រភេទទិន្នន័យបឋម (Primitive Data Types)

  មាន ៧ ប្រភេទ៖

  1. String (ខ្សែអក្សរ)
  2. Number (លេខ)
  3. Boolean (តម្លៃ true/false)
  4. Undefined (មិនបានកំណត់តម្លៃ)

5. Null (គ្មានតម្លៃ)
6. Symbol (សម្រាប់ identifier តែមួយគត់)
7. BigInt (សម្រាប់លេខធំៗ)

- ប្រភេទទិន្នន័យដែលមិនមែនជាបឋម (Non-Primitive Data Types)

ប្រភេទទាំងនេះអាចផ្ទុកតម្លៃ និងអនុគមន៍ជាច្រើន៖

- Objects: តំណាងឱ្យទិន្នន័យជា key-value pairs ({}).
- Array: ជាបញ្ជីទិន្នន័យតាមលំដាប់លំដោយ ([])។
- Functions: គឺជា objects ដែលប្រើដើម្បីអនុវត្តការងារ។

មេរៀនទី ៤: ប្រមាណវិធី (Operators)

មេរៀននេះពន្យល់អំពីប្រភេទ ប្រមាណវិធី (Operators) ផ្សេងៗគ្នាដែលប្រើក្នុង JavaScript ដើម្បីអនុវត្តប្រតិបត្តិការលើទិន្នន័យ។

ប្រភេទសំខាន់ៗនៃប្រមាណវិធី៖

1. ប្រមាណវិធីនព្វន្ធ (Arithmetic Operators): សម្រាប់គណនាគណិតវិទ្យា (+, -, , /, %, , ++, --) ។

2. ប្រមាណវិធីប្រៀបធៀប (Comparison Operators): ប្រើដើម្បីប្រៀបធៀបតម្លៃពីរ និងត្រឡប់ Boolean (==, ===, !=, !==, <, >)។ ចំណាំ === ពិនិត្យទាំងតម្លៃ និងប្រភេទទិន្នន័យ។

3. ប្រមាណវិធីតក្កវិជ្ជា (Logical Operators): ប្រើដើម្បីភ្ជាប់ ឬបញ្ច្រាសលក្ខខណ្ឌ (&& (AND), || (OR), ! (NOT))។

4. ប្រមាណវិធីផ្តល់តម្លៃ (Assignment Operators): សម្រាប់កំណត់ ឬកែប្រែតម្លៃអថេរ (=, +=, -=, etc.)

5. ប្រមាណវិធីត្រីភាគ (Ternary Operator): វិធីសង្ខេបដើម្បីសរសេរលក្ខខណ្ឌ if...else (condition ? expr1 : expr2)

- លំដាប់ និងទំនាក់ទំនង (Precedence and Associativity)
  - Precedence: កំណត់លំដាប់នៃការអនុវត្តប្រតិបត្តិការ (ឧទាហរណ៍៖ គុណ និងចែក មុនបូក និងដក)។
  - Associativity: កំណត់ទិសដៅនៃការគណនា (ឆ្វេងទៅស្តាំ ឬ ស្តាំទៅឆ្វេង) ។

មេរៀនទី ៥: Control Flow Statements

មេរៀននេះគ្របដណ្តប់លើ Control Flow Statements ដែលអនុញ្ញាតឱ្យកម្មវិធីធ្វើការសម្រេចចិត្តដោយផ្អែកលើលក្ខខណ្ឌ និងអនុវត្តកូដដដែលៗ។

- Conditional Statements (លក្ខខណ្ឌ)

  អនុញ្ញាតឱ្យប្រតិបត្តិកូដដោយផ្អែកលើលក្ខខណ្ឌជាក់លាក់៖

  1. if, else if, else: ប្រើដើម្បីពិនិត្យលក្ខខណ្ឌជាបន្តបន្ទាប់។
  2. switch: ប្រើដើម្បីជំនួស if...else if ជាច្រើន ដោយលោតទៅកាន់ case ដែលត្រូវគ្នា។

- រង្វិលជុំ (Loops)

  ត្រូវបានប្រើដើម្បីប្រតិបត្តិប្លុកកូដម្តងហើយម្តងទៀត រហូតទាល់តែលក្ខខណ្ឌឈប់ពិត៖

  - for Loop: ប្រើនៅពេលដែលដឹងចំនួននៃការធ្វើម្តងទៀត។
  - while Loop: ពិនិត្យលក្ខខណ្ឌមុននឹងប្រតិបត្តិកូដ។
  - do...while Loop: ធានាថាប្លុកកូដនឹងប្រតិបត្តិយ៉ាងហោចណាស់ម្តង មុននឹងពិនិត្យលក្ខខណ្ឌ។
  - for...in Loop: សម្រាប់ properties (keys) នៃ Objects។
  - for...of Loop: សម្រាប់ values នៃ iterable objects (ដូចជា Arrays)។

ជំពូកទី ៣៖ លក្ខណៈពិសេសនៃ ES2024

មេរៀនទី ១៖ Class Fields និង Private Methods

នៅក្នុង Modern JavaScript, Classes បានវិវឌ្ឍដើម្បីផ្តល់នូវ advanced features បន្ថែមទៀត ដែលបង្កើនវិធីដែលយើងគ្រប់គ្រង object-oriented programming (OOP)។ ជាមួយនឹងការចេញផ្សាយ ECMAScript 2024 យើងឃើញការបន្តធ្វើឱ្យប្រសើរឡើងនូវ class រូបមន្ត ជាពិសេសជាមួយនឹង class fields and private methods ។ មេរៀននេះនឹងស្វែងយល់អំពីលក្ខណៈពិសេសទាំងនេះ ដោយពន្យល់ពីរបៀបដែលពួកគេធ្វើការ និងរបៀបដែលពួកគេអាចប្រើប្រាស់យ៉ាងមានប្រសិទ្ធភាពនៅក្នុង Code របស់អ្នក។

១.១ ការប្រកាស Class Fields

Class fields គឺជាការបន្ថែមថ្មីទៅ JavaScript ដែលផ្តល់នូវវិធីសង្ខេប និងច្បាស់លាស់បន្ថែមទៀតដើម្បី define properties នៅលើ class instance ។ មុនពេលលក្ខណៈពិសេសនេះ defining properties នៅក្នុង constructor method គឺជាបទដ្ឋាន ដែលអាចនាំឱ្យ Code ដដែលៗ និងមិនអាច readable code ។ ជាមួយនឹង class fields ឥឡូវនេះយើងអាច define properties ដោយផ្ទាល់នៅក្នុង class body ។

Class fields ត្រូវបានប្រកាសដោយផ្ទាល់នៅក្នុង class body, ក្រៅវិធីសាស្ត្រណាមួយ។ ពួកវាត្រូវបានចាប់ផ្តើមទៅជា default value ឬអាចត្រូវបានទុកចោលដោយមិនបានកំណត់រហូតដល់ត្រូវបានកំណត់យ៉ាងច្បាស់លាស់។

រូបមន្ត៖

```
class MyClass {
    fieldName = initialValue;
  }
```

ឧទាហរណ៍៖

```
class Person {
    name = 'John Doe';
    age = 30;
    constructor(name, age) {
```

```
        if (name) this.name = name;
        if (age) this.age = age;
    }
    introduce() {
        console.log(`Hello, my name is ${this.name} and I am ${this.age} years old.`);
    }
}
const person1 = new Person();
person1.introduce(); // Output: Hello, my name is John Doe and I am 30 years old.
const person2 = new Person('Alice', 25);
person2.introduce(); // Output: Hello, my name is Alice and I am 25 years old.
```

ក្នុងឧទាហរណ៍ខាងលើ name and age គឺជាវាលClassដែលជា default values ។ នៅពេលដែល Person class ត្រូវបានធ្វើឱ្យសកម្ម fields ទាំងនេះត្រូវបានចាប់ផ្តើមជាមួយនឹងតម្លៃដែលផ្តល់ដោយ constructor ឬ default ទៅតម្លៃដំបូងរបស់ពួកគេ។

១.២ Private Fields and Methods

Private Fields and Methods ត្រូវបានណែនាំនៅក្នុង ECMAScript 2022 ហើយត្រូវបានកែលម្អបន្ថែមទៀតនៅក្នុង ES 2024។ ពួកវាត្រូវបានប្រើដើម្បីរៀបចំខ្ចប់ទិន្នន័យ និង behavior within a class ដោយធានាថា properties and methods មួយចំនួនមិនអាចចូលប្រើបានពីខាងក្រៅ class នោះទេ។ ត្រូវបានសម្រេចដោយប្រើ # prefix។

១.២.១ ការប្រកាស Private Fields

Private fields ត្រូវបានកំណត់ដោយ prefixing the field name ដោយសញ្ញា # ។ ពួកគេអាចចូលប្រើ និងកែប្រែបានតែនៅក្នុងថ្នាក់ដែលពួកគេត្រូវបានប្រកាស។

រូបមន្ត៖

```
class MyClass {
  #privateField = 'This is private';
  getPrivateField() {
    return this.#privateField;
  }
}
```

ឧទាហរណ៍៖

```
class BankAccount {
  #balance = 0;
  deposit(amount) {
    if (amount > 0) {
      this.#balance += amount;
    }
  }
  withdraw(amount) {
    if (amount > 0 && amount <= this.#balance) {
      this.#balance -= amount;
    }
  }
  getBalance() {
    return this.#balance;
  }
}
const account = new BankAccount();
account.deposit(100);
console.log(account.getBalance()); // Output: 100
account.withdraw(50);
```

```
console.log(account.getBalance()); // Output: 50
```

នៅក្នុង BankAccount Class, #Balance គឺជា private field ដែលមិនអាចចូលប្រើដោយផ្ទាល់ពីខាងក្រៅ class បានទេ។ The methods deposit, withdraw និង getBalance ផ្តល់នូវការចូលប្រើប្រាស់និងគ្រប់គ្រងទៅកាន់ private data នេះ។

១.២.២ ការកំណត់ Private Methods

Private methods ត្រូវបានកំណត់ស្រដៀងគ្នាទៅនឹង private fields ដោយប្រើ # prefix ។ Methods ទាំងនេះអាចចូលប្រើបានតែនៅក្នុង Class ប៉ុណ្ណោះ ហើយមិនអាចហៅពីខាងក្រៅបានទេ។

រូបមន្ត៖

```
class MyClass {
  #privateMethod() {
    console.log('This is a private method');
  }
  publicMethod() {
    this.#privateMethod();
  }
}
```

ឧទាហរណ៍៖

```
class User {
  #password;
  constructor(password) {
    this.#password = password;
  }
  #hashPassword() {
    // A simplistic example of hashing (not secure)
    return `hashed_${this.#password}`;
  }

  authenticate(inputPassword) {
```

```
        return this.#hashPassword() === `hashed_${inputPassword}`;
    }
}
const user = new User('securePassword');
console.log(user.authenticate('securePassword')); // Output: true
console.log(user.#hashPassword());    //    Error:    Private    field
'#hashPassword' must be declared in an enclosing class
```

នៅក្នុង User class, #hashPassword គឺជា private method ដែលប្រើនៅខាងក្នុងដើម្បីបំបែកលេខសម្ងាត់។ The authenticate method ប្រើ private method នេះដើម្បីផ្ទៀងផ្ទាត់ពាក្យសម្ងាត់។

១.៣ លំហាត់

1. តើ Class Fields គឺជាអ្វី? ហើយវាត្រូវបានប្រើនៅកន្លែងណា?
2. តើប្រយោជនភាពនៃការប្រើ Private Fields ក្នុង JavaScript គឺជាអ្វី?
3. តើនិយមន័យនៃ Private Fields ត្រូវបានសរសេរជាដួងដូចម្តេច? ជាមួយនឹងឧទាហរណ៍។
4. តើ Private Methods ត្រូវបានកំណត់ដោយសញ្ញាអ្វី? ហើយវាដំណើរការលើកាលវិបត្តិដូចម្តេច?
5. តើ Class Fields មានProperty ដែលផ្ទៀងផ្ទាត់ពី Constructor ជានិច្ចឬ? ដោយពន្យល់។
6. តើគន្លឹះសំខាន់ណាខ្លះដែលគួររាប់អំពីការប្រើ Private Fields ក្នុង OOP Design?
7. តើ រូបមន្ត មួយដែលប្រើ Private Methods ក្នុងការគ្រប់គ្រងសុវត្ថិភាពទិន្នន័យមានអ្វីខ្លះ? សូមផ្តល់ឧទាហរណ៍។
8. តើអ្វីទៅជាប្រៀបធៀបទៅនឹង Fields និង Methods ដែលមាន Accessibility public និង private?
9. តើមូលហេតុអ្វីបានជាវាកាន់តែប្រសើរដើម្បីប្រើ Class Fields ជំនួសការប្រើ Properties នៅក្នុង Constructor?
10. តើ Private Fields អាចឲ្យយកទៅប្រើក្រៅ Class ដោយផ្ទាល់ដៃឬទេ? ប្រសិនបើមិនបាន សូមពន្យល់។
11. បង្កើត Class Product ដែលមាន Public Fields សម្រាប់៖
    - name (ឈ្មោះផលិតផល)
    - price (តម្លៃផលិតផល)
    - បន្ថែម Constructor ដើម្បីបញ្ជូនតម្លៃដើមឲ្យ Public Fields។

12. បង្កើត Class Account ដែលមាន Private Fields៖

- #username
- #password
- បន្ថែម Methods សាធារណៈដើម្បី៖
- កំណត់តម្លៃ username និង password។
- ទាញយក username (មិនអាចទាញយក password)។

13. បង្កើត Class Calculator ដែលមាន Private Methods៖

- #add(a, b) (បូក)
- #subtract(a, b) (ដក)
- បន្ថែម Public Method calculate(operation, a, b) ដើម្បីហៅ Private Methods នៅក្នុង Class ដោយប្រើ operation (បន្ទាត់សាច់រឿង add ឬ subtract)។

មេរៀនទី ២៖ Static Properties និង Methods

នៅក្នុង JavaScript, Class អាចមាន properties and methods ដែលត្រូវបានផ្សារភ្ជាប់ជាមួយនឹង Class ខ្លួនឯងជាជាងជាមួយ instances នៃClass។ ទាំងនេះត្រូវបានគេស្គាល់ថាជា static properties and methods ។ មេរៀននេះនឹងពិភាក្សាអំពីរបៀបដែល static properties and methods ដំណើរការ និងប្រើប្រាស់របស់វា និងរបៀបប្រើប្រាស់ពួកវាប្រកបដោយប្រសិទ្ធភាពនៅក្នុង Code JavaScript របស់អ្នក។

២.១ Understanding Static Properties and Methods

Static properties and methods ត្រូវបានកំណត់ដោយប្រើពាក្យគន្លឹះ static ក្នុង Class មួយ។ មិនដូច properties and methods ដែលត្រូវបានចូលប្រើតាមរយៈ instances នៃ Class នោះទេ។ Static properties and methods ត្រូវបានចូលប្រើដោយផ្ទាល់នៅលើ class ខ្លួនវាផ្ទាល់។

២.១.១ Static Methods

Static methods ត្រូវបានហៅនៅលើ Class ជាជាងលើ instances នៃClass។ ពួកវាត្រូវបានប្រើជាញឹកញាប់សម្រាប់មុខងារ utility functions ដែលទាក់ទងនឹង Class ប៉ុន្តែមិនត្រូវការការចូលប្រើទិន្នន័យជាក់លាក់ជាក់លាក់ទេ។

ឧទាហរណ៍៖

```
class MathUtils {
  static add(x, y) {
    return x + y;
  }
}
// Accessing static method directly on the class
console.log(MathUtils.add(5, 3)); // Output: 8
```

២.១.២ Static Properties

Static properties ដំណើរការស្រដៀងនឹង static methods ប៉ុន្តែត្រូវបានប្រើដើម្បីរក្សាទុកទិន្នន័យជាជាង functionality ។ពួកវាអាចមានប្រយោជន៍សម្រាប់ការរក្សាទុកទិន្នន័យថេរ ឬការកំណត់រចនាសម្ព័ន្ធដែលពាក់ព័ន្ធនឹង Class ទាំងមូល។

ឧទាហរណ៍៖

```
class Circle {
  static PI = 3.14159;

  constructor(radius) {
    this.radius = radius;
  }
  getArea() {
    return Circle.PI * this.radius * this.radius;
  }
}
// Accessing static property directly on the class
console.log(Circle.PI); // Output: 3.14159
```

២.២ ប្រើករណី Static Properties and Methods

Static properties and methods មានប្រយោជន៍ក្នុងស្ថានភាពផ្សេងៗ៖

- Utility Functions ៖ មុខងារដែលអនុវត្តប្រតិបត្តិការមិនអាស្រ័យលើទិន្នន័យជាក់លាក់នៃ instance data ។ ឧទាហរណ៍ Class មួយដែលមាន utility methods សម្រាប់ប្រតិបត្តិការគណិតវិទ្យា ឬការបំប្លែងទិន្នន័យ។
- Constants ៖ តម្លៃដែលនៅថេរនៅទូទាំង instances of a class ទាំងអស់។ ដូចជា mathematical constants or configuration options ។
- Factory Methods ៖ Static methods អាចត្រូវបានប្រើដើម្បីបង្កើត instances of the class ក្នុងលក្ខណៈគ្រប់គ្រងដោយផ្តល់ជម្រើសជំនួសដល់ constructor ។

ឧទាហរណ៍៖

```
class Person {
  static createAdult(name) {
    return new Person(name, 18);
  }
```

```
    constructor(name, age) {
      this.name = name;
      this.age = age;
    }
}
const adultPerson = Person.createAdult('John');
console.log(adultPerson); // Output: Person { name: 'John', age: 18 }
```

២.៣ ការចូលប្រើ Static Properties and Methods

Static properties and methods ត្រូវបានចូលប្រើដោយប្រើ class name មិនមែនតាមរយៈ instances of the class នោះទេ។ នេះជានាថាពួកវាត្រូវបានប្រើសម្រាប់ប្រតិបត្តិការ និងទិន្នន័យដែលមានលក្ខណៈ global សម្រាប់ Class។

ឧទាហរណ៍៖

```
class MathUtil {
  static baseNumber = 10;
  static multiplyByBase(number) {
    return number * MathUtil.baseNumber;
  }
}
console.log(MathUtil.baseNumber); // Output: 10
console.log(MathUtil.multiplyByBase(5)); // Output: 50
```

២.៤ ភាពខុសគ្នារវាង Static and Instance Members

ការយល់ដឹងពីភាពខុសគ្នារវាង static and instance members គឺមានសារៈសំខាន់ណាស់៖

- Static Members ៖ ជាកម្មសិទ្ធិរបស់ Class ខ្លួនឯង ហើយត្រូវបានចែករំលែកនៅទូទាំងគ្រប់ករណី។ ពួកវាអាចចូលប្រើបានដោយប្រើclass name និងត្រូវបានប្រើសម្រាប់ class-level data and functionality ។

- Instance Members៖ ជាកម្មសិទ្ធិរបស់ instances បុគ្គលនៃ class ហើយត្រូវបានចូលប្រើតាមរយៈ instances ។ ពួកវាច្រើនតែដំណើរការលើទិន្នន័យជាក់លាក់ជាក់លាក់។

២.៥ លំហាត់

1. Static Properties និង Static Methods គឺជាអ្វី? ចូលសរសេរៀបរាប់ពីភាពខុសគ្នារវាង Static Properties និង Static Methods។
2. Static Methods ត្រូវបានប្រើនៅពេលណា? ចូលលើកឧទាហរណ៍ពីករណីដែលអាចប្រើ Static Methods។
3. ធ្វើដូចម្តេចដើម្បីចូលប្រើ Static Properties និង Methods? ចូលសរសេរដំណើរការនៃការចូលប្រើ Static Properties និង Methods។
4. Static Members ខុសពី Instance Members យ៉ាងដូចម្តេច? ចូលសរសេរជាប្រយោគពីសារីភាព និងការចូលប្រើ។
5. សរសេរ Code JavaScript ដើម្បីបង្កើត class ដោយមាន Properties static name = "Utility" និង Method static getName( ), ដែលបង្ហាញ name ដោយប្រើ console.log។
6. បង្កើត class ឈ្មោះ MathUtils ដែលមាន Static Method square ដើម្បីគណនាដង់ស៊ីរបស់ចំនួន។
7. បង្កើត class ឈ្មោះ Circle ដែលមាន Static Property PI = 3.14 និង Method getArea(radius) ដែលត្រឡប់តម្លៃផ្ទៃនៃរង្វង់។
8. បង្កើត class ឈ្មោះ Person ដែលមាន Static Method createAdult(name) ដើម្បីបង្កើត Object Person ដែលមានអាយុ 18 ដោយលំនាំដើម។
9. បង្កើត class ឈ្មោះ Converter ដែលមាន Static Property conversionRate = 0.85 និង Static Method toEuro(dollars) ដើម្បីបំលែងចំនួននៅក្នុងដុល្លារទៅយ៉ូរ៉ូ។
10. បង្កើត class ឈ្មោះ User ដែលមាន Static Property totalUsers ដែលចាប់ផ្តើមតម្លៃជា 0។ រាល់ពេលដែលអ្នកបង្កើត Object ថ្មីពី User, តម្លៃ totalUsers ត្រូវកើនឡើងមួយ។
11. បង្កើត class ឈ្មោះ IDGenerator ដែលមាន Static Property currentID (ចាប់ផ្តើម 0) និង Static Method generateID( ) ដែលបង្កើតលេខសម្គាល់តែមួយសម្រាប់អ្នកប្រើប្រាស់នីមួយៗ។
12. បង្កើត class ឈ្មោះ TaxCalculator ដែលមាន Static Property taxRate (ចាប់ផ្តើម 0.1) និង Static Method calculateTax(amount) ដែលគណនាពន្ធសរុប។

13. បង្កើត class ឈ្មោះ Weather ដែលមាន Static Property defaultCity = "New York" និង Static Method getForecast(city) ដែលប្រសិនបើ city មិនត្រូវបានផ្តល់វាទៅប្រើ defaultCity ជំនួស។
14. បង្កើត class ឈ្មោះ GradesManager ដែលមាន Static Method calculateAverage(grades) ដែលយកអារេយធំ (array) នៃចំនួនជាមូលដ្ឋាន និងត្រឡប់តម្លៃមធ្យម។
15. បង្កើត class ឈ្មោះ Summation ដែលមាន Static Method sum(numbers) ដែលយកអារេយធំ (array) នៃចំនួន ហើយត្រឡប់ចំនួនបូកសរុប។
16. បង្កើត class ឈ្មោះ BankAccount ដែលមាន Static Property totalBalance (ចាប់ផ្តើមតម្លៃ 0) និង Methods ដូចខាងក្រោម៖
    - Static Method deposit(amount) ដើម្បីបន្ថែមចំនួនទៅ totalBalance។
    - Static Method withdraw(amount) ដើម្បីដកចំនួនពី totalBalance។
    - Static Method getBalance( ) ដើម្បីត្រឡប់ totalBalance។

មេរៀនទី ៣៖ Optional Chaining

JavaScript ត្រូវបានគេប្រើយ៉ាងទូលំទូលាយសម្រាប់ការចាត់ចែងទិន្នន័យក្នុងទម្រង់ជា objects, arrays និងstructures ផ្សេងទៀត។ ការចូលប្រើលក្ខណសម្បត្តិរបស់ objects or arrays ដែលបង្កប់យ៉ាងជ្រៅ គឺជាកិច្ចការទូទៅមួយ ប៉ុន្តែវាអាចនាំឲ្យមានកំហុសពេល Run នៅពេលជួប null or undefined values។ Optional chaining, ជាជម្រើសដែលបានណែនាំនៅក្នុង ECMAScript 2020 (ES11) ផ្តល់នូវវិធីសង្ខេប និងសុវត្ថិភាពក្នុងការចូលប្រើ properties ទាំងនេះដោយមិនមានកំហុសឆ្គង ធ្វើឱ្យ Code កាន់តែស្អាត និងងាយស្រួលក្នុងការថែទាំ។

៣.១ តើអ្វីជា Optional Chaining?

Optional chaining ជាជម្រើសគឺជា រូបមន្ត ដែលអនុញ្ញាតឱ្យអ្នកអភិវឌ្ឍន៍ចូលប្រើ properties, methods, or array elements ដែលបានដាក់ដោយសុវត្ថិភាព ទោះបីជាតម្លៃមួយចំនួននៃតម្លៃទាំងនោះជា null or undefined ។ ជារួម វាអនុញ្ញាតឱ្យអ្នកអភិវឌ្ឍន៍ធ្វើការស្វែងរកអចលនទ្រព្យដោយមិនចាំបាច់ពិនិត្យដោយដៃថាតើគ្រប់កម្រិតនៃរចនាសម្ព័ន្ធObjectមានឬអត់។

ឧទាហរណ៍៖

```
const user = {
  profile: {
    address: {
      street: '123 Main St',
    },
  },
};
// Trying to access nested properties
const streetName = user.profile.address.street;
```

នៅក្នុង Code នេះ ប្រសិនបើ profile or address មិនត្រូវបានកំណត់។ ការចូលប្រើ street នឹងបោះកំហុស TypeError។ ជាមួយនឹងការ optional chaining នេះអាចដោះស្រាយបានកាន់តែល្អ៖

```
const streetName = user.profile?.address?.street; // undefined if profile or address is missing
```

The ?. operator អនុញ្ញាតឱ្យលេខ Code ព្យាយាមចូលប្រើ user.profile.address.street ដោយសុវត្ថិភាព។ ប្រសិនបើ property ណាមួយនៅតាម chain is undefined or null កន្សោមទាំងមូលនឹងវាយតម្លៃទៅមិនបានកំណត់ ជំនួសឱ្យការបោះចោលកំហុស។

៣.២ រូបមន្ត Optional Chaining

Optional chaining អាចត្រូវបានប្រើនៅក្នុងបរិបទផ្សេងៗ រួមទាំងការចូលប្រើប្រាស់ property ការហៅ method និងការចូលប្រើ array element ។

រូបមន្ត៖

```
object?.property;

object?.[expression];

object?.method();
```

៣.២.១ Property Access

អ្នកអាចប្រើ ?. ដើម្បីចូលប្រើ properties របស់ object ដោយសុវត្ថិភាព។

```
const streetName = user.profile?.address?.street;
```

៣.២.២ Array Access

Optional chaining ដំណើរការស្រដៀងគ្នាសម្រាប់ការចូលប្រើ array elements ។

```
const firstElement = array?.[0]; // Safe access to the first element
```

៣.២.៣ Method Calls

Optional chaining ក៏ជួយហៅ methods ដោយសុវត្ថិភាពផងដែរ។

```
const result = user.profile?.getDetails?.(); // No error if `getDetails` is undefined
```

ប្រសិនបើផ្នែកណាមួយនៃខ្សែ chain គឺ null or undefined នោះលទ្ធផលនៃ entire chain នឹង undefined ហើយ script នឹងបន្តដំណើរការដោយមិនមានកំហុស។

៣.៣ អត្ថប្រយោជន៍នៃ Optional Chaining

៣.៣.១ ធ្វើអោយប្រសើរឡើងនូវ Readability

code គឺស្អាត និងងាយស្រួលយល់ជាងបើប្រៀបធៀបទៅនឹងវិធីសាស្ត្របែបប្រពៃណី ដែលពឹងផ្អែកលើជាច្រើនដូចជា if statements or logical operators (&&) សម្រាប់ពិនិត្យមើលថាតើមាន propertyដែរឬទេ។

ក. Without optional chaining៖

```
const streetName = user && user.profile && user.profile.address ? user.profile.address.street : undefined;
```

ខ. With optional chaining៖

```
const streetName = user.profile?.address?.street;
```

៣.៣.២ ការការពារកំហុស

Optional chaining ការពារកំហុស runtime ដែលអាចកើតមានដោយសារការព្យាយាមចូលប្រើ properties នៅលើ null ឬ undefined។

៣.៣.៣ Code សន្ទូបបន្ថែមទៀត

កាត់បន្ថយ Code boilerplate ដោយលុបបំបាត់តម្រូវការសម្រាប់ការត្រួតពិនិត្យ null យ៉ាងទូលំទូលាយ ដែលបណ្តាលឱ្យមាន Code សង្ខេប និង maintainable code។

៣.៤ ការប្រើប្រាស់ជាក់ស្តែង

៣.៤.១ ការចូលប្រើប្រាស់ Nested Properties

នៅក្នុងស្ថានភាពដែលអ្នកកំពុងធ្វើការជាមួយ data structures ដែលអាចត្រូវបានកំណត់ដោយផ្នែក (ខ. fetching data from APIs) optional chaining ក្លាយជាមានប្រយោជន៍ខ្ពស់។

ឧទាហរណ៍៖

```
const data = {
  user: {
    profile: null,
  },
};
```

```
const streetName = data.user?.profile?.address?.street; // No error, returns undefined
```

៣.៤.២ Safely Calling Methods៖

Optional chaining អាចត្រូវបានប្រើដើម្បីហៅ method ដោយសុវត្ថិភាពលុះត្រាតែវាមាន។ ឧទាហរណ៍៖

```
const user = {
  profile: {
    getDetails: function() {
      return 'Details';
    },
  },
};
const details = user.profile?.getDetails?.(); // 'Details'
```

ប្រសិនបើ method មិនមានទេ, code នឹង safely return undefined ដោយមិនបង្កឱ្យមានកំហុស

៣.៤.៣ ការចូលប្រើ Dynamic Property **ជាមួយ**Array

Optional chaining មិនត្រូវបានកំណត់ចំពោះ objects ទេវាក៏ដំណើរការជាមួយ arrays ផងដែរ នៅពេលចូលប្រើធាតុ dynamically។

ឧទាហរណ៍៖

```
const data = {
  users: [
    { name: 'Alice' },
    { name: 'Bob' },
  ],
};
const firstUserName = data.users?.[0]?.name; // 'Alice'
```

```
const secondUserName = data.users?.[1]?.name; // 'Bob'
```

៣.៤.៤ ការចូលប្រើ Deeply Nested Data

ពិចារណាឧទាហរណ៍មួយដែលទិន្នន័យត្រូវបានទាញយកពី API ហើយវាលជាក់លាក់អាចមានឬប្រហែលជាមិនមាន។ Optional chaining អាចធ្វើឱ្យ Code របស់អ្នកមានភាពធន់នឹងទិន្នន័យដែលបាត់

ឧទាហរណ៍៖

```
const apiResponse = {
    data: {
      user: {
        profile: {
          address: null,
        },
      },
    },
  };
  const city = apiResponse.data?.user?.profile?.address?.city; // undefined, no error
```

៣.៤.៥ រួមបញ្ចូលគ្នាទៅ Optional Chaining និង Other Operators

Optional chaining អាចត្រូវបានរួមបញ្ចូលគ្នាជាមួយ operators ផ្សេងទៀតដូចជា nullish coalescing operator (??) ដើម្បីផ្តល់ default values នៅពេលដែល properties គឺ undefined ។

ឧទាហរណ៍៖

```
const user = {
    profile: {
      address: {
        city: 'New York',
      },
```

```
    },
  };
const city = user.profile?.address?.city ?? 'Unknown'; // 'New York'
```

ក្នុងករណីនេះ ប្រសិនបើទី city គឺ undefined តម្លៃជំនួស 'Unknown' នឹងត្រូវបានប្រើ។

៣.៤.៦ ដែនកំណត់ និងការពិចារណា

ខណៈពេល optional chaining គឺជាឧបករណ៍ដ៏មានឥទ្ធិពល មានដែនកំណត់មួយចំនួនដែលត្រូវចងចាំ៖

ក. Not for Assignment

Optional chaining មិនអាចប្រើនៅផ្នែកខាងឆ្វេងនៃកិច្ចការបានទេ។

```
user.profile?.address?.city = 'Los Angeles'; // This will throw an error
```

ខ. ការប្រើប្រាស់ហួសកំរិត

ខណៈពេលដែល optional chaining មានប្រយោជន៍សម្រាប់ការគ្រប់គ្រងទិន្នន័យមិនច្បាស់លាស់ ការប្រើប្រាស់វាច្រើនពេកអាចធ្វើឱ្យ Code របស់អ្នកពិបាកក្នុងការបំបាត់កំហុស (debug)។ ត្រូវប្រាកដថាអ្នកមិនប្រើ optional chaining ដើម្បីបិទបាំងបញ្ហាផ្សេងទៀតនៅក្នុង Code របស់អ្នក ដូចជាការបរាជ័យក្នុងការធ្វើឱ្យមានសុពលភាពត្រឹមត្រូវនៃធាតុចូល។

គ. ផលប៉ះពាល់លើការអនុវត្ត

ខណៈពេលដែលតម្លៃនៃការអនុវត្តគឺតិចតួចបំផុតនៅក្នុងករណីភាគច្រើន ការប្រើ optional chaining យ៉ាងទូលំទូលាយនៅក្នុងកម្មវិធីដែលសំខាន់ក្នុងការអនុវត្តអាចតម្រូវឱ្យមានការពិចារណាបន្ថែមទៀតដោយសារតែការត្រួតពិនិត្យបន្ថែម។

៣.៥ លំហាត់

1. អ្នកមាន object ដូចខាងក្រោម:

```
const car = {
  make: 'Toyota',
  model: 'Corolla',
  owner: {
```

```
    name: 'John Doe',
    contact: {
      email: 'john@example.com',
    },
  },
};
```

- ចូលប្រើអ៊ីមែលរបស់ម្ចាស់ដោយប្រើខ្សែ optional chaining ។
- ព្យាយាមចូលប្រើលេខទូរស័ព្ទរបស់ម្ចាស់(ដែលមិនមានវត្តមាន)ដោយប្រើ optional chaining
- Print តម្លៃទាំងពីរក្នុង Console។

2. ពិចារណា object ខាងក្រោមដែល properties មួយចំនួនអាចនឹងបាត់៖

```
const company = {
  name: 'Tech Corp',
  employees: [
    {
      name: 'Alice',
      role: 'Developer',
      details: {
        skills: ['JavaScript', 'React'],
      },
    },
    {
      name: 'Bob',
      role: 'Designer',
    },
  ],
```

```
};
```

- Print ជំនាញដំបូងរបស់ Alice ដោយប្រើ optional chaining ។
- Print ជំនាញដំបូងរបស់ Bob ដោយសុវត្ថិភាព (Bob has no skills listed)។
- Print ទីតាំងរបស់ក្រុមហ៊ុន (ដែលមិនត្រូវបានកំណត់) ។

3. អ្នកកំពុងធ្វើការជាមួយសំណុំទិន្នន័យដែលមានបញ្ហីអ្នកប្រើប្រាស់៖

```
const users = [
  {
    id: 1,
    name: 'Alice',
    posts: [
      { title: 'JavaScript Tips', likes: 15 },
      { title: 'React Guide', likes: 20 },
    ],
  },
  {
    id: 2,
    name: 'Bob',
    posts: [],
  },
];
```

- Print ចំណងជើងនៃប្រកាសដំបូងរបស់ Alice ដោយប្រើ optional chaining។
- Print ចំណងជើងនៃប្រកាសដំបូងរបស់ Bob ដោយសុវត្ថិភាព។
- Print ចំនួននៃការចូលចិត្តនៅលើប្រកាសដំបូងរបស់ Bob (should return undefined)។

4. Optional Chaining with Functions

```
const library = {
  books: [
    {
      title: 'The Great Gatsby',
      getSummary() {
        return 'A novel written by F. Scott Fitzgerald.';
      },
    },
  ],
};
```

- Call the getSummary នៃសៀវភៅទីមួយដោយប្រើ optional chaining។
- Try calling getSummary សម្រាប់សៀវភៅទីពីរដែលមិនមានប្រើ optional chaining ។
- Print the results.

មេរៀនទី ៤៖ Nullish Coalescing Operator

Nullish Coalescing Operator (??) គឺជាការបន្ថែមថ្មីចំពោះ JavaScript ដែលបានណែនាំនៅក្នុង ECMAScript 2020 (ES11)។ វាផ្តល់នូវវិធីមួយដើម្បីកំណត់តម្លៃលំនាំដើមសម្រាប់អថេរ នៅពេលដែលពួកវាជា null or undefined ដោយមិនត្រលប់មកវិញនូវតម្លៃក្លែងក្លាយដូចជា 0, NaN ឬ empty string ។ មេរៀននេះនឹងស្វែងយល់ពីគោលបំណងនៃ Nullish Coalescing Operator, រូបមន្ត និងករណីប្រើប្រាស់ជាក់ស្តែងក្នុងការអភិវឌ្ឍន៍ JavaScript ទំនើប។

៤.១ បញ្ហាជាមួយតម្លៃក្លែងក្លាយ

នៅក្នុង JavaScript គោលគំនិតនៃតម្លៃ " falsy " គឺមានសារៈសំខាន់ក្នុងការយល់ដឹងពីរបៀបដែលប្រតិបត្តិការជាក់លាក់មានឥរិយាបទ។ តម្លៃក្លែងក្លាយគឺជាតម្លៃមួយដែលវាយតម្លៃទៅមិនពិតនៅក្នុង boolean context ។ ទាំងនេះរួមមាន៖

- false
- 0
- -0
- 0n (BigInt zero)
- "" (empty string)
- null
- undefined
- NaN

មុនពេលការណែនាំនៃ Nullish Coalescing Operator, developers ជាញឹកញាប់បានប្រើ logical OR operator (||) ដើម្បីផ្តល់ default values ។ ទោះជាយ៉ាងណាក៏ដោយ វិធីសាស្ត្រនេះអាចនាំឱ្យមាន unintended behavior នៅពេលដោះស្រាយជាមួយតម្លៃដែលត្រូវបានចាត់ទុកថា falsy ប៉ុន្តែនៅតែមាន valid or meaningful ដូចជា 0, "" ឬ false ។

ឧទាហរណ៍៖

```
let userInput = 0;
let defaultValue = userInput || 10;
console.log(defaultValue); // Output: 10
```

នៅក្នុង Code នេះ userInput គឺ 0 ដែល falsy ។ ដោយសារការប្រព្រឹត្តនៃការ || operator, defaultValue ត្រូវបានកំណត់ទៅ 10 ទោះបីជា 0 អាចជាការបញ្ចូលត្រឹមត្រូវ និងមានបំណងក៏ដោយ។ នេះអាចនាំឱ្យមានកំហុស និង unintended behavior នៅក្នុងកម្មវិធីដែលភាពខុសគ្នាបែបនេះនិងមានសារៈសំខាន់។

៤.២ ប្រមាណវិធី Nullish Coalescing Operator (? ?)

Nullish Coalescing Operator (? ?) ដោះស្រាយបញ្ហាដោយគ្រាន់តែពិចារណាទិន្នន័យ null ឬ undefined។ បើមានតម្លៃដែលត្រូវជំនួសដោយ default ។ វាមិនគិតពី falsy values ផ្សេងទៀតដូចជា 0 ឬ "" ទេ

រូបមន្ត៖

```
let result = expression1 ?? expression2;
```

នៅក្នុង expression, ប្រសិនបើ expression1 មិនមែនជា null nor undefined នោះលទ្ធផលនឹងត្រូវបានកំណត់ទៅជា expression1។ ប្រសិនបើ expression1 ជា null or undefined លទ្ធផលនឹងត្រូវបានកំណត់ទៅជា expression2។

៤.២.១ ការប្រើប្រាស់មូលដ្ឋាន

ឧទាហរណ៍៖

```
let userInput = null;

let defaultValue = userInput ?? 10;

console.log(defaultValue); // Output: 10
```

ក្នុងឧទាហរណ៍នេះ userInput គឺ null ដូច្នេះ defaultValue ត្រូវបានផ្តល់តម្លៃ 10។

៤.២.២ តម្លៃមិនបានកំណត់

ឧទាហរណ៍៖

```
let userInput;

let defaultValue = userInput ?? 10;

console.log(defaultValue); // Output: 10
```

នៅទីនេះ userInput គឺ undefined ដែលបណ្តាលឱ្យ defaultValue ជា 10 ផងដែរ។

៤.២.៣ តម្លៃមិនពិត

ឧទាហរណ៍៖

```
let userInput = 0;
let defaultValue = userInput ?? 10;
console.log(defaultValue); // Output: 0
```

ក្នុងករណីនេះ userInput គឺ 0 ដែលមិនមែនជា null ឬ undefined ដូច្នេះ defaultValue គឺ 0

៤.៣ ការប្រើប្រាស់ជាក់ស្តែង

៤.៣.១ ប៉ារ៉ាម៉ែត្រអនុគមន៍

នៅពេលធ្វើការជាមួយ functions, default parameters អាចត្រូវបានកំណត់ដោយប្រើ Nullish Coalescing Operator ដើម្បីដោះស្រាយករណីដែល arguments អាចត្រូវបានកំណត់យ៉ាងច្បាស់ថាជា null or undefined ។

ឧទាហរណ៍៖

```
function greet(name) {
    let greeting = name ?? 'Guest';
    console.log(`Hello, ${greeting}!`);
}
greet();           // Output: Hello, Guest!
greet('Alice');    // Output: Hello, Alice!
greet(null);       // Output: Hello, Guest!
greet(undefined);  // Output: Hello, Guest!
```

៤.៣.២ Object Properties

នៅក្នុងស្ថានភាពដែល object properties អាចមិនត្រូវបានកំណត់នោះ Nullish Coalescing Operator ជួយផ្តល់តម្លៃ default values ។

ឧទាហរណ៍៖

```
let config = {
    timeout: 0
```

```
};
let timeout = config.timeout ?? 5000;
console.log(timeout); // Output: 0
```

នៅទីនេះ config.timeout គឺ 0 ជា falsy value ប៉ុន្តែមិនមែនជា null or undefined ទេ ដូច្នេះអស់ពេលត្រូវបានកំណត់ជា 0 មិនមែនជា default 5000 ទេ។

៤.៤ អន្តរកម្មជាមួយប្រមាណវិធីតក្កវិជ្ជា

វាជាការសំខាន់ក្នុងការកត់សម្គាល់ថាប្រមាណវិធី Nullish Coalescing មានអាទិភាពទាបជាង operators ដទៃទៀត។ នេះមានន័យថា វាគួរតែត្រូវបានប្រើជាមួយវង់ក្រចក នៅពេលរួមបញ្ចូលគ្នាជាមួយ operators ផ្សេងទៀត ដើម្បីធានាបាននូវលំដាប់នៃការវាយតម្លៃ។

ឧទាហរណ៍៖

```
let value = 0;
let result = value ?? (true || false); // true || false is true, so result is 0 ?? true
console.log(result); // Output: 0
```

ក្នុងឧទាហរណ៍ខាងលើ true || false វាយតម្លៃទៅ true ហើយបន្ទាប់មក 0 ?? លទ្ធផលពិតក្នុង 0 ។

៤.៥ លំហាត់

1. ចូលបង្កើតអថេរ userAge ហើយកំណត់តម្លៃ undefined ។ ប្រើ Nullish Coalescing Operator ដើម្បីកំណត់ default age នៃ 18 ទៅអថេរដែលមានឈ្មោះថា finalAge . Print finalAge.
2. សរសេរមុខងារដែលហៅថា greet ដែលយក parameter name ។ ប្រើ Nullish Coalescing Operator ដើម្បីផ្តល់ default value of "Guest" នៅពេលដែល name គឺ null ឬ undefined ។ហៅមុខងារជាមួយ៖
   - No argument
   - "Alice"
   - Null
   - Undefined

មេរៀនទី ៥៖ ការផ្គូផ្គង Pattern

Pattern matching គឺជាគំនិតដ៏មានអានុភាពក្នុងការសរសេរកម្មវិធី ដែលអនុញ្ញាតឱ្យអ្នកទាញយក និងរៀបចំតម្លៃដោយផ្អែកលើ structure របស់វា។ នៅក្នុង JavaScript (ES 2024) pattern matching ត្រូវបានធ្វើឱ្យប្រសើរឡើងតាមរយៈ new language features ជាពិសេសការកែលម្អរបៀបដែលយើងធ្វើការជាមួយទិន្នន័យ។ មេរៀននេះស្វែងយល់ពីការប្រើប្រាស់ការផ្គូផ្គងលំនាំជាមួយ switch statements និង destructuring ដែលជាបច្ចេកទេសសំខាន់ពីរដែលអាចជួយសម្រួល និងបញ្ជាក់ Code ។

៥.១ ការផ្គូផ្គង Pattern ជាមួយ Switch

switch statement នៅក្នុង JavaScript គឺជារចនាសម្ព័ន្ធលំហូរObjectបញ្ជាដែលអនុញ្ញាតឱ្យអ្នកប្រតិបត្តិប្លុក Code ផ្សេងៗគ្នាដោយផ្អែកលើតម្លៃនៃកន្សោមមួយ។ ជាប្រពៃណី switch statements ត្រូវបានប្រើជាមួយនឹងតម្លៃសាមញ្ញ ប៉ុន្តែពួកវាក៏អាចប្រើ pattern matching ដើម្បីដោះស្រាយ scenarios ដែលស្មុគស្មាញជាងនេះ។

ឧទាហរណ៍៖

```
function getDayName(day) {
    switch (day) {
      case 0:
        return "Sunday";
      case 1:
        return "Monday";
      case 2:
        return "Tuesday";
      case 3:
        return "Wednesday";
      case 4:
        return "Thursday";
```

```
      case 5:
        return "Friday";
      case 6:
        return "Saturday";
      default:
        return "Invalid day";
    }
  }
  console.log(getDayName(2)); // Output: Tuesday
```

ក្នុងឧទាហរណ៍នេះ switch statement វាយតម្លៃ value of day ហើយត្រូវគ្នានឹងករណីដែលបានផ្តល់។ ប្រសិនបើការផ្គូផ្គងត្រូវបានរកឃើញ ប្លុក Code ដែលត្រូវគ្នាត្រូវបានប្រតិបត្តិ។

៥.២ ការផ្គូផ្គង Pattern ជាមួយ Switch អម្រិតខ្ពស់

នៅក្នុង ES 2024 The switch statement ត្រូវបានធ្វើឱ្យប្រសើរឡើងដើម្បីគាំទ្រ advanced pattern matching បន្ថែមទៀត ដែលអនុញ្ញាតឱ្យមានលក្ខខណ្ឌស្មុគស្មាញ និង structural matching ។ អ្នកអាចប្រើលំនាំ patterns មិនត្រឹមតែ simple values ប៉ុណ្ណោះទេ ប៉ុន្តែក៏មាន objects and arrays ផងដែរ។

ឧទាហរណ៍៖

```
function getVehicleType(vehicle) {
    switch (vehicle) {
      case { type: 'car' }:
        return "Car";
      case { type: 'bike' }:
        return "Bike";
      case { type: 'truck' }:
        return "Truck";
```

```
      default:
        return "Unknown vehicle";
    }
  }
  console.log(getVehicleType({ type: 'bike' })); // Output: Bike
```

ក្នុងឧទាហរណ៍នេះ switch statement ត្រូវគ្នានឹង structure របស់ object រចៀយន្តប្រឆាំងនឹងលំនាំផ្សេងៗគ្នា។ នេះអនុញ្ញាតឱ្យមាន matching ច្បាស់លាស់ជាងមុនដោយផ្អែកលើ properties of the object ។

៥.៣ Pattern Matching with Destructuring

Destructuring គឺជាលក្ខណៈពិសេសមួយនៅក្នុង JavaScript ដែលអនុញ្ញាតឱ្យអ្នកស្រាយតម្លៃពី arrays or properties ពី objects ទៅជាអថេរផ្សេងគ្នា។ វាអាចត្រូវបានប្រើជាមួយនឹង pattern matching ដើម្បីទាញយក និងធ្វើការជាមួយនឹង data structures ដែលងាយស្រួល។

ឧទាហរណ៍៖

```
function getCoordinates(point) {
    const [x, y] = point;
    return `X: ${x}, Y: ${y}`;
  }
  console.log(getCoordinates([10, 20])); // Output: X: 10, Y: 20
```

ក្នុងឧទាហរណ៍នេះ getCoordinates function ប្រើការ array destructuring ដើម្បីទាញយកតម្លៃ x និង y ពី point array ។

៥.៤ Nested Destructuring

Destructuring ក៏អាចត្រូវបានដាក់ជា nested ដើម្បីគ្រប់គ្រង data structures ស្មុគស្មាញបន្ថែមទៀត។

ឧទាហរណ៍៖

```
function getUserInfo(user) {
    const { name, address: { city, country } } = user;
```

```
    return `Name: ${name}, City: ${city}, Country: ${country}`;
  }

  const user = {
    name: 'Alice',
    address: {
      city: 'Wonderland',
      country: 'Imaginary'
    }
  };
  console.log(getUserInfo(user)); // Output: Name: Alice, City: Wonderland, Country: Imaginary
```

ចំពោះ getUserInfo function ប្រើ nested destructuring ដើម្បីទាញយក name, city, and country ចេញពី user object ។ នេះអនុញ្ញាតឱ្យមាន Code សង្ខេប និងអាចអានបាននៅពេលធ្វើការជាមួយ data structures ដែលមានមូលដ្ឋានយ៉ាងជ្រៅ។

៥.៥ ការរួមបញ្ចូលគ្នារវាង Switch និង Destructuring

អ្នកក៏អាចបញ្ចូលគ្នានូវ switch statements ជាមួយនឹង destructuring ដើម្បីបង្កើត powerful patterns ។

ឧទាហរណ៍៖

```
function handleEvent(event) {
  switch (event.type) {
    case 'click':
      const { x, y } = event.coordinates;
      console.log(`Click at X: ${x}, Y: ${y}`);
      break;
```

```
        case 'keydown':
          const { key } = event;
          console.log(`Key pressed: ${key}`);
          break;
        default:
          console.log('Unknown event type');
      }
    }
    const clickEvent = { type: 'click', coordinates: { x: 100, y: 200 }
};
    handleEvent(clickEvent); // Output: Click at X: 100, Y: 200
```

ក្នុងឧទាហរណ៍នេះ handleEvent function ប្រើទាំង switch និង destructuring ដើម្បីដំណើរការប្រភេទផ្សេងគ្នានៃព្រឹត្តិការណ៍ប្រកបដោយប្រសិទ្ធភាព។

៥.៦ លំហាត់

1. តើ switch គឺជាអ្វី?
2. តើសមាសធាតុ Object អាចប្រើជាមួយ switch ដោយផ្ទាល់បានទេ? បញ្ជាក់ដោយផ្តល់ឧទាហរណ៍
3. សូមពិពណ៌នាអំពី Destructuring និងអត្ថប្រយោជន៍នៃការប្រើវា។
4. តើ JavaScript អាចផ្គូផ្គងអរិភាពក្នុង Arrays ដោយប្រើ Destructuring បានយ៉ាងដូចម្តេច? សូមផ្តល់ឧទាហរណ៍។
5. តើអ្នកអាចប្រើ Destructuring ដើម្បីយកតម្លៃពី Nested Object ដោយរបៀបណា? សូមផ្តល់ឧទាហរណ៍។
6. តើ Default Values អាចប្រើក្នុង Destructuring បានយ៉ាងដូចម្តេច? សូមបញ្ជាក់។
7. តើអ្នកអាចប្រើ Destructuring ក្នុង Function Parameter ដោយរបៀបណា? សូមផ្តល់ឧទាហរណ៍
8. តើអ្នកអាចប្រើ REST Operator (...rest) ក្នុង Destructuring ដើម្បីប្រមូលតម្លៃនៅសល់បានយ៉ាងដូចម្តេច?
9. តើ Optional Chaining (?.) អាចប្រើជាមួយ Switch Statement ដើម្បីជៀសវាង Errors បានដូចម្តេច?

10. សរសេរមុខងារមួយហៅថា describeShape ដែលទទួលយក shape object ដែលមាន properties { type, side }។ ប្រើ switch statement ដើម្បី return ការពិពណ៌នាដោយផ្អែកលើប្រភេទ

```
// Example Usage:

describeShape({ type: 'triangle', sides: 3 });

// Output: "A triangle has 3 sides."

describeShape({ type: 'square', sides: 4 });

// Output: "A square has 4 sides."

describeShape({ type: 'circle' });

// Output: "A circle has no sides."
```

11. បង្កើតមុខងារ getAnimalType ដែលទទួលយក animal object ហើយប្រើ switch ដើម្បីកំណត់ប្រភេទសត្វដោយផ្អែកលើប្រភេទរបស់វា៖

```
// Example Usage:

getAnimalType({ species: 'dog' });

// Output: "This is a dog."

getAnimalType({ species: 'cat' });

// Output: "This is a cat."

getAnimalType({ species: 'bird' });

// Output: "This is a bird."

getAnimalType({ species: 'fish' });

// Output: "Unknown animal."
```

12. សរសេរមុខងារ getStudentScores ដែលទទួលយកArrayនៃពិន្ទុតេស្ត ហើយផ្តល់លទ្ធផលពិន្ទុខ្ពស់បំផុត និងពិន្ទុមធ្យមដោយប្រើ Destructuring Arrays៖

```
// Example Usage:

console.log(getStudentScores([90, 80, 70, 100]));

// Output: "Highest Score: 100, Average Score: 85"
```

13. សរសេរអនុគមន៍ getFullAddress ដែលទទួលយក object អ្នកប្រើប្រាស់ និង destructures Objectអាសយដ្ឋាននៅក្នុងវា ដើម្បី return អាស័យដ្ឋានពេញលេញ៖

```
const user = {
  name: 'John Doe',
  address: {
    street: '123 Elm St',
    city: 'Metropolis',
    zip: '12345'
  }
};

// Example Usage:

console.log(getFullAddress(user));

// Output: "123 Elm St, Metropolis, 12345"
```

សេចក្តីសន្និដ្ឋានជំពូកទី ៣៖ លក្ខណៈពិសេសនៃ ES2024

មេរៀនទី ១៖ Class Fields និង Private Methods (ES2024)

មេរៀននេះពន្យល់អំពីការកែលម្អលើ Class នៅក្នុង ES2024 ដែលជួយពង្រឹងការអនុវត្ត Object-Oriented Programming (OOP)។

- ការប្រកាស Class Fields:

Class Fields អនុញ្ញាតឱ្យអ្នកកំណត់ properties នៅក្នុង class body ដោយផ្ទាល់ ដូចជា `name = 'John Doe';` ។ វិធីនេះធ្វើឱ្យកូដកាន់តែសង្ខេប និងងាយស្រួលគ្រប់គ្រងជាងការប្រើ constructor

- Private Fields និង Private Methods:

មុខងារនេះត្រូវបានណែនាំដើម្បីអនុវត្ត Encapsulation (ការរៀចខ្ចប់ទិន្នន័យ) ដោយកំណត់ properties ឬ methods ឱ្យប្រើប្រាស់បានតែនៅក្នុង Class នោះប៉ុណ្ណោះ។ វាត្រូវបានសម្គាល់ដោយប្រើសញ្ញា # នៅពីមុខឈ្មោះ (ឧទាហរណ៍៖ #balance ឬ #hashPassword()) ។ ការប្រើប្រាស់ Private Members ជួយការពារទិន្នន័យផ្ទៃក្នុងរបស់ Class ពីការចូលប្រើពីខាងក្រៅ។

មេរៀនទី ២៖ Static Properties និង Methods

មេរៀននេះរៀបរាប់អំពី Static Properties និង Static Methods ដែលជាលក្ខណៈសម្បត្តិ និងអនុគមន៍ដែលទាក់ទងនឹង Class ខ្លួនឯង មិនមែន instance (វត្ថុ) នៃ Class នោះទេ។

- ការស្វែងយល់អំពី Static Members:

Static members ត្រូវបានកំណត់ដោយប្រើពាក្យគន្លឹះ static។ ពួកវាត្រូវបានចូលប្រើដោយផ្ទាល់នៅលើឈ្មោះ Class (ឧទាហរណ៍៖ ClassName.method()) ។

- Static Methods: ត្រូវបានប្រើជាញឹកញាប់សម្រាប់ Utility Functions (មុខងារជំនួយ) ដែលមិនត្រូវការទិន្នន័យជាក់លាក់នៃ instance ណាមួយ។
- Static Properties: ត្រូវបានប្រើដើម្បីរក្សាទុក Constants (តម្លៃថេរ) ឬទិន្នន័យដែលចែករំលែកដោយ Class ទាំងមូល។

Static members ជាកម្មសិទ្ធិរបស់ Class ខ្លួនឯង ហើយត្រូវបានចែករំលែកនៅទូទាំងគ្រប់ instances ទាំងអស់ ខុសពី Instance Members ដែលជាកម្មសិទ្ធិរបស់ instances បុគ្គលនីមួយៗ។

មេរៀនទី ៣៖ Optional Chaining (?.)

មេរៀននេះណែនាំអំពី Optional Chaining (?.) ដែលជាមុខងារ សម្រាប់ចូលប្រើ properties ឬ methods ដែលបង្កប់យ៉ាងជ្រៅនៅក្នុង object ឬ array ប្រកបដោយសុវត្ថិភាព។

- តើ Optional Chaining គឺជាអ្វី?

Optional Chaining អនុញ្ញាតឱ្យអ្នកចូលប្រើទិន្នន័យដោយមិនបោះចោលកំហុស (TypeError) ប្រសិនបើ properties ណាមួយនៅក្នុងខ្សែ chain គឺ null ឬ undefined។ ប្រសិនបើខ្សែ chain បាត់តម្លៃ វានឹងត្រឡប់មកវិញ undefined វិញ។

- អត្ថប្រយោជន៍:
  - ការការពារកំហុស: ការពារ runtime errors ពេលចូលប្រើទិន្នន័យដែលបាត់។
  - កូដស្រស់ស្អាត និងសង្ខេប: កាត់បន្ថយតម្រូវការសម្រាប់ការត្រួតពិនិត្យ null ច្រើនដោយដៃ ធ្វើឱ្យកូដកាន់តែងាយស្រួលអាន។

មេរៀនទី ៤៖ Nullish Coalescing Operator (??)

មេរៀននេះគ្របដណ្តប់លើ Nullish Coalescing Operator (??) ដែលផ្តល់នូវវិធីសាស្ត្រសុវត្ថិភាពសម្រាប់ការកំណត់តម្លៃលំនាំដើម ដោយដោះស្រាយបញ្ហាពីការប្រើប្រាស់ Logical OR (||)។

- បញ្ហាជាមួយ Logical OR (||):

ប្រតិបត្តិករ || នឹងផ្តល់តម្លៃលំនាំដើម ប្រសិនបើតម្លៃខាងឆ្វេងជា falsy (ដូចជា 0, , false, null, ឬ undefined)។ នេះអាចនាំឱ្យមានឥរិយាបថដែលមិនចង់បាន ប្រសិនបើ 0 ឬ គឺជាការបញ្ចូលដែលត្រឹមត្រូវ។

- ប្រមាណវិធី Nullish Coalescing Operator (??):

ប្រតិបត្តិករ ?? ពិនិត្យតែតម្លៃ null និង undefined ប៉ុណ្ណោះ។ ប្រសិនបើតម្លៃខាងឆ្វេងមិនមែនជា null ឬ undefined ទេ វានឹងត្រឡប់តម្លៃនោះ។ ប្រសិនបើវាជា null ឬ undefined វានឹងត្រឡប់តម្លៃខាងស្តាំ (តម្លៃលំនាំដើម)។

មេរៀនទី ៥៖ ការផ្គូផ្គង Pattern (Pattern Matching)

មេរៀននេះពន្យល់អំពី Pattern Matching ដែលអនុញ្ញាតឱ្យអ្នកទាញយក និងរៀបចំតម្លៃដោយផ្អែកលើរចនាសម្ព័ន្ធទិន្នន័យរបស់វា។

- ការផ្គូផ្គង Pattern ជាមួយ switch:

នៅក្នុង ES2024, switch ត្រូវបានធ្វើឱ្យប្រសើរឡើងដើម្បីគាំទ្រ advanced pattern matching ដែលអនុញ្ញាតឱ្យផ្គូផ្គងរចនាសម្ព័ន្ធ objects និង arrays នៅក្នុង case ដើម្បីដោះស្រាយ scenarios ស្មុគស្មាញ។

- Pattern Matching ជាមួយ Destructuring:

Destructuring គឺជាទម្រង់សំខាន់មួយនៃការផ្គូផ្គង pattern ដែលអនុញ្ញាតឱ្យអ្នក ស្រាយ (unpack) តម្លៃពី arrays ឬ properties ពី objects ដោយផ្ទាល់ទៅក្នុងអថេរផ្សេងគ្នា។ នេះធ្វើឱ្យការចូលប្រើទិន្នន័យកាន់តែងាយស្រួល៖

- Object Destructuring: ប្រើ {} សម្រាប់ objects។
- Array Destructuring: ប្រើ [] សម្រាប់ arrays តាមលំដាប់លំដោយ។

ការបញ្ចូលគ្នារវាង switch និង Destructuring ផ្តល់នូវវិធីសាស្ត្រដ៏មានអានុភាពសម្រាប់ការគ្រប់គ្រងទិន្នន័យស្មុគស្មាញ។

ជំពូកទី ៤៖ ការអនុវត្តន៍ JavaScript

មេរៀនទី ១៖ អនុគមន៍ (Functions)

Functions គឺជាសមាសធាតុសំខាន់នៅក្នុងកម្មវិធី JavaScript ដែលអនុញ្ញាតឱ្យអ្នករៀបចំ និងគ្រប់គ្រង Code ប្រកបដោយប្រសិទ្ធភាព។ នៅក្នុងមេរៀននេះ យើងនឹងស្វែងយល់អំពីប្រភេទមុខងារផ្សេងៗដែលមាននៅក្នុង JavaScript រួមទាំងការ ការប្រកាសអនុគមន៍, កន្សោមអនុគមន៍, arrow functions, generator functions, and asynchronous functions យើងក៏នឹងពិភាក្សាអំពី function scope, closures និងរបៀបដោះស្រាយ arguments និង return values ។

១.១ ការប្រកាសអនុគមន៍

ការប្រកាសអនុគមន៍ដែលត្រូវបានគេស្គាល់ផងដែរថាជា function statements គឺជាវិធីសាមញ្ញបំផុតមួយក្នុងការកំណត់មុខងារមួយ។ ពួកគេត្រូវបានលើកទៅកំពូលនៃវិសាលភាពរបស់ពួកគេ ដែលមានន័យថាពួកគេអាចត្រូវបានគេហៅ មុនពេលដែលពួកគេត្រូវបានកំណត់នៅក្នុង Code ។

រូបមន្ត៖

```
function functionName(parameters) {
    // Code to execute
}
```

ឧទាហរណ៍៖

```
function greet(name) {
    return `Hello, ${name}!`;
}
console.log(greet('Alice')); // Output: Hello, Alice!
```

១.២ កន្សោមអនុគមន៍

កន្សោមអនុគមន៍បានកំណត់ អនុគមន៍ជាផ្នែកនៃកន្សោម ហើយអាចដាក់ឈ្មោះ ឬអនាមិក។ មិនដូច function declarations ទេ function expressions មិនត្រូវបានលើកទេ។ ពួកវាត្រូវបានកំណត់ទៅអថេរ និងអាចត្រូវបានប្រើតែបន្ទាប់ពីការកំណត់របស់ពួកគេ។

រូបមន្ត៖

```
const functionName = function(parameters) {
    // Code to execute
};
```

ឧទាហរណ៍៖

```
const add = function(a, b) {
    return a + b;
};
console.log(add(5, 3)); // Output: 8
```

Named Function Expression៖

```
const multiply = function multiply(a, b) {
    return a * b;
};
console.log(multiply(4, 5)); // Output: 20
```

១.៣ អនុគមន៍ Arrow

អនុគមន៍ Arrow ដែលណែនាំនៅក្នុង ECMAScript 6 (ES6) ផ្តល់នូវរូបមន្តសង្ខេបសម្រាប់មុខងារសរសេរ។ ពួកគេមិនមានបរិបទនេះផ្ទាល់របស់ពួកគេទេ ដែលមានប្រយោជន៍ក្នុងសេណារីយ៉ូជាក់លាក់ ជាពិសេសនៅពេលដោះស្រាយជាមួយនឹងការ callbacks និង methods in classes ។

រូបមន្ត៖

```
const functionName = (parameters) => {
    // Code to execute
};
```

ឧទាហរណ៍៖

```
const add = (a, b) => a + b;
console.log(add(5, 3)); // Output: 8
```

១.៤ អនុគមន៍ Generator

Generator functions ដែលណែនាំនៅក្នុង ES6 អនុញ្ញាតឱ្យអ្នកកំណត់ក្បួនដោះស្រាយដដែលៗដោយប្រើ function* រូបមន្ត and yield expressions ។ ពួកវាត្រឡប់Objectដែលប្រើឡើងវិញដែលអាចត្រូវបានប្រើដើម្បីឆ្លងកាត់តាម series of values ។

រូបមន្ត៖

```
function* generatorFunction() {
    yield value1;
    yield value2;
}
```

ឧទាហរណ៍៖

```
function* countUpTo(max) {
    let count = 1;
    while (count <= max) {
        yield count;
        count++;
    }
}
const counter = countUpTo(3);
console.log(counter.next().value); // Output: 1
console.log(counter.next().value); // Output: 2
console.log(counter.next().value); // Output: 3
console.log(counter.next().value); // Output: undefined
```

១.៥ អនុគមន៍ Asynchronous

Asynchronous functions ដែលកំណត់ដោយប្រើពាក្យគន្លឹះ async អនុញ្ញាតឱ្យអ្នកសរសេរ Code ដែលដំណើរការ asynchronous operations កាន់តែងាយស្រួល។ ពួកគេ return ចំពោះតម្លៃដែល returned by the function ។

រូបមន្ត៖

```
async function functionName(parameters) {
    // Code to execute
}
```

ឧទាហរណ៍៖

```
async function fetchData() {
    let response = await fetch('https://api.example.com/data');
    let data = await response.json();
    return data;
}
fetchData().then(data => console.log(data));
```

១.៦ អនុគមន៍ Scope and Closures

១.៦.១ Scope

Scope សំដៅលើលទ្ធភាពប្រើប្រាស់នៃ variables and functions នៅក្នុងផ្នែកផ្សេងៗនៃ Code របស់អ្នក។ មុខងារ JavaScript បង្កើត scope ផ្ទាល់ខ្លួនរបស់ពួកគេ មានន័យថាអថេរដែលបានកំណត់នៅខាងក្នុង function គឺមិនអាចចូលប្រើបាននៅខាងក្រៅវាទេ។

ឧទាហរណ៍៖

```
function outerFunction() {
    let outerVariable = 'I am outside!';

    function innerFunction() {
        console.log(outerVariable);
    }
```

```
    innerFunction(); // Output: I am outside!
}
outerFunction();
console.log(outerVariable); // Error: outerVariable is not defined
```

១.៦.២ Closures

Closures គឺជាលក្ខណៈពិសេសមួយដែល function ខាងក្នុងរក្សាការចូលប្រើអថេរពី function ខាងក្រៅរបស់វា សូម្បីតែបន្ទាប់ពី function ខាងក្រៅបានបញ្ចប់ការប្រតិបត្តិក៏ដោយ។ Closures គឺមានប្រយោជន៍សម្រាប់ការបញ្ចូលទិន្នន័យ និងបង្កើត private variables ។

ឧទាហរណ៍៖

```
function createCounter() {
    let count = 0;
    return function() {
        count++;
        return count;
    };
}
const counter = createCounter();
console.log(counter()); // Output: 1
console.log(counter()); // Output: 2
```

១.៧ ការត្រឡប់ត្រឡប Arguments and Return Values

Functions អាចដោះស្រាយ arguments ច្រើននិង return values តាមវិធីផ្សេងៗ។

១.៧.១ Default Parameters

Default Parameters ផ្តល់ default values សម្រាប់ function parameters ប្រសិនបើគ្មានការឆ្លងកាត់។

ឧទាហរណ៍៖

```
function greet(name = 'Guest') {
    return `Hello, ${name}!`;
}
console.log(greet()); // Output: Hello, Guest!
```

១.៧.២ Rest Parameters

Rest Parameters តំណាងឱ្យចំនួន arguments មិនកំណត់ជា array ។ ឧទាហរណ៍៖

```
function sum(...numbers) {
    return numbers.reduce((total, num) => total + num, 0);
}
console.log(sum(1, 2, 3, 4)); // Output: 10
```

១.៧.៣ Returning Multiple Values

Functions អាច return objects or arrays ដើម្បីបញ្ជូលតម្លៃច្រើន។ ឧទាហរណ៍៖

```
function getUserInfo() {
    return {
        name: 'Alice',
        age: 30
    };
}
const user = getUserInfo();
console.log(user.name); // Output: Alice
console.log(user.age); // Output: 30
```

១.៨ លំហាត់

1. តើមុខងារ (Function) គឺជាអ្វី?
2. តើ Function Declaration មានរបៀបដូចម្តេចក្នុង JavaScript? ចូលលើកឧទាហរណ៍។
3. Arrow Function មានភាពខុសពី Function Normal យ៉ាងដូចម្តេច? ប្រៀបធៀបការប្រើប្រាស់ function និង => សម្រាប់អនុគមន៍។
4. តើហេតុអ្វីបានជា JavaScript អនុញ្ញាតឱ្យប្រើ Rest Parameters? តើមានអត្ថប្រយោជន៍អ្វីខ្លះក្នុងការប្រើ Rest Parameters?
5. Function Expression ត្រូវបានប្រើនៅពេលណា និងមានភាពខុសពី Function Declaration ដូចម្តេច?
6. តើ this នៅក្នុង arrow functions មានអ្វីខុសពី Normal function? ប្រៀបធៀបការប្រើប្រាស់ this នៅក្នុង arrow function និង Normal function ?
7. តើ arguments គឺជាអ្វី និងមានភាពខុសគ្នាដូចម្តេចរវាង arrow function និង Normal function?
8. សូមបង្ហាញអំពីមុខងារ arguments និងការប្រើប្រាស់វា។
9. តើមុខងារ async និង await គឺជាអ្វី និងតើវាអាចជួយដល់ការបញ្ជារង្វៀងអនុគមន៍ Asynchronous ដូចម្តេច?
10. ពិភាក្សាអំពីការប្រើ async និង await ដើម្បីរៀបចំការចូលប្រើ API ឬបញ្ចូលទិន្នន័យនៅក្នុងមុខងារ។
11. តើហេតុអ្វីបានជា JavaScript យើងគួរតែប្រើ function ប្រសិនបើមុខងារ ត្រូវបានប្រើលើកទាំងអស់ក្នុងការបំពេញការងារ?
12. តើអ្នកអាចតំណាងនូវ Function ជា Object ក្នុង JavaScript ដូចម្តេច? ចូលបំលែង function ទៅជា object និងប្រើវា។
13. ចូលសរសេរមុខងារដែលអាចទទួលបានលេខពី 2 ទៅ 10 ហើយបង្ហាញកំហុស (Error) ប្រសិនបើលេខមិនគ្រប់គ្រាន់។ ប្រើ Function Declaration និងប្រើការប្រកាសប្រាក់កាសនៃលេខដែលអ្នកផ្តល់។
14. ចូលសរសេរមុខងារ arrow function ដែលសម្រាប់គណនាអាយុសរុបរបស់បុគ្គលពីការបង្ហាញពីឆ្នាំខុសគ្នារវាងឆ្នាំបច្ចុប្បន្ន និងឆ្នាំកំណើត។ ដោយប្រើ ES6 Arrow Functions និង Default Parameter។

15. ចូលបង្កើតមុខងារ Generator ដែលបង្ហាញលេខពី 1 ដល់ 5។ ប្រើ yield ដើម្បីបញ្ជូនតម្លៃទាំងនេះមួយៗ។
16. ចូលសរសេរមុខងារ sum ដែលអាចគណនាតម្លៃសរុបពីបញ្ជីលេខមួយ ដែលអ្នកប្រើប្រាស់បានផ្តល់ដោយ Rest Parameter។ ប្រើ reduce ក្នុងការគណនាសរុប។
17. ចូលបង្កើតមុខងារ createPerson ដែលទទួលឈ្មោះ និងអាយុជាប៉ារ៉ាម៉ែត្រ និងបញ្ជូនវា។ មុខងារនេះត្រូវតែត្រឡប់មកវិញជា Object ដែលមាន Property name និង age ។
18. បង្កើតមុខងារ arrow function ដែលគណនាអាយុបានមកពីកំណត់លេខរបស់ឆ្នាំកំណើត និងឆ្នាំបច្ចុប្បន្ន ដោយប្រើ Default Parameters និង Rest Parameters ក្នុងការកំណត់ចំនួនពីរប៉ារ៉ាម៉ែត្រ។ ប្រើ Math ដើម្បីគណនាអាយុ។
19. បង្កើតមុខងារ Generator ដែលបង្ហាញអក្សរទាំងអស់ពី String មួយដែលផ្តល់ជារបស់អ្នកក្នុងលំដាប់តំណាងជាមួយ yield។ ប្រើ yield ដើម្បីប្រើប្រាស់ Character តាមលំដាប់។
20. បង្កើតមុខងារ addNumbers ដែលអាចដាក់លេខជាច្រើនទៀត និងរកប្រាក់កាសនៃការបូកទាំងអស់នេះដោយប្រើ Rest Parameter និង reduce។ ប្រើ reduce នៅក្នុង arrow function ដើម្បីគណនាលទ្ធផលនៃការបូកលេខទាំងអស់។
21. បង្កើតមុខងារ counter ដោយប្រើ Closures ដែលរាប់ចាប់ពី 0 និងបន្ថែមតម្លៃដោយបញ្ជូនប៉ារ៉ាម៉ែត្រ increment មួយ។ ប្រើ return ដើម្បីត្រឡប់លទ្ធផលនៃការពង្រីកបរិមាណ។
22. ប្រើ async និង await ដើម្បីសរសេរមុខងារ fetchData ដែលធ្វើការទាញយកព័ត៌មានពី URL មួយ ហើយបង្ហាញលទ្ធផល។ ប្រើ async function និង await នៅក្នុង function ដើម្បីទទួលការឆ្លើយតបពី API។

មេរៀនទី ២៖ Objects

នៅក្នុង JavaScript objects គឺជាប្រភេទ fundamental data មួយដែលត្រូវបានប្រើដើម្បីតំណាង និងគ្រប់គ្រងការប្រមូលទិន្នន័យ។ មេរៀននេះនឹងគ្របដណ្តប់លើចំណុចសំខាន់ៗនៃការ creating, accessing, modifying, and iterating លើ objects ដោយប្រើស្តង់ដារ ECMAScript 2024 (ES2024)។

២.១ ការបង្កើត Objects

Objects នៅក្នុង JavaScript គឺជាបណ្តុំនៃគូតម្លៃ key-value ដែលកូនសោនីមួយៗគឺជា string (or a Symbol) ហើយតម្លៃនីមួយៗអាចជាប្រភេទទិន្នន័យ JavaScript ត្រឹមត្រូវ។ មានវិធីជាច្រើនដើម្បីបង្កើតObject៖

២.១.១ Object Literal រូបមន្ត

វិធីសាស្ត្រសាមញ្ញបំផុតក្នុងការបង្កើត object មួយគឺការប្រើ object literal ។ ឧទាហរណ៍៖

```
const person = {
    name: 'Alice',
    age: 30,
    greet: function() {
        console.log('Hello, ' + this.name);
    }
};
```

ក្នុងឧទាហរណ៍នេះ person ជា object ដែលមានលក្ខណៈបីយ៉ាងគឺ name, age, and greet ។ greet property គឺជាវិធីសាស្ត្រដែលបោះពុម្ពការស្វាគមន៍ទៅកាន់ console ។

២.១.២ អនុគមន៍ Constructor

វិធីមួយទៀតដើម្បីបង្កើត objects គឺដោយប្រើ constructor function ។ ឧទាហរណ៍៖

```
function Person(name, age) {
    this.name = name;
    this.age = age;
    this.greet = function() {
```

```
        console.log('Hello, ' + this.name);
    };
}
const person1 = new Person('Bob', 25);
```

Person function ដើរតួជាប្លង់មេសម្រាប់បង្កើត objects ។ ដោយប្រើ new keyword, new Person object ត្រូវបានបង្កើតដោយ name and age properties របស់វា។

២.១.៣ Object.create( )

វិធីសាស្ត្រ Object.create( ) បង្កើត object ថ្មីជាមួយនឹង prototype object និងproperties ឧទាហរណ៍៖

```
const proto = {
    greet: function() {
        console.log('Hello, ' + this.name);
    }
};
const person2 = Object.create(proto);
person2.name = 'Charlie';
```

នៅទីនេះ person2 ទទួលមរតកពី proto ដែលមានន័យថាវាមានសិទ្ធិចូលប្រើ greet method ដែលបានកំណត់នៅលើ proto ។

២.២ ការចូលប្រើ Object Properties

Properties របស់ object អាចចូលប្រើបានដោយប្រើសញ្ញាចុច. ឬសញ្ញាតង្កៀប [] ។

២.២.១ សញ្ញាចុច.

នេះគឺជាវិធីសាមញ្ញបំផុត និងត្រង់បំផុតក្នុងការចូលប្រើ properties ។

ឧទាហរណ៍៖

```
console.log(person.name); // Output: Alice
```

២.២.២ សញ្ញាតង្កៀប[]

នេះអនុញ្ញាតឱ្យមានការចូលប្រើប្រាស់ dynamic កាន់តែច្រើន ជាពិសេសមានប្រយោជន៍នៅពេលដែល property names ត្រូវបានរក្សាទុកក្នុងអថេរ ឬមិនមែនជាការកំណត់អត្តសញ្ញាណត្រឹមត្រូវ។

ឧទាហរណ៍៖

```
console.log(person['age']); // Output: 30

const propertyName = 'name';

console.log(person[propertyName]); // Output: Alice
```

២.៣ ការកែប្រែ Object Properties

ដើម្បីកែប្រែតម្លៃនៃ existing property អ្នកគ្រាន់តែកំណត់វាឡើងវិញ។

២.៣.១ ប្រើសញ្ញាចុច.៖

ឧទាហរណ៍៖

```
person.age = 31;
```

២.៣.១ ប្រើសញ្ញាតង្កៀប[]៖

ឧទាហរណ៍៖

```
person['name'] = 'Alicia';
```

អ្នកក៏អាចបន្ថែម properties ថ្មីតាមរបៀបដូចគ្នា។

២.៤ Iterating Over Objects

JavaScript ផ្តល់នូវវិធីសាស្ត្រជាច្រើនដើម្បីធ្វើការត្រួតពិនិត្យ properties របស់ object ។

២.៤.១ for...in Loop

This loop ធ្វើម្តងទៀតលើ properties ដែលអាចរាប់បានទាំងអស់នៃ object ដែលត្រូវបានចុចដោយ strings.

ឧទាហរណ៍៖

```
for (let key in person) {
    if (person.hasOwnProperty(key)) {
        console.log(key + ': ' + person[key]);
```

}}

The hasOwnProperty method ត្រូវបានប្រើដើម្បីធានាថាមានតែលក្ខណៈសម្បត្តិផ្ទាល់ខ្លួនរបស់ object ប៉ុណ្ណោះ (មិនត្រូវបានទទួលមរតក) ត្រូវបានធ្វើឡើងវិញ។

២.៤.២ Object.keys( )

វិធីសាស្ត្រនេះreturns an array នៃproperty namesដែលអាចរាប់បានផ្ទាល់ខ្លួនរបស់ object ។ ឧទាហរណ៍៖

```
Object.keys(person).forEach(key => {
    console.log(key + ': ' + person[key]);
});
```

២.៤.៣ Object.values( )៖

វិធីសាស្ត្រនេះ returns an array នៃ property values ដែលអាចរាប់បានផ្ទាល់ខ្លួនរបស់ object ឧទាហរណ៍៖

```
Object.values(person).forEach(value => {
    console.log(value);
});
```

២.៤.៤ Object.entries( )

វិធីសាស្ត្រនេះ returns an array នៃ property ដែលអាចរាប់បានផ្ទាល់ខ្លួនរបស់ object [key, value] pairs។

ឧទាហរណ៍៖

```
Object.entries(person).forEach(([key, value]) => {
    console.log(key + ': ' + value);
});
```

២.៥ លំហាត់

1. តើ object ក្នុង JavaScript ជាអ្វី? អាចបញ្ជាក់អត្ថប្រយោជន៍របស់វា?
2. តើមានវិធីបង្កើត object អ្វីខ្លះក្នុង JavaScript?
3. ចូរពន្យល់ពីភាពខុសគ្នារវាង dot notation (.) និង bracket notation ([]) ក្នុងការចូលប្រើ property របស់ object។
4. តើវិធីសាស្ត្រដែលអាចប្រើសម្រាប់ធ្វើ iteration លើ object មានអ្វីខ្លះ?
5. តើ Object.create( ) ខុសពី object literal ({}) យ៉ាងដូចម្តេច?
6. តើ Object.keys( ), Object.values( ), និង Object.entries( ) មានតួនាទីអ្វី?
7. ចូរសរសេរ function ដែលចូលប្រើ និងផ្លាស់ប្តូរទិន្នន័យក្នុង object ដោយប្រើ bracket notation។
8. តើ for...in loop និង Object.keys( ) ខុសគ្នាយ៉ាងដូចម្តេច?
9. តើ hasOwnProperty( ) មានតួនាទីអ្វី?
10. តើ Object.assign( ) និង spread operator (...) ខុសគ្នាយ៉ាងដូចម្តេចនៅក្នុងការចម្លង object?
11. បង្កើត Object និងចូលប្រើ Property
    - បង្កើត object car ដែលមាន property brand, model, និង year។
    - ចូលប្រើ និងបង្ហាញ brand និង year ដោយប្រើ dot notation និង bracket notation
12. បន្ថែម និងផ្លាស់ប្តូរទិន្នន័យក្នុង Object
    - បន្ថែម property color ទៅក្នុង object car។
    - ផ្លាស់ប្តូរ year ទៅជា 2024។
    - លុប property model ចេញពី object។
13. Iteration លើ Object
    - ប្រើ for...in loop ដើម្បី iterate លើ property របស់ object car។
    - ប្រើ Object.keys( ) ដើម្បីទាញយក key ទាំងអស់។
    - ប្រើ Object.values( ) ដើម្បីទាញយក value ទាំងអស់។
14. សរសេរ function updateProperty(obj, key, value) ដែលអាចផ្លាស់ប្តូរ property មួយក្នុង object obj។

មេរៀនទី ៣៖ អារេ (Arrays)

៣.១ អារេ (Arrays)

អារេ គឺជា Objects ដែរអនុញ្ញាតឱ្យអ្នករក្សាទុកបណ្តុំនៃតម្លៃសំខាន់ៗ។ ប៉ុន្តែជាញឹកញាប់យើងឃើញថាយើងត្រូវការ ការប្រមូលតាមលំដាប់ ដែលយើងមានធាតុទី 1 ទី 2 ទី 3 ជាដើម។ ឧទាហរណ៍ យើងត្រូវការវាដើម្បីទុកបញ្ជីរបស់អ្វីមួយ៖ អ្នកប្រើ ទំនិញ ធាតុ HTML ជាដើម។

វាមិនងាយស្រួលប្រើObjectនៅទីនេះទេ ព្រោះវាមិនមានវិធីសាស្ត្រដើម្បីគ្រប់គ្រងលំដាប់នៃធាតុ។ យើងមិនអាចបញ្ចូលProperty ថ្មី "between" Property ដែលមានស្រាប់បានទេ។ Objectsគឺគ្រាន់តែមិនមានន័យសម្រាប់ការប្រើប្រាស់បែបនេះ។ មានរចនាសម្ព័ន្ធទិន្នន័យពិសេសមួយដែលមានឈ្មោះថា Arrayដើម្បីរក្សាទុកបណ្តុំដែលបានបញ្ជាទិញ។

៣.១.១ ការប្រកាស Arrays

មាន រូបមន្ត ពីរសម្រាប់បង្កើតArrayទទេ៖

```
let arr = new Array();
let arr = [];
```

ស្ទើរតែគ្រប់ពេលវេលា រូបមន្តទីពីរត្រូវបានប្រើ។ យើងអាចផ្តត់ផ្តង់ធាតុដំបូងនៅក្នុងតង្កៀប៖

```
let fruits = ["Apple", "Orange", "Plum"];
```

ធាតុArrayត្រូវបានដាក់លេខដោយចាប់ផ្តើមពីសូន្យ។ យើងអាចទទួលបានធាតុមួយដោយលេខរបស់វាក្នុងតង្កៀបការ៉ែ៖

```
let fruits = ["Apple", "Orange", "Plum"];

alert( fruits[0] ); // Apple
alert( fruits[1] ); // Orange
alert( fruits[2] ); // Plum
```

យើងអាចជំនួសធាតុមួយ៖

```
fruits[2] = 'Pear'; // now ["Apple", "Orange", "Pear"]
```

...ឬបន្ថែមថ្មីមួយទៅArray៖

```
fruits[3] = 'Lemon'; // now ["Apple", "Orange", "Pear", "Lemon"]
```

ចំនួនសរុបនៃធាតុនៅក្នុងArrayគឺរបស់វា length៖

```
let fruits = ["Apple", "Orange", "Plum"];

alert( fruits.length ); // 3
```

យើងក៏អាចប្រើ alertដើម្បីបង្ហាញArrayទាំងមូលផងដែរ។

```
let fruits = ["Apple", "Orange", "Plum"];

alert( fruits ); // Apple,Orange,Plum
```

Arrayអាចផ្ទុកធាតុនៃប្រភេទណាមួយ។

ឧទាហរណ៍៖

```
// mix of values
let arr = [ 'Apple', { name: 'John' }, true, function() {
alert('hello'); } ];

// get the object at index 1 and then show its name
alert( arr[1].name ); // John

// get the function at index 3 and run it
arr[3](); // hello
```

- សញ្ញាក្បៀសតាមក្រោយ

Array ដូចជា Object មួយ អាចបញ្ចប់ដោយសញ្ញាក្បៀស៖

```
let fruits = [
  "Apple",
  "Orange",
  "Plum",
];
```

រចនាប័ទ្ម "សញ្ញាក្បៀសនៅខាងក្រោយ" ធ្វើឱ្យវាកាន់តែងាយស្រួលក្នុងការបញ្ចូល/យកធាតុចេញ ពីព្រោះបន្ទាត់ទាំងអស់ប្រែជាដូចគ្នា។

៣.១.២ ការទាញយកធាតុចុងក្រោយដោយប្រើ "at"

ឧបមាថាយើងចង់បានធាតុចុងក្រោយនៃArray។ ភាសាសរសេរកម្មវិធីមួយចំនួនអនុញ្ញាតឱ្យប្រើលិបិក្រមអវិជ្ជមានសម្រាប់គោលបំណងដូចគ្នា ដូចជា fruits[-1]។ ទោះបីជា, នៅក្នុង JavaScript វានឹងមិនដំណើរការ។ លទ្ធផលនឹងជា undefinedពីព្រោះសន្ទស្សន៍ក្នុងតង្កៀបការ៉េត្រូវបានចាត់ទុកតាមព្យញ្ជនៈ។ យើងអាចគណនាលិបិក្រមធាតុចុងក្រោយដោយច្បាស់លាស់ ហើយបន្ទាប់មកចូលប្រើវា៖ fruits[fruits.length - 1]។

```
let fruits = ["Apple", "Orange", "Plum"];

alert( fruits[fruits.length-1] ); // Plum
```

ពិបាកបន្តិចមែនទេ? យើងត្រូវសរសេរឈ្មោះអថេរពីរដង។ ជាសំណាងល្អ មានរូបមន្តខ្លីជាងនេះ៖ fruits.at(-1)៖

```
let fruits = ["Apple", "Orange", "Plum"];

// same as fruits[fruits.length-1]
alert( fruits.at(-1) ); // Plum
```

នៅក្នុងពាក្យផ្សេងទៀត arr.at(i)៖

- គឺដូចគ្នាទៅនឹង arr[i], if i >= 0.
- សម្រាប់តម្លៃអវិជ្ជមាន វាដើរថយក្រោយពីចុងបញ្ចប់នៃArray។

៣.១.៣ **ទិន្នន័យសាស្ត្រ** pop/push, shift/unshift

Queue គឺជាផ្នែកមួយនៃការប្រើប្រាស់ទូទៅបំផុតនៃArrayមួយ ។ នៅក្នុងវិទ្យាសាស្ត្រកុំព្យូទ័រ នេះមានន័យថាការប្រមូលធាតុដែលបានបញ្ជាដែលគាំទ្រប្រតិបត្តិការពីរ៖

- pushបន្ថែមធាតុមួយទៅចុងក្រោយ។
- shiftទទួលបានធាតុមួយពីដំបូង រុញជួរ ដូច្នេះធាតុទី 2 ក្លាយជាធាតុទី 1 ។

រូបភាព 9៖ Methods Push/Shift

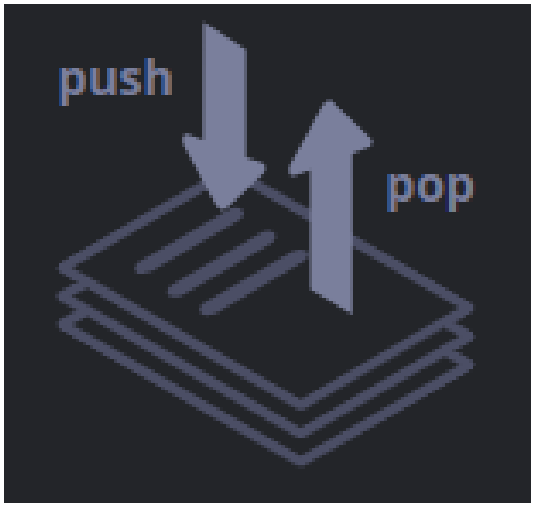
Arrayគាំទ្រប្រតិបត្តិការទាំងពីរ។ នៅក្នុងការអនុវត្តយើងត្រូវការវាជាញឹកញាប់។ ឧទាហរណ៍ Queue នៃសារដែលត្រូវការបង្ហាញនៅលើអេក្រង់។ មានករណីប្រើប្រាស់មួយផ្សេងទៀតសម្រាប់Array រចនាសម្ព័ន្ធទិន្នន័យដែលមានឈ្មោះថា Stack ។

វាគាំទ្រប្រតិបត្តិការពីរ៖

- pushបន្ថែមធាតុមួយនៅចុងក្រោយ។
- popដកធាតុចុងក្រោយ។

ដូច្នេះធាតុថ្មីត្រូវបានបន្ថែមឬយកចេញពី "ចុងបញ្ចប់" ជានិច្ច។

Stack មួយជាធម្មតាត្រូវបានបង្ហាញជាកញ្ចប់នៃសន្លឹកបៀ៖ សន្លឹកបៀថ្មីត្រូវបានបន្ថែមទៅកំពូ៖



រូបភាព 10៖ Methods Push/Pop

សម្រាប់ Stacks ជាតុដែលបានរុញចុងក្រោយបំផុតត្រូវបានទទួលមុនគេ ដែលត្រូវបានគេហៅថាគោលការណ៍ LIFO (Last-In-First-Out) ផងដែរ។ សម្រាប់ Queues យើងមាន FIFO (First-In-First-Out)។

Arrayក្នុង JavaScript អាចដំណើរការទាំងជា Queues និងជា Stacks។ ពួកវាអនុញ្ញាតឱ្យអ្នកបន្ថែម / ដកចេញធាតុតាំងពីដើមឬចុងបញ្ចប់។ នៅក្នុងវិទ្យាសាស្ត្រកុំព្យូទ័រ រចនាសម្ព័ន្ធទិន្នន័យដែលអនុញ្ញាតឱ្យវាត្រូវបានគេហៅថា deque ។ វិធីសាស្ត្រដែលធ្វើការជាមួយចុងបញ្ចប់នៃArray៖

- Pop ដកធាតុចុងក្រោយនៃArray៖

```
let fruits = ["Apple", "Orange", "Pear"];

alert( fruits.pop() ); // remove "Pear" and alert it

alert( fruits ); // Apple, Orange
```

ទាំងពីរ fruits.pop( )និង fruits.at(-1) Return ជាតុចុងក្រោយនៃArray។ ប៉ុន្តែ fruits.pop( ) ក៏កែប្រែArrayដោយដកវាចេញ។

- Push បន្ថែមធាតុទៅចុងក្រោយនៃArray៖

```
let fruits = ["Apple", "Orange"];

fruits.push("Pear");

alert( fruits ); // Apple, Orange, Pear
```

ការហៅ fruits.push(...)គឺស្មើនឹង fruits[fruits.length] = ...។

- Shift ដកធាតុដំបូងនៃArray៖

```
let fruits = ["Apple", "Orange", "Pear"];

alert( fruits.shift() ); // remove Apple and alert it

alert( fruits ); // Orange, Pear
```

- Unshift បន្ថែមធាតុទៅដើមនៃArray៖

```
let fruits = ["Orange", "Pear"];

fruits.unshift('Apple');

alert( fruits ); // Apple, Orange, Pear
```

វិធីសាស្ត្រ pushនិង unshiftអាចបន្ថែមធាតុជាច្រើនក្នុងពេលតែមួយ៖

```
let fruits = ["Apple"];

fruits.push("Orange", "Peach");
fruits.unshift("Pineapple", "Lemon");

// ["Pineapple", "Lemon", "Apple", "Orange", "Peach"]
alert( fruits );
```

៣.១.៤ Internals

Arrayគឺជាប្រភេទ Object ពិសេស។ តង្កៀបការ៉េដែលប្រើដើម្បីចូលប្រើ property arr[0]ពិតគឺមកពី object រូបមន្ត។ នោះជាការសំខាន់ដូចគ្នានឹង object obj[key]។ ដែល arr ជា object ខណៈពេលដែលលេខត្រូវបានប្រើជាកូនសោរ។

ពួកវាពង្រីក object ដែលផ្តល់វិធីសាស្ត្រពិសេស ដើម្បីធ្វើការជាមួយការប្រមូលទិន្នន័យតាមលំដាប់ និង length property ផងដែរ។ ប៉ុន្តែនៅស្នូលវានៅតែជា Object។ ArrayគឺជាObjectមួយ ហើយដូច្នេះមានឥរិយាបទដូច Object មួយ។

ឧទាហរណ៍៖

```
let fruits = ["Banana"]

let arr = fruits; // copy by reference (two variables reference the same array)

alert( arr === fruits ); // true

arr.push("Pear"); // modify the array by reference
```

```
alert( fruits ); // Banana, Pear - 2 items now
```

... ប៉ុន្តែអ្វីដែលធ្វើឱ្យArrayពិតជាពិសេសគឺតំណាង internal របស់វា។ ម៉ាស៊ីនព្យាយាមរក្សាទុកជាតុរបស់វានៅក្នុងតំបន់អង្គចងចាំជាប់គ្នា ពីមួយទៅមួយ ដូចដែលបានបង្ហាញនៅលើរូបភាពនៅក្នុងជំពូកនេះ ហើយមានការបង្កើនប្រសិទ្ធភាពផ្សេងទៀតផងដែរ ដើម្បីធ្វើឱ្យArrayដំណើរការយ៉ាងរហ័ស។

ប៉ុន្តែពួកវាទាំងអស់នឹងខូច ប្រសិនបើយើងឈប់ធ្វើការជាមួយArrayដូចទៅនឹង "ordered collection" ហើយចាប់ផ្តើមធ្វើការជាមួយវាដូចជា Object ធម្មតា។

ឧទាហរណ៍ តាមបច្ចេកទេស យើងអាចធ្វើដូចនេះបាន៖

```
let fruits = []; // make an array

fruits[99999] = 5; // assign a property with the index far greater than its length

fruits.age = 25; // create a property with an arbitrary name
```

វាអាចទៅរួច ព្រោះArrayគឺជាObjectនៅមូលដ្ឋានរបស់វា។ យើងអាចបន្ថែមProperty ណាមួយទៅពួកគេ។ ប៉ុន្តែម៉ាស៊ីននឹងយើញថាយើងកំពុងធ្វើការជាមួយArrayដូចទៅនឹងObjectធម្មតាដែរ។ ការបង្កើនប្រសិទ្ធភាពជាក់លាក់នៃArrayមិនសំក្តិសមសម្រាប់ករណីបែបនេះទេ ហើយនឹងត្រូវបានបិទ អត្ថប្រយោជន៍របស់ពួកគេនឹងរលាយបាត់។

វិធីប្រើArrayខុស៖

- បន្ថែម property ដែលមិនមែនជាលេខដូចជា arr.test = 5.
- Make holes៖ បន្ថែម arr[0]ហើយបន្ទាប់មក arr[1000](និងគ្មានអ្វីរវាងពួកវា)។
- បំពេញArrayក្នុងលំដាប់បញ្ច្រាស ដូចជា arr[1000]ជាដើម arr[999]។

សូមគិតពីArrayជារចនាសម្ព័ន្ធពិសេស ដើម្បីធ្វើការជាមួយ ទិន្នន័យដែលបានបញ្ជាទិញ ។ ពួកគេផ្តល់វិធីសាស្ត្រពិសេសសម្រាប់នោះ។ Arrayត្រូវបានកែសម្រួលដោយប្រុងប្រយ័ត្ននៅក្នុងម៉ាស៊ីនJavaScript ដើម្បីធ្វើការជាមួយទិន្នន័យដែលបានបញ្ជាទិញជាប់គ្នា សូមប្រើពួកវាតាមវិធីនេះ។ ហើយប្រសិនបើអ្នកត្រូវការកូនសោរ ឱកាសគឺខ្ពស់ដែលអ្នកពិតជាត្រូវការObjectធម្មតា {}។

៣.១.៥ Performance

វិធីសាស្ត្រ push/popដំណើរការលឿន ខណៈពេល shift/unshiftដែលយឺត។

រូបភាព 11៖ Performance

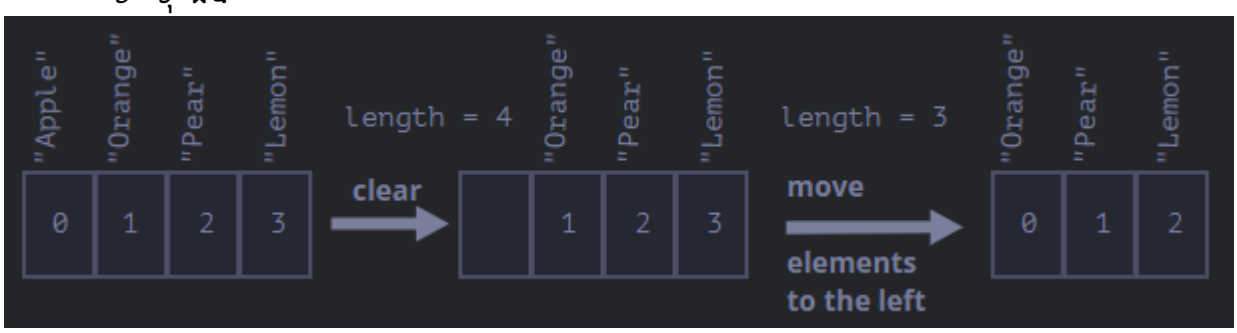
ហេតុអ្វីបានជាវាលឿនជាងក្នុងការធ្វើការជាមួយចុងក្រោយនៃArrayជាងការចាប់ផ្តើមរបស់វា? គោះមើលថាមានអ្វីកើតឡើងក្នុងពេលប្រតិបត្តិ៖

```
fruits.shift(); // take 1 element from the start
```

វាមិនគ្រប់គ្រាន់ទេក្នុងការយក និងលុបធាតុដោយប្រើ Index 0។ ធាតុផ្សេងទៀតក៏ចាំបាច់ត្រូវប្តូរលេខផងដែរ។

ប្រតិបត្តិការ shiftត្រូវធ្វើ ៣ យ៉ាង៖

1. យកធាតុចេញដោយប្រើ Index 0។
2. ផ្លាស់ទីធាតុទាំងអស់ទៅខាងឆ្វេង ប្តូរលេខពួកវាពី Index 1ទៅ 0ពី 2ទៅ 1ជាដើម។
3. ធ្វើបច្ចុប្បន្នភាព length property។



រូបភាព 12៖ shift

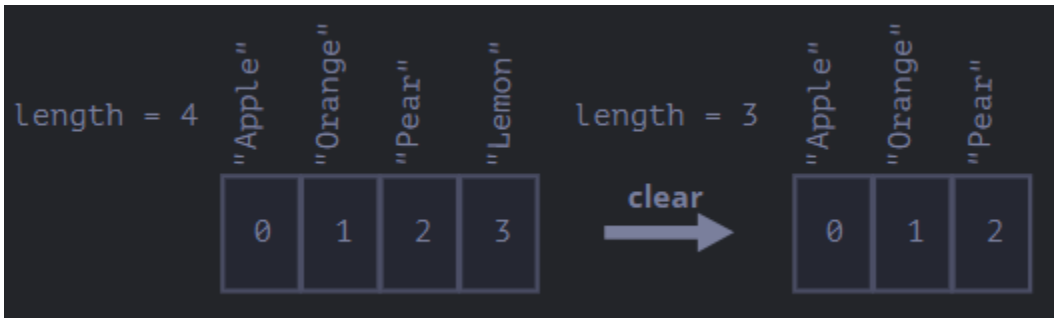
ធាតុកាន់តែច្រើននៅក្នុងArray ពេលវេលាកាន់តែច្រើនដើម្បីផ្លាស់ទីពួកវា ប្រតិបត្តិការក្នុងអង្គចងចាំកាន់តែច្រើន។

រឿងស្រដៀងគ្នានេះកើតឡើងជាមួយ unshift៖ ដើម្បីបន្ថែមធាតុមួយទៅការចាប់ផ្តើមនៃArray យើងត្រូវផ្លាស់ទីធាតុដែលមានស្រាប់ទៅខាងស្តាំជាមុនសិន ដោយបង្កើនសន្ទស្សន៍របស់វា។ ហើយនៅ

ជាមួយអ្វី push/pop? ពួកគេមិនចាំបាច់ផ្លាស់ទីអ្វីទេ។ ដើម្បីដកធាតុមួយចេញពីចុងក្រោយ popវិធីសាស្ត្រសម្អាត Index និងធ្វើឱ្យខ្លី length។

សកម្មភាពសម្រាប់ popប្រតិបត្តិការ៖

```
fruits.pop(); // take 1 element from the end
```



រូបភាព 13៖ Pop

វិធីសាស្ត្រ pop មិនចាំបាច់ផ្លាស់ទីអ្វីទេព្រោះធាតុផ្សេងទៀតរក្សាសន្ទស្សន៍របស់វា។ នោះហើយជាមូលហេតុដែលវារឿនខ្លាំង។ រឿងស្រដៀងគ្នាជាមួយ pushវិធីសាស្ត្រ។

៣.១.៦ Loops

វិធីមួយក្នុងចំណោមវិធីចាស់បំផុតក្នុងការបង្វិលធាតុArrayគឺ for loop លើ Index៖

```
let arr = ["Apple", "Orange", "Pear"];

for (let i = 0; i < arr.length; i++) {
  alert( arr[i] );
}
```

ប៉ុន្តែសម្រាប់Arrayមានទម្រង់ loop មួយទៀតគឺ for..of៖

```
let fruits = ["Apple", "Orange", "Plum"];

// iterates over array elements
for (let fruit of fruits) {
  alert( fruit );
}
```

លេខនេះ for..ofមិនផ្តល់សិទ្ធិចូលប្រើចំនួនធាតុបច្ចុប្បន្នទេ គ្រាន់តែតម្លៃរបស់វា ប៉ុន្តែក្នុងករណីភាគច្រើនវាគ្រប់គ្រាន់ហើយ។

តាមបច្ចេកទេស ដោយសារArrayជាObject វាក៏អាចប្រើបានដែរ for..in៖

```
let arr = ["Apple", "Orange", "Pear"];

for (let key in arr) {
  alert( arr[key] ); // Apple, Orange, Pear
```

```
}
```

ប៉ុន្តែតាមពិត នោះជាគំនិតអាក្រក់។ មានបញ្ហាដែលអាចកើតមានជាមួយវា៖

- loop for..in ធ្វើម្តងទៀតលើ Property ទាំងអស់ មិនត្រឹមតែជាលេខប៉ុណ្ណោះទេ។ មានObject ដែលហៅថា "array-like" នៅក្នុងកម្មវិធីរុករកតាមអ៊ីនធឺណិត និងក្នុងបរិយាកាសផ្សេងទៀត ដែល មើលទៅដូចArray ។ នោះគឺពួកគេមាន lengthនិងធ្វើ indexes properties ប៉ុន្តែពួកវាក៏អាចមានProperty និងវិធីសាស្ត្រដែលមិនមែនជាលេខផ្សេងទៀត ដែលជាធម្មតាយើងមិនត្រូវការ។ ង្វិល for..inជុំនឹងរាយបញ្ជីពួកគេ។ ដូច្នេះប្រសិនបើយើងត្រូវការធ្វើការជាមួយObjectដូច Array នោះProperty "extra" ទាំងនេះអាចក្លាយជាបញ្ហា។
- loop for..in ត្រូវបានធ្វើឱ្យប្រសើរសម្រាប់Objectទូទៅ មិនមែនArrayទេ ហើយដូច្នេះវាយឺតជាង 10-100 ដង។ ជាការពិតណាស់វានៅតែលឿនណាស់។ ការបង្កើនល្បឿនអាចមានបញ្ហាតែក្នុងបញ្ហាប៉ុណ្ណោះ។ ប៉ុន្តែនៅតែយើងគួរតែដឹងពីភាពខុសគ្នា។ ជាទូទៅ យើងមិនគួរប្រើ for..in សម្រាប់Arrayទេ។

៣.១.៧ អំពី "length"

length property ធ្វើបច្ចុប្បន្នភាពដោយស្វ័យប្រវត្តិនៅពេលយើងកែប្រែArray។ ដើម្បីឱ្យច្បាស់លាស់ វាមិនមែនជាចំនួននៃតម្លៃនៅក្នុងArrayនោះទេ ប៉ុន្តែជាសន្ទស្សន៍លេខដ៏អស្ចារ្យបំផុតបូកមួយ។

ឧទាហរណ៍៖

```
let fruits = [];
fruits[123] = "Apple";

alert( fruits.length ); // 124
```

ចំណាំថាជាធម្មតាយើងមិនប្រើArrayដូចនោះទេ។ អ្វីដែលគួរឱ្យចាប់អារម្មណ៍មួយទៀតអំពី length property គឺអាចសរសេរបាន។

ប្រសិនបើយើងបង្កើនវាដោយដៃ គ្មានអ្វីគួរឱ្យចាប់អារម្មណ៍កើតឡើងទេ។ ប៉ុន្តែប្រសិនបើយើងបន្ថយវា Arrayត្រូវបានកាត់។ ដំណើរការគឺមិនអាចត្រឡប់វិញបានទេ នេះជាឧទាហរណ៍៖

```
let arr = [1, 2, 3, 4, 5];

arr.length = 2; // truncate to 2 elements
alert( arr ); // [1, 2]

arr.length = 5; // return length back
alert( arr[3] ); // undefined: the values do not return
```

៣.១.៨ new Array( )

មាន រូបមន្ត មួយបន្ថែមទៀតដើម្បីបង្កើតArrayមួយ៖

```
let arr = new Array("Apple", "Pear", "etc");
```

កម្រប្រើណាស់ ព្រោះតង្កៀបរាងការ៉េ []ខ្លីជាង។ វាក៏មានមុខងារដ៏ពិបាកមួយផងដែរ។ ប្រសិនបើ new Arrayត្រូវបានហៅដោយArgumentតែមួយដែលជាលេខ នោះវានឹងបង្កើតArray ដោយមិនមានធាតុ ប៉ុន្តែជាមួយនឹងប្រវែងដែលបានផ្តល់ ។

ឧទាហរណ៍៖

```
let arr = new Array(2); // will it create an array of [2] ?

alert( arr[0] ); // undefined! no elements.

alert( arr.length ); // length 2
```

ដើម្បីជៀសវាងការភ្ញាក់ផ្អើលបែបនេះ យើងជាធម្មតាប្រើតង្កៀបការ៉េ លុះត្រាតែយើងពិតជាដឹងថាយើងកំពុងធ្វើអ្វីមួយ។

៣.១.៩ Multidimensional arrays

ArrayអាចមានធាតុដែលជាArrayផងដែរ។ យើងអាចប្រើវាសម្រាប់Arrayពហុវិមាត្រ ឧទាហរណ៍ដើម្បីរក្សាទុកម៉ាទ្រីស៖

```
let matrix = [
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9]
];

alert( matrix[0][1] ); // 2, the second value of the first inner array
```

៣.១.១០ toString

Arrayមានការអនុវត្ត toStringវិធីសាស្ត្រផ្ទាល់ខ្លួនរបស់វាដែលត្រឡប់បញ្ជីធាតុដែលបំបែកដោយសញ្ញាក្បៀស។

ឧទាហរណ៍៖

```
let arr = [1, 2, 3];

alert( arr ); // 1,2,3
alert( String(arr) === '1,2,3' ); // true
```

ផងដែរ តោះសាកល្បងវិធីនេះ៖

```
alert( [] + 1 ); // "1"
alert( [1] + 1 ); // "11"
alert( [1,2] + 1 ); // "1,21"
```

Arrayមិនមាន Symbol.toPrimitiveទាំងមិនបិតថេរ valueOfពួកវាអនុវត្តតែ toStringការបំប្លែងដូច្នេះនៅទីនេះ []ក្លាយជាខ្សែអក្សរទទេ [1]ក្លាយជា "1"និង [1,2]ក្លាយជា "1,2".

នៅពេលដែល binary plus "+"operator បន្ថែមអ្វីមួយទៅ string វាបំប្លែងវាទៅជា string ផងដែរ ដូច្នេះដំហានបន្ទាប់មើលទៅដូចនេះ៖

```
alert( "" + 1 ); // "1"
alert( "1" + 1 ); // "11"
alert( "1,2" + 1 ); // "1,21"
```

៣.២ Array Methods

ក្នុង JavaScript មាន Array Methods មួយចំនួនដែលគេបង្កើតឡើងផ្តល់ភាពងាយស្រួលសម្រាប់ប្រើប្រាស់ក្នុងការប្រតិបត្តិការគណនា។

ទាំងនេះគឺជា Array Methods៖

តារាង 6៖ Array Methods

Method	Description
concat( )	joins two or more arrays and returns a result
indexOf( )	searches an element of an array and returns its position
findIndex( )	returns the first index of an array element that passes a test រូបមន្ត៖ array. find(*function(currentValue, index, arr),this value*)
forEach( )	method executes a provided function for each array element
includes( )	checks if an array contains a specified element or not.
push( )	adds a new element to the end of an array and returns the new length of an array

unshift( )	adds a new element to the beginning of an array and returns the new length of an array
pop( )	removes the last element of an array and returns the removed element
shift( )	removes the first element of an array and returns the removed element
sort( )	sorts the elements alphabetically in strings and in ascending order
slice( )	selects the part of an array and returns the new array
splice( )	removes or replaces existing elements and/or adds new elements
toString( )	Returns the string representation of an array
join( )	Concatenates the array elements to a string
length	Returns the number of elements in an array
reverse( )	Returns the array in reverse order

៣.៣ លំហាតអនុវត្ត

1. តើអាចបង្កើត array បានប៉ុន្មានវិធី? សូមផ្តល់ឧទាហរណ៍។
2. តើការប្រើ Array.of( ) និង new Array( ) មានភាពខុសគ្នាអ្វីខ្លះ?
3. តើសូចនាករ index ក្នុង array ចាប់ផ្តើមពីលេខប៉ុន្មាន?
4. តើភាពខុសគ្នារវាង push( ) និង unshift( ) ជាអ្វី?
5. តើមុខងារ splice( ) និង slice( ) មានភាពខុសគ្នាដូចម្តេច?
6. តើធ្វើដូចម្តេចដើម្បីចូលប្រើតម្លៃចុងក្រោយនៃ array ដោយមិនចាំបាច់ដឹងប្រវែងរបស់វា?
7. តើអាចប្រើ for...of លើ object បានដែរឬទេ? ហេតុអ្វី?
8. តើពាក្យបញ្ជា Array.isArray( ) មានតួនាទីអ្វី?
9. សរសេរ Code ដើម្បីបង្កើត array ដែលផ្ទុកឈ្មោះនៃប្រទេស 5 និងបង្ហាញវាតាម console។

10. សរសេរ Code ដើម្បីបន្ថែម និងលុបធាតុក្នុង array៖
    - បង្កើត array fruits ដែលមាន 'Apple', 'Banana', 'Cherry'។
    - បន្ថែម 'Mango' នៅចុងបញ្ចប់។
    - លុប 'Banana' ចេញពី array។
    - បង្ហាញ array ដែលបានផ្លាស់ប្តូរ។
11. បង្កើត array numbers ដែលមាន [1, 2, 3, 4, 5] ហើយប្រើ forEach( ) ដើម្បីបង្ហាញតម្លៃទាំងអស់។
12. បង្កើត array numbers = [2, 4, 6, 8] ហើយប្រើ map( ) ដើម្បីបង្កើត array ថ្មីដែលមានតម្លៃទាំងអស់បង្កើនឡើង 2 ដង។ បង្ហាញ array ថ្មីនោះ។
13. បង្កើត array ages = [10, 15, 18, 21, 25] ហើយប្រើ filter( ) ដើម្បីបង្កើត array ថ្មីដែលមានតែអាយុចាប់ពី 18 ឡើង។
14. ប្រើ find( ) ដើម្បីរកតម្លៃដំបូងដែលធំជាង 18 ក្នុង array [5, 12, 18, 25, 30]។
15. បង្កើត array scores = [50, 20, 75, 90, 10]។ ប្រើ sort( ) ដើម្បីរៀបតម្លៃតាមលំដាប់ឡើង។
16. បង្កើត array arr1 = [1, 2, 3] និង arr2 = [4, 5, 6]។ ប្រើ concat( ) ឬ spread operator (...) ដើម្បីបង្កើត array ថ្មី ដែលមានទាំងតម្លៃពី arr1 និង arr2។
17. ប្រើ includes( ) ដើម្បីពិនិត្យថា 'JavaScript' មានក្នុង array ['Python', 'JavaScript', 'C++'] ដែរឬទេ ?
18. ប្រើ Set ដើម្បីយកតម្លៃស្ទួនចេញពី array [1, 2, 3, 3, 4, 4, 5]។

មេរៀនទី ៤៖ ការគ្រប់គ្រង DOM

Document Object Model (DOM) គឺជាចំណុចប្រទាក់សរសេរកម្មវិធីសម្រាប់ឯកសារគេហទំព័រ។ វាតំណាងឱ្យរចនាសម្ព័ន្ធនៃឯកសារជាមែកធាងនៃ objects ដែលអនុញ្ញាតឱ្យអ្នកអភិវឌ្ឍន៍រៀបចំ content, structure និង style នៃគេហទំព័រដោយ dynamic ។ នៅក្នុងមេរៀននេះ យើងនឹងស្វែងយល់ពីរបៀបដែល JavaScript ធ្វើអន្តរកម្មជាមួយ DOM ដើម្បីកែប្រែគេហទំព័រតាមពេលវេលាជាក់ស្តែង។

៤.១ ស្វែងយល់អំពី DOM

DOM គឺជារចនាសម្ព័ន្ធដូចដើមឈើ ដែលថ្នាំងនីមួយៗតំណាងឱ្យផ្នែកនៃ document ។ នេះអាចជា element, an attribute, or text ។ ឧទាហរណ៍ ឯកសារ HTML ត្រូវបានតំណាងជាមែកធាងដែលថ្នាំងឯកសារជាឫស ហើយធាតុដូចជា <html> <head> និង <body> គឺជាកូនរបស់វា។

៤.១.១ DOM Nodes

- Element Nodes៖ តំណាងឱ្យធាតុ HTML (ឧ. , <p>, <a>)។
- Text Nodes៖ តំណាងឱ្យ text content នៅខាងក្នុងធាតុ។
- Attribute Nodes៖ តំណាងឱ្យ attributes នៃធាតុ (ឧ. id, class)។

៤.២ ការចូលប្រើធាតុ DOM

JavaScript ផ្តល់នូវវិធីសាស្ត្រជាច្រើនសម្រាប់ការចូលប្រើ DOM elements ៖

- getElementById៖ ជ្រើសរើសធាតុមួយដោយ id attribute ។

```
let element = document.getElementById('myElement');
```

- getElementsByClassName៖ ជ្រើសរើសធាតុដោយ class attribute ។

```
let elements = document.getElementsByClassName('myClass');
```

- getElementsByTagName៖ ជ្រើសរើសធាតុដោយ tag name របស់ពួកគេ។

```
let elements = document.getElementsByTagName('div');
```

- querySelector៖ ជ្រើសរើសធាតុទីមួយដែលត្រូវគ្នានឹង CSS Selector។

```
let element = document.querySelector('.myClass');
```

- querySelectorAll៖ ជ្រើសរើសធាតុទាំងអស់ដែលត្រូវគ្នានឹង CSS Selector។

```
let elements = document.querySelectorAll('div.myClass');
```

៤.៣ ការកែប្រែធាតុ DOM

នៅពេលដែលធាតុមួយត្រូវបានចូលប្រើ អ្នកអាចកែប្រែ properties និង content របស់វាបាន៖

- Changing Text Content៖

```
element.textContent = 'New Text Content';
```

- Changing HTML Content៖

```
element.innerHTML = '<p>New HTML Content</p>';
```

- Changing Attributes៖

```
element.setAttribute('id', 'newId');
```

- Modifying Styles៖

```
element.style.backgroundColor = 'blue';
```

៤.៤ ការបន្ថែម និងលុបធាតុ

អ្នកក៏អាចបន្ថែម និងលុបធាតុចេញពី DOM ផងដែរ៖

- Creating New Elements៖

```
 let newElement = document.createElement('div');
 newElement.textContent = 'I am a new element';
```

- Appending Elements៖

```
document.body.appendChild(newElement);
```

- Removing Elements៖

```
document.body.removeChild(newElement);
```

៤.៥ Event Handling

DOM អនុញ្ញាតឱ្យអ្នកធ្វើអន្តរកម្មជាមួយសកម្មភាពរបស់អ្នកប្រើតាមរយៈព្រឹត្តិការណ៍៖

- Adding Event Listeners៖

ឧទាហរណ៍៖

```
 element.addEventListener('click', function() {
```

```
  alert('Element clicked!');
});
```

- Adding Event Listeners៖

ឧទាហរណ៍៖

```
function handleClick() {
  alert('Element clicked!');
}
element.addEventListener('click', handleClick);
            element.removeEventListener('click',
handleClick);
```

៤.៦ ឧទាហរណ៍ជាក់ស្តែង

នេះគឺជាឧទាហរណ៍ជាក់ស្តែងដែលរួមបញ្ចូលគ្នានូវការរៀបចំ DOM និងការដោះស្រាយព្រឹត្តិការណ៍៖

ឧទាហរណ៍៖

```
<!DOCTYPE html>
<html>
<head>
  <title>DOM Manipulation Example</title>
</head>
<body>
  <button id="myButton">Click Me</button>
  <div id="myDiv">Original Content</div>
  <script>
    let button = document.getElementById('myButton');
    let div = document.getElementById('myDiv');

    button.addEventListener('click', function() {
```

```
        div.textContent = 'Content has been changed!';
        div.style.color = 'red';
      });
    </script>
  </body>
  </html>
```

ក្នុងឧទាហរណ៍នេះ ការចុចប៊ូតុងផ្លាស់ប្តូរខ្លឹមសារអត្ថបទរបស់ ហើយកែប្រែពណ៌អត្ថបទរបស់វា។

៤.៧ លំហាត់

1. អ្នកអាចបញ្ជាក់អំពី querySelector( ) និង querySelectorAll( ) ម្តងទៀតបានទេ?
2. តើភាពខុសគ្នារវាង innerHTML និង textContent ជាអ្វី?
3. ឲ្យឧទាហរណ៍នៃការបង្កើត និងបន្ថែមធាតុថ្មីទៅក្នុង DOM ដោយប្រើ JavaScript។
4. តើ removeChild( ) និង replaceChild( ) ប្រើសម្រាប់អ្វី?
5. អ្នកអាចពណ៌នាពីវិធីសាស្ត្រដែលប្រើសម្រាប់ផ្លាស់ប្តូរស្ទីលរបស់ធាតុ DOM បានទេ?
6. តើ window និង document មានភាពខុសគ្នាក្នុងការគ្រប់គ្រង DOM ដោយរបៀបណា?
7. ចំណាកស្រុកប្រើ setAttribute( ) និង removeAttribute( ) ត្រូវបានប្រើសម្រាប់អ្វី?
8. អ្នកអាចបញ្ជាក់ពីភាពខុសគ្នារវាង appendChild( ) និង innerHTML បានទេ?
9. តើអ្នកអាចបង្កើត និងបន្ថែមធាតុ <button> ទៅក្នុង ដោយប្រើ JavaScript ដោយរបៀបណា?
10. បង្ហាញឧទាហរណ៍នៃការកំណត់ classList.add( ) និង classList.remove( ) ដើម្បីបន្ថែមនិងលុប class នៅក្នុង DOM។
11. ចូលបង្កើតទំព័រ HTML ដែលមាន <button> មួយ។ ប្រើ JavaScript ដើម្បីផ្លាស់ប្តូរចំណងជើង <h1> នៅពេលចុចប៊ូតុង។
12. ចូលបង្កើត <ul> នៅលើទំព័រ។ បន្ថែម <li> ថ្មីៗចូលក្នុង ul ពេលចុចប៊ូតុង។
13. ចូលបង្កើត ដែលផ្លាស់ប្តូរសេរីពណ៌ផ្ទៃខាងក្រោយនៅពេលចុចលើវា។
14. ចូលបង្កើត <button> ដែលផ្លាស់ប្តូរសំរបសំរួល (font-size) និងពណ៌អក្សរ (color) របស់ <p> នៅពេលចុចប៊ូតុង។

15. ចូលបង្កើត
    - បន្ថែមប៊ូតុងដែលបង្កើត <p> ថ្មីនៅក្នុង
    - បន្ថែមប៊ូតុងផ្សេងទៀតដើម្បីលុប <p> ចុងក្រោយក្នុង
16. ចូលបង្កើត form ដែលអាចបន្ថែមចន្លោះបំពេញអាសយដ្ឋានថ្មីៗដោយប្រើប៊ូតុង
17. ចូលបង្កើត <ul> ដែលអាចចុចលើ ដើម្បីប្តូរស្ទីល (toggle CSS class)
18. ចូលបង្កើតប៊ូតុងដែលអាចបិទ (disable) និងបើក (enable) ជាតុ <input>

មេរៀនទី ៥៖ កម្មវិធី Asynchronous

Asynchronous programming គឺជាមូលដ្ឋានគ្រឹះនៃការអភិវឌ្ឍន៍ JavaScript ទំនើប ដែលអនុញ្ញាតឱ្យកម្មវិធីអាចដំណើរការ non-blocking operations ។ មេរៀននេះនឹងស្វែងយល់ពីភាពស្មុគស្មាញនៃ asynchronous operations ដោយផ្តោតលើការ callbacks, promises, async/await និង Fetch API។ តាមរយៈការយល់ដឹងអំពីគំនិតទាំងនេះ អ្នកនឹងត្រូវបានបំពាក់ដើម្បីដោះស្រាយសេណារីយ៉ូស្មុគស្មាញ ដែលប្រតិបត្តិការត្រូវដំណើរការស្របគ្នា ឬរង់ចាំធនធានខាងក្រៅ។

៥.១ ស្វែងយល់អំពីប្រតិបត្តិការ Asynchronous

នៅក្នុង JavaScript ប្រតិបត្តិការជាច្រើនគឺ asynchronous មានន័យថាពួកគេមិនរារាំងការប្រតិបត្តិនៃ Code ជាបន្តបន្ទាប់នោះទេ។ នេះមានសារៈសំខាន់ជាពិសេសសម្រាប់កិច្ចការដែលត្រូវការពេលវេលា ដូចជា reading files, making network requests, ឬ querying a database ។ បើគ្មាន asynchronous programming ប្រតិបត្តិការទាំងនេះនឹងបញ្ឈប់កម្មវិធីទាំងមូល ដែលនាំឱ្យអ្នកប្រើប្រាស់មានបទពិសោធន៍មិនល្អ និងការប្រើប្រាស់ធនធានគ្មានប្រសិទ្ធភាព។

៥.១.១ Event Loop

JavaScript ប្រើគំរូដែលដំរុញដោយព្រឹត្តិការណ៍ជាមួយនឹង event loop ដែលគ្រប់គ្រង asynchronous operations ។ event loop ត្រួតពិនិត្យជានិច្ចនូវ call stack និង task queue (or callback queue) ។ នៅពេលដែល call stack គឺច្បាស់ event loop ដំណើរការការកិច្ចពីជួរ ដោយធានាថា asynchronous code ត្រូវបានប្រតិបត្តិនៅពេលដែលខ្សែស្រឡាយមេឥតគិតថ្លៃ។

៥.១.២ អនុគមន៍ Callback

callbacks ជាវិធីសាស្ត្រចម្បងសម្រាប់ដោះស្រាយ asynchronous operations ។ callback គឺជាមុខងារមួយដែលត្រូវបានបញ្ជូនជាArgumentទៅមុខងារមួយផ្សេងទៀត ហើយត្រូវបានប្រតិបត្តិបន្ទាប់ពីបញ្ចប់ asynchronous task ។

ឧទាហរណ៍៖

```
function fetchData(callback) {
  setTimeout(() => {
      const data = "Some data";
      callback(data);
  }, 1000);
```

```
}
fetchData((data) => {
  console.log(data); // Output after 1 second: "Some data"
});
```

ខណៈពេលដែលមានប្រសិទ្ធភាព callbacks អាចនាំទៅដល់ " callback hell " នៅពេលដោះស្រាយជាមួយ multiple asynchronous operations ដែលបណ្តាលឱ្យមាន Code សម្ងាត់យ៉ាងជ្រៅដែលពិបាកគ្រប់គ្រង។

៥.២ Promises

Promises ត្រូវបានណែនាំដើម្បីកែលម្អការគ្រប់គ្រង asynchronous operations ដោយផ្តល់នូវវិធីសាស្ត្រកាន់តែស្អាត និងអាចគ្រប់គ្រងបាន។ promise តំណាងឱ្យការបញ្ចប់ចុងក្រោយ (ឬការបរាជ័យ) នៃ asynchronous operation និងតម្លៃលទ្ធផលរបស់វា។

៥.២.១ ការបង្កើត Promise

promise ត្រូវបានបង្កើតដោយប្រើ Promise constructor ដែលយកមុខងារមួយជាមួយនឹង Argumentពីរ៖ resolve និង reject ។ resolve function ត្រូវបានហៅនៅពេលដែលប្រតិបត្តិការជោគជ័យ ខណៈពេលដែល reject ត្រូវបានហៅប្រសិនបើមានកំហុសកើតឡើង។

ឧទាហរណ៍៖

```
let myPromise = new Promise((resolve, reject) => {
  setTimeout(() => {
      const success = true;
      if (success) {
          resolve("Operation succeeded!");
      } else {
          reject("Operation failed.");
      }
  }, 1000);
});
```

```
myPromise.then(result => {
   console.log(result);  //  Output  after  1  second:  "Operation
succeeded!"
}).catch(error => {
  console.error(error);
});
```

៥.២.២ ការចង Promises

Promises អនុញ្ញាតឱ្យដាក់ខ្សែសង្វាក់ដោយប្រើវិធីសាស្ត្រ. then ( ) និង. catch ( ) ដែលធ្វើឱ្យវាកាន់តែងាយស្រួលក្នុងការដោះស្រាយលំដាប់នៃ asynchronous operations ។

ឧទាហរណ៍៖

```
myPromise
    .then(result => {
        console.log(result);
        return "Another value";
    })
    .then(newResult => {
        console.log(newResult);
    })
    .catch(error => {
        console.error(error);
    })
```

៥.៣ កំហុសក្នុងការគ្រប់គ្រង Promises

Promise chains គឺអស្ចារ្យណាស់ក្នុងការដោះស្រាយកំហុស។ នៅពេលដែលការPromise បដិសេធ ការគ្រប់គ្រងនឹងលោតទៅអ្នកដោះស្រាយការបដិសេធដែលនៅជិតបំផុត។ វាងាយស្រួលណាស់ក្នុងការអនុវត្ត។

ឧទាហរណ៍ ក្នុង Code ខាងក្រោម URL fetchខុស (គ្មានគេហទំព័របែបនេះទេ) និង .catch ដោះស្រាយកំហុស៖

```
fetch('https://no-such-server.blabla') // rejects
  .then(response => response.json())
  .catch(err => alert(err)) // TypeError: failed to fetch (the text
may vary)
```

ដូចដែលអ្នកអាចមើលឃើញ .catchមិនចាំបាច់ភ្លាមៗនោះទេ។ វាអាចលេចឡើងបន្ទាប់ពីមួយ ឬប្រហែលជាច្រើនដង .then។

ឬ ប្រហែលជាអ្វីៗទាំងអស់គឺត្រឹមត្រូវជាមួយគេហទំព័រ ប៉ុន្តែការឆ្លើយតបគឺមិនត្រឹមត្រូវ JSON ទេ។ មធ្យោបាយងាយស្រួលបំផុតដើម្បីចាប់កំហុសទាំងអស់គឺត្រូវបន្ថែម .catchទៅចុងបញ្ចប់នៃខ្សែសង្វាក់៖

```
fetch('/article/promise-chaining/user.json')
  .then(response => response.json())
  .then(user => fetch(`https://api.github.com/users/${user.name}`))
  .then(response => response.json())
  .then(githubUser => new Promise((resolve, reject) => {
    let img = document.createElement('img');
    img.src = githubUser.avatar_url;
    img.className = "promise-avatar-example";
    document.body.append(img);

    setTimeout(() => {
      img.remove();
      resolve(githubUser);
    }, 3000);
  }))
  .catch(error => alert(error.message));
```

ជាធម្មតា វា .catchមិនបង្កអ្វីទាំងអស់។ ប៉ុន្តែប្រសិនបើការPromiseណាមួយខាងលើបដិសេធ (បញ្ហាបណ្តាញ ឬ json មិនត្រឹមត្រូវ ឬអ្វីក៏ដោយ) នោះវានឹងចាប់វាបាន។

៥.៣.១ Implicit try…catch

The code of a promise ប្រតិបត្តិករ និងអ្នកដោះស្រាយការPromiseមាន "មើលមិនឃើញ try..catch" នៅជុំវិញវា។ ប្រសិនបើករណីលើកលែងកើតឡើង វាត្រូវបានចាប់ និងចាត់ទុកជាការបដិសេធ

ឧទាហរណ៍ Code នេះ៖

```
new Promise((resolve, reject) => {
  throw new Error("Whoops!");
```

```
}).catch(alert); // Error: Whoops!
```

... ដំណើរការដូចគ្នាទៅនឹងនេះ៖

```
new Promise((resolve, reject) => {
  reject(new Error("Whoops!"));
}).catch(alert); // Error: Whoops!
```

"មើលមិនឃើញ try..catch" នៅជុំវិញប្រតិបត្តិករចាប់យកកំហុសដោយស្វ័យប្រវត្តិហើយប្រែទៅជាការPromiseដែលត្រូវបានបដិសេធ។

វាកើតឡើងមិនត្រឹមតែនៅក្នុងមុខងារប្រតិបត្តិប៉ុណ្ណោះទេប៉ុន្តែនៅក្នុងឧបករណ៍ដោះស្រាយរបស់វាផងដែរ។ ប្រសិនបើយើង throwនៅខាងក្នុង .thenអ្នកដោះស្រាយ នោះមានន័យថាជាការPromise ដែលត្រូវបានបដិសេធ ដូច្នេះការគ្រប់គ្រងនឹងលោតទៅអ្នកដោះស្រាយកំហុសដែលនៅជិតបំផុត។

នេះជាឧទាហរណ៍៖

```
new Promise((resolve, reject) => {
  resolve("ok");
}).then((result) => {
  throw new Error("Whoops!"); // rejects the promise
}).catch(alert); // Error: Whoops!
```

វាកើតឡើងចំពោះកំហុសទាំងអស់ មិនមែនគ្រាន់តែបណ្តាលមកពី throwសេចក្តីថ្លែងការណ៍នោះទេ។ ឧទាហរណ៍ កំហុសកម្មវិធី៖

```
new Promise((resolve, reject) => {
  resolve("ok");
}).then((result) => {
  blabla(); // no such function
}).catch(alert); // ReferenceError: blabla is not defined
```

វាគ្គផ្តាច់ព្រ័ត្រ .catchមិនត្រឹមតែចាប់បានការបដិសេធយ៉ាងច្បាស់លាស់ប៉ុណ្ណោះទេ ប៉ុន្តែក៏មានកំហុសដោយចៃដន្យនៅក្នុងអ្នកដោះស្រាយខាងលើផងដែរ។

៥.៣.២ Unhandled rejections

តើមានអ្វីកើតឡើងនៅពេលដែលកំហុសមិនត្រូវបានដោះស្រាយ? ជាឧទាហរណ៍ យើងភ្លេចបន្ថែម .catchទៅចុងបញ្ចប់នៃខ្សែសង្វាក់ ដូចជានៅទីនេះ៖

```
new Promise(function() {
  noSuchFunction(); // Error here (no such function)
})
```

```
  .then(() => {
    // successful promise handlers, one or more
  }); // without .catch at the end!
```

ក្នុងករណីមានកំហុស ការPromiseត្រូវបានបដិសេធ ហើយការប្រតិបត្តិគួរតែទៅអ្នកដោះស្រាយការបដិសេធដែលនៅជិតបំផុត។ ប៉ុន្តែមិនមានទេ។ ដូច្នេះកំហុស "ជាប់គាំង" ។ មិនមានលេខ Code ដើម្បីដោះស្រាយវាទេ។

នៅក្នុងការអនុវត្ត ដូចទៅនឹងកំហុសដែលមិនបានដោះស្រាយជាទៀងទាត់នៅក្នុង Code វាមានន័យថាមានអ្វីមួយខុសឆ្គងយ៉ាងខ្លាំង។

តើមានអ្វីកើតឡើងនៅពេលដែលកំហុសធម្មតាកើតឡើងហើយមិនត្រូវបានចាប់ដោយ try..catch? ស្ត្រីបស្លាប់ជាមួយនឹងសារនៅក្នុងកុងសូល។ រឿងស្រដៀងគ្នានេះកើតឡើងជាមួយនឹងការបដិសេធការPromiseដែលមិនអាចដោះស្រាយបាន។

ម៉ាស៊ីន JavaScript តាមដានការបដិសេធបែបនេះ និងបង្កើតកំហុសសកលនៅក្នុងករណីនោះ។ អ្នកអាចឃើញវានៅក្នុងកុងសូល ប្រសិនបើអ្នកដំណើរការឧទាហរណ៍ខាងលើ។ នៅក្នុងBrowser យើងអាចចាប់កំហុសបែបនេះដោយប្រើព្រឹត្តិការណ៍ unhandledrejection៖

```
window.addEventListener('unhandledrejection', function(event) {
  // the event object has two special properties:
   alert(event.promise); // [object Promise] - the promise that generated the error
  alert(event.reason); // Error: Whoops! - the unhandled error object
});

new Promise(function() {
  throw new Error("Whoops!");
}); // no catch to handle the error
```

ព្រឹត្តិការណ៍គឺជាផ្នែកនៃ ស្តង់ដារ HTML ។ ប្រសិនបើមានកំហុសកើតឡើង ហើយមិនមានទេ .catchអ្នក unhandledrejectionដោះស្រាយនឹងចាប់ផ្តើម ហើយទទួលបាន eventObjectជាមួយនឹងព័ត៌មានអំពីកំហុស ដូច្នេះយើងអាចធ្វើអ្វីមួយបាន។

ជាធម្មតា កំហុសបែបនេះមិនអាចយកមកវិញបានទេ ដូច្នេះវិធីល្អបំផុតរបស់យើងគឺត្រូវជូនដំណឹងដល់អ្នកប្រើប្រាស់អំពីបញ្ហា ហើយប្រហែលជារាយការណ៍អំពីឧបទ្ទវហេតុទៅម៉ាស៊ីនមេ។ នៅក្នុងបរិស្ថានដែលមិនមែនជាកម្មវិធីរុករកដូចជា Node.js មានវិធីផ្សេងទៀតដើម្បីតាមដានកំហុសដែលមិនបានដោះស្រាយ។

៥.៤ Promise API

មានវិធីសាស្ត្របិតិវន្តចំនួន 6 នៅក្នុង PromiseClass។ យើងនឹងគ្របដណ្តប់ករណីប្រើប្រាស់របស់ពួកគេយ៉ាងធាប់រហ័សនៅទីនេះ។

៥.៤.១ Promise.all

ចូរនិយាយថាយើងចង់បានការPromiseជាច្រើនដើម្បីអនុវត្តស្របគ្នា ហើយរង់ចាំរហូតដល់ពួកគេទាំងអស់រួចរាល់។ ឧទាហរណ៍ ទាញយក URLs ជាច្រើនស្របគ្នា ហើយដំណើរការមាតិការនៅពេលពួកវារួចរាល់។នោះហើយជាអ្វី Promise.allដែលសម្រាប់។

រូបមន្ត៖

```
let promise = Promise.all(iterable);
```

Promise.allទទួលយកការPromiseមួយ (ជាធម្មតាArrayនៃការPromise) ហើយត្រឡប់ការPromiseថ្មី។ ការPromiseថ្មីនឹងដោះស្រាយនៅពេលដែលការPromiseដែលបានរាយបញ្ជីទាំងអស់ត្រូវបានដោះស្រាហើយArrayនៃលទ្ធផលរបស់ពួកគេក្លាយជាលទ្ធផលរបស់វា។

ឧទាហរណ៍ Promise.allខាងក្រោមដោះស្រាយបន្ទាប់ពី 3 វិនាទីហើយបន្ទាប់មកលទ្ធផលរបស់វាគឺArrayមួយ [1, 2, 3]៖

```
Promise.all([
  new Promise(resolve => setTimeout(() => resolve(1), 3000)), // 1
  new Promise(resolve => setTimeout(() => resolve(2), 2000)), // 2
  new Promise(resolve => setTimeout(() => resolve(3), 1000))  // 3
]).then(alert); // 1,2,3 when promises are ready: each promise
contributes an array member
```

សូមចំណាំថាលំដាប់នៃសមាជិកArrayលទ្ធផលគឺដូចគ្នាទៅនឹងការPromiseប្រភពរបស់វា។ ទោះបីជាការPromiseដំបូងត្រូវចំណាយពេលយូរបំផុតដើម្បីដោះស្រាយក៏ដោយ វានៅតែជាលទ្ធផលដំបូងនៅក្នុងArrayនៃលទ្ធផល។

ល្បិចធម្មតាមួយគឺត្រូវគូសផែនទីArrayនៃទិន្នន័យការងារទៅក្នុងArrayនៃការPromise ហើយបន្ទាប់មករុំវាទៅក្នុង Promise.all.

ឧទាហរណ៍ ប្រសិនបើយើងមានArrayនៃ URLs យើងអាចយកពួកវាទាំងអស់ដូចនេះ៖

```
let urls = [
  'https://api.github.com/users/iliakan',
  'https://api.github.com/users/remy',
  'https://api.github.com/users/jeresig'
];
```

```
// map every url to the promise of the fetch
let requests = urls.map(url => fetch(url));

// Promise.all waits until all jobs are resolved
Promise.all(requests)
  .then(responses => responses.forEach(
    response => alert(`${response.url} ${response.status}`)
  ));
```

ឧទាហរណ៍ធំជាងជាមួយនឹងការទៅយកព័ត៌មានអ្នកប្រើប្រាស់សម្រាប់Arrayនៃអ្នកប្រើប្រាស់ GitHub តាមឈ្មោះរបស់ពួកគេ (យើងអាចទៅយកArrayនៃទំនិញដោយលេខសម្គាល់របស់ពួកគេ តក្កវិជ្ជាគឺដូចគ្នាបេះបិទ)៖

```
let names = ['iliakan', 'remy', 'jeresig'];

let requests = names.map(name => fetch(`https://api.github.com/users/${name}`));

Promise.all(requests)
  .then(responses => {
    // all responses are resolved successfully
    for(let response of responses) {
      alert(`${response.url} ${response.status}`); // shows 200 for every url
    }

    return responses;
  })
  // map array of responses into an array of response.json() to read their content
  .then(responses => Promise.all(responses.map(r => r.json())))
  // all JSON answers are parsed: "users" is the array of them
  .then(users => users.forEach(user => alert(user.name)));
```

ប្រសិនបើការPromiseណាមួយត្រូវបានបដិសេធ នោះការPromiseដែលត្រឡប់មកវិញដោយ Promise.allបដិសេធភ្លាមៗជាមួយនឹងកំហុសនោះ។

ឧទាហរណ៍៖

```
Promise.all([
  new Promise((resolve, reject) => setTimeout(() => resolve(1), 1000)),
```

```
  new Promise((resolve, reject) => setTimeout(() => reject(new
Error("Whoops!")), 2000)),
  new Promise((resolve, reject) => setTimeout(() => resolve(3), 3000))
]).catch(alert); // Error: Whoops!
```

នៅទីនេះការPromiseទីពីរបដិសេធក្នុងរយៈពេលពីរវិនាទី។ នោះនាំទៅរកការបដិសេធភ្លាមៗ នៃ Promise.all, ដូច្នេះ .catchប្រតិបត្តិ៖ កំហុសការបដិសេធភ្លាយជាលទ្ធផលនៃទាំងមូល Promise.all.

៥.៤.២ Promise.all Settled

Promise.allបដិសេធទាំងស្រុង ប្រសិនបើការPromiseណាមួយបដិសេធ។ នោះជាការល្អ សម្រាប់ករណី "ទាំងអស់ ឬគ្មានអ្វី" នៅពេលដែលយើងត្រូវការ លទ្ធផល ទាំងអស់ ដោយជោគជ័យដើម្បីបន្ត៖

```
Promise.all([
  fetch('/template.html'),
  fetch('/style.css'),
  fetch('/data.json')
]).then(render); // render method needs results of all fetches
```

Promise.allSettledគ្រាន់តែរង់ចាំការPromiseទាំងអស់ដើម្បីដោះស្រាយដោយមិនគិតពីលទ្ធផល។ Arrayលទ្ធផលមាន៖

- {status:"fulfilled", value:result}សម្រាប់ការឆ្លើយតបជោគជ័យ,
- {status:"rejected", reason:error}សម្រាប់កំហុស។

ឧទាហរណ៍ យើងចង់ទៅយកព័ត៌មានអំពីអ្នកប្រើប្រាស់ច្រើន។ ទោះបីជាសំណើមួយបរាជ័យក៏ដោយ យើងនៅតែចាប់អារម្មណ៍ចំពោះសំណើផ្សេងទៀត។

តោះប្រើ Promise.allSettled៖

```
let urls = [
  'https://api.github.com/users/iliakan',
  'https://api.github.com/users/remy',
  'https://no-such-url'
];
```

```
Promise.allSettled(urls.map(url => fetch(url)))
  .then(results => { // (*)
    results.forEach((result, num) => {
      if (result.status == "fulfilled") {
        alert(`${urls[num]}: ${result.value.status}`);
      }
      if (result.status == "rejected") {
        alert(`${urls[num]}: ${result.reason}`);
      }
    });
  });
```

នៅក្នុង resultsបន្ទាត់ (*)ខាងលើនឹងមាន៖

```
[
  {status: 'fulfilled', value: ...response...},
  {status: 'fulfilled', value: ...response...},
  {status: 'rejected', reason: ...error object...}
]
```

ដូច្នេះសម្រាប់ការPromiseនីមួយៗ យើងទទួលបានហាន: និង value/error.

ក. Polyfill

ប្រសិនបើកម្មវិធីរុករកមិនគាំទ្រ Promise.allSettledវាងាយស្រួលក្នុងការបំពេញបន្ថែម៖

```
if (!Promise.allSettled) {
  const rejectHandler = reason => ({ status: 'rejected', reason });

  const resolveHandler = value => ({ status: 'fulfilled', value });

  Promise.allSettled = function (promises) {
          const convertedPromises = promises.map(p => Promise.resolve(p).then(resolveHandler, rejectHandler));
    return Promise.all(convertedPromises);
  };
}
```

ក្នុង Code នេះ promises.mapយកតម្លៃបញ្ចូល បង្វែរវាទៅជាការPromise (គ្រាន់តែក្នុងករណីដែលការមិនPromiseត្រូវបានឆ្លងកាត់) ជាមួយនឹង p => Promise.resolve(p), ហើយបន្ទាប់មកបន្ថែម .thenអ្នកដោះស្រាយទៅគ្រប់គ្នា។

អ្នកដោះស្រាយនោះប្រែលទ្ធផលជោគជ័យ valueទៅជា {status:'fulfilled', value}, និង កំហុស reasonទៅជា {status:'rejected', reason}. នោះហើយជាទម្រង់នៃ Promise.allSettled.

ឥឡូវនេះយើងអាចប្រើ Promise.allSettledដើម្បីទទួលបានលទ្ធផលនៃ ការPromiseដែល បានផ្តល់ឱ្យ ទាំងអស់ ទោះបីជាពួកគេមួយចំនួនបដិសេធក៏ដោយ។

៥.៤.៣ Promise.race

ស្រដៀងនឹង Promise.all, ប៉ុន្តែរង់ចាំតែការPromiseដែលបានដោះស្រាយលើកដំបូង និង ទទួលបានលទ្ធផលរបស់វា (ឬកំហុស)។

រូបមន្ត៖

```
let promise = Promise.race(iterable);
```

ឧទាហរណ៍នៅទីនេះលទ្ធផលនឹង 1មាន:

```
Promise.race([
   new Promise((resolve, reject) => setTimeout(() => resolve(1),
1000)),
   new Promise((resolve, reject) => setTimeout(() => reject(new
Error("Whoops!")), 2000)),
  new Promise((resolve, reject) => setTimeout(() => resolve(3), 3000))
]).then(alert); // 1
```

ការPromiseដំបូងនៅទីនេះគឺលឿនបំផុត ដូច្នេះវាបានក្លាយជាលទ្ធផល។ បន្ទាប់ពីការPromise ដែលបានដោះស្រាយលើកដំបូង "ឈ្នះការប្រណាំង" លទ្ធផល / កំហុសបន្ថែមទៀតទាំងអស់មិនត្រូវ បានអើពើ

៥.៤.៤ Promise.any

ស្រដៀងនឹង Promise.raceប៉ុន្តែរង់ចាំតែការPromiseដែលបានបំពេញដំបូង និងទទួលបាន លទ្ធផលរបស់វា។ ប្រសិនបើការPromiseដែលបានផ្តល់ឱ្យទាំងអស់ត្រូវបានច្រានចោល នោះការ Promiseដែលបានប្រគល់មកវិញត្រូវបានបដិសេធជាមួយនឹង AggregateError- Objectដែលមាន កំហុសពិសេសដែលរក្សាទុករាល់កំហុសPromiseនៅក្នុង errorsProperty របស់វា។

រូបមន្ត៖

```
let promise = Promise.any(iterable);
```

ឧទាហរណ៍នៅទីនេះលទ្ធផលនឹង 1មាន:

```
Promise.any([
   new Promise((resolve, reject) => setTimeout(() => reject(new
Error("Whoops!")), 1000)),
```

```
  new Promise((resolve, reject) => setTimeout(() => resolve(1), 2000)),
  new Promise((resolve, reject) => setTimeout(() => resolve(3), 3000))
]).then(alert); // 1
```

ការPromiseទីមួយនៅទីនេះលឿនបំផុត ប៉ុន្តែវាត្រូវបានបដិសេធ ដូច្នេះការPromiseទីពីរបានក្លាយជាលទ្ធផល។ បន្ទាប់ពីការPromiseដែលបានបំពេញជាលើកដំបូង "ឈ្នះការប្រណាំង" លទ្ធផលបន្ថែមទៀតទាំងអស់មិនត្រូវបានអើពើ។

នេះជាឧទាហរណ៍មួយ នៅពេលដែលការPromiseទាំងអស់បរាជ័យ៖

```
Promise.any([
  new Promise((resolve, reject) => setTimeout(() => reject(new Error("Ouch!")), 1000)),
  new Promise((resolve, reject) => setTimeout(() => reject(new Error("Error!")), 2000))
]).catch(error => {
  console.log(error.constructor.name); // AggregateError
  console.log(error.errors[0]); // Error: Ouch!
  console.log(error.errors[1]); // Error: Error!
});
```

ដូចដែលអ្នកអាចឃើញObjectកំហុសសម្រាប់ការPromiseដែលបរាជ័យមាននៅក្នុង errors លក្ខណសម្បត្តិរបស់ AggregateErrorObject។

៥.៤.៥ Promise.resolve/reject

វិធីសាស្ត្រ Promise.resolveនិង Promise.rejectកម្រត្រូវការជាចាំបាច់ក្នុង Code ទំនើបពីព្រោះ async/awaitរួបមន្ត (យើងនឹងគ្របដណ្តប់វា នៅពេលក្រោយ) ធ្វើឱ្យពួកវាលែងប្រើបន្តិច។ យើងគ្របដណ្តប់វានៅទីនេះសម្រាប់ភាពពេញលេញ និងសម្រាប់អ្នកដែលមិនអាចប្រើប្រាស់ async/awaitដោយហេតុផលមួយចំនួន។

ក. Promise.resolve

Promise.resolve(value)បង្កើតការPromiseដែលបានដោះស្រាយជាមួយនឹងលទ្ធផល value ដូចគ្នានឹង៖

```
let promise = new Promise(resolve => resolve(value));
```

វិធីសាស្ត្រត្រូវបានប្រើសម្រាប់compatibility។ នៅពេលដែលមុខងារមួយត្រូវបានគេរំពឹងថានឹងត្រលប់មកវិញនូវការPromise។

ឧទាហរណ៍ loadCachedមុខងារខាងក្រោមទាញយក URL ហើយចងចាំ (ឃ្លាំងសម្ងាត់) មាតិការបស់វា។ សម្រាប់ការហៅទូរសព្ទនាពេលខាងមុខជាមួយនឹង URL ដូចគ្នា វាទទួលបានមាតិកាពីមុនភ្លាមៗពីឃ្លាំងសម្ងាត់ ប៉ុន្តែប្រើ Promise.resolveដើម្បីធ្វើការPromiseរបស់វា ដូច្នេះតម្លៃដែលបានត្រឡប់មកវិញគឺតែងតែជាការPromise៖

```
let cache = new Map();

function loadCached(url) {
  if (cache.has(url)) {
    return Promise.resolve(cache.get(url)); // (*)
  }

  return fetch(url)
    .then(response => response.text())
    .then(text => {
      cache.set(url,text);
      return text;
    });
}
```

យើងអាចសរសេរបាន loadCached(url).then(…)ព្រោះមុខងារត្រូវបានធានាថានឹងត្រឡប់ការPromiseវិញ។ យើងតែងតែអាចប្រើ .thenបន្ទាប់ពី loadCached។ នោះហើយជាគោលបំណងនៃ Promise.resolveបន្ទាត់ (*)។

ខ. Promise.reject

Promise.reject(error)បង្កើតការPromiseបដិសេធជាមួយ error។ ដូចគ្នានឹង៖

```
let promise = new Promise((resolve, reject) => reject(error));
```

នៅក្នុងការអនុវត្តវិធីសាស្ត្រនេះស្ទើរតែមិនដែលប្រើ។

៥.៥ ការបំលែងទៅជា Promise (Promisification)

"Promisification" គឺជាពាក្យដ៏វែងសម្រាប់ការផ្លាស់ប្តូរដ៏សាមញ្ញមួយ។ វាជាការបំប្លែងមុខងារដែលទទួលការហៅត្រឡប់ទៅជាមុខងារដែលត្រឡប់ការPromise។ ការបំប្លែងបែបនេះច្រើនតែទាមទារនៅក្នុងជីវិតពិត ព្រោះមុខងារ និងបណ្ណាល័យជាច្រើនមានមូលដ្ឋានលើការហៅត្រឡប់មកវិញ។ ប៉ុន្តែការPromiseមានភាពងាយស្រួលជាង ដូច្នេះវាសមហេតុផលក្នុងការPromiseពួកគេ។ ដើម្បីយល់កាន់តែច្បាស់ សូមមើលឧទាហរណ៍មួយ។

ឧទាហរណ៍ យើងមាន loadScript(src, callback)ពីជំពូក សេចក្តីណែនាំ៖ ការហៅត្រឡប់វិញ

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;

  script.onload = () => callback(null, script);
  script.onerror = () => callback(new Error(`Script load error for
${src}`));

  document.head.append(script);
}

// usage:
// loadScript('path/script.js', (err, script) => {...})
```

មុខងារផ្ទុកស្គ្រីបជាមួយនឹងអ្វីដែលបានផ្តល់ឱ្យ srcហើយបន្ទាប់មកហៅទៅ callback(err)ក្នុងករណីមានកំហុស ឬ callback(null, script)ក្នុងករណីនៃការផ្ទុកជោគជ័យ។ នោះជាកិច្ចព្រមព្រៀងដ៏ទូលំទូលាយសម្រាប់ការប្រើប្រាស់ការហៅត្រឡប់មកវិញ យើងបានឃើញវាពីមុនមក។

យើងនឹងបង្កើតមុខងារថ្មីមួយ loadScriptPromise(src)ដែលធ្វើដូចគ្នា (ផ្ទុកស្គ្រីប) ប៉ុន្តែត្រឡប់ការPromiseជំនួសឱ្យការប្រើការហៅត្រឡប់មកវិញ។

ម្យ៉ាងវិញទៀត យើងឆ្លងកាត់វាតែប៉ុណ្ណោះ src(ទេ callback) ហើយទទួលបានការPromiseជាថ្នូរនឹងការដោះស្រាយ scriptនៅពេលដែលការផ្ទុកបានជោគជ័យ ហើយបដិសេធជាមួយនឹងកំហុសបើមិនដូច្នេះទេ។

នេះគឺ៖

```
let loadScriptPromise = function(src) {
  return new Promise((resolve, reject) => {
    loadScript(src, (err, script) => {
      if (err) reject(err);
      else resolve(script);
    });
  });
};

// usage:
// loadScriptPromise('path/script.js').then(...)
```

ដូចដែលយើងអាចឃើញមុខងារថ្មីគឺរុំជុំវិញ loadScriptមុខងារដើម។ វាហៅវាថាផ្តល់ការហៅត្រឡប់របស់វាផ្ទាល់ដែលបកប្រែទៅនឹងការPromise resolve/reject។ ឥឡូវនេះ loadScriptPromise សមល្អនៅក្នុង Code ផ្អែកលើការPromise។ ប្រសិនបើយើងចូលចិត្តការPromiseច្រើនជាងការហៅត្រ

ឡប់មកវិញ (ហើយឆាប់ៗនេះយើងនឹងឃើញហេតុផលបន្ថែមទៀតសម្រាប់រឿងនោះ) នោះយើងនឹងប្រើវាជំនួសវិញ។

នៅក្នុងការអនុវត្ត យើងប្រហែលជាត្រូវPromiseជាមួយមុខងារច្រើនជាងមួយ ដូច្នេះវាសមហេតុផលក្នុងការប្រើជំនួយការ។ យើងនឹងហៅវាថា promisify(f)៖ វាទទួលយកមុខងារPromise ។និងត្រឡប់ wrapper function។

```
function promisify(f) {
  return function (...args) { // return a wrapper-function (*)
    return new Promise((resolve, reject) => {
      function callback(err, result) { // our custom callback for f
(**)
        if (err) {
          reject(err);
        } else {
          resolve(result);
        }
      }

      args.push(callback); // append our custom callback to the end
of f arguments

      f.call(this, ...args); // call the original function
    });
  };
}

// usage:
let loadScriptPromise = promisify(loadScript);
loadScriptPromise(...).then(...);
```

Code អាចមើលទៅស្មុគស្មាញបន្តិច ប៉ុន្តែវាសំខាន់ដូចគ្នាទៅនឹងអ្វីដែលយើងបានសរសេរខាងលើ ខណៈពេលដែល loadScriptមុខងារPromise។ ការហៅឱ្យ promisify(f)ត្រឡប់រុំជុំវិញ f (*)។ កញ្ចប់នោះត្រឡប់ការPromise ហើយបញ្ជូនបន្តការហៅទៅដើម ។ដោយតាមដានលទ្ធផលនៅក្នុងការហៅត្រឡប់មកវិញផ្ទាល់ខ្លួន (**)។

នៅទីនេះ promisifyសន្មត់ថាមុខងារដើមរំពឹងការហៅត្រឡប់មកវិញជាមួយនឹងArgumentពីរយ៉ាងពិតប្រាកដ (err, result)។ នោះហើយជាអ្វីដែលយើងជួបប្រទះញឹកញាប់បំផុត។ បន្ទាប់មកការហៅត្រឡប់មកវិញផ្ទាល់ខ្លួនរបស់យើងគឺពិតជាទម្រង់ត្រឹមត្រូវ ហើយ promisifyដំណើរការល្អសម្រាប់

ករណីបែបនេះ។ ប៉ុន្តែចុះយ៉ាងណាបើដើម អំពីងថានឹងមានការហៅត្រឡប់មកវិញជាមួយនឹង Argumentបន្ថែមទៀត callback(err, res1, res2, ...)?

យើងអាចកែលម្អជំនួយរបស់យើង។ តោះបង្កើតកំណែកម្រិតខ្ពស់ជាងនេះ promisify។

- នៅពេលដែលគេហៅថា promisify(f)វាគួរតែដំណើរការស្រដៀងនឹងកំណែខាងលើ។
- នៅពេលហៅជា promisify(f, true)វាគួរតែត្រឡប់ការPromiseដែលដោះស្រាយជាមួយនឹង Arrayនៃលទ្ធផលហៅត្រឡប់មកវិញ។ នោះពិតជាសម្រាប់ការហៅត្រឡប់មកវិញជាមួយនឹង Argumentជាច្រើន។

```
// promisify(f, true) to get array of results
function promisify(f, manyArgs = false) {
  return function (...args) {
    return new Promise((resolve, reject) => {
      function callback(err, ...results) { // our custom callback for
f
        if (err) {
          reject(err);
        } else {
         // resolve with all callback results if manyArgs is specified
          resolve(manyArgs ? results : results[0]);

        }
      }

      args.push(callback);

      f.call(this, ...args);
    });
  };
}

// usage:
f = promisify(f, true);
f(...).then(arrayOfResults => ..., err => ...);
```

ដូចដែលអ្នកអាចឃើញវាសំខាន់ដូចគ្នានឹងខាងលើដែរ ប៉ុន្តែ resolveត្រូវបានហៅដោយអំណះអំណាងតែមួយ ឬទាំងអស់អាស្រ័យលើថាតើ manyArgsជាការពិត។

សម្រាប់ទម្រង់ហៅត្រឡប់មកវិញកម្រនិងអសកម្មជាច្រើនទៀត ដូចជាទម្រង់ដែលមិនមាន err អ្វីទាំងអស់៖ callback(result)យើងអាចPromiseមុខងារបែបនេះដោយដៃដោយមិនប្រើជំនួយ។ វាក៏

មានម៉ូឌុលដែលមានមុខងារPromiseដែលអាចបត់បែនបានបន្តិច ឧទាហរណ៍ es6-promisify ។ នៅក្នុង Node.js មាន util.promisifyមុខងារភ្ជាប់មកជាមួយសម្រាប់នោះ។

៥.៦ Microtasks

អ្នក ដោះស្រាយការPromise .then/.catch/.finally តែងតែអសមកាល។ ទោះបីជានៅពេលដែលការPromiseត្រូវបានដោះស្រាយភ្លាមៗក៏ដោយ Code នៅលើបន្ទាត់ ខាងក្រោម .then/.catch/.finally នៅតែ ប្រតិបត្តិមុនពេលអ្នកដោះស្រាយទាំងនេះ។

នេះជាការបង្ហាញ៖

```
let promise = Promise.resolve();

promise.then(() => alert("promise done!"));

alert("code finished"); // this alert shows first
```

ប្រសិនបើអ្នកដំណើរការវា អ្នកឃើញ code finishedមុន ហើយបន្ទាប់មក promise done!។ នោះជារឿងចម្លែកព្រោះពាក្យPromiseច្បាស់ជាធ្វើតាំងពីដើមមក។ ហេតុអ្វីបានជា .then trigger បន្ទាប់មក ? តើមានអ្វីកើតឡើង ?

៥.៦.១ Microtasks queue

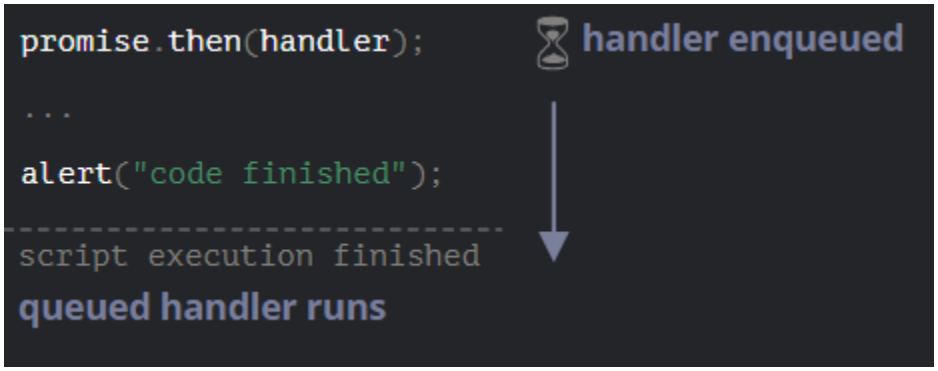
Asynchronous tasks ត្រូវការការគ្រប់គ្រងត្រឹមត្រូវ។ សម្រាប់នោះ ស្តង់ដារ ECMA បញ្ជាក់ជួរ ខាងក្នុង PromiseJobsដែលជារឿយៗគេហៅថា “microtask queue” (V8 term) ។

ដូចដែលបានបញ្ជាក់នៅក្នុង ការបញ្ជាក់ ៖

- ជួរគឺចេញមុនគេ៖ កិច្ចការដែលដាក់ជាជួរមុនគេត្រូវបាន run មុនគេ។
- ការអនុវត្តការកិច្ចត្រូវបានផ្តួចផ្តើមតែនៅពេលដែលមិនមានអ្វីផ្សេងទៀតកំពុងដំណើរការ។

និយាយឱ្យសាមញ្ញជាងនេះទៅទៀត នៅពេលដែលការPromiseរួចរាល់ .then/.catch/.finally អ្នកចាត់ការរបស់វាត្រូវបានដាក់ចូលទៅក្នុងជួរ។ ពួកគេមិនទាន់ត្រូវបានគេសម្លាប់នៅឡើយទេ។ នៅពេលដែលម៉ាស៊ីន JavaScript ទំនេរពី Code បច្ចុប្បន្ន វាត្រូវការកិច្ចការពីជួរ ហើយប្រតិបត្តិវា។

នោះហើយជាមូលហេតុដែល " Code បញ្ចប់" នៅក្នុងឧទាហរណ៍ខាងលើបង្ហាញមុន។



រូបភាព 14៖ Microtasks queue

អ្នកដោះស្រាយការPromiseតែងតែឆ្លងកាត់ជួរខាងក្នុងនេះ។ ប្រសិនបើមានខ្សែសង្វាក់ច្រើន .then/.catch/.finallyនោះពួកវានីមួយៗត្រូវបានប្រតិបត្តិដោយអសមកាល។ នោះគឺជាដំបូងវាត្រូវបានដាក់ជាជួរ បន្ទាប់មកប្រតិបត្តិនៅពេលដែល Code បច្ចុប្បន្នត្រូវបានបញ្ចប់ ហើយអ្នកដោះស្រាយដែលបានដាក់ជួរពីមុនត្រូវបានបញ្ចប់។

ចុះបើការបញ្ជាទិញសំខាន់សម្រាប់យើង? តើយើងអាច code finishedលេចមុខ ដោយរបៀបណា promise done? ងាយៗ គ្រាន់តែដាក់វាចូលទៅក្នុងជួរជាមួយ .then៖

```
Promise.resolve()
  .then(() => alert("promise done!"))
  .then(() => alert("code finished"));
```

៥.៦.២ Unhandled rejection

ចងចាំ unhandledrejectionព្រឹត្តិការណ៍ពីអត្ថបទការដោះស្រាយកំហុសជាមួយនឹងការPromise? ឥឡូវនេះយើងអាចមើលឃើញយ៉ាងច្បាស់អំពីរបៀបដែល JavaScript រកឃើញថាមានការបដិសេធដែលមិនអាចគ្រប់គ្រងបាន។ "ការបដិសេធដែលមិនបានដោះស្រាយ" កើតឡើងនៅពេលដែលកំហុសការPromiseមិនត្រូវបានដោះស្រាយនៅចុងបញ្ចប់នៃជួរ microtask ។

ជាធម្មតា ប្រសិនបើយើងរំពឹងថានឹងមានកំហុស យើងបន្ថែម .catchទៅខ្សែសង្វាក់Promise ដើម្បីដោះស្រាយវា៖

```
let promise = Promise.reject(new Error("Promise Failed!"));
promise.catch(err => alert('caught'));

// doesn't run: error handled
window.addEventListener('unhandledrejection', event => alert(event.reason));
```

ប៉ុន្តែប្រសិនបើយើងភ្លេចបន្ថែម .catchនោះបន្ទាប់ពីជួរ microtask គឺទទេ ម៉ាស៊ីននឹងបង្កព្រឹត្តិការណ៍៖

```
let promise = Promise.reject(new Error("Promise Failed!"));

// Promise Failed!
window.addEventListener('unhandledrejection',        event        =>
alert(event.reason));
```

ចុះបើយើងដោះស្រាយកំហុសនៅពេលក្រោយ? ដូចនេះ៖

```
let promise = Promise.reject(new Error("Promise Failed!"));
setTimeout(() => promise.catch(err => alert('caught')), 1000);

// Error: Promise Failed!
window.addEventListener('unhandledrejection',        event        =>
alert(event.reason));
```

ឥឡូវនេះ ប្រសិនបើយើងដំណើរការវា យើងនឹងឃើញ Promise Failed!មុនគេ ហើយបន្ទាប់មក caught។ ប្រសិនបើយើងមិនបានដឹងអំពីជួរ microtasks នោះយើងអាចធ្វល់ថា "ហេតុអ្វីបានជា unhandledrejectionhandler វត់? យើងបានចាប់និងដោះស្រាយកំហុស!"

ប៉ុន្តែឥឡូវនេះយើងយល់ថាវា unhandledrejectionត្រូវបានបង្កើតនៅពេលដែលជួរ microtask បានបញ្ចប់៖ ម៉ាស៊ីនពិនិត្យការPromise ហើយប្រសិនបើពួកគេណាមួយស្ថិតក្នុងស្ថានភាព "បដិសេធ" នោះព្រឹត្តិការណ៍នឹងកើតឡើង។ នៅក្នុងឧទាហរណ៍ខាងលើ .catchបន្ថែមដោយ setTimeouttriggers ផងដែរ។ ប៉ុន្តែវាកើតឡើងនៅពេលក្រោយ បន្ទាប់ពី unhandledrejectionបានកើតឡើងរួចហើយ ដូច្នេះវាមិនមានអ្វីផ្លាស់ប្តូរទេ។

៥.៧ Async/Await

មាន រូបមន្ត ពិសេសមួយដើម្បីធ្វើការជាមួយការPromiseក្នុងទម្រង់ដែលងាយស្រួលជាង ហៅថា "async/await"។ វាគួរឱ្យភ្ញាក់ផ្អើលងាយស្រួលក្នុងការយល់និងប្រើ។

៥.៧.១ អនុគមន៍ Async

ចូរចាប់ផ្តើមជាមួយ asyncពាក្យគន្លឹះ។ វាអាចត្រូវបានដាក់នៅមុខមុខងារដូចនេះ៖

```
async function f() {
  return 1;
}
```

ពាក្យ "async" មុនពេលអនុគមន៍មានន័យថារឿងសាមញ្ញមួយ៖ មុខងារតែងតែត្រឡប់ promise។ តម្លៃផ្សេងទៀតត្រូវបានរុំដោយការPromiseដែលបានដោះស្រាយដោយស្វ័យប្រវត្តិ។

ឧទាហរណ៍ មុខងារនេះត្រឡប់ការPromiseដែលបានដោះស្រាយជាមួយនឹងលទ្ធផលនៃ 1; តោះសាកល្បង៖

```
async function f() {
  return 1;
}

f().then(alert); // 1
```

... យើងអាចត្រឡប់ការPromiseវិញយ៉ាងច្បាស់ ដែលនឹងដូចគ្នា៖

```
async function f() {
  return Promise.resolve(1);
}

f().then(alert); // 1
```

ដូច្នេះ asyncជានាថាមុខងារត្រឡប់ការPromise ហើយបិទការមិនPromiseនៅក្នុងវា។ សាមញ្ញគ្រប់គ្រាន់ហើយមែនទេ? ប៉ុន្តែមិនត្រឹមតែប៉ុណ្ណឹងទេ។ មានពាក្យគន្លឹះមួយទៀត awaitដែលដំណើរការតែក្នុង asyncមុខងារប៉ុណ្ណោះ ហើយវាពិតជាឡូយណាស់។

៥.៧.២ Await

រូបមន្ត៖

```
// works only inside async functions
let value = await promise;
```

ពាក្យគន្លឹះ awaitធ្វើឱ្យ JavaScript រង់ចាំរហូតដល់ការPromiseនោះដោះស្រាយ ហើយត្រឡប់លទ្ធផលរបស់វា។

នេះជាឧទាហរណ៍ជាមួយការPromiseដែលដោះស្រាយក្នុងរយៈពេល 1 វិនាទី៖

```
async function f() {

  let promise = new Promise((resolve, reject) => {
    setTimeout(() => resolve("done!"), 1000)
  });

  let result = await promise; // wait until the promise resolves (*)

  alert(result); // "done!"
}

f();
```

ការប្រតិបត្តិមុខងារ "pauses" នៅបន្ទាត់ (*)ហើយបន្តនៅពេលដែលការPromiseបានដោះស្រាយ ដោយ resultក្លាយជាលទ្ធផលរបស់វា។ ដូច្នេះលេខ Code ខាងលើបង្ហាញថា "រួចរាល់!" ក្នុងមួយវិនាទី។

សូមបញ្ជាក់៖ awaitផ្អាកការប្រតិបត្តិមុខងារតាមព្យញ្ជនៈ រហូតដល់ការPromiseដោះស្រាយ ហើយបន្ទាប់មកបន្តវាឡើងវិញជាមួយនឹងលទ្ធផលPromise។ វាមិនចំណាយធនធាន CPU ទេ ព្រោះម៉ាស៊ីន JavaScript អាចធ្វើការងារផ្សេងទៀតក្នុងពេលនេះ៖ ប្រតិបត្តិស្គ្រីបផ្សេងៗ ដោះស្រាយព្រឹត្តិការណ៍។ល។

វាគ្រាន់តែជារូបមន្តដ៏ធើតឆាយនៃការទទួលបានលទ្ធផលPromiseជាង promise.then. ហើយវាកាន់តែងាយស្រួលអាន និងសរសេរ។

- មិនអាចប្រើ awaitក្នុងមុខងារធម្មតាបាន ទេ។

ប្រសិនបើយើងព្យាយាមប្រើ awaitក្នុងមុខងារមិនធ្វើសមកាលកម្ម វានឹងមានកំហុសរូបមន្ត៖

```
function f() {
  let promise = Promise.resolve(1);
  let result = await promise; // រូបមន្ត error
}
```

យើងអាចនឹងទទួលបានកំហុសនេះ ប្រសិនបើយើងភ្លេចដាក់ asyncមុនមុខងារ។ ដូចដែលបានបញ្ជាក់ពីមុន awaitដំណើរការតែនៅក្នុង asyncមុខងារមួយ។

ចូរយក showAvatar()ឧទាហរណ៍ពីជំពូក Promises chaining ហើយសរសេរវាឡើងវិញដោយប្រើ async/await៖

1. យើងនឹងត្រូវដំនួស .thenការហៅទូរសព្ទ awaitវិញ។
2. យើងក៏គួរតែបង្កើតមុខងារ asyncឱ្យពួកគេដំណើរការដែរ។

```
async function showAvatar() {

  // read our JSON
  let response = await fetch('/article/promise-chaining/user.json');
  let user = await response.json();

  // read github user
  let githubResponse = await fetch(`https://api.github.com/users/${user.name}`);
  let githubUser = await githubResponse.json();

  // show the avatar
```

```
  let img = document.createElement('img');
  img.src = githubUser.avatar_url;
  img.className = "promise-avatar-example";
  document.body.append(img);

  // wait 3 seconds
  await new Promise((resolve, reject) => setTimeout(resolve, 3000));

  img.remove();

  return githubUser;
}

showAvatar();
```

ស្អាតហើយស្រួលអានណាស់មែនទេ ? ប្រសើរជាងមុនច្រើន។

៥.៧.៣ ការគ្រប់គ្រងកំហុស

ប្រសិនបើការPromiseដោះស្រាយជាធម្មតា នោះ await promiseលទ្ធផលនឹងត្រលប់មកវិញ។ ប៉ុន្តែក្នុងករណីបដិសេធ វាបោះចោលកំហុស ដូចជាមាន throwសេចក្តីថ្លែងការណ៍នៅបន្ទាត់នោះ។

Code នេះ៖

```
async function f() {
  await Promise.reject(new Error("Whoops!"));
}
```

...គឺដូចគ្នានេះ៖

```
async function f() {
  throw new Error("Whoops!");
}
```

ក្នុងស្ថានភាពជាក់ស្តែង ការPromiseអាចចំណាយពេលខ្លះ មុនពេលវាបដិសេធ។ ក្នុងករណីនេះវានឹងមានការពន្យាពេលមុន awaitពេលបោះកំហុស។ យើងអាចចាប់កំហុសនោះដោយប្រើ try..catchវិធីដូចគ្នានឹងការធម្មតាដែរ throw៖

```
async function f() {

  try {
    let response = await fetch('http://no-such-url');
  } catch(err) {
```

```
    alert(err); // TypeError: failed to fetch
  }
}

f();
```

ក្នុងករណីមានកំហុស Objectបញ្ជូនលោតទៅ catchប្លុក។ យើងក៏អាចរុំបន្ទាត់ជាច្រើន៖

```
async function f() {

  try {
    let response = await fetch('/no-user-here');
    let user = await response.json();
  } catch(err) {
    // catches errors both in fetch and response.json
    alert(err);
  }
}

f();
```

ប្រសិនបើយើងមិនមាន try..catchនោះការPromiseដែលបង្កើតដោយការហៅរបស់មុខងារ async f( )នឹងបដិសេធ។ យើងអាចបន្ថែម .catchដើម្បីដោះស្រាយវា៖

```
async function f() {
  let response = await fetch('http://no-such-url');
}

// f() becomes a rejected promise
f().catch(alert); // TypeError: failed to fetch // (*)
```

ប្រសិនបើយើងភ្លេចបន្ថែម .catchនៅទីនោះ នោះយើងទទួលបានកំហុសPromiseដែលមិនបានដោះស្រាយ (អាចមើលបានក្នុងកុងសូល)។ យើងអាចចាប់កំហុសបែបនេះដោយប្រើ unhandledrejectionកម្មវិធីដោះស្រាយព្រឹត្តិការណ៍សកលដូចដែលបានពិពណ៌នាក្នុងជំពូក ការដោះស្រាយកំហុសជាមួយនឹងការPromise ។

- async/awaitនិងpromise.then/catch

ពេលយើងប្រើ async/awaitយើងកម្រត្រូវការណាស់ .thenព្រោះ awaitដោះស្រាយការរង់ចាំយើង។ ហើយយើងអាចប្រើធម្មតា try..catchជំនួសឱ្យ .catch. នោះជាធម្មតា (ប៉ុន្តែមិនតែងតែ) ងាយស្រួលជាង។

ប៉ុន្តែនៅកម្រិតកំពូលនៃ Code នៅពេលដែលយើងនៅខាងក្រៅ asyncមុខងារណាមួយ យើងមិនអាចប្រើវាបានទេ awaitដូច្នេះវាជាការអនុវត្តធម្មតាក្នុងការបន្ថែម .then/catchដើម្បីដោះស្រាយលទ្ធផលចុងក្រោយ ឬកំហុសធ្លាក់ចូលដូចក្នុងឧទាហរណ៍ (*)ខាងលើ។

- async/awaitដំណើរការល្អជាមួយPromise.all

នៅពេលដែលយើងត្រូវរង់ចាំការPromiseជាច្រើន យើងអាចបញ្ចប់វា Promise.allហើយបន្ទាប់មក await៖

```
// wait for the array of results
let results = await Promise.all([
  fetch(url1),
  fetch(url2),
  ...
]);
```

ក្នុងករណីមានកំហុស វាបន្តផ្សាយដូចធម្មតា ពីការPromiseដែលបរាជ័យ Promise.allហើយបន្ទាប់មកក្លាយជាករណីលើកលែងដែលយើងអាចចាប់បានដោយប្រើ try..catchជុំវិញការហៅទូរសព្ទ។

៥.៨ Fetch API

Fetch API ផ្តល់នូវវិធីទំនើប និងអាចបត់បែនបានក្នុងការ network requests ។ វាត្រូវបានបង្កើតឡើងដោយ promises និងអនុញ្ញាតឱ្យអ្នកដោះស្រាយការឆ្លើយតបកាន់តែងាយស្រួលបើប្រៀបធៀបទៅនឹងវិធីសាស្ត្រចាស់ៗដូចជា XMLHttpRequest ជាដើម។

៥.៨.១ ការធ្វើសំណើ

មុខងារ fetch( ) ចាប់ផ្តើមសំណើទៅកាន់ URL ដែលបានបញ្ជាក់ ហើយ returns a promise ដែលដោះស្រាយចំពោះ Response object ដែលតំណាងឱ្យការឆ្លើយតបទៅនឹងសំណើ។

ឧទាហរណ៍៖

```
fetch('https://api.example.com/data')

    .then(response => response.json())

    .then(data => console.log(data))

    .catch(error => console.error('Error:', error))
```

៥.៨.២ ការគ្រប់គ្រងការឆ្លើយតប

Response object រួមបញ្ចូលវិធីសាស្ត្រដូចជា .json( ), .text( ), និង .blob( ) ដើម្បីគ្រប់គ្រងប្រភេទទិន្នន័យផ្សេងៗគ្នា។

ឧទាហរណ៍៖

```
async function getData() {
  try {
      const response = await fetch('https://api.example.com/data');
      if (!response.ok) {
          throw new Error('Network response was not ok');
      }
      const data = await response.json();
      console.log(data);
  } catch (error) {
      console.error('Error:', error);
  }
}
getData();
```

៥.៨.៣ ជម្រើសនៃសំណើ

មុខងារ fetch( ) ក៏គាំទ្រជម្រើសផ្សេងៗផងដែរ ដូចជា HTTP methods, headers, and body content សម្រាប់សំណើស្មុគស្មាញជាងនេះ។

ឧទាហរណ៍៖

```
fetch('https://api.example.com/submit', {
  method: 'POST',
  headers: {
```

```
        'Content-Type': 'application/json'
    },
    body: JSON.stringify({ key: 'value' })
})
    .then(response => response.json())
    .then(data => console.log(data))
    .catch(error => console.error('Error:', error));
```

៥.៩ លំហាត់

1. ពន្យល់អំពី event loop នៅក្នុង JavaScript ហើយបង្ហាញឧទាហរណ៍នៃ asynchronous operation ដែល event loop ដោះស្រាយ។
2. អត្ថប្រយោជន៍នៃ promises ធៀបនឹង callback functions គឺអ្វីខ្លះ?
3. async/await និង. then( ).catch( ) មានភាពខុសគ្នាអ្វីខ្លះ?
4. ពន្យល់អំពីចំណុចខ្សោយនៃ Fetch API និងតើមានវិធីណាដើម្បីកែសំរួលវា?
5. អត្ថបទខាងក្រោមមានបញ្ហាអ្វីខ្លះ? តើត្រូវកែសម្រួលយ៉ាងដូចម្តេច?
6. ពន្យល់អំពី "callback hell" ហើយបង្ហាញឧទាហរណ៍អំពីរបៀបដែល Code asynchronous អាចក្លាយទៅជា Code មិនងាយអាន។
7. តើចំណុចខុសគ្នារវាង Promise.all( ) និង Promise.race( ) ជាអ្វី? បង្ហាញឧទាហរណ៍។
8. ក្នុង Fetch API, តើការប្រើ .then( ).catch( ) និង async/await មានភាពខុសគ្នាអ្វី?
9. តើត្រូវប្រើ Promise.all( ) ឬ Promise.allSettled( ) នៅពេលណា? អាចផ្តល់ឧទាហរណ៍បានទេ?
10. តើ event loop ដោះស្រាយ asynchronous operations ដោយរបៀបណា?
11. ចូលកែប្រែ function ខាងក្រោមពី callback-based ទៅ Promise-based

```
function getUserData(callback) {
  setTimeout(() => {
      callback({ name: "Alice", age: 25 });
  }, 1000);
}

getUserData((user) => {
  console.log(user);
});
```

12. សរសេរ function ដែលប្រើ async/await ដើម្បីទាញយក JSON data ពី API https ៖//jsonplaceholder.typicode.com/todos/1 ហើយបង្ហាញលទ្ធផល។

13. បង្កើត function fetchMultipleData ដែល fetch data ពី URLs ពីរ និង console.log លទ្ធផលរួម

```
const url1 = "https://jsonplaceholder.typicode.com/posts/1";
const url2 = "https://jsonplaceholder.typicode.com/posts/2";
```

14. ចូលសរសេរ function fetchDataWithErrorHandling ដែលទាញយក data ពី API មួយ ហើយប្រើ try...catch ដើម្បីចាប់យក error (simulate error ដោយប្រើ wrong URL)

មេរៀនទី ៦៖ ការគ្រប់គ្រងកំហុស (Error handling)

៦.១ ការគ្រប់គ្រងកំហុស "try...catch"

មិនថាយើងពូកែសរសេរកម្មវិធីប៉ុណ្ណាទេ ពេលខ្លះស្គ្រីបរបស់យើងមានកំហុស។ ពួកវាអាចកើតឡើងដោយសារតែកំហុសរបស់យើង ការបញ្ចូលអ្នកប្រើប្រាស់ដែលមិនបានរំពឹងទុក ការឆ្លើយតបរបស់ម៉ាស៊ីនមេដែលមានកំហុស និងសម្រាប់ហេតុផលរាប់ពាន់ផ្សេងទៀត។ ជាធម្មតា ស្គ្រីប "ងាប់" (ឈប់ភ្លាមៗ) ក្នុងករណីមានកំហុស បោះពុម្ពវាទៅកុងសូល។

ប៉ុន្តែមានការស្ថាបនារូបមន្ត try...catch ដែលអនុញ្ញាតឱ្យយើង "ចាប់" កំហុស ដូច្នេះស្គ្រីបអាចជំនួសឱ្យការស្លាប់ ធ្វើអ្វីមួយដែលសមហេតុផលជាងនេះ។

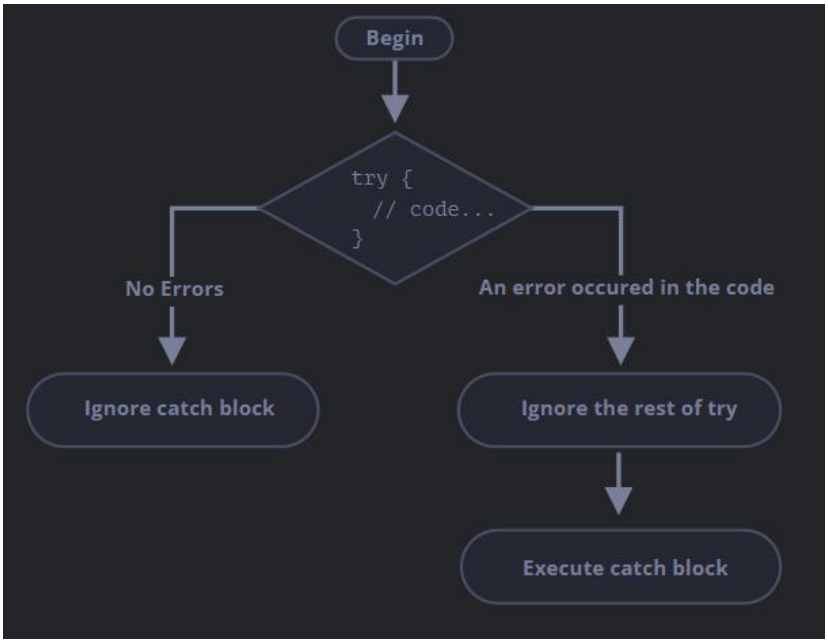
៦.១.១ រូបមន្ត "try…catch"

try...catch construct មានប្លុកសំខាន់ពីរ try៖ ហើយបន្ទាប់មក catch៖

```
try {

  // code...

} catch (err) {

  // error handling

}
```

វាដំណើរការដូចនេះ៖

1. ជាដំបូង Code នៅក្នុង try {...}ត្រូវបានប្រតិបត្តិ។
2. ប្រសិនបើគ្មានកំហុសទេ នោះ catch (err)មិនត្រូវបានអើពើ៖ ការប្រតិបត្តិឈានដល់ទីបញ្ចប់ tryហើយបន្តទៅមុខ ដោយរំលង catch។
3. ប្រសិនបើមានកំហុសកើតឡើង នោះ tryការប្រតិបត្តិត្រូវបានបញ្ឈប់ ហើយការគ្រប់គ្រងហូរទៅដើម catch (err). អថេរ err(យើងអាចប្រើឈ្មោះណាមួយសម្រាប់វា) នឹងមានObjectកំហុសដែលមានព័ត៌មានលម្អិតអំពីអ្វីដែលបានកើតឡើង។



រូបភាព 15៖ រូបមន្ត "try...catch"

ដូច្នេះ កំហុសនៅក្នុង try {...}ប្តូកមិនសម្លាប់ស្ត្រីបទេ យើងមានឱកាសដោះស្រាយវានៅក្នុង catch។ សូមក្រឡេកមើលឧទាហរណ៍ខ្លះ។

- ឧទាហរណ៍ដែលគ្មានកំហុស៖ បង្ហាញ alert (1)និង (2)៖

```
try {

  alert('Start of try runs');  // (1) <--

  // ...no errors here

  alert('End of try runs');   // (2) <--

} catch (err) {

  alert('Catch is ignored, because there are no errors'); // (3)

}
```

- ឧទាហរណ៍ដែលមានកំហុស៖ បង្ហាញ (1)និង (3)៖

```
try {

  alert('Start of try runs');  // (1) <--

  lalala; // error, variable is not defined!

  alert('End of try (never reached)');  // (2)
```

```
} catch (err) {

  alert(`Error has occurred!`); // (3) <--

}
```

ចំណាំ

- try...catchដំណើរការសម្រាប់តែកំហុសពេលដំណើរការប៉ុណ្ណោះ។

ដើម្បី try...catchដំណើរការ Code ត្រូវតែអាចដំណើរការបាន។ នៅក្នុងពាក្យផ្សេងទៀត វាគួរតែជា JavaScript ត្រឹមត្រូវ។ វានឹងមិនដំណើរការទេ ប្រសិនបើ Code ខុសរូបមន្ត ឧទាហរណ៍ វាមានដង្កៀបអង្កាញ់ដែលមិនអាចផ្គូផ្គងបាន៖

```
try {
  {{{{{{{{{{{{
} catch (err) {
  alert("The engine can't understand this code, it's invalid");
}
```

ម៉ាស៊ីន JavaScript ដំបូងអាន Code ហើយបន្ទាប់មកដំណើរការវា។ កំហុសដែលកើតឡើងក្នុងដំណាក់កាលអានត្រូវបានគេហៅថា "parse-time" errors ហើយមិនអាចយកវិញបាន (ពីខាងក្នុង Code នោះ)។ នោះគឺដោយសារតែម៉ាស៊ីនមិនអាចយល់ Code ។

ដូច្នេះ try...catchអាចដោះស្រាយបានតែកំហុសដែលកើតឡើងនៅក្នុង Code ត្រឹមត្រូវ។ កំហុសបែបនេះត្រូវបានគេហៅថា "កំហុសពេលដំណើរការ" ឬជួនកាល "ករណីលើកលែង" ។

- try...catchដំណើរការស្របគ្នា។

ប្រសិនបើករណីលើកលែងកើតឡើងនៅក្នុង Code "បានកំណត់ពេល" ដូចជានៅក្នុង setTimeoutនោះ try...catchនឹងមិនចាប់វាទេ៖

```
try {
  setTimeout(function() {
    noSuchVariable; // script will die here
  }, 1000);
} catch (err) {
  alert( "won't work" );
}
```

នោះដោយសារតែមុខងារខ្លួនវាត្រូវបានប្រតិបត្តិនៅពេលក្រោយ នៅពេលដែលម៉ាស៊ីនបានចាកចេញពី try...catchការសោងសង់រួចហើយ។

ដើម្បីចាប់យកករណីលើកលែងមួយនៅក្នុងមុខងារដែលបានកំណត់ពេល try...catchត្រូវតែស្ថិតនៅក្នុងមុខងារនោះ៖

```
setTimeout(function() {
  try {
    noSuchVariable; // try...catch handles the error!
  } catch {
    alert( "error is caught here!" );
  }
}, 1000);
```

៦.១.២ Error object

នៅពេលដែលមានកំហុសកើតឡើង JavaScript បង្កើតObjectមួយដែលមានព័ត៌មានលម្អិតអំពីវា។ បន្ទាប់មកObjectត្រូវបានបញ្ជូនជាArgumentទៅ catch៖

សម្រាប់កំហុសដែលភ្ជាប់មកជាមួយទាំងអស់ ObjectកំហុសមានProperty សំខាន់ពីរ៖

ក. name

Error name៖ ឧទាហរណ៍ សម្រាប់អថេរដែលមិនបានកំណត់នោះគឺ "ReferenceError"។

ខ. message

សារជាអក្សរអំពីព័ត៌មានលម្អិតអំពីបញ្ហា។ មាន non-standard properties ផ្សេងទៀតដែលមាននៅក្នុងបរិស្ថានភាគច្រើន។ មួយក្នុងចំនោមគេប្រើយ៉ាងទូលំទូលាយ និងគាំទ្រគឺ៖

គ. stack

Current call stack៖៖ ខ្សែអក្សរដែលមានព័ត៌មានអំពីលំដាប់នៃការហៅទូរសព្ទដែលភ្ជាប់មកជាមួយដែលនាំឱ្យមានបញ្ហា។ ប្រើសម្រាប់គោលបំណងបំបាត់កំហុស។

ឧទាហរណ៍៖

```
try {
  lalala; // error, variable is not defined!
} catch (err) {
  alert(err.name); // ReferenceError
  alert(err.message); // lalala is not defined
   alert(err.stack); // ReferenceError: lalala is not defined at
(...call stack)

  // Can also show an error as a whole
  // The error is converted to string as "name: message"
  alert(err); // ReferenceError: lalala is not defined
}
```

៦.១.៣ Optional “catch” binding

ប្រសិនបើយើងមិនត្រូវការព័ត៌មានលម្អិតអំពីបញ្ហាទេ catchអាចលុបចោលវា៖

```
try {
  // ...
} catch { // <-- without (err)
  // ...
}
```

៦.១.៤ ការប្រើប្រាស់ “try…catch”

តោះស្វែងយល់ពីករណីប្រើប្រាស់ក្នុងជីវិតពិតនៃ try...catch. ដូចដែលយើងដឹងរួចហើយ JavaScript គាំទ្រ វិធីសាស្ត្រ JSON.parse(str) ដើម្បីអានតម្លៃដែលបានអ៊ិន Code JSON ។ ជាធម្មតាវាត្រូវបានប្រើដើម្បីឌិ Code ទិន្នន័យដែលទទួលបានតាមរយៈបណ្តាញ ពីម៉ាស៊ីនមេ ឬប្រភពផ្សេងទៀត។

យើងទទួលវា ហើយហៅ JSON.parseដូចនេះ៖

```
let json = '{"name":"John", "age": 30}'; // data from the server

let user = JSON.parse(json); // convert the text representation to JS
object

// now user is an object with properties from the string
alert( user.name ); // John
alert( user.age );  // 30
```

អ្នកអាចស្វែងរកព័ត៌មានលម្អិតបន្ថែមអំពី JSON នៅក្នុង វិធី JSON ដល់ ជំពូក JSON ។ ប្រសិនបើ jsonមានទម្រង់ខុស JSON.parseបង្កើតកំហុស ដូច្នេះស្ក្រីប "ស្លាប់"។

តើយើងគួរតែពេញចិត្តនឹងរឿងនោះទេ? ជាការពិតណាស់មិនមែនទេ!

វិធីនេះ ប្រសិនបើមានអ្វីមួយខុសជាមួយទិន្នន័យ អ្នកចូលមើលនឹងមិនដឹងរឿងនោះទេ (លុះត្រាតែពួកគេបើកកុងសូលអ្នកអភិវឌ្ឍន៍)។ ហើយមនុស្សពិតជាមិនចូលចិត្តនៅពេលដែលអ្វីមួយ "ទើបតែស្លាប់" ដោយគ្មានសារកំហុសណាមួយឡើយ។

តោះប្រើ try...catchដើម្បីដោះស្រាយកំហុស៖

```
let json = "{ bad json }";

try {

  let user = JSON.parse(json); // <-- when an error occurs...
  alert( user.name ); // doesn't work
```

```
} catch (err) {
  // ...the execution jumps here
  alert( "Our apologies, the data has errors, we'll try to request it
one more time." );
  alert( err.name );
  alert( err.message );
}
```

នៅទីនេះយើងប្រើ catch ប្លុកដើម្បីបង្ហាញសារតែប៉ុណ្ណោះ ប៉ុន្តែយើងអាចធ្វើបានច្រើនទៀត៖ ផ្ញើសំណើបណ្តាញថ្មី ស្នើជម្រើសជំនួសអ្នកទស្សនា ផ្ញើព័ត៌មានអំពីកំហុសទៅកន្លែងកាប់ឈើ ... ។ ទាំងអស់ប្រសើរជាងគ្រាន់តែស្លាប់។

៦.១.៥ បោះចោលកំហុសរបស់យើង

ចុះបើ jsonរូបមន្តត្រឹមត្រូវ ប៉ុន្តែមិនមាន name property ដែលត្រូវការ?

ដូចនេះ៖

```
let json = '{ "age": 30 }'; // incomplete data

try {

  let user = JSON.parse(json); // <-- no errors
  alert( user.name ); // no name!

} catch (err) {
  alert( "doesn't execute" );
}
```

នៅទីនេះ JSON.parseដំណើរការជាធម្មតា ប៉ុន្តែអវត្តមាន nameគឺពិតជាកំហុសសម្រាប់ពួកយើង។ ដើម្បីបង្រួបបង្រួមការដោះស្រាយកំហុស យើងនឹងប្រើ throwប្រតិបត្តិករ។

៦.១.៦ “Throw” operator

ប្រតិបត្តិ throwករបង្កើតកំហុស។ រូបមន្ត៖

```
throw <error object>
```

តាមបច្ចេកទេស យើងអាចប្រើអ្វីមួយជាObjectកំហុស។ នោះអាចជាបុព្វបទ ដូចជាលេខ ឬខ្សែអក្សរ ប៉ុន្តែវាជាការប្រសើរក្នុងការប្រើប្រាស់Object និយមជាមួយ nameនិង messageProperty (ដើម្បីរក្សាភាពឆបគ្នាខ្លះជាមួយកំហុសដែលភ្ជាប់មកជាមួយ)។

JavaScript មានអ្នកសាងសង់ដែលមានស្រាប់ជាច្រើនសម្រាប់កំហុសស្តង់ដារ៖ Error, រូបមន្ត Error, ReferenceError, TypeErrorនិងផ្សេងទៀត។ យើងអាចប្រើពួកវាដើម្បីបង្កើត error object ផងដែរ។

រូបមន្ត៖

```
let error = new Error(message);
// or
let error = new រូបមន្តError(message);
let error = new ReferenceError(message);
// ...
```

សម្រាប់កំហុសដែលភ្ជាប់មកជាមួយ (មិនមែនសម្រាប់Objectណាមួយទេ គ្រាន់តែសម្រាប់កំហុស) nameProperty គឺពិតជាឈ្មោះរបស់អ្នកសាងសង់។ ហើយ messageត្រូវបានដកចេញពីArgument។

ឧទាហរណ៍៖

```
let error = new Error("Things happen o_O");

alert(error.name); // Error
alert(error.message); // Things happen o_O
```

តោះមើលថាតើកំហុសប្រភេទណា JSON.parseដែលបង្កើតឡើង៖

```
try {
  JSON.parse("{ bad json o_O }");
} catch (err) {
  alert(err.name); // រូបមន្តError
  alert(err.message); // Unexpected token b in JSON at position 2
}
```

ដូចដែលយើងអាចមើលឃើញ នោះគឺជា រូបមន្តError. ហើយនៅក្នុងករណីរបស់យើង អវត្តមាននៃ nameគឺជាកំហុសមួយ ដោយសារអ្នកប្រើប្រាស់ត្រូវតែមាន name.

So let's throw it៖

```
let json = '{ "age": 30 }'; // incomplete data

try {

  let user = JSON.parse(json); // <-- no errors

  if (!user.name) {
```

```
    throw new រូបមន្តError("Incomplete data: no name"); // (*)
  }

  alert( user.name );

} catch (err) {
  alert( "JSON Error: " + err.message ); // JSON Error: Incomplete data
: no name
}
```

នៅក្នុងបន្ទាត់ (*)ប្រតិបត្តិ throwករបង្កើត a រូបមន្តErrorជាមួយនឹងការផ្តល់ឱ្យ message តាមរបៀបដូចគ្នានឹង JavaScript នឹងបង្កើតវាដោយខ្លួនឯង។ ការប្រតិបត្តិ tryការបញ្ឈប់ជាបន្ទាន់និងលំហូរការត្រួតពិនិត្យលោតចូលទៅក្នុង catch.

ឥឡូវនេះ catchបានក្លាយជាកន្លែងតែមួយសម្រាប់ការដោះស្រាយកំហុសទាំងអស់៖ ទាំងសម្រាប់ JSON.parseនិងករណីផ្សេងទៀត។

៦.១.៧ Rethrowing

ក្នុងឧទាហរណ៍ខាងលើ យើងប្រើ try...catchដើម្បីដោះស្រាយទិន្នន័យមិនត្រឹមត្រូវ។ ប៉ុន្តែវាអាចទៅរួចដែលថា មានកំហុសដែលមិននឹកស្មានដល់មួយទៀតកើតឡើងនៅក្នុង try {...}ប្លុក? ដូចជាកំហុសក្នុងការសរសេរកម្មវិធី (អថេរមិនត្រូវបានកំណត់) ឬអ្វីផ្សេងទៀត មិនមែនគ្រាន់តែជា "ទិន្នន័យមិនត្រឹមត្រូវ" នេះទេ។

ឧទាហរណ៍៖

```
let json = '{ "age": 30 }'; // incomplete data

try {
  user = JSON.parse(json); // <-- forgot to put "let" before user

  // ...
} catch (err) {
  alert("JSON Error: " + err); // JSON Error: ReferenceError: user is
not defined
  // (no JSON Error actually)
}
```

ជាការពិតណាស់ អ្វីៗអាចទៅរួច! អ្នកសរសេរកម្មវិធីមានកំហុស។ សូម្បីតែនៅក្នុងឧបករណ៍ប្រើប្រាស់ប្រភពបើកចំហដែលត្រូវបានប្រើប្រាស់ដោយមនុស្សរាប់លានអស់ជាច្រើនទសវត្សរ៍មកហើយ ស្រាប់តែមានកំហុសមួយអាចត្រូវបានរកឃើញដែលនាំទៅដល់ការលួចចូលដ៏គួរឱ្យភ័យខ្លាច។

ក្នុងករណីរបស់យើង try...catchត្រូវបានដាក់ដើម្បីចាប់កំហុស "ទិន្នន័យមិនត្រឹមត្រូវ" ។ ប៉ុន្តែដោយធម្មជាតិរបស់វា catchទទួលបាន កំហុស ទាំងអស់try ពី . នៅទីនេះវាទទួលបានកំហុសដែលមិនបានរំពឹងទុក ប៉ុន្តែនៅតែបង្ហាញ "JSON Error"សារដដែល។ វាខុស ហើយថែមទាំងធ្វើឱ្យ Code កាន់តែពិបាកក្នុងការបំបាត់កំហុស។

ដើម្បីជៀសវាងបញ្ហាបែបនេះ យើងអាចប្រើបច្ចេកទេស "បោះចោល"។ ច្បាប់គឺសាមញ្ញ៖

Catch គួរតែដំណើរការតែកំហុសដែលវាដឹង និង " rethrowing" អ្នកផ្សេងទៀតទាំងអស់។ បច្ចេកទេស "ការដកថយ" អាចត្រូវបានពន្យល់លម្អិតបន្ថែមទៀតដូចជា៖

1. Catch ទទួលបានកំហុសទាំងអស់។
2. នៅក្នុង catch (err) {...}ប្លុកយើងវិភាគObjectកំហុស err។
3. បើយើងមិនចេះដោះស្រាយទេ យើងធ្វើ throw err។

ជាធម្មតា យើងអាចពិនិត្យប្រភេទ Error ដោយប្រើ instanceofOperator៖

```
try {
  user = { /*...*/ };
} catch (err) {
  if (err instanceof ReferenceError) {
     alert('ReferenceError'); // "ReferenceError" for accessing an
undefined variable
  }
}
```

យើងក៏អាចទទួលបានឈ្មោះClassកំហុសពី err.nameលក្ខណសម្បត្តិ។ កំហុសដើមទាំងអស់មានវា។ ជម្រើសមួយទៀតគឺការអាន err.constructor.name។

ក្នុង Code ខាងក្រោម យើងប្រើការដកវិញដូច្នេះ catchបានតែដោះស្រាយ រួបមន្តError៖

```
let json = '{ "age": 30 }'; // incomplete data
try {

  let user = JSON.parse(json);

  if (!user.name) {
    throw new រួបមន្តError("Incomplete data: no name");
  }
```

```
  blabla(); // unexpected error

  alert( user.name );

} catch (err) {

  if (err instanceof រូបមន្តError) {
    alert( "JSON Error: " + err.message );
  } else {
    throw err; // rethrow (*)
  }

}
```

កំហុសដែលបោះនៅលើបន្ទាត់ (*)ពីប្លុកខាងក្នុង catch"ធ្លាក់ចេញ" try...catchហើយអាចត្រូវបានចាប់បានដោយ try...catchការសាងសង់ខាងក្រៅ (ប្រសិនបើវាមាន) ឬវាសម្លាប់ស្គ្រីប។ ដូច្នេះ catchប្លុកពិតជាដោះស្រាយតែកំហុសដែលវាដឹងពីរបៀបដោះស្រាយនិង "រំលង" ទាំងអស់ផ្សេងទៀត។

ឧទាហរណ៍ខាងក្រោមបង្ហាញពីរបៀបដែលកំហុសបែបនេះអាចត្រូវបានចាប់បានដោយកម្រិតមួយទៀតនៃ try...catch៖

```
function readData() {
  let json = '{ "age": 30 }';

  try {
    // ...
    blabla(); // error!
  } catch (err) {
    // ...
    if (!(err instanceof រូបមន្តError)) {
      throw err; // rethrow (don't know how to deal with it)
    }
  }
}

try {
  readData();
} catch (err) {
  alert( "External catch got: " + err ); // caught it!
}
```

នៅទីនេះ readDataដឹងតែពីរបៀបដោះស្រាយ រួបមន្តErrorចំណែកខាងក្រៅ try...catchដឹងពីរបៀបដោះស្រាយគ្រប់យ៉ាង។

៦.១.៨ try…catch…finally

The try...catch construct អាចមានឃ្លា Code មួយទៀត៖ finally។ ប្រសិនបើមាន វាដំណើរការក្នុងគ្រប់ករណីទាំងអស់៖

- បន្ទាប់ពី try, ប្រសិនបើមិនមានកំហុស,
- បន្ទាប់ពី catchប្រសិនបើមានកំហុស។

រួបមន្ត៖

```
try {
   ... try to execute the code ...
} catch (err) {
   ... handle errors ...
} finally {
   ... execute always ...
}
```

សាកល្បងដំណើរការ Code នេះ៖

```
try {
  alert( 'try' );
  if (confirm('Make an error?')) BAD_CODE();
} catch (err) {
  alert( 'catch' );
} finally {
  alert( 'finally' );
}
```

Code មានវិធីពីរយ៉ាងក្នុងការប្រតិបត្តិ៖

1. ប្រសិនបើអ្នកឆ្លើយថា "បាទ / ចាស" ទៅ "បង្កើតកំហុស ?" នោះ try -> catch -> finally.
2. ប្រសិនបើអ្នកនិយាយថា "ទេ" នោះ try -> finally.

ប្រយោគ finallyត្រូវបានគេប្រើជាញឹកញាប់នៅពេលដែលយើងចាប់ផ្តើមធ្វើអ្វីមួយ ហើយចង់បញ្ចប់វានៅក្នុងករណីនៃលទ្ធផលណាមួយ។ ជាឧទាហរណ៍ យើងចង់វាស់ពេលវេលាដែលមុខងារលេខ Fibonacci fib(n)ប្រើ។ តាមធម្មជាតិ យើងអាចចាប់ផ្តើមវាស់មុនពេលវាដំណើរការ និងបញ្ចប់នៅពេលក្រោយ។ ប៉ុន្តែចុះយ៉ាងណាបើមានកំហុសអំឡុងពេលហៅមុខងារ? ជាពិសេស ការអនុវត្ត fib(n)នៅក្នុង Code ខាងក្រោម ត្រឡប់កំហុសសម្រាប់លេខអវិជ្ជមាន ឬមិនមែនចំនួនគត់។ ឃ្លា finallyគឺជាកន្លែងដ៏ល្អមួយដើម្បីបញ្ចប់ការវាស់វែងមិនថាមានបញ្ហាអ្វីនោះទេ។

នៅទីនេះ finallyជានាថាពេលវេលានឹងត្រូវបានវាស់វែងយ៉ាងត្រឹមត្រូវក្នុងស្ថានភាពទាំងពីរ - ក្នុងករណីដែលការប្រតិបត្តិដោយជោគជ័យ fibនិងក្នុងករណីមានកំហុសនៅក្នុងវា៖

```
let num = +prompt("Enter a positive integer number?", 35)

let diff, result;

function fib(n) {
  if (n < 0 || Math.trunc(n) != n) {
    throw new Error("Must not be negative, and also an integer.");
  }
  return n <= 1 ? n : fib(n - 1) + fib(n - 2);
}

let start = Date.now();

try {
  result = fib(num);
} catch (err) {
  result = 0;
} finally {
  diff = Date.now() - start;
}

alert(result || "error occurred");

alert( `execution took ${diff}ms` );
```

អ្នកអាចពិនិត្យដោយដំណើរការ Code ដោយបញ្ចូល 35ទៅក្នុង prompt– វាដំណើរការដោយធម្មតា finallyបន្ទាប់ពី try. ហើយបន្ទាប់មកបញ្ចូល -1- វានឹងមានកំហុសភ្លាមៗ ហើយការប្រតិបត្តិនឹងកើតឡើង 0msៗ ការវាស់វែងទាំងពីរត្រូវបានធ្វើត្រឹមត្រូវ។ នៅក្នុងពាក្យផ្សេងទៀត មុខងារអាចបញ្ចប់ដោយ returnឬ throwវាមិនមានបញ្ហាៗ ប្រយោគ finallyប្រតិបត្តិក្នុងករណីទាំងពីរ។

- អថេរគឺស្ថិតនៅក្នុងមូលដ្ឋានtry...catch...finally

  សូមចំណាំថា resultនិង diffអថេរនៅក្នុង Code ខាងលើត្រូវបានប្រកាស ពីមុន try...catch ។

  បើមិនដូច្នេះទេ ប្រសិនបើយើងប្រកាស letនៅក្នុង tryប្លុក វានឹងអាចមើលឃើញតែនៅខាងក្នុងរបស់វាប៉ុណ្ណោះ។

- finallyនិងreturn

ប្រយោគ finallyដំណើរការសម្រាប់ ច្រកចេញ ណាមួយ ពី try...catch. នោះរួមបញ្ចូលទាំងការ ពន្យល់ច្បាស់លាស់ return។

នៅក្នុងឧទាហរណ៍ខាងក្រោម returnមាន try. ក្នុងករណីនេះ finallyត្រូវបានប្រតិបត្តិមុនពេល Objectបញ្ជាត្រឡប់ទៅលេខ Code ខាងក្រៅវិញ។

```
function func() {

  try {
    return 1;

  } catch (err) {
    /* ... */
  } finally {
    alert( 'finally' );
  }
}

alert( func() ); // first works alert from finally, and then this one
```

- try...finally

ការសាងសង់ try...finallyដោយគ្មាន catchឃ្លាក៏មានប្រយោជន៍ផងដែរ។ យើងអនុវត្តវានៅ ពេលដែលយើងមិនចង់ដោះស្រាយកំហុសនៅទីនេះ (អនុញ្ញាតឱ្យពួកគេឆ្លងកាត់) ប៉ុន្តែចង់ប្រាកដថា ដំណើរការដែលយើងបានចាប់ផ្តើមត្រូវបានបញ្ចប់។

```
function func() {
  // start doing something that needs completion (like measurements)
  try {
    // ...
  } finally {
    // complete that thing even if all dies
  }
}
```

នៅក្នុង Code ខាងលើ កំហុសនៅខាងក្នុង tryតែងតែលេចចេញជានិច្ច ព្រោះវាគ្មាន catch។ ប៉ុន្តែ finallyដំណើរការមុនពេលលំហូរប្រតិបត្តិចាកចេញពីមុខងារ។

៦.១.៥ Global catch

សូមស្រមៃថាយើងមានកំហុសធ្ងន់ធ្ងរនៅខាងក្រៅ try...catchហើយស្ត្រីបបានងាប់។ ដូចជា កំហុសកម្មវិធី ឬរឿងដ៏គួរឱ្យភ័យខ្លាចផ្សេងទៀត។ តើមានវិធីដើម្បីប្រតិកម្មចំពោះការកើតឡើងបែបនេះ

ទេ? យើងប្រហែលជាចង់កត់ត្រាកំហុស បង្ហាញអ្វីមួយដល់អ្នកប្រើ (ជាធម្មតាពួកគេមិនឃើញសារកំហុស) ជាដើម។ មិនមានអ្វីនៅក្នុងការបញ្ជាក់នោះទេ ប៉ុន្តែបរិស្ថានជាធម្មតាផ្តល់វា ព្រោះវាពិតជាមានប្រយោជន៍។ ឧទាហរណ៍ Node.js មាន process.on("uncaughtException") សម្រាប់នោះ។ ហើយនៅក្នុងBrowserយើងអាចកំណត់មុខងារមួយទៅ លក្ខណសម្បត្តិ window.onerror ពិសេស ដែលនឹងដំណើរការក្នុងករណីមានកំហុសដែលមិនអាចចាប់បាន។

រូបមន្ត៖

```
window.onerror = function(message, url, line, col, error) {
  // ...
};
```

- message៖ សារកំហុស។
- url៖ URL នៃស្គ្រីបដែលកំហុសបានកើតឡើង។
- line,col៖ លេខជួរ និងជួរដែលមានកំហុសកើតឡើង។
- error៖ Objectមានកំហុស។

ឧទាហរណ៍៖

```
<script>
  window.onerror = function(message, url, line, col, error) {
    alert(`${message}\n At ${line}:${col} of ${url}`);
  };

  function readData() {
    badFunc(); // Whoops, something went wrong!
  }

  readData();
</script>
```

តួនាទីរបស់អ្នកដោះស្រាយសកល window.onerrorគឺជាធម្មតាមិនមែនដើម្បីសង្គ្រោះការប្រតិបត្តិស្គ្រីបទេ - វាប្រហែលជាមិនអាចទៅរួចទេក្នុងករណីមានកំហុសក្នុងការសរសេរកម្មវិធី ប៉ុន្តែដើម្បីផ្ញើសារកំហុសទៅអ្នកអភិវឌ្ឍន៍។ វាក៏មានសេវាគេហទំព័រដែលផ្តល់ការកត់ត្រាកំហុសសម្រាប់ករណីបែបនេះ ដូចជា https://muscula.com ឬ https://www.sentry.io ។

ពួកគេធ្វើការដូចនេះ៖

1. យើងចុះឈ្មោះនៅសេវាកម្ម និងទទួលបានបំណែកនៃ JS (ឬ script URL) ពីពួកគេដើម្បីបញ្ចូលនៅលើទំព័រ។

2. ស្ដ្រីប JS នោះកំណត់ window.onerrorមុខងារផ្ទាល់ខ្លួន។
3. នៅពេលមានកំហុសកើតឡើង វាផ្ញើសំណើបណ្តាញអំពីវាទៅសេវាកម្ម។
4. យើងអាចចូលទៅកាន់ចំណុចប្រទាក់បណ្តាញសេវា និងមើលឃើញកំហុស។

៦.២ Custom errors, extending Error

នៅពេលយើងអភិវឌ្ឍអ្វីមួយ យើងតែងតែត្រូវការClassកំហុសផ្ទាល់ខ្លួនរបស់យើង ដើម្បីឆ្លុះបញ្ចាំងពីអ្វីដែលជាក់លាក់ដែលអាចខុសនៅក្នុងកិច្ចការរបស់យើង។ សម្រាប់កំហុសក្នុងប្រតិបត្តិការបណ្តាញ យើងប្រហែលជាត្រូវការ HttpErrorសម្រាប់ប្រតិបត្តិការមូលដ្ឋានទិន្នន័យ DbErrorសម្រាប់ប្រតិបត្តិការស្វែងរក NotFoundErrorនិងអ្វីៗផ្សេងទៀត។ កំហុសរបស់យើងគួរតែគាំទ្រProperty កំហុសជាមូលដ្ឋានដូចជា message, nameនិង, និយម, stack. ប៉ុន្តែពួកវាក៏អាចមានProperty ផ្សេងទៀតរបស់ពួកគេផងដែរ ឧទាហរណ៍ HttpErrorObjectអាច statusCodeមានតម្លៃដូច 404ឬ 403ឬ 500។

JavaScript អនុញ្ញាតឱ្យប្រើ throwជាមួយArgumentណាមួយ ដូច្នេះតាមបច្ចេកទេស Class កំហុសផ្ទាល់ខ្លួនរបស់យើងមិនចាំបាច់ទទួលមរតកពី Error. ប៉ុន្តែប្រសិនបើយើងទទួលមរតក នោះវាអាចប្រើ obj instanceof Errorដើម្បីកំណត់អត្តសញ្ញាណObjectដែលមានកំហុស។ ដូច្នេះវាជាការប្រសើរក្នុងការទទួលមរតកពីវា។ នៅពេលដែលកម្មវិធីរីកចម្រើន កំហុសរបស់យើងផ្ទាល់បង្កើតជាឋានានុក្រម។ ឧទាហរណ៍ HttpTimeoutErrorអាចទទួលមរតកពី HttpErrorជាដើម។

៦.២.១ Extending Error

ជាឧទាហរណ៍ សូមពិចារណាមុខងារ readUser(json)ដែលគួរអាន JSON ជាមួយទិន្នន័យអ្នកប្រើប្រាស់។ នេះជាឧទាហរណ៍នៃរបៀបដែលសុពលភាព jsonអាចមើលទៅ៖

```
let json = `{ "name": "John", "age": 30 }`;
```

នៅខាងក្នុងយើងនឹងប្រើ JSON.parse។ ប្រសិនបើវាទទួលខុស jsonវានឹងបោះចោល រូបមន្ត Error។ ប៉ុន្តែទោះបីជា jsonវាត្រឹមត្រូវក៏ដោយ វាមិនមែនមានន័យថាវាជាអ្នកប្រើប្រាស់ត្រឹមត្រូវទេមែនទេ? វាអាចខកខានទិន្នន័យចាំបាច់។ ជាឧទាហរណ៍ វាអាចមិនមាន nameនិង ageProperty ដែលចាំបាច់សម្រាប់អ្នកប្រើប្រាស់របស់យើង។

មុខងាររបស់យើង readUser(json)នឹងមិនត្រឹមតែអាន JSON ប៉ុណ្ណោះទេ ប៉ុន្តែពិនិត្យមើល("ធ្វើឱ្យមានសុពលភាព") ទិន្នន័យ។ ប្រសិនបើមិនមានវាលដែលត្រូវការ ឬទម្រង់ខុស នោះជាកំហុស។ ហើយនោះមិនមែនជា ទេ រូបមន្តErrorពីព្រោះទិន្នន័យគឺត្រឹមត្រូវ រូបមន្ត ប៉ុន្តែប្រភេទនៃកំហុសមួយផ្សេង

ទៀត។ យើងនឹងហៅវា ValidationErrorហើយបង្កើតClassសម្រាប់វា។ កំហុសនៃប្រភេទនោះក៏គួរមានព័ត៌មានអំពីកន្លែងបំពានផងដែរ។

Class របស់យើង ValidationErrorគួរតែទទួលមរតកពី ErrorClass។ Class Errorត្រូវបានភ្ជាប់មកជាមួយ ប៉ុន្តែនេះគឺជា Code ប្រហាក់ប្រហែលរបស់វា ដូច្នេះយើងអាចយល់ពីអ្វីដែលយើងកំពុងពង្រីក៖

```
// The "pseudocode" for the built-in Error class defined by JavaScript
itself
class Error {
  constructor(message) {
    this.message = message;
    this.name = "Error"; // (different names for different built-in
error classes)
    this.stack = <call stack>; // non-standard, but most environments
support it
  }
}
```

ឥឡូវនេះ ចូរយើងទទួលមរតក ValidationErrorពីវា ហើយសាកល្បងវានៅក្នុងសកម្មភាព៖

```
class ValidationError extends Error {
  constructor(message) {
    super(message); // (1)
    this.name = "ValidationError"; // (2)
  }
}

function test() {
  throw new ValidationError("Whoops!");
}

try {
  test();
} catch(err) {
  alert(err.message); // Whoops!
  alert(err.name); // ValidationError
  alert(err.stack); // a list of nested calls with line numbers for
each
}
```

សូមចំណាំ៖ នៅក្នុងបន្ទាត់ (1)យើងហៅអ្នកបង្កើតមេ។ JavaScript តម្រូវឱ្យយើងហៅ super ទៅកាន់អ្នកបង្កើតកូន ដូច្នេះវាជាកាតព្វកិច្ច។ អ្នកសាងសង់មេកំណត់ messageProperty ។

អ្នកបង្កើតមេក៏កំណត់ nameលក្ខណសម្បត្តិ ផងដែរ "Error"ដូច្នេះក្នុងជួរ (2)យើងកំណត់វាឡើងវិញទៅតម្លៃត្រឹមត្រូវ។

តោះសាកល្បងប្រើក្នុង readUser(json)៖

```
class ValidationError extends Error {
  constructor(message) {
    super(message);
    this.name = "ValidationError";
  }
}

// Usage
function readUser(json) {
  let user = JSON.parse(json);

  if (!user.age) {
    throw new ValidationError("No field: age");
  }
  if (!user.name) {
    throw new ValidationError("No field: name");
  }

  return user;
}

// Working example with try..catch

try {
  let user = readUser('{ "age": 25 }');
} catch (err) {
  if (err instanceof ValidationError) {
    alert("Invalid data: " + err.message); // Invalid data: No field: name
  } else if (err instanceof រូបមន្តError) { // (*)
    alert("JSON រូបមន្ត Error: " + err.message);
  } else {
    throw err; // unknown error, rethrow it (**)
  }
}
```

ប្លុក try..catchក្នុង Code ខាងលើគ្រប់គ្រងទាំងរបស់យើង ValidationErrorនិងការភ្ជាប់មកជាមួយ រូបមន្តErrorពី JSON.parse។ សូមក្រឡេកមើលពីរបៀបដែលយើងប្រើ instanceofដើម្បីពិនិត្យមើលប្រភេទកំហុសជាក់លាក់នៅក្នុងបន្ទាត់ (*)។

យើងក៏អាចមើលផងដែរ err.nameដូចនេះ៖

```
// ...
// instead of (err instanceof រូបមន្តError)
} else if (err.name == "រូបមន្តError") { // (*)
  // ...
```

កំណែ instanceofគឺកាន់តែល្អប្រសើរជាងមុន ព្រោះនៅពេលអនាគតយើងនឹងពង្រីក ValidationErrorបង្កើតប្រភេទរងរបស់វាដូចជា PropertyRequiredError. ហើយ instanceofធីកនឹងបន្តដំណើរការសម្រាប់Classទទួលមរតកថ្មី។ ដូច្នេះវាជាកស្តុតាងអនាគត។

វាក៏សំខាន់ផងដែរដែលថាប្រសិនបើ catchជួបកំហុសដែលមិនស្គាល់នោះវានឹងបោះវាឡើងវិញនៅក្នុងបន្ទាត់ (* *)។ ប្លុក catchដឹងតែពីរបៀបដោះស្រាយភាពត្រឹមត្រូវ និងកំហុសរូបមន្តប៉ុណ្ណោះ ប្រភេទផ្សេងទៀត (បណ្តាលមកពីការវាយអក្សរនៅក្នុង Code ឬមូលហេតុដែលមិនស្គាល់ផ្សេងទៀត) គួរតែធ្លាក់។

១.២.២ មរតកបន្ត (Further inheritance)

Class ValidationErrorគឺមានលក្ខណៈទូទៅណាស់។ រឿងជាច្រើនអាចនឹងខុស។ Property អាចនឹងអវត្តមាន ឬវាអាចមានទម្រង់ខុស (ដូចជាតម្លៃខ្សែអក្សរ ageជំនួសឲ្យលេខ)។ ចូរបង្កើតClass បេតុងបន្ថែមទៀត PropertyRequiredErrorពិតប្រាកដសម្រាប់Property អវត្តមាន។ វានឹងមានព័ត៌មានបន្ថែមអំពីProperty ដែលបាត់។

```
class ValidationError extends Error {
  constructor(message) {
    super(message);
    this.name = "ValidationError";
  }
}

class PropertyRequiredError extends ValidationError {
  constructor(property) {
    super("No property: " + property);
    this.name = "PropertyRequiredError";
    this.property = property;
  }
```

```
}

// Usage
function readUser(json) {
  let user = JSON.parse(json);

  if (!user.age) {
    throw new PropertyRequiredError("age");
  }
  if (!user.name) {
    throw new PropertyRequiredError("name");
  }

  return user;
}

// Working example with try..catch

try {
  let user = readUser('{ "age": 25 }');
} catch (err) {
  if (err instanceof ValidationError) {
    alert("Invalid data: " + err.message); // Invalid data: No property
: name
    alert(err.name); // PropertyRequiredError
    alert(err.property); // name
  } else if (err instanceof រូបមន្តError) {
    alert("JSON រូបមន្ត Error: " + err.message);
  } else {
    throw err; // unknown error, rethrow it
  }
}
```

Classថ្មី PropertyRequiredErrorងាយស្រួលប្រើ៖ យើងគ្រាន់តែបញ្ជូលឈ្មោះអចលនទ្រព្យ៖ new PropertyRequiredError(property). មនុស្សអាចអានបាន messageត្រូវបានបង្កើតឡើងដោយអ្នកសាងសង់។

សូមចំណាំថា this.nameនៅក្នុង PropertyRequiredErrorconstructor ត្រូវបានកំណត់ម្តងទៀតដោយដៃ។ នោះអាចក្លាយជាការធុញទ្រាន់បន្តិច - ដើម្បីចាត់តាំង this.name = <class name> នៅក្នុងគ្រប់Classកំហុសផ្ទាល់ខ្លួន។ យើងអាចជៀសវាងវាដោយបង្កើតClass "កំហុសមូលដ្ឋាន" ផ្ទាល់

ខ្លួនរបស់យើងដែល this.name = this.constructor.nameកំណត់ ហើយបន្ទាប់មកទទួលមរតករាល់កំហុសផ្ទាល់ខ្លួនរបស់យើងពីវា។ ចូរហៅវា MyError។

នេះជា Code ជាមួយ MyErrorនិងClassកំហុសផ្ទាល់ខ្លួនផ្សេងទៀត ដែលបានធ្វើឲ្យសាមញ្ញ៖

```
class MyError extends Error {
  constructor(message) {
    super(message);
    this.name = this.constructor.name;
  }
}

class ValidationError extends MyError { }

class PropertyRequiredError extends ValidationError {
  constructor(property) {
    super("No property: " + property);
    this.property = property;
  }
}

// name is correct
alert( new PropertyRequiredError("field").name ); // PropertyRequiredError
```

ឥឡូវនេះ កំហុសផ្ទាល់ខ្លួនគឺខ្លីជាង ជាពិសេស ValidationErrorនៅពេលដែលយើងកម្ចាត់ "this.name = ..."បន្ទាត់នៅក្នុង constructor ។

៦.២.៣ ការលើកលែងការខេចខ្ចប់ (Wrapping exceptions)

គោលបំណងនៃមុខងារ readUserនៅក្នុង Code ខាងលើគឺ "អានទិន្នន័យអ្នកប្រើប្រាស់"។ វាអាចមានប្រភេទផ្សេងគ្នានៃកំហុសនៅក្នុងដំណើរការ។ ឥឡូវនេះយើងមាន រូបមន្តErrorហើយ Validation

Errorប៉ុន្តែនៅពេលអនាគត readUserមុខងារអាចនឹងរីកចម្រើន ហើយប្រហែលជាបង្កើតប្រភេទកំហុសផ្សេងទៀត។

លេខ Code ដែលការហៅទូរស័ព្ទ readUserគួរតែដោះស្រាយកំហុសទាំងនេះ។ ឥឡូវនេះវាប្រើ ifs ច្រើននៅក្នុង catchប្លុក ដែលពិនិត្យមើលClass និងដោះស្រាយកំហុសដែលគេស្គាល់ ហើយបោះចោលអ្នកដែលមិនស្គាល់ឡើងវិញ។

គ្រោងការណ៍គឺដូចនេះ៖

```
try {
  ...
  readUser()  // the potential error source
  ...
} catch (err) {
  if (err instanceof ValidationError) {
    // handle validation errors
  } else if (err instanceof រូបមន្តError) {
    // handle រូបមន្ត errors
  } else {
    throw err; // unknown error, rethrow it
  }
}
```

នៅក្នុង Code ខាងលើ យើងអាចមើលឃើញកំហុសពីរប្រភេទ ប៉ុន្តែអាចមានច្រើនជាងនេះ។ ប្រសិនបើ readUserមុខងារបង្កើតកំហុសជាច្រើនប្រភេទ នោះយើងគួរតែសួរខ្លួនយើងថា តើយើងពិតជាចង់ពិនិត្យមើលប្រភេទកំហុសទាំងអស់ម្តងមួយៗរាល់ពេលមែនទេ?

ជារឿយៗ ចម្លើយគឺ "ទេ"៖ យើងចង់ក្លាយជា "កម្រិតមួយលើសពីអ្វីទាំងអស់"។ យើងគ្រាន់តែចង់ដឹងថាតើមាន "កំហុសក្នុងការអានទិន្នន័យ" - ហេតុអ្វីបានជាវាកើតឡើងជាញឹកញាប់មិនពាក់ព័ន្ធ (សារកំហុសពិពណ៌នាអំពីវា)។ ឬប្រសើរជាងនេះទៅទៀត យើងចង់មានវិធីដើម្បីទទួលបានព័ត៌មានលម្អិតអំពីបញ្ហា ប៉ុន្តែប្រសិនបើយើងត្រូវការតែប៉ុណ្ណោះ។ បច្ចេកទេសដែលយើងពិពណ៌នានៅទីនេះត្រូវបានគេហៅថា "ការលើកលែងរុំ" ។

1. យើងនឹងបង្កើតClassថ្មីមួយ ReadErrorដើម្បីតំណាងឱ្យកំហុស "ការអានទិន្នន័យ" ទូទៅ។
2. មុខងារនេះ readUserនឹងចាប់កំហុសក្នុងការអានទិន្នន័យដែលកើតឡើងនៅក្នុងវា ដូចជា ValidationErrorនិង រូបមន្តErrorនិងបង្កើត ReadErrorជំនួសវិញ។
3. Object ReadErrorនឹងរក្សាឯកសារយោងទៅកំហុសដើមនៅក្នុង causeProperty របស់វា។

បន្ទាប់មក លេខ Code ដែលហៅចេញ readUserនឹងត្រូវពិនិត្យ ReadErrorមិនមែនសម្រាប់រាល់កំហុសក្នុងការអានទិន្នន័យនោះទេ។ ហើយប្រសិនបើវាត្រូវការព័ត៌មានលម្អិតបន្ថែមអំពីកំហុស វាអាចពិនិត្យមើល causeProperty របស់វា។

នេះជា Code ដែលកំណត់ ReadErrorនិងបង្ហាញពីការប្រើប្រាស់របស់វានៅក្នុង readUserនិង try..catch៖

ឧទាហរណ៍៖

```
class ReadError extends Error {
```

```
  constructor(message, cause) {
    super(message);
    this.cause = cause;
    this.name = 'ReadError';
  }
}

class ValidationError extends Error { /*...*/ }
class PropertyRequiredError extends ValidationError { /* ... */ }

function validateUser(user) {
  if (!user.age) {
    throw new PropertyRequiredError("age");
  }

  if (!user.name) {
    throw new PropertyRequiredError("name");
  }
}

function readUser(json) {
  let user;

  try {
    user = JSON.parse(json);
  } catch (err) {
    if (err instanceof រូបមន្តError) {
      throw new ReadError("រូបមន្ត Error", err);
    } else {
      throw err;
    }
  }

  try {
    validateUser(user);
  } catch (err) {
    if (err instanceof ValidationError) {
      throw new ReadError("Validation Error", err);
    } else {
      throw err;
    }
  }
```

```
}

try {
  readUser('{bad json}');
} catch (e) {
  if (e instanceof ReadError) {
    alert(e);
    // Original error: រូបមន្តError: Unexpected token b in JSON at position
1
    alert("Original error: " + e.cause);
  } else {
    throw e;
  }
}
```

នៅក្នុង Code ខាងលើ readUserដំណើរការយ៉ាងពិតប្រាកដដូចដែលបានពិពណ៌នា – ចាប់កំហុសរូបមន្ត និងសុពលភាព ហើយបោះ ReadErrorកំហុសជំនួសវិញ (កំហុសមិនស្គាល់ត្រូវបានបញ្ជូនឡើងវិញដូចធម្មតា)។

ដូច្នេះលេខ Code ខាងក្រៅពិនិត្យ instanceof ReadErrorហើយនោះជាវា។ មិនចាំបាច់រាយបញ្ជីប្រភេទកំហុសដែលអាចកើតមានទាំងអស់។ វិធីសាស្ត្រត្រូវបានគេហៅថា "wrapping exceptions" ពីព្រោះយើងយកការលើកលែង "low level" ហើយ "wrap" ពួកវាទៅក្នុង ReadErrorនោះមានលក្ខណៈអរូបីជាង។ វាត្រូវបានគេប្រើយ៉ាងទូលំទូលាយក្នុងការសរសេរកម្មវិធីតម្រង់ទិសObject។

៦.៣ លំហាត់

១. ប្រៀបធៀបបំណែក Code ទាំងពីរ។

- ទីមួយប្រើ finallyដើម្បីប្រតិបត្តិ Code បន្ទាប់ពី try...catch៖

```
try {
  work work
} catch (err) {
  handle errors
} finally {
  cleanup the working space
}
```

- បំណែកទីពីរដាក់ការសម្អាតភ្លាមៗបន្ទាប់ពី try...catch៖

```
try {
  work work
} catch (err) {
  handle errors
```

```
}

cleanup the working space
```

យើងពិតជាត្រូវការការសម្អាតបន្ទាប់ពីការងារ មិនថាមានកំហុសឬអត់នោះទេ។

២. តើមានអត្ថប្រយោជន៍នៅទីនេះក្នុងការប្រើប្រាស់ finallyឬបំណែក Code ទាំងពីរស្មើគ្នា ? បើមានអត្ថប្រយោជន៍បែបនេះ សូមលើកឧទាហរណ៍មួយពេលវាសំខាន់។

៣. បង្កើតClass FormatErrorដែលទទួលមរតកពី រូបមន្តErrorClassដែលភ្ជាប់មកជាមួយ។ វាគួរតែគាំទ្រ message, nameនិង stackProperty ។

ឧទាហរណ៍នៃការប្រើប្រាស់៖

```
let err = new FormatError("formatting error");

alert( err.message ); // formatting error
alert( err.name ); // FormatError
alert( err.stack ); // stack

alert( err instanceof FormatError ); // true
alert( err instanceof រូបមន្តError ); // true (because inherits from រូបមន្ត
Error)
```

សេចក្តីសន្និដ្ឋានជំពូកទី ៤៖ ការអនុវត្តន៍ JavaScript

ជំពូកនេះគ្របដណ្តប់លើអនុគមន៍ Functions, Objects, Arrays, ការគ្រប់គ្រង DOM, កម្មវិធី Asynchronous និងការគ្រប់គ្រងកំហុស (Error Handling)។

មេរៀនទី ១៖ អនុគមន៍ (Functions)

មេរៀននេះរៀបរាប់អំពី អនុគមន៍ (Functions) ដែលជាសមាសធាតុគ្រឹះសម្រាប់រៀបចំ និងគ្រប់គ្រងកូដ JavaScript។

- ប្រភេទនៃអនុគមន៍
  - Function Declarations: វិធីសាមញ្ញបំផុតក្នុងការប្រកាសអនុគមន៍ ដែលត្រូវបាន hoisted (អាចហៅបានមុនពេលប្រកាស)។
  - Function Expressions: កំណត់អនុគមន៍ទៅអថេរ។ ពួកគេ មិនត្រូវបាន hoisted ទេ។
  - Arrow Functions (ES6): ផ្តល់នូវ រូបមន្ត សង្ខេប និងមិនមានបរិបទ this ផ្ទាល់ខ្លួនទេ។
  - Generator Functions: ប្រើ yield ដែលអនុញ្ញាតឱ្យអនុគមន៍ផ្អាក និងបន្តការប្រតិបត្តិ។
  - Asynchronous Functions: ប្រើ async និង await ដើម្បីគ្រប់គ្រង Promises ប្រកបដោយភាពងាយស្រួល។
- Scope និង Closures
  - Scope: កំណត់លទ្ធភាពប្រើប្រាស់នៃអថេរ (variables) និងអនុគមន៍នៅក្នុងកូដ។
  - Closures: កើតឡើងនៅពេលដែលអនុគមន៍ខាងក្នុង (inner function) រក្សាការចូលប្រើអថេរពីអនុគមន៍ខាងក្រៅ (outer function) សូម្បីតែបន្ទាប់ពីអនុគមន៍ខាងក្រៅបានបញ្ចប់ការប្រតិបត្តិ។ Closures មានប្រយោជន៍សម្រាប់ការបង្កើតអថេរឯកជន (private variables) ។
- ការគ្រប់គ្រង Arguments និង Return Values
  - Default Parameters: អនុញ្ញាតឱ្យកំណត់តម្លៃលំនាំដើមសម្រាប់ parameters។
  - Rest Parameters (...): ប្រើដើម្បីតំណាងឱ្យចំនួន arguments មិនកំណត់ជា array។

មេរៀនទី ២៖ Objects

មេរៀននេះផ្តោតលើ Objects ដែលជាប្រភេទទិន្នន័យគ្រឹះសម្រាប់តំណាងឱ្យបណ្តុំនៃគូ key-value។

- ការបង្កើត Objects
  - Object Literal រូបមន្ត: ប្រើ {} ដូចជា ។

- Constructor Function: ប្រើអនុគមន៍ជា blueprint រួមជាមួយពាក្យគន្លឹះ new។
- Object.create( ): បង្កើត object ថ្មីដោយផ្អែកលើ prototype object ដែលបានផ្តល់ឱ្យ។

- ការចូលប្រើ និង Iterating

Properties អាចចូលប្រើបានដោយប្រើ Dot Notation (.) ឬ Bracket Notation ([])។ ការធ្វើ Iteration លើ properties អាចប្រើវិធីសាស្ត្រដូចជា for...in loop, Object.keys( ), Object.values( ), និង Object.entries( ) ។

មេរៀនទី ៣៖ អារេ (Arrays)

មេរៀននេះពន្យល់អំពី អារេ (Arrays) ដែលជារចនាសម្ព័ន្ធទិន្នន័យសម្រាប់រក្សាទុកបណ្តុំនៃតម្លៃតាមលំដាប់លំដោយ។

- ការប្រកាស និងការចូលប្រើ

អារេត្រូវបានប្រកាសដោយប្រើ [] (ឧ. let fruits = [Apple, Orange];)។ ធាតុត្រូវបានចូលប្រើតាមរយៈសូចនាករ (Index) ដោយចាប់ផ្តើមពីសូន្យ (0) (ឧ. fruits[0]) ឬប្រើ arr.at( ) សម្រាប់សន្ទស្សន៍អវិជ្ជមាន។

- វិធីសាស្ត្រសំខាន់ៗ

- push( ) / pop( ): បន្ថែម/ដកធាតុពី ចុងបញ្ចប់ នៃអារេ។
- shift( ) / unshift( ): ដក/បន្ថែមធាតុទៅ ដើមដំបូង នៃអារេ (ដែលយឺតជាង push/pop)។
- for...of loop: វិធីសាស្ត្រដែលត្រូវបានណែនាំសម្រាប់ការធ្វើ iteration លើ elements របស់អារេ។
- Array Methods ផ្សេងទៀត: រួមមាន concat( ), forEach( ), sort( ), slice( ), splice( ), indexOf( ), includes( ), toString( ), និង join( )។

មេរៀនទី៤៖ ការគ្រប់គ្រង DOM (Document Object Model)

មេរៀននេះពន្យល់អំពី DOM (Document Object Model) ដែលជាចំណុចប្រទាក់សម្រាប់ JavaScript ធ្វើអន្តរកម្ម និងកែប្រែឯកសារគេហទំព័រ (HTML) ក្នុងទម្រង់ជា មែកធាង នៃ Nodes។

- ការចូលប្រើ និងការកែប្រែធាតុ

- ការចូលប្រើធាតុ: ប្រើ methods ដូចជា getElementById( ), querySelector( ), និង querySelectorAll( ) ដើម្បីជ្រើសរើស elements។
- ការកែប្រែធាតុ: ប្រើ textContent (សម្រាប់អត្ថបទសុទ្ធ), innerHTML (សម្រាប់ខ្លឹមសារ HTML), setAttribute( ) (សម្រាប់ attributes) និង element.style (សម្រាប់ CSS styles)

- ការបន្ថែម និងលុបធាតុ

អ្នកអាចបង្កើតធាតុថ្មីដោយប្រើ document.createElement( ) ហើយបន្ថែម ឬលុបពួកវាដោយប្រើ appendChild( ) និង removeChild( )។

- Event Handling

Event Handling អនុញ្ញាតឱ្យអ្នកឆ្លើយតបទៅនឹងសកម្មភាពរបស់អ្នកប្រើប្រាស់ (ដូចជាការចុច) ដោយប្រើ addEventListener( )។

មេរៀនទី ៥៖ កម្មវិធី Asynchronous (Asynchronous Programming)

មេរៀននេះណែនាំអំពី Asynchronous Programming ដែលអនុញ្ញាតឱ្យកម្មវិធីដំណើរការប្រតិបត្តិការដែលត្រូវការពេលវេលា (non-blocking operations) ដោយប្រើ Event Loop។

- Promises

Promises តំណាងឱ្យលទ្ធផលចុងក្រោយនៃប្រតិបត្តិការ Asynchronous ហើយដោះស្រាយបញ្ហា Callback Hell។

  - .then( ) ត្រូវបានប្រើសម្រាប់លទ្ធផលជោគជ័យ។
  - .catch( ) ត្រូវបានប្រើសម្រាប់កំហុស។

- Promise APIs

JavaScript ផ្តល់នូវ methods សម្រាប់គ្រប់គ្រង Promises ជាច្រើន៖

  - Promise.all( ): រង់ចាំរហូតដល់ Promises ទាំងអស់ជោគជ័យ។
  - Promise.allSettled( ): រង់ចាំ Promises ទាំងអស់បញ្ចប់ (ជោគជ័យ ឬបរាជ័យ)។
  - Promise.race( ) និង Promise.any( ): ត្រឡប់លទ្ធផលរបស់ Promise ទីមួយដែលដោះស្រាយ។

- Async/Await

async/await គឺជា រូបមន្ត ដែលធ្វើឱ្យការងារជាមួយ Promises កាន់តែងាយស្រួល និងអាចអានបាន។

  - async: កំណត់អនុគមន៍ Asynchronous។
  - await: ផ្អាកការប្រតិបត្តិអនុគមន៍រហូតដល់ Promise ដោះស្រាយ។

- Fetch API

Fetch API គឺជាវិធីសាស្ត្រទំនើបសម្រាប់ធ្វើ network requests (ឧទាហរណ៍ ការហៅ API)។

  - មុខងារ fetch( ) ត្រឡប់ Promise ដែលដោះស្រាយទៅជា Response object។

- ការគ្រប់គ្រងកំហុស: Fetch API បដិសេធ (reject) Promise តែនៅពេលមាន Network Error ទេ ដូច្នេះអ្នកត្រូវពិនិត្យមើល response.ok ដោយដៃសម្រាប់ HTTP errors (ដូចជា 404)។

មេរៀនទី ៦៖ ការគ្រប់គ្រងកំហុស (Error Handling)

មេរៀននេះពន្យល់ពីរបៀបដែល JavaScript គ្រប់គ្រងកំហុសដែលកើតឡើងក្នុងពេលប្រតិបត្តិ (runtime errors) ដើម្បីការពារកម្មវិធីពីការឈប់ដំណើរការ។

- ការប្រើប្រាស់ try...catch

try...catch ត្រូវបានប្រើដើម្បីចាប់ និងដោះស្រាយកំហុសនៅក្នុងប្លុក try។ ប្រសិនបើកំហុសកើតឡើង ការប្រតិបត្តិផ្លាស់ទីទៅប្លុក catch(err)។

- Throw និង Rethrowing
  - throw: ប្រើដើម្បីបង្កើតកំហុសដោយចេតនា (ឧ. នៅពេលទិន្នន័យមិនត្រឹមត្រូវ)។
  - Rethrowing: អនុញ្ញាតឱ្យប្លុក catch គ្រប់គ្រងតែកំហុសដែលវាស្គាល់ ហើយបោះចោលកំហុសដែលមិនបានរំពឹងទុកឡើងវិញ។
- Try...Catch...Finally

ប្លុក finally ត្រូវបានបន្ថែមទៅ try...catch ហើយដំណើរការក្នុងគ្រប់ករណីទាំងអស់ (ទាំងជោគជ័យ ឬបរាជ័យ)។ វាត្រូវបានប្រើជាចម្បងសម្រាប់ការសម្អាត (cleanup) ធនធាន។

- Custom Error Classes និង Wrapping Exceptions

អ្នកអាចបង្កើត Class កំហុសផ្ទាល់ខ្លួន ដោយពង្រីក Class Error របស់ JavaScript ដើម្បីបង្កើតកំហុសដែលឆ្លុះបញ្ចាំងពីលក្ខណៈជាក់លាក់នៃកម្មវិធីរបស់អ្នក។ Wrapping Exceptions គឺជាបច្ចេកទេសនៃការចាប់កំហុសកម្រិតទាប ហើយបោះចោលកំហុសកម្រិតខ្ពស់ជំនួសវិញ។

ជំពូកទី ៥៖ JavaScript កម្រិតអ្នកជំនាញ

មេរៀនទី ១៖ Generators, advanced iteration

១.១ Generators

មុខងារធម្មតា ត្រឡប់តម្លៃតែមួយ តម្លៃតែមួយ (ឬគ្មានអ្វី)។ Generators អាចត្រឡប់ ("yield") តម្លៃច្រើន មួយបន្ទាប់ពីមួយផ្សេងទៀត តាមតម្រូវការ។ ពួកវាដំណើរការបានល្អណាស់ជាមួយ iterables ដែលអនុញ្ញាតឱ្យបង្កើតស្ទ្រីមទិន្នន័យយ៉ាងងាយស្រួល។

១.១.១ មុខងារ៖ Generator (Generator functions)

ដើម្បីបង្កើត Generator យើងត្រូវការរចនាសម្ព័ន្ធរូបមន្តពិសេស៖ function* ដែលគេហៅថា "Generator function"។

ដូចនេះ៖

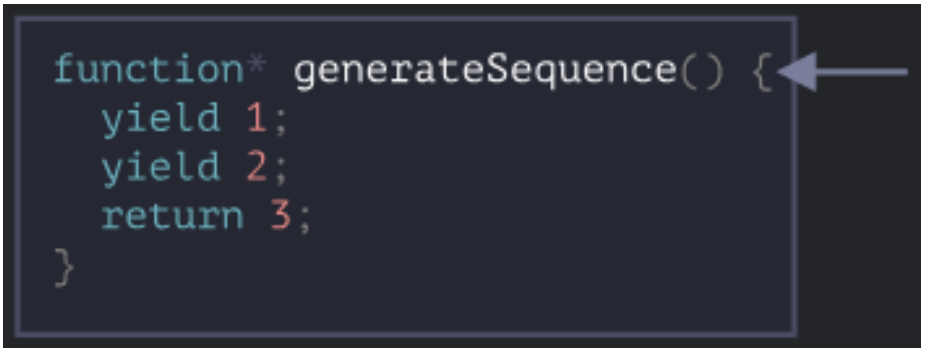
```
function* generateSequence() {
  yield 1;
  yield 2;
  return 3;
}
```

មុខងាររបស់ Generator មានឥរិយាបទខុសពីមុខងារធម្មតា។ នៅពេលដែលមុខងារបែបនេះត្រូវបានហៅ វាមិនដំណើរការ Code របស់វាទេ។ ជំនួសមកវិញ វាត្រឡប់Objectពិសេសមួយ ដែលហៅថា "generator object " ដើម្បីគ្រប់គ្រងការប្រតិបត្តិ។

នៅទីនេះ សូមមើល៖

```
function* generateSequence() {
  yield 1;
  yield 2;
  return 3;
}

// "generator function" creates "generator object"
let generator = generateSequence();
alert(generator); // [object Generator]
```

ការប្រតិបត្តិ Code មុខងារមិនទាន់បានចាប់ផ្តើមនៅឡើយទេ៖



រូបភាព 16៖ Generator functions

វិធីសាស្ត្រសំខាន់នៃ Generator គឺ next( )។ នៅពេលហៅមក វាដំណើរការការប្រតិបត្តិរហូតដល់ yield <value>សេចក្តីថ្លែងការណ៍ដែលនៅជិតបំផុត (valueអាចត្រូវបានលុបចោល បន្ទាប់មកវាជា undefined)។ បន្ទាប់មកការប្រតិបត្តិមុខងារផ្អាក ហើយលទ្ធផល valueត្រូវបានត្រឡប់ទៅលេខ Code ខាងក្រៅ។ លទ្ធផល next( )គឺតែងតែជាObjectដែលមានProperty ពីរ៖

- value៖ តម្លៃដែលបានទទួល។
- done៖ trueប្រសិនបើ Code មុខងារបានបញ្ចប់ បើមិនដូច្នោះទេ false.

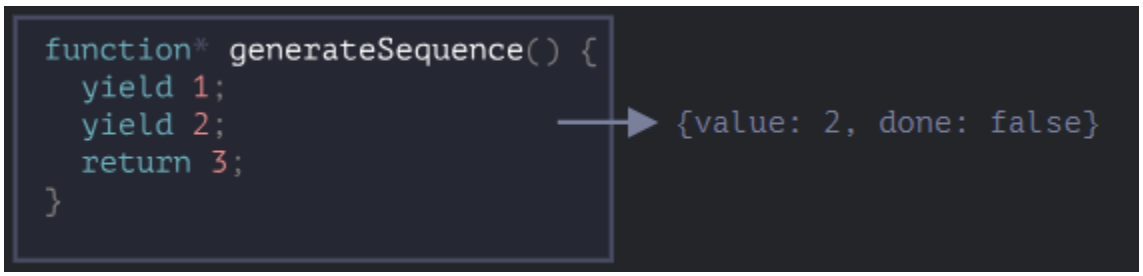
ជាឧទាហរណ៍ នៅទីនេះយើងបង្កើតម៉ាស៊ីនភ្លើង ហើយទទួលបានតម្លៃទិន្នផលដំបូងរបស់វា៖

```
function* generateSequence() {
  yield 1;
  yield 2;
  return 3;
}

let generator = generateSequence();

let one = generator.next();

alert(JSON.stringify(one)); // {value: 1, done: false}
```

មកដល់ពេលនេះ យើងទទួលបានតម្លៃដំបូងតែប៉ុណ្ណោះ ហើយការប្រតិបត្តិមុខងារគឺស្ថិតនៅលើបន្ទាត់ទីពីរ៖

គោះហៅ generator.next( )ម្តងទៀត។ វាបន្តការប្រតិបត្តិ Code ហើយត្រឡប់បន្ទាប់ទៀត yield៖

```
let two = generator.next();

alert(JSON.stringify(two)); // {value: 2, done: false}
```



រូបភាព 17៖ generator.next( )

ហើយប្រសិនបើយើងហៅវាជាលើកទីបី ការប្រតិបត្តិឈានដល់ return statement ដែលបញ្ចប់មុខងារ៖

```
let three = generator.next();

alert(JSON.stringify(three)); // {value: 3, done: true}
```

ឥឡូវនេះ generator រួចរាល់។ យើងគួរតែមើលវាពី done៖trueនិងដំណើរការ value៖3ជាលទ្ធផលចុងក្រោយ។ ការហៅថ្មីដើម្បី generator.next( )លែងមានន័យទៀតហើយ។ បើយើងធ្វើវា គេត្រឡប់Objectដដែល៖ {done៖ true}.

- function* f(…)ឬ function *f(…)?

រូបមន្ត ទាំងពីរគឺត្រឹមត្រូវ។ ប៉ុន្តែជាធម្មតា រូបមន្ត ទីមួយត្រូវបានគេពេញចិត្ត ដោយសារផ្កាយ * បង្ហាញថាវាជាមុខងារ Generatorវាពិពណ៌នាអំពីប្រភេទ មិនមែនជាឈ្មោះ ដូច្នេះវាគួរតែនៅជាប់នឹង functionពាក្យគន្លឹះ។

១.១.២ Generators អាចធ្លាស់ម្តងបាន (Generators are iterable)

ដូចដែលអ្នកប្រហែលជាបានទាយរួចហើយ ក្រឡេកមើល next( )វិធីសាស្ត្រ នោះ Generator គឺ អាចប្រើវាបាន ។

យើងអាចត្រួតលើតម្លៃរបស់វាដោយប្រើ for..of៖

```
function* generateSequence() {
  yield 1;
  yield 2;
  return 3;
}

let generator = generateSequence();

for(let value of generator) {
  alert(value); // 1, then 2 }
```

មើលទៅស្អាតជាងការហៅ .next( ).valueមែនទេ ?

...ប៉ុន្តែសូមចំណាំ៖ ឧទាហរណ៍ខាងលើបង្ហាញ ហើយ 1បន្ទាប់មក 2នោះហើយជាទាំងអស់។ វាមិនបង្ហាញទេ 3! វាដោយសារតែ for..ofការធ្វើឡើងវិញមិនអើពើចុងក្រោយ valueគឺនៅពេលណា done៖ true. ដូច្នេះ ប្រសិនបើយើងចង់ឱ្យលទ្ធផលទាំងអស់ត្រូវបានបង្ហាញដោយ for..ofយើងត្រូវបញ្ជូនពួកគេមកវិញជាមួយ yield៖

```
function* generateSequence() {
  yield 1;
  yield 2;
  yield 3;
}

let generator = generateSequence();

for(let value of generator) {
  alert(value); // 1, then 2, then 3
}
```

ដោយសារ Generator អាចប្រើវាវិញបាន យើងអាចហៅមុខងារដែលទាក់ទងទាំងអស់ ឧ. រូបមន្តរីករាលដាល ...៖

```
function* generateSequence() {
  yield 1;
  yield 2;
  yield 3;
}

let sequence = [0, ...generateSequence()];

alert(sequence); // 0, 1, 2, 3
```

នៅក្នុង Code ខាងលើ ...generateSequence( )បង្វែរObjectបង្កើតដែលអាចប្រើវាបានទៅជាArrayនៃធាតុ (អានបន្ថែមអំពីរូបមន្តរីករាលដាលក្នុងជំពូក ប៉ារ៉ាម៉ែត្រសម្រាក និងរូបមន្តរីករាលដាល)

១.១.៣ ការប្រើប្រាស់ generators (Using generators for iterables)

មួយរយៈមុននេះ នៅក្នុងជំពូក Iterables យើងបានបង្កើត rangeObjectដែលអាចផ្លាស់ប្តូរបានដែលត្រឡប់តម្លៃ from..to។

នៅទីនេះ ចូរយើងចងចាំ Code ៖

```
let range = {
  from: 1,
  to: 5,

  // for..of range calls this method once in the very beginning
  [Symbol.iterator]() {
    // ...it returns the iterator object:
    // onward, for..of works only with that object, asking it for next values
    return {
      current: this.from,
      last: this.to,

      // next() is called on each iteration by the for..of loop
      next() {
        // it should return the value as an object {done:.., value :...}
        if (this.current <= this.last) {
          return { done: false, value: this.current++ };
        } else {
          return { done: true };
        }
      }
    };
  }
};

// iteration over range returns numbers from range.from to range.to
alert([...range]); // 1,2,3,4,5
```

យើងអាចប្រើមុខងារ generator សម្រាប់ការធ្វើឡើងវិញដោយផ្តល់វាជា Symbol.iterator។ នេះដូចគ្នា rangeប៉ុន្តែបង្រួមច្រើនជាង

```
let range = {
  from: 1,
  to: 5,

  *[Symbol.iterator]() { // a shorthand for [Symbol.iterator]: function*()
```

```
    for(let value = this.from; value <= this.to; value++) {
      yield value;
    }
  }
};

alert( [...range] ); // 1,2,3,4,5
```

វាដំណើរការ ពីព្រោះ range[Symbol.iterator]( )ឥឡូវនេះត្រឡប់ម៉ាស៊ីនភ្លើង ហើយវិធីសាស្ត្រ generator គឺពិតជាអ្វីដែល for..ofរំពឹងទុក៖

- វាមាន .next( )វិធីសាស្ត្រ មួយ។
- ដែលត្រឡប់តម្លៃក្នុងទម្រង់{value៖ ..., done៖ true/false}

នោះមិនមែនជារឿងចៃដន្យទេ ជាការពិត។ generator ងត្រូវបានបន្ថែមទៅភាសា JavaScript ជាមួយនឹងអ្នកសរសេរឡើងវិញក្នុងចិត្ត ដើម្បីអនុវត្តវាយ៉ាងងាយស្រួល។ បំរែបំរួលជាមួយ generator គឺមានភាពសង្ខេបជាង Code ដើមដែលអាចប្តូរបាន rangeហើយរក្សាមុខងារដដែល។

- Generator អាចបង្កើតតម្លៃជារៀងរហូត

នៅក្នុងឧទាហរណ៍ខាងលើ យើងបានបង្កើតលំដាប់កំណត់ ប៉ុន្តែយើងក៏អាចបង្កើត generator ដែលផ្តល់តម្លៃជារៀងរហូតផងដែរ។ ជាឧទាហរណ៍ លំដាប់មិនបញ្ចប់នៃលេខចៃដន្យ។

នោះប្រាកដជាត្រូវការ break(ឬ return) នៅក្នុង for..of generator បែបនេះ។ បើមិនដូច្នោះទេ រង្វិលជុំនឹងកើតឡើងជារៀងរហូត ហើយព្យួរ។

១.១.៤ សមាសភាព Generator (Generator composition)

សមាសភាព Generator គឺជាលក្ខណៈពិសេសរបស់ម៉ាស៊ីនភ្លើងដែលអនុញ្ញាតឱ្យ "embed" Generatorដោយតម្លាភាពនៅក្នុងគ្នាទៅវិញទៅមក។

ឧទាហរណ៍ យើងមានមុខងារដែលបង្កើតលំដាប់លេខ៖

```
function* generateSequence(start, end) {
  for (let i = start; i <= end; i++) yield i;
}
```

ឥឡូវនេះយើងចង់ប្រើវាឡើងវិញដើម្បីបង្កើតលំដាប់ស្មុគស្មាញជាងនេះ៖

- ទីមួយខ្ទង់ 0..9(ជាមួយលេខ Code តួអក្សរ 48...57)
- អមដោយអក្សរធំ A..Z(Code តួអក្សរ 65...90)
- បន្តដោយអក្សរតូច a..z(Code តួអក្សរ 97...122)

យើងអាចប្រើលំដាប់នេះ ឧ. ដើម្បីបង្កើតពាក្យសម្ងាត់ដោយជ្រើសរើសតួអក្សរពីវា (អាចបន្ថែមតួអក្សររូបមន្តផងដែរ) ប៉ុន្តែសូមបង្កើតវាជាមុនសិន។

នៅក្នុងមុខងារធម្មតា ដើម្បីផ្សំលទ្ធផលពីមុខងារផ្សេងទៀតជាច្រើន យើងហៅពួកវា រក្សាទុកលទ្ធផល ហើយបន្ទាប់មកចូលរួមនៅចុងបញ្ចប់។ សម្រាប់ Generator មាន yield* រូបមន្តពិសេសមួយដើម្បី "embed" (តែង) Generator មួយទៅម៉ាស៊ីនមួយទៀត។

Generator ដែលផ្សំឡើង៖

```
function* generateSequence(start, end) {
  for (let i = start; i <= end; i++) yield i;
}

function* generatePasswordCodes() {

  // 0..9
  yield* generateSequence(48, 57);

  // A..Z
  yield* generateSequence(65, 90);

  // a..z
  yield* generateSequence(97, 122);

}

let str = '';

for(let code of generatePasswordCodes()) {
  str += String.fromCharCode(code);
}

alert(str); // 0..9A..Za..z
```

ការណែនាំទិន្នផល* ផ្ទេរការប្រតិបត្តិទៅ generator ផ្សេងទៀត។ ពាក្យនេះមានន័យថា ទិន្នផល* gen កើតឡើងលើ generator gen ហើយបញ្ជូនទិន្នផលរបស់វាទៅខាងក្រៅដោយតម្លាភាព។ ដូចជាប្រសិនបើតម្លៃត្រូវបានផ្តល់ដោយ generator ខាងក្រៅ។

លទ្ធផលគឺដូចគ្នានឹងការដាក់បញ្ចូល Code ពី generator ដែលបានដាក់បញ្ចូលគ្នា៖

```
function* generateSequence(start, end) {
  for (let i = start; i <= end; i++) yield i;
}
```

```
function* generateAlphaNum() {

  // yield* generateSequence(48, 57);
  for (let i = 48; i <= 57; i++) yield i;

  // yield* generateSequence(65, 90);
  for (let i = 65; i <= 90; i++) yield i;

  // yield* generateSequence(97, 122);
  for (let i = 97; i <= 122; i++) yield i;

}

let str = '';

for(let code of generateAlphaNum()) {
  str += String.fromCharCode(code);
}

alert(str); // 0..9A..Za..z
```

សមាសភាព generator គឺជាវិធីធម្មជាតិដើម្បីបញ្ចូលលំហូរនៃ generator មួយទៅក្នុងម៉ាស៊ីនមួយទៀត។ វាមិនប្រើអង្គចងចាំបន្ថែមដើម្បីរក្សាទុកលទ្ធផលកម្រិតមធ្យមទេ។

១.១.៥ "ទិន្នផល" គឺជាផ្លូវពីរ (“yield” is a two-way street)

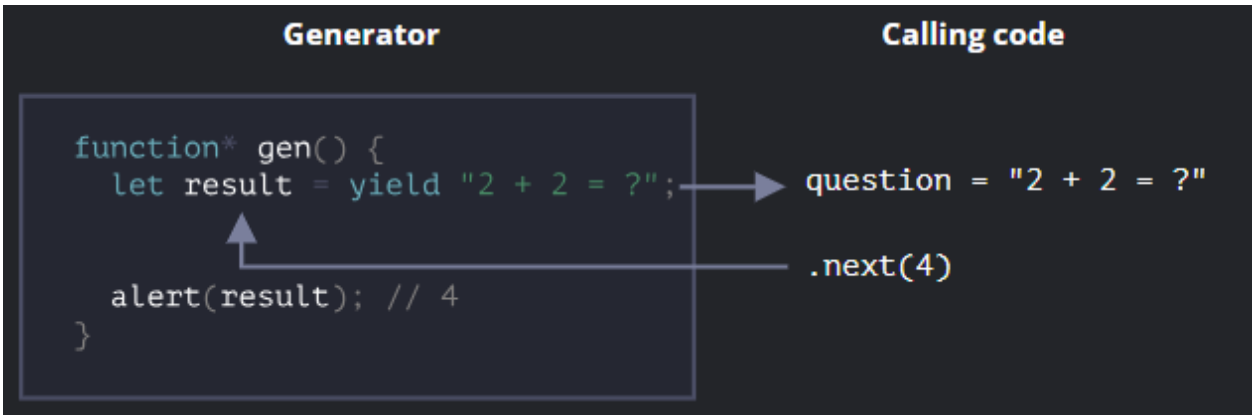
រហូតមកដល់ពេលនេះ generator គឺស្រដៀងទៅនឹងObjectដែលអាចផ្លាស់ប្តូរបាន ដោយមានរូបមន្តពិសេសដើម្បីបង្កើតតម្លៃ។ ប៉ុន្តែតាមពិត ពួកវាមានថាមពលខ្លាំងជាង និងអាចបត់បែនបាន។ នោះហើយ yieldជាដោយសារតែផ្លូវពីរផ្លូវ៖ វាមិនត្រឹមតែបញ្ជូនលទ្ធផលទៅខាងក្រៅប៉ុណ្ណោះទេ ប៉ុន្តែវាក៏អាចហុចតម្លៃនៅខាងក្នុងម៉ាស៊ីនភ្លើងផងដែរ។

ដើម្បីធ្វើដូច្នេះ យើងគួរតែហៅទូរសព្ទ generator.next(arg)ដោយមានអំណះអំណាង។ Argumentនោះក្លាយជាលទ្ធផលនៃ yield.

តោះមើលឧទាហរណ៍៖

```
function* gen() {
  // Pass a question to the outer code and wait for an answer
  let result = yield "2 + 2 = ?"; // (*)

  alert(result);
}
```

```
let generator = gen();

let question = generator.next().value; // <-- yield returns the value

generator.next(4); // --> pass the result into the generator
```



រូបភាព 18៖ "yield" is a two-way street

1. ការហៅទូរសព្ទលើកដំបូង generator.next( )គួរតែត្រូវបានធ្វើឡើងដោយគ្មានអំណះអំណាង (Argumentមិនត្រូវបានអើពើប្រសិនបើឆ្លងកាត់) ។វាចាប់ផ្តើមការប្រតិបត្តិនិងត្រឡប់លទ្ធផលនៃការដំបូង yield "2+2=?". នៅចំណុចនេះ ម៉ាស៊ីនភ្លើងផ្អាកការប្រតិបត្តិ ខណៈពេលដែលកំពុងស្ថិតនៅលើបន្ទាត់ (*)។
2. បន្ទាប់មកដូចដែលបានបង្ហាញនៅរូបភាពខាងលើ លទ្ធផលនៃ yieldការទទួលបានចូលទៅក្នុង questionអថេរនៅក្នុងលេខ Code ហៅទូរសព្ទ។
3. បើក generator.next(4) generator ចាប់ផ្តើមឡើងវិញ ហើយ 4ចូលជាលទ្ធផល៖ let result = 4។

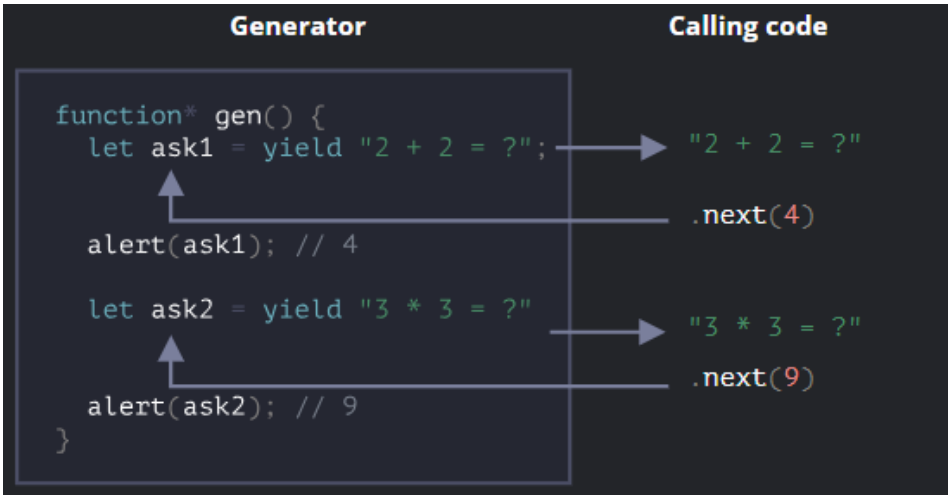
សូមចំណាំ៖ Code ខាងក្រៅមិនចាំបាច់ហៅភ្លាមៗទេ next(4)។ វាអាចត្រូវការពេលវេលា។ នោះមិនមែនជាបញ្ហាទេ៖ generator នឹងរង់ចាំ។

ឧទាហរណ៍៖

```
// resume the generator after some time
setTimeout(() => generator.next(4), 1000);
```

ដូចដែលយើងអាចមើលឃើញ មិនដូចមុខងារធម្មតាទេ Generator និង Code ហៅទូរសព្ទអាចផ្លាស់ប្តូរលទ្ធផលដោយឆ្លងកាត់តម្លៃក្នុង next/yield។ ដើម្បីធ្វើឱ្យអ្វីៗកាន់តែច្បាស់ នេះជាឧទាហរណ៍មួយទៀត ជាមួយនឹងការហៅទូរសព្ទកាន់តែច្រើន៖

```
function* gen() {
  let ask1 = yield "2 + 2 = ?";

  alert(ask1); // 4

  let ask2 = yield "3 * 3 = ?"

  alert(ask2); // 9
}

let generator = gen();

alert( generator.next().value ); // "2 + 2 = ?"

alert( generator.next(4).value ); // "3 * 3 = ?"

alert( generator.next(9).done ); // true
```



រូបភាព 19៖ Generator and Calling Code

1. ទីមួយ .next( )ចាប់ផ្តើមការប្រតិបត្តិ... វាឈានដល់ទីមួយ yield។
2. លទ្ធផលត្រូវបានត្រលប់ទៅ Code ខាងក្រៅ។
3. ទីពីរ .next(4)បញ្ជូន 4ត្រឡប់ទៅ generator វិញជាលទ្ធផលនៃទីមួយ yieldហើយបន្តការប្រតិបត្តិ។
4. ...វាឈានដល់ទីពីរ yieldដែលក្លាយជាលទ្ធផលនៃ Generator call ។
5. ទីបី next(9)ចូល 9ទៅក្នុង generator ដែលជាលទ្ធផលនៃទីពីរ yieldហើយបន្តការប្រតិបត្តិដែលឈានដល់ចុងបញ្ចប់នៃមុខងារដូច្នេះ done៖ true.

វាដូចជាល្បែង "ping-pong" ។ នីមួយៗ next(value)(មិនរាប់បញ្ចូលលេខទីមួយ) ឆ្លងកាត់តម្លៃមួយទៅក្នុងម៉ាស៊ីនបង្កើត ដែលក្លាយជាលទ្ធផលនៃចរន្ត yieldហើយបន្ទាប់មកទទួលបានលទ្ធផលនៃបន្ទាប់ yield។

១.១.៦ generator.throw

ដូចដែលយើងបានសង្កេតនៅក្នុងឧទាហរណ៍ខាងលើ Code ខាងក្រៅអាចបញ្ជូនតម្លៃមួយទៅក្នុង generator ដែលជាលទ្ធផលនៃ yield។

...ប៉ុន្តែវាក៏អាចចាប់ផ្តើម (throw) កំហុសនៅទីនោះផងដែរ។ នោះជាធម្មជាតិព្រោះកំហុសគឺជាលទ្ធផលមួយប្រភេទ។ ដើម្បីឆ្លងផុតកំហុសមួយ yieldយើងត្រូវហៅទូរស័ព្ទទៅ generator.throw(err). ក្នុងករណីនោះ errត្រូវបានបោះចោលនៅក្នុងជួរជាមួយនោះ yield។ ឧទាហរណ៍ នៅទីនេះ ទិន្នផល "2 + 2 = ?"នាំទៅរកកំហុស៖

```
function* gen() {
  try {
    let result = yield "2 + 2 = ?"; // (1)

    alert("The execution does not reach here, because the exception
is thrown above");
  } catch(e) {
    alert(e); // shows the error
  }
}

let generator = gen();

let question = generator.next().value;

generator.throw(new Error("The answer is not found in my database"));
// (2)
```

កំហុសដែលបានបោះចូលទៅក្នុង generator នៅបន្ទាត់ (2)នាំឱ្យមានករណីលើកលែងមួយ (1)ស្របតាម yield. នៅក្នុងឧទាហរណ៍ខាងលើ try..catchចាប់វាហើយបង្ហាញវា។ ប្រសិនបើយើងមិនចាប់វាទេ ដូចករណីលើកលែងណាមួយដែរ វា "falls out" generatorចូលទៅក្នុងលេខ Code ហៅទូរសព្ទ

បន្ទាត់បច្ចុប្បន្ននៃ Code ហៅគឺជាបន្ទាត់ដែល generator.throwមានស្លាកជា (2). ដូច្នេះយើងអាចចាប់វាបាននៅទីនេះ៖

```
function* generate() {
```

```
  let result = yield "2 + 2 = ?"; // Error in this line
}

let generator = generate();

let question = generator.next().value;

try {
   generator.throw(new Error("The answer is not found in my
database"));
} catch(e) {
  alert(e); // shows the error
}
```

ប្រសិនបើយើងមិនចាប់កំហុសនៅទីនោះទេ នោះដូចជាធម្មតា វាធ្លាក់ចូលទៅលេខ Code ហៅក្រៅ (បើមាន) ហើយបើមិនចាប់បាន នោះនឹងសម្លាប់ស្គ្រីប។

១.១.៧ generator.return

generator.return(value)បញ្ចប់ការប្រតិបត្តិ generator ហើយត្រឡប់បានផ្តល់ឱ្យវិញ value

```
function* gen() {
  yield 1;
  yield 2;
  yield 3;
}

const g = gen();

g.next();        // { value: 1, done: false }
g.return('foo'); // { value: "foo", done: true }
g.next();        // { value: undefined, done: true }
```

ប្រសិនបើយើងប្រើម្តងទៀត generator.return( )នៅក្នុងម៉ាស៊ីនដែលបានបញ្ចប់ វានឹងត្រឡប់តម្លៃនោះម្តងទៀត (MDN)។

ជារឿយៗយើងមិនប្រើវាទេ ព្រោះភាគច្រើនយើងចង់ទទួលបានតម្លៃត្រឡប់មកវិញទាំងអស់ ប៉ុន្តែវាអាចមានប្រយោជន៍នៅពេលដែលយើងចង់បញ្ឈប់ម៉ាស៊ីនភ្លើងក្នុងលក្ខខណ្ឌជាក់លាក់មួយ។

១.២ Async iteration and generators

ការធ្វើឡើងវិញ Async អនុញ្ញាតឱ្យយើងធ្វើម្តងទៀតលើទិន្នន័យដែលមកដោយ Async តាមតម្រូវការ។ ឧទាហរណ៍ ដូចជានៅពេលយើងទាញយកអ្វីមួយជាដុំៗតាមបណ្តាញ។ ហើយ asyncs generator ធ្វើឱ្យវាកាន់តែងាយស្រួល។

សូមមើលឧទាហរណ៍សាមញ្ញជាមុនសិន ដើម្បីយល់រូបមន្ត ហើយបន្ទាប់មកពិនិត្យមើលករណីប្រើប្រាស់ក្នុងជីវិតពិត។

១.២.១ រំលឹកឡើងវិញ (Recall iterables)

ចូរយើងរំលឹកឡើងវិញនូវប្រធានបទអំពី Objectដែលអាចកើតមានបាន។ គំនិតគឺថាយើងមាន Objectមួយដូចជា rangeនៅទីនេះ៖

```
let range = {
  from: 1,
  to: 5
};
```

ហើយយើងចង់ប្រើ for..ofវង្វិលជុំនៅលើវាដូចជាfor(value of range) ដើម្បីយកតម្លៃពី 1ទៅ 5

ម្យ៉ាងទៀត យើងចង់បន្ថែម សមត្ថភាពធ្វើដដែលៗ ទៅObject។ វាអាចត្រូវបានអនុវត្តដោយប្រើវិធីសាស្ត្រពិសេសមួយដែលមានឈ្មោះ Symbol.iterator៖

វិធីសាស្ត្រនេះត្រូវបានហៅដោយ for..ofconstruct នៅពេលដែលវង្វិលជុំត្រូវបានចាប់ផ្តើម ហើយវាគួរតែត្រឡប់Objectមួយជាមួយនឹង nextវិធីសាស្ត្រ។ សម្រាប់ការធ្វើឡើងវិញនីមួយៗ next( ) វិធីសាស្ត្រត្រូវបានហៅសម្រាប់តម្លៃបន្ទាប់។

គួរតែ next( )ត្រឡប់តម្លៃក្នុងទម្រង់ {done៖ true/false, value៖<loop value>}ដែល done៖ trueមានន័យថាចុងបញ្ចប់នៃវង្វិលជុំ។ នេះជាការអនុវត្តសម្រាប់ iterable នេះ range៖

```
let range = {
  from: 1,
  to: 5,

  [Symbol.iterator]() { // called once, in the beginning of for..of
    return {
      current: this.from,
      last: this.to,

      next() { // called every iteration, to get the next value
        if (this.current <= this.last) {
```

```
          return { done: false, value: this.current++ };
        } else {
          return { done: true };
        }
      }
    };
  }
};

for(let value of range) {
  alert(value); // 1 then 2, then 3, then 4, then 5
}
```

ប្រសិនបើមានអ្វីមិនច្បាស់លាស់ សូមចូលទៅកាន់ជំពូក Iterables វាផ្តល់ព័ត៌មានលម្អិតទាំងអស់អំពី Iterables ធម្មតា។

១.២.២ Async iterables

ការធ្វើឡើងវិញ Async គឺត្រូវការពេលដែលតម្លៃមកជា Async ៖ បន្ទាប់ពី setTimeoutការពន្យារពេលមួយប្រភេទទៀត។ ករណីទូទៅបំផុតគឺថាObjectត្រូវការដើម្បីធ្វើសំណើបណ្តាញដើម្បីផ្តល់តម្លៃបន្ទាប់ យើងនឹងឃើញឧទាហរណ៍ជាក់ស្តែងរបស់វា។

ដើម្បីធ្វើឱ្យObjectអាចធ្វើដំណើរបានដោយអសមកាល៖

1. ប្រើ Symbol.asyncIteratorជំនួស Symbol.iterator។
2. វិធី next( )សាស្ត្រគួរតែត្រឡប់ការPromiseមួយ (ដើម្បីបំពេញជាមួយនឹងតម្លៃបន្ទាប់) ។
    - ពាក្យគន្លឹះ asyncដោះស្រាយវា យើងអាចបង្កើតបានយ៉ាងសាមញ្ញ async next( )។
3. ដើម្បីធ្វើឡើងវិញលើObjectបែបនេះ យើងគួរប្រើ for await (let item of iterable)រង្វិលជុំ។
    - ចំណាំ awaitពាក្យ។

ជាឧទាហរណ៍ចាប់ផ្តើម យើងបង្កើត rangeObjectដែលអាចផ្លាស់ប្តូរបាន ស្រដៀងនឹងObject ពីមុន ប៉ុន្តែឥឡូវនេះវានឹងត្រឡប់តម្លៃអសមកាល មួយក្នុងមួយវិនាទី។

អ្វីដែលយើងត្រូវធ្វើគឺធ្វើការជំនួសមួយចំនួននៅក្នុង Code ខាងលើ៖

```
let range = {
  from: 1,
  to: 5,

  [Symbol.asyncIterator]() { // (1)
    return {
      current: this.from,
```

```
      last: this.to,

      async next() { // (2)

        // note: we can use "await" inside the async next:
        await new Promise(resolve => setTimeout(resolve, 1000)); // (3)

        if (this.current <= this.last) {
          return { done: false, value: this.current++ };
        } else {
          return { done: true };
        }
      }
    };
  }
};

(async () => {

  for await (let value of range) { // (4)
    alert(value); // 1,2,3,4,5
  }

})()
```

ដូចដែលយើងអាចមើលឃើញរចនាសម្ព័ន្ធគឺស្រដៀងទៅនឹង iterators ធម្មតា៖

1. ដើម្បីធ្វើឱ្យObjectអាចធ្វើវាបានដោយអសមកាល វាត្រូវតែមានវិធីសាស្ត្រមួយ Symbol.asyncIterator (1)។
2. វិធីសាស្ត្រនេះត្រូវតែត្រឡប់Objectជាមួយនឹង next( )វិធីសាស្ត្រត្រឡប់ការPromise (2)។
3. វិធី next( )សាស្ត្រមិនចាំបាច់ទេ asyncវាអាចជាវិធីសាស្ត្រធម្មតាដែលត្រឡប់ការPromise ប៉ុន្តែ asyncអនុញ្ញាតឱ្យយើងប្រើ awaitដូច្នេះងាយស្រួល។ នៅទីនេះយើងគ្រាន់តែពន្យារពេលមួយវិនាទី (3)។
4. ដើម្បីរំលឹកឡើងវិញ យើងប្រើ for await(let value of range) (4)ពោលគឺបន្ថែម "រង់ចាំ" បន្ទាប់ពី "សម្រាប់" ។ វាហៅ range[Symbol.asyncIterator]( )ម្តង ហើយបន្ទាប់មកវា next( )សម្រាប់តម្លៃ។

នេះគឺជាតារាងតូចមួយដែលមានភាពខុសគ្នា៖

	Iterators	Async iterators
Object method to provide iterator	Symbol.iterator	Symbol.asyncIterator
next( ) return value is	any value	Promise
to loop, use	for..of	for await..of

តារាង 7៖ Async iterables

- The spread រូបមន្ត ... doesn't work asynchronously

លក្ខណៈពិសេសដែលតម្រូវឱ្យអ្នកធ្វើសមកាលកម្មទៀងទាត់ មិនដំណើរការជាមួយអសមកាលទេ។ ឧទាហរណ៍៖ Spread រូបមន្តនឹងមិនដំណើរការទេ៖

នោះជាធម្មជាតិ ដូចដែលគេរំពឹងថានឹងរកឃើញ Symbol.iteratorមិនមែនទេ Symbol.Async Iterator។ វាក៏ជាករណីសម្រាប់ for..of៖ រូបមន្តដោយគ្មាន await តម្រូវការ Symbol.iterator។

១.២.៣ Recall generators

ឥឡូវនេះ យើងចូរ Recall generators ព្រោះវាអនុញ្ញាតឱ្យធ្វើ Code ដែលៗខ្លីជាង។ ភាគច្រើន នៅពេលដែលយើងចង់បង្កើតវា យើងនឹងប្រើ generators ។ សម្រាប់ភាពសាមញ្ញបំផុត ដោយលុបចោលនូវObjectសំខាន់ៗមួយចំនួន ពួកវាជា "មុខងារដែលបង្កើតតម្លៃ (yield)"។ ពួកវាត្រូវបានពន្យល់យ៉ាងលម្អិតនៅក្នុងជំពូក Generators ។

Generators ត្រូវបានដាក់ស្លាកជាមួយ function*(ចំណាំផ្កាយ) ហើយប្រើ yieldដើម្បីបង្កើតតម្លៃ បន្ទាប់មកយើងអាចប្រើ for..ofដើម្បីរង្វិលជុំលើពួកវា។

ឧទាហរណ៍នេះបង្កើតលំដាប់នៃតម្លៃពី startទៅ end៖

```
function* generateSequence(start, end) {
  for (let i = start; i <= end; i++) {
    yield i;
  }
}

for(let value of generateSequence(1, 5)) {
  alert(value); // 1, then 2, then 3, then 4, then 5
}
```

ដូចដែលយើងដឹងរួចមកហើយថា ដើម្បីធ្វើឱ្យObjectមួយអាចផ្លាស់ប្តូរបាន យើងត្រូវបន្ថែម Symbol.iteratorវាទៅវា។

```
let range = {
  from: 1,
  to: 5,
  [Symbol.iterator]() {
    return <object with next to make range iterable>
  }
}
```

ការអនុវត្តជាទូទៅ Symbol.iteratorគឺការបញ្ជូន generator មកវិញ វាធ្វើឱ្យ Code ខ្លីជាងមុន ដូចដែលអ្នកបានឃើញ៖

```
let range = {
  from: 1,
  to: 5,

   *[Symbol.iterator]() {  //  a  shorthand  for  [Symbol.iterator]:
function*()
    for(let value = this.from; value <= this.to; value++) {
      yield value;
    }
  }
};

for(let value of range) {
  alert(value); // 1, then 2, then 3, then 4, then 5
}
```

សូមមើលជំពូក Generators ប្រសិនបើអ្នកចង់បានព័ត៌មានលម្អិតបន្ថែម។នៅក្នុង Generators ធម្មតា យើងមិនអាចប្រើបានទេ await។ តម្លៃទាំងអស់ត្រូវតែមកក្នុងពេលដំណាលគ្នា តាមតម្រូវការដោយ for..ofសំណង់។ ចុះបើយើងចង់បង្កើតតម្លៃអសមកាល? ឧទាហរណ៍ពីសំណើបណ្តាញ។

ចូរប្តូរទៅ asynchronous generators ដើម្បីធ្វើឱ្យវាអាចធ្វើទៅបាន។

១.២.៤ Async generators (finally)

សម្រាប់កម្មវិធីជាក់ស្តែងភាគច្រើន នៅពេលដែលយើងចង់បង្កើតObjectដែលបង្កើតលំដាប់នៃតម្លៃអសមកាល យើងអាចប្រើ asynchronous generator ។

The រូបមន្ត is simple៖ ភ្ជាប់ function*ជាមួយ async. នោះធ្វើឱ្យ generator asynchronous ។ ហើយបន្ទាប់មកប្រើ for await (...)ដើម្បីរំលឹកវាដូចនេះ៖

```
async function* generateSequence(start, end) {

  for (let i = start; i <= end; i++) {

    // Wow, can use await!
    await new Promise(resolve => setTimeout(resolve, 1000));

    yield i;
  }

}

(async () => {

  let generator = generateSequence(1, 5);
  for await (let value of generator) {
    alert(value); // 1, then 2, then 3, then 4, then 5 (with delay
between)
  }

})();
```

ដោយសារ generator is asynchronous យើងអាចប្រើ awaitនៅខាងក្នុងវា ពឹងផ្អែកលើការ Promise អនុវត្តសំណើបណ្តាញជាដើម។

- Under-the-hood difference

តាមបច្ចេកទេស ប្រសិនបើអ្នកជាអ្នកអានកម្រិតខ្ពស់ដែលចង់ចាំព័ត៌មានលម្អិតអំពី generators វាមានភាពខុសគ្នាខាងក្នុង។ សម្រាប់ async generators generator.next( )វិធីសាស្ត្រគឺអសមកាល វាត្រឡប់ការPromise។

នៅក្នុង generator ធម្មតា យើងនឹងប្រើ result = generator.next( )ដើម្បីទទួលបានតម្លៃ។ នៅក្នុង async generator យើងគួរបន្ថែម awaitដូចនេះ៖

```
result = await generator.next(); // result = {value: ..., done:
true/false}
```

នោះហើយជាមូលហេតុដែលម៉ាស៊ីនភ្លើង async ដំណើរការជាមួយ for await...of.

១.២.៥ Async iterable range

Regular generators អាចប្រើ Symbol.iteratorដើម្បីធ្វើឲ្យ Code សរសេរឡើងវិញខ្លីជាង។ ស្រដៀងគ្នានឹងនោះ ម៉ាស៊ីនបង្កើតអសមកាលអាចត្រូវបានប្រើ Symbol.asyncIteratorដើម្បីអនុវត្ត asynchronous iteration ។

ឧទាហរណ៍ យើងអាចធ្វើឲ្យ rangeObjectបង្កើតតម្លៃអសមកាល ម្តងក្នុងមួយវិនាទី ដោយជំនួសសមកាលកម្ម Symbol.iteratorជាមួយអសមកាល Symbol.asyncIterator៖

```
let range = {
  from: 1,
  to: 5,

  // this line is same as [Symbol.asyncIterator]: async function*() {
  async *[Symbol.asyncIterator]() {
    for(let value = this.from; value <= this.to; value++) {

      // make a pause between values, wait for something
      await new Promise(resolve => setTimeout(resolve, 1000));

      yield value;
    }
  }
};

(async () => {

  for await (let value of range) {
    alert(value); // 1, then 2, then 3, then 4, then 5
  }

})();
```

ឥឡូវនេះតម្លៃមកជាមួយការពន្យាពេល 1 វិនាទីរវាងពួកវា។

សូមចំណាំ៖ តាមបច្ចេកទេស យើងអាចបន្ថែមទាំងពីរ Symbol.iteratorនិង Symbol.asyncIterատորទៅObject ដូច្នេះវាទាំង synchronously (for..of) និង asynchronously (for await..of) iterable។ ទោះជាយ៉ាងណា ក្នុងការអនុវត្ត វានឹងជារឿងចម្លែកដែលត្រូវធ្វើ។

១.២.៦ ឧទាហរណ៍ក្នុងជីវិតពិត៖ អ៊ីនធឺហ្វេសដែលបានបិទភ្ជាប់

រហូតមកដល់ពេលនេះ យើងបានឃើញឧទាហរណ៍ជាមូលដ្ឋាន ដើម្បីទទួលបានការយល់ដឹង។ ឥឡូវនេះ សូមពិនិត្យមើលករណីប្រើប្រាស់ក្នុងជីវិតពិត។

មានសេវាកម្មអនឡាញជាច្រើនដែលផ្តល់ទិន្នន័យ paginated ។ ឧទាហរណ៍ នៅពេលដែលយើងត្រូវការបញ្ជីអ្នកប្រើប្រាស់ សំណើមួយនឹងត្រឡប់ចំនួនដែលបានកំណត់ជាមុន (ឧ. អ្នកប្រើប្រាស់ 100) – "ទំព័រមួយ" ហើយផ្តល់ URL ទៅទំព័របន្ទាប់។

គំរូនេះគឺជារឿងធម្មតាណាស់។ វាមិនមែនអំពីអ្នកប្រើប្រាស់ទេ ប៉ុន្តែគ្រាន់តែអំពីអ្វីទាំងអស់។ ជាឧទាហរណ៍ GitHub អនុញ្ញាតឱ្យយើងទាញយកការប្រព្រឹត្តិក្នុងទម្រង់ដូចគ្នាដែលបិទភ្ជាប់៖

1. យើងគួរតែធ្វើការស្នើសុំ ក្នុង fetchទម្រង់https://api.github.com/repos/<repo>/commits
2. វាឆ្លើយតបជាមួយនឹង JSON នៃ 30 commits ហើយក៏ផ្តល់នូវតំណភ្ជាប់ទៅកាន់ទំព័របន្ទាប់នៅក្នុង Linkបឋមកថាផងដែរ។
3. បន្ទាប់មក យើងអាចប្រើតំណនោះសម្រាប់សំណើបន្ទាប់ ដើម្បីទទួលបានការប្តេជ្ញាចិត្តបន្ថែមទៀត។ល។

សម្រាប់ Code របស់យើង យើងចង់បានវិធីសាមញ្ញជាងនេះដើម្បីទទួលបានការPromise។ ចូរបង្កើតមុខងារ fetchCommits(repo)ដែលទទួលបានការPromiseសម្រាប់ពួកយើង ដោយធ្វើការស្នើសុំនៅពេលណាដែលចាំបាច់។ ហើយទុកឱ្យវាខ្វល់ពីគ្រប់ Object នៃការដាក់ទំព័រ។ សម្រាប់ពួកយើង វានឹងក្លាយជាការធ្វើសមកាលកម្មដ៏សាមញ្ញមួយ for await..of។

ដូច្នេះការប្រើប្រាស់នឹងដូចនេះ៖

```
for await (let commit of fetchCommits("username/repository")) {
  // process commit
}
```

នេះគឺជាមុខងារបែបនេះ ដែលត្រូវបានអនុវត្តជា async generator ៖

```
async function* fetchCommits(repo) {
  let url = `https://api.github.com/repos/${repo}/commits`;

  while (url) {
    const response = await fetch(url, { // (1)
      headers: {'User-Agent': 'Our script'}, // github needs any user-
agent header
    });
```

```
    const body = await response.json(); // (2) response is JSON (array
of commits)

    // (3) the URL of the next page is in the headers, extract it
      let  nextPage  =  response.headers.get('Link').match(/<(.*?)>;
rel="next"/);
    nextPage = nextPage?.[1];

    url = nextPage;

    for(let commit of body) { // (4) yield commits one by one, until
the page ends
      yield commit;
    }
  }
}
```

ការពន្យល់បន្ថែមអំពីរបៀបដែលវាដំណើរការ៖

1. យើងប្រើ វិធី ទាញយក កម្មវិធីរុករកតាមអ៊ីនធឺណិត ដើម្បីទាញយកការPromise។
   - URL ដំបូងគឺ https://api.github.com/repos/<repo>/commitsហើយទំព័របន្ទាប់នឹងស្ថិតនៅក្នុង Linkបឋមកថានៃការឆ្លើយតប។
   - វិធី fetchសាស្ត្រអនុញ្ញាតឱ្យយើងផ្តត់ផ្តង់ការអនុញ្ញាត និងបឋមកថាផ្សេងទៀត ប្រសិនបើចាំបាច់ នៅទីនេះ GitHub ទាមទារ User-Agent។
2. ការប្តេជ្ញាចិត្តត្រូវបានប្រគល់មកវិញជាទម្រង់ JSON ។
3. យើងគួរតែទទួលបាន URL ទំព័របន្ទាប់ពី Linkបឋមកថានៃការឆ្លើយតប។ វាមានទម្រង់ពិសេសដូច្នេះយើងប្រើកន្សោមធម្មតាសម្រាប់វា (យើងនឹងរៀនលក្ខណៈនេះក្នុង កន្សោមធម្មតា)។
   - URL ទំព័របន្ទាប់អាចមើលទៅដូចhttps://api.github.com/repositories/93253246/commits?page=2. វាត្រូវបានបង្កើតឡើងដោយ GitHub ខ្លួនឯង។
4. បន្ទាប់មក យើងផ្តល់ការប្តេជ្ញាចិត្តដែលបានទទួលម្តងមួយៗ ហើយនៅពេលដែលពួកគេបញ្ចប់ while(url)ការធ្វើម្តងទៀតបន្ទាប់នឹងបង្កឡើង ដោយធ្វើការស្នើសុំមួយបន្ថែមទៀត។

ឧទាហរណ៍នៃការប្រើប្រាស់ (បង្ហាញអ្នកនិពន្ធក្នុងកុងសូល)៖

```
(async () => {

  let count = 0;
```

```
    for (const commit of fetchCommits('javascript-tutorial/en.javascript.info')) {

    console.log(commit.author.login);

    if (++count == 100) { // let's stop at 100 commits
      break;
    }
  }

})();

// Note: If you are running this in an external sandbox, you'll need to paste here the function fetchCommits described above
```

នោះជាអ្វីដែលយើងចង់បាន។ មេកានិកខាងក្នុងនៃសំណើដែលបានបិទភ្ជាប់គឺមិនអាចមើលឃើញពីខាងក្រៅ។ សម្រាប់ពួកយើង វាគ្រាន់តែជាម៉ាស៊ីនភ្លើង async ដែលត្រឡប់ commits ប៉ុណ្ណោះ។

១.៣ លំហាត់

1. Pseudo-random generator

មានផ្នែកជាច្រើនដែលយើងត្រូវការទិន្នន័យចៃដន្យ។ មួយក្នុងចំណោមពួកគេគឺការធ្វើតេស្ត។ យើងប្រហែលជាត្រូវការទិន្នន័យចៃដន្យ៖ អត្ថបទ លេខ ជាដើម ដើម្បីសាកល្បងអ្វីៗឱ្យបានល្អ។នៅក្នុង JavaScript យើងអាចប្រើ Math.random( ). ប៉ុន្តែប្រសិនបើមានអ្វីខុស យើងចង់ធ្វើតេស្តម្តងទៀតដោយប្រើទិន្នន័យដូចគ្នាទាំងស្រុង។

សម្រាប់នោះគេហៅថា "seeded pseudo-random generators" ត្រូវបានគេប្រើ។ ពួកគេយក "seed" ដែលជាតម្លៃទីមួយ ហើយបន្ទាប់មកបង្កើតគ្រាប់បន្ទាប់ដោយប្រើរូបមន្តដើម្បីឱ្យ seedដូចគ្នាផ្តល់ទិន្នផលតាមលំដាប់ដូចគ្នា ដូច្នេះហើយលំហូរទាំងមូលអាចបន្តពូជបានយ៉ាងងាយស្រួល។ យើងគ្រាន់តែត្រូវចងចាំ seed ធ្វើវាឡើងវិញ។

ឧទាហរណ៍នៃរូបមន្តបែបនេះ ដែលបង្កើតតម្លៃចែកចាយស្មើៗគ្នា៖

```
next = previous * 16807 % 2147483647
```

ប្រសិនបើយើងប្រើ 1ជា seed នោះ តម្លៃនឹងមាន:

- 16807
- 282475249
- 1622650073

- ...ហើយបន្តទៀត...

កិច្ចការគឺដើម្បីបង្កើតមុខងារ Generator pseudoRandom(seed)ដែលយក seedនិងបង្កើត Generator ជាមួយរូបមន្តនេះ។

ឧទាហរណ៍នៃការប្រើប្រាស់៖

```
let generator = pseudoRandom(1);

alert(generator.next().value); // 16807
alert(generator.next().value); // 282475249
alert(generator.next().value); // 1622650073
```

មេរៀនទី ២៖ Modules

នៅក្នុង modern JavaScript development ការរៀបចំ Code ចូលទៅក្នុងម៉ូឌុលដែលអាចប្រើឡើងវិញបានគឺជាការអនុវត្តដ៏សំខាន់ដែលបង្កើនការរក្សាបាន លទ្ធភាពអាន និងការធ្វើមាត្រដ្ឋាន។ JavaScript ES6 (ECMAScript 2015) បានណែនាំប្រព័ន្ធម៉ូឌុលស្តង់ដារដែលអនុញ្ញាតឱ្យអ្នកអភិវឌ្ឍន៍បញ្ចូលមុខងារទៅជាឯកតាដាច់ដោយឡែក ដែលបន្ទាប់មកអាចត្រូវបាននាំចូល និងប្រើប្រាស់នៅក្នុងផ្នែកផ្សេងទៀតនៃកម្មវិធីមួយ។ មេរៀននេះស្វែងយល់ពីមូលដ្ឋានគ្រឹះនៃម៉ូឌុល JavaScript រួមទាំង រូបមន្ត អត្ថប្រយោជន៍ និងការអនុវត្តល្អបំផុតរបស់ពួកគេ។

២.១ តើអ្វីគឺជា Modules ?

Modules គឺជាបំណែកនៃ Code ដែលមានដោយខ្លួនឯង ដែលនាំចេញមុខងារ ឬទិន្នន័យជាក់លាក់។ ពួកគេជួយក្នុងការបែងចែកកម្មវិធីទៅជាផ្នែកតូចៗដែលអាចគ្រប់គ្រងបាន។ module នីមួយៗមាន variables ផ្ទាល់ខ្លួន ដែលមានន័យថាអថេរ និងមុខងារដែលបានកំណត់ក្នុងម៉ូឌុលគឺមិនអាចចូលប្រើបានក្រៅពីវាទេ លុះត្រាតែនាំចេញយ៉ាងច្បាស់លាស់។

២.២ រូបមន្ត និង ការប្រើប្រាស់

JavaScript ES6 បានណែនាំវិធីចម្បងពីរដើម្បីធ្វើការជាមួយម៉ូឌុល៖ របាយការណ៍នាំចូល និងនាំចេញ (import and export statements) ។

២.៣ Exporting

ដើម្បីធ្វើឱ្យ Code មានម៉ូឌុលផ្សេងទៀត អ្នកអាចប្រើពាក្យគន្លឹះនាំចេញ(export)។ ការនាំចេញមានពីរប្រភេទសំខាន់ៗ៖ ការនាំចេញដែលមានឈ្មោះ និងការនាំចេញលំនាំដើម(named exports and default exports)។

២.៣.១ Named Exports

Named exports អនុញ្ញាតឱ្យអ្នកនាំចេញជាតុជាច្រើនពីម៉ូឌុលមួយ។

ឧទាហរណ៍៖

```
// mathUtils.js
export const add = (a, b) => a + b;
export const subtract = (a, b) => a - b;
```

នៅទីនេះ add និង subtract ត្រូវបានដាក់ឈ្មោះនាំចេញនៃម៉ូឌុល mathUtils.js ។

២.៣.២ Default Exports

module ក៏អាចមាននាំចេញលំនាំដើមតែមួយផងដែរ។ default export មានប្រយោជន៍នៅពេលដែលម៉ូឌុលមួយមានបំណងនាំចេញObjectសំខាន់មួយ។

ឧទាហរណ៍៖

```
// calculator.js
const multiply = (a, b) => a * b;
export default multiply;
```

ក្នុងករណីនេះ គុណគឺជាការនាំចេញលំនាំដើមនៃម៉ូឌុល calculator.js ។

២.៤ Importing

ដើម្បីប្រើមុខងារដែលបាននាំចេញពីម៉ូឌុលផ្សេងទៀត អ្នកប្រើពាក្យគន្លឹះនាំចូល (import)។

២.៤.១ Importing Named Exports

ដើម្បី import named exports សូមប្រើដង្កៀបអង្កាញ់(curly braces)៖

ឧទាហរណ៍៖

```
// app.js
import { add, subtract } from './mathUtils.js';
console.log(add(2, 3));       // Output: 5
console.log(subtract(5, 2));  // Output: 3
```

២.៤.២ Importing Default Exports

Default exports អាចត្រូវបាននាំចូលដោយគ្មាន curly braces ហើយអាចប្តូរឈ្មោះបាន៖

ឧទាហរណ៍៖

```
// main.js
import multiply from './calculator.js';
console.log(multiply(4, 5));  // Output: 20
```

២.៥ អត្ថប្រយោជន៍នៃការប្រើប្រាស់ Modules

២.៥.១ ការលាក់ព័ត៌មាន (Encapsulation)

មុខងារ Modules encapsulate ដែលធ្វើឱ្យវាកាន់តែងាយស្រួលក្នុងការគ្រប់គ្រង និងយល់ Code ។ ម៉ូ module នីមួយៗមានវិសាលភាពផ្ទាល់ខ្លួន ដោយកាត់បន្ថយលទ្ធភាពនៃការដាក់ឈ្មោះជម្លោះ និងអន្តរកម្មដោយអចេតនារវាងផ្នែកផ្សេងៗនៃកម្មវិធី។

២.៥.២ ការប្រើឡើងវិញ (Reusability)

តាមរយៈការបំបែក Code ទៅជាម៉ូឌុល អ្នកអាចប្រើមុខងារដូចគ្នាឡើងវិញនៅផ្នែកផ្សេងៗនៃកម្មវិធី ឬសូម្បីតែនៅទូទាំងគម្រោងផ្សេងៗគ្នា។ នេះលើកកម្ពស់គោលការណ៍ DRY(Don't Repeat Yourself)។

២.៥.៣ ការថែទាំ (Maintainability)

Modules ធ្វើឱ្យកាន់តែងាយស្រួលក្នុងការថែទាំ និងធ្វើបច្ចុប្បន្នភាព Code ។ នៅពេលដែលមុខងារត្រូវបានរៀបចំទៅជាម៉ូឌុលដាច់ដោយឡែក ការផ្លាស់ប្តូរទៅម៉ូឌុលមួយទំនងជាមិនសូវមានឥទ្ធិពលលើអ្នកដទៃទេ ដែលធ្វើឱ្យវាកាន់តែងាយស្រួលក្នុងការសាកល្បង និង refactor ។

២.៥.៤ Dependency Management

Modules ជួយគ្រប់គ្រងភាពអាស្រ័យដោយបញ្ជាក់យ៉ាងច្បាស់ថាម៉ូឌុលមួយណាដែល Code ពឹងផ្អែកលើ។ នេះធ្វើឱ្យវាច្បាស់ថាតើភាពអាស្រ័យអ្វីខ្លះត្រូវបានទាមទារ និងសម្របសម្រួលការរៀបចំមូលដ្ឋាន Code កាន់តែប្រសើរ។

២.៦ លក្ខណៈពិសេស Module

២.៦.១ Dynamic Imports

Dynamic imports អនុញ្ញាតឱ្យអ្នកផ្ទុកម៉ូឌុលតាមតម្រូវការជាជាងនៅពេលផ្ទុកដំបូង។ នេះអាចធ្វើឱ្យប្រសើរឡើងនូវដំណើរការដោយកាត់បន្ថយពេលវេលាផ្ទុកដំបូងនៃកម្មវិធី។

ឧទាហរណ៍៖

```
// asyncLoad.js
async function loadModule() {
  const module = await import('./module.js');
  module.doSomething()}
```

២.៦.២ Module Path Resolution

Modern JavaScript environments តែងតែគាំទ្រយុទ្ធសាស្ត្រដោះស្រាយម៉ូឌុលផ្ទាល់ខ្លួន ដែលអនុញ្ញាតឱ្យអ្នកកំណត់ផ្លូវក្លែងក្លាយសម្រាប់ការនាំចូលម៉ូឌុលកាន់តែងាយស្រួល។

ឧទាហរណ៍៖

```
// tsconfig.json or jsconfig.json
{
  "compilerOptions": {
    "baseUrl": ".",
    "paths": {
      "@utils/*": ["src/utils/*"]
    }
  }
}
// Usage
import { someUtil } from '@utils/someUtil';
```

២.៧ លំហាត់

1. តើ Modules ក្នុង JavaScript មានតួនាទីអ្វី?
2. អ្វីទៅជា Named Export និង Default Export? ចូរសរសេរឧទាហរណ៍នៃ Export ទាំងពីរ។
3. តើការប្រើ Modules មានអត្ថប្រយោជន៍អ្វីខ្លះសម្រាប់ការអភិវឌ្ឍកម្មវិធី?
4. តើ Dynamic Import មានអត្ថប្រយោជន៍អ្វីខ្លះ?
5. ចូរពន្យល់អំពី Module Scope និងវិធីដើម្បីចៀសវាង Namespace Conflicts។
6. តើអ្វីខ្លះដែលត្រូវមានដើម្បីប្រើ ES Modules ក្នុង Node.js?
7. តើ import * as utils from './mathUtils.js' មានន័យដូចម្តេច? ចូរពន្យល់។
8. តើការ Import Module ជា Alias (import { func as myFunc } from './module.js') មានប្រយោជន៍អ្វី?

9. តើអ្វីជាការប្រើ Module Bundlers (e.g., Webpack, Rollup) ហើយតើពួកវាជួយយើងអ្វីខ្លះ?
10. តើ export { functionA, functionB } និង export { functionA as newFunctionA } ផ្សេងគ្នាដោយវិធីណា?
11. ចូលបង្កើត Module mathUtils.js ដែលមានមុខងារ sum(a, b), multiply(a, b) និង divide(a, b)។ នាំចេញមុខងារទាំងនេះប្រើ Named Export។ បន្ទាប់មក បង្កើត app.js ហើយ Import មុខងារទាំងអស់ពី mathUtils.js និងសាកល្បងប្រើ។
12. ចូលបង្កើត Module greeting.js ដែលមានមុខងារ sayHello(name) និង Export វាជា Default Export។ បង្កើត main.js ហើយ Import sayHello ពី greeting.js និងសាកល្បងប្រើវាជាមួយអត់ថេរមួយ។
13. បង្កើត Module dataFetcher.js ដែលមានមុខងារ fetchData(url) និងប្រើ Dynamic Import ក្នុង app.js ដើម្បី Import fetchData តាមការចាំបាច់។
14. ចូលបង្កើត Module stringUtils.js ដែលមានមុខងារ capitalize(word), lowercase(word) និង reverseString(word)។ Export មុខងារទាំងនេះប្រើ Named Export។ បន្ទាប់មក Import មុខងារទាំងនេះក្នុង main.js ហើយសាកល្បងប្រើ។
15. បង្កើត Module config.js ដែលមាន Default Export ដែលជាObject (object) មាន apiUrl និង port។ Import វា ក្នុង server.js ហើយបង្ហាញតម្លៃនៃ apiUrl និង port។
16. បង្កើត Module logger.js ដែលមានមុខងារ logInfo( ), logWarning( ), និង logError( )
    - Export logInfo( ) និង logWarning( ) ជា Named Export និង logError( ) ជា Default Export។
    - Import logInfo និង logWarning ដោយប្រើ {} និង Import logError ដោយប្រើ Default Import ក្នុង app.js។
17. បង្កើត Module dateUtils.js ដែលមានមុខងារ getCurrentDate( )។ ប្រើ Dynamic Import ក្នុង index.js ដើម្បី Import និងប្រើ getCurrentDate( ) នៅពេលចាំបាច់។
18. បង្កើត Module mathOperations.js ដែលមានមុខងារ add(a, b), subtract(a, b), និង multiply(a, b)។ បង្កើត Module calculator.js ហើយ Import មុខងារ add និង multiply ពី mathOperations.js ហើយប្រើវា។

មេរៀនទី ៣៖ តេស្ត (Testing)

Testing គឺជាផ្នែកសំខាន់មួយនៃការអភិវឌ្ឍន៍កម្មវិធីទំនើប ដោយធានាថា Code មានដំណើរការដូចការរំពឹងទុក និងកំណត់បញ្ហាមុនពេលវាប៉ះពាល់ដល់អ្នកប្រើប្រាស់ចុងក្រោយ។ នៅក្នុងមេរៀននេះ យើងនឹងរៀបរាប់ពីចំណុចសំខាន់ៗនៃការធ្វើតេស្ត Code JavaScript រួមទាំងការធ្វើតេស្តឯកតា ការធ្វើតេស្តរួមបញ្ចូល និង testing frameworks ដ៏ពេញនិយមដូចជា Jest និង Mocha ជាដើម។

៣.១ Unit Testing

Unit testing ពាក់ព័ន្ធនឹងការសាកល្បងសមាសធាតុនីមួយៗ ឬមុខងារនៃ Code របស់អ្នក ដោយឡែក ដើម្បីធានាថាផ្នែកនីមួយៗដំណើរការបានត្រឹមត្រូវ។ គោលដៅគឺដើម្បីធ្វើឱ្យមានសុពលភាពថាឯកតានៃ Code នីមួយៗដំណើរការដូចបំណង ដោយឯករាជ្យពីឯកតាផ្សេងទៀត។

៣.១.១ Why Unit Testing is Important

- Early Detection of Bugs ៖ តាមរយៈការធ្វើតេស្តសមាសធាតុនៅក្នុងភាពឯកោ អ្នកអាចចាប់បានបញ្ហានៅដំណាក់កាលអភិវឌ្ឍន៍ដំបូង។
- Simplifies Debugging ៖ នៅពេលដែលការធ្វើតេស្តបរាជ័យ វាផ្តល់នូវមតិកែលម្អជាក់លាក់អំពីអង្គភាពណាមួយដែលបរាជ័យ ដែលធ្វើឱ្យវាកាន់តែងាយស្រួលក្នុងការធ្វើរោគវិនិច្ឆ័យ និងដោះស្រាយបញ្ហា។
- Facilitates Refactoring ៖ការធ្វើតេស្តឯកតាធានាថាការផ្លាស់ប្តូរ Code របស់អ្នកមិនធ្វើឱ្យខូចមុខងារដែលមានស្រាប់នោះទេ ដែលមានតម្លៃជាពិសេសក្នុងអំឡុងពេលជួសជុល ឬបន្ថែមមុខងារថ្មីៗ។

៣.១.២ Writing Unit Tests in JavaScript

- Identify Test Cases៖ កំណត់មុខងារ ឬសមាសធាតុណាមួយដែលត្រូវធ្វើតេស្ត និងកំណត់លទ្ធផលរំពឹងទុក។
- Set Up Test Framework៖ ប្រើ testing framework ដើម្បីសរសេរ និងអនុវត្តការធ្វើតេស្តរបស់អ្នក។
- Write Test Functions៖ អនុវត្ត test functions ដែលពិនិត្យមើលសេណារីយ៉ូផ្សេងៗ និង edge cases ។
- Run Tests៖ អនុវត្តការធ្វើតេស្តរបស់អ្នកដើម្បីផ្ទៀងផ្ទាត់ថាសមាសធាតុទាំងអស់ដំណើរការដូចការរំពឹងទុក។

- Refactor Code៖ ផ្អែកលើលទ្ធផលតេស្ត ធ្វើការផ្លាស់ប្តូរជាចាំបាច់ចំពោះ Code របស់អ្នក ហើយដំណើរការការធ្វើតេស្តឡើងវិញ ដើម្បីធានាថាអ្វីៗនៅតែដំណើរការបានត្រឹមត្រូវ។

៣.២ Integration Testing

Integration testing ផ្តោតលើការធ្វើឯកតាសមាសធាតុ ឬប្រព័ន្ធផ្សេងៗដំណើរការជាមួយគ្នាដូចបំណង។ មិនដូចការធ្វើតេស្តឯកតាទេ ដែលសាកល្បងឯកតានីមួយៗក្នុងភាពឯកោ ការធ្វើតេស្តរួមបញ្ចូលវាយតម្លៃថាតើសមាសធាតុជាច្រើនមានអន្តរកម្មយ៉ាងដូចម្តេច។

៣.២.១ Why Integration Testing is Important៖

- Ensures Proper Communication ៖ Integration tests ពិនិត្យមើលថាផ្នែកផ្សេងៗនៃកម្មវិធីរបស់អ្នកប្រាស្រ័យទាក់ទង និងដំណើរការជាមួយគ្នាយ៉ាងត្រឹមត្រូវ។
- Identifies Issues in Data Flow ៖ ការធ្វើតេស្តទាំងនេះជួយស្វែងរកបញ្ហាក្នុងការផ្លាស់ប្តូរទិន្នន័យ ឬការរួមបញ្ចូលចំណុចរវាងសមាសធាតុ។
- Validates System Functionality ៖ តាមរយៈការធ្វើតេស្តសមាសធាតុរួមបញ្ចូលគ្នា អ្នកអាចធានាថាប្រព័ន្ធទាំងមូលដំណើរការដូចការរំពឹងទុក។

៣.២.២ Writing Integration Tests in JavaScript៖

1. Define Integration Points៖ កំណត់សមាសធាតុ ឬប្រព័ន្ធដែលត្រូវការអន្តរកម្ម និងកំណត់សេណារីយ៉ូដែលត្រូវសាកល្បង។
2. Set Up Test Environment៖ រៀបចំ environment ដើម្បីក្លែងធ្វើ real-world interaction នៃសមាសធាតុ។
3. Write Integration Test Cases៖ បង្កើតករណីសាកល្បងដែលពិនិត្យមើលអន្តរកម្ម និងលំហូរទិន្នន័យរវាងសមាសធាតុ។
4. Execute Tests៖ ដំណើរការការធ្វើតេស្តរួមបញ្ចូល ដើម្បីធ្វើឱ្យធានាថាសមាសធាតុដំណើរការជាមួយគ្នាដូចការរំពឹងទុក។
5. Analyze Results៖ ពិនិត្យមើលលទ្ធផលតេស្តដើម្បីកំណត់អត្តសញ្ញាណ និងដោះស្រាយបញ្ហាជាមួយអន្តរកម្មសមាសធាតុ។

៣.៣ Testing Frameworks

Testing frameworks ផ្តល់ឧបករណ៍ និងឧបករណ៍ប្រើប្រាស់ដើម្បីសរសេរ រៀបចំ និងប្រតិបត្តិការធ្វើតេស្តឱ្យកាន់តែមានប្រសិទ្ធភាព។ testing frameworks ដ៏ពេញនិយមពីរសម្រាប់ JavaScript គឺ Jest និង Mocha ។

៣.៣.១ Jest

Overview ៖ Jest គឺជាក្របខ័ណ្ឌសាកល្បងដ៏ទូលំទូលាយមួយដែលបង្កើតឡើងដោយ Facebook ។ វារួមបញ្ចូលឧបករណ៍រត់សាកល្បង assertion library និងសមត្ថភាពក្លែងបន្លំដែលភ្ជាប់មកជាមួយ។

Features៖

Snapshot Testing៖ អនុញ្ញាតឱ្យអ្នកថតរូបនៃលទ្ធផលនៃសមាសធាតុ ហើយប្រៀបធៀបពួកវាទៅនឹងរូបថតនាពេលអនាគត ដើម្បីរកមើលការផ្លាស់ប្តូរ។

Mocking៖ ផ្តល់ឧបករណ៍ប្រើប្រាស់ដើម្បីមើលមុខងារ និងម៉ូឌុល ដែលអនុញ្ញាតឱ្យមានការសាកល្បងដាច់ដោយឡែកនៃសមាសភាគ។

Zero Configuration៖ ភ្ជាប់មកជាមួយ defaults ដែលអាចយល់បាន និងតម្រូវឱ្យមានការដំឡើងតិចតួចបំផុត។

ឧទាហរណ៍៖

```
// Jest test example

test('adds 1 + 2 to equal 3', () => {
  expect(1 + 2).toBe(3);
});
```

៣.៣.២ Mocha

Overview ៖ Mocha គឺជា testing framework ដែលអាចបត់បែនបាន និងប្រើប្រាស់យ៉ាងទូលំទូលាយ ដែលផ្តល់នូវអ្នករត់ការសាកល្បង និងអនុញ្ញាតឱ្យមានntegrationជាមួយ assertion libraries ផ្សេងទៀត និង mocking tools។

Features៖

- Flexible Assertions៖ ធ្វើការជាមួយ assertion libraries ផ្សេងៗដូចជា Chai ។
- Asynchronous Testing៖ គាំទ្រការធ្វើតេស្ត asynchronous code ជាមួយនឹងយន្តការដែលភ្ជាប់មកជាមួយ។

- Customizable៖ អាចកំណត់រចនាសម្ព័ន្ធបានខ្ពស់ ដើម្បីបំពេញតម្រូវការសាកល្បងផ្សេងៗ។

ឧទាហរណ៍៖

```
// Mocha test example with Chai assertion library
const assert = require('chai').assert;

describe('Array', () => {
  it('should start empty', () => {
    let arr = [];
    assert.equal(arr.length, 0);
  });
});
```

៣.៤ Choosing a Framework

- Jest គឺសំក្តិសមសម្រាប់គម្រោងដែលទាមទារដំណោះស្រាយសាកល្បងទាំងអស់ក្នុងមួយជាមួយនឹងការកំណត់រចនាសម្ព័ន្ធតិចតួចបំផុត។
- Mocha គឺល្អសម្រាប់គម្រោងដែលត្រូវការភាពបត់បែនបន្ថែមទៀត និងការប្តូរតាមបំណងក្នុងការរៀបចំការសាកល្បងរបស់ពួកគេ។

៣.៥ លំហាត់

1. តើ unit testing មានអត្ថប្រយោជន៍អ្វីខ្លះក្នុងការអភិវឌ្ឍកម្មវិធី JavaScript?
2. តើខុសគ្នារវាង unit testing និង integration testing អ្វីខ្លះ?
3. អ្វីទៅជា testing framework ហើយហេតុអ្វីជាមូលហេតុនៃការប្រើ Jest ឬ Mocha?
4. តើ mocking មានតួនាទីយ៉ាងដូចម្តេចក្នុងការប្រើប្រាស់ testing frameworks?
5. លើកលែងតម្លៃខុសៗគ្នា 3 ដែលអាចបណ្តាលឲ្យ unit test មួយបរាជ័យ?
6. តើការប្រើប្រាស់ assertions ក្នុង unit testing មានអត្ថប្រយោជន៍យ៉ាងដូចម្តេច?
7. តើ Test-Driven Development (TDD) ជាដូចម្តេច ហើយតើវាស្តីពីដំណើរការផលិតកម្មយ៉ាងដូចម្តេច?
8. តើ mocking និង stubbing ខុសគ្នាអ្វី?
9. តើ asynchronous testing មានសារៈសំខាន់យ៉ាងដូចម្តេចក្នុងការប្រើប្រាស់ JavaScript?

10. តើនៅពេលណាដែលគួរតែធ្វើ integration testing នៅលើប្រព័ន្ធការងារនៅក្នុងកម្មវិធី JavaScript?
11. តើចំនុចដែលមានឥទ្ធិពលសំខាន់ក្នុងការប្រើ Jest ដោយប្រើ watch mode គឺមានអ្វីខ្លះ?
12. បង្កើត function សម្រាប់បញ្ជាក់វាលទិន្នន័យក្នុង form ហើយសរសេរ unit tests ដើម្បីធ្វើការផ្ទៀងផ្ទាត់ថាការបញ្ចូលទិន្នន័យសម្រាប់អ្នកប្រើប្រាស់មិនអនុញ្ញាតឱ្យបញ្ចូលព័ត៌មានដែលមិនត្រឹមត្រូវ។
13. បង្កើត asynchronous function ដែលត្រូវតែកំណត់ការកំណត់ពេលវេលាសម្រាប់ការទទួលបាន JSON ពី API ហើយសរសេរ Mocha test ដើម្បីផ្ទៀងផ្ទាត់ថាប្រព័ន្ធនោះត្រូវបានទាញយកនូវ JSON data។
14. សរសេរ test case ដែលផ្ទៀងផ្ទាត់ថា function findLargestNumber(numbers) ត្រឡប់តម្លៃចំនួនធំបំផុតនៅក្នុងអារ៉េ។
15. សរសេរ asynchronous test ដើម្បីផ្ទៀងផ្ទាត់ថា function fetchUserData(id) ត្រឡប់តម្លៃបន្ទាប់ពី 2 វិនាទី។ ប្រើ done() បញ្ជាក់ថាបន្ទប់អាជីព asynchronous បានបញ្ចប់ត្រឹមត្រូវ។
16. ប្រើ Jest mock function ដើម្បី mock API call និងសរសេរ test ដើម្បីបញ្ជាក់ថាការហៅ API ត្រូវបាន mock និងមានលទ្ធផលត្រឹមត្រូវ។
17. បង្កើត test coverage report សម្រាប់ Code ដែលសរសេរ ហើយចង្អុលបង្ហាញបែបផែនពីការប្រើ Jest coverage options។
18. បង្កើត configuration សំរាប់ continuous integration ដែលបង្កើតការប្រតិបត្តិ testing ក្នុងប្រព័ន្ធ CI/CD (Continuous Integration/Continuous Deployment) ដើម្បីទទួលបាន Code ដែលត្រូវបានសាកល្បងយ៉ាងគ្រប់គ្រាន់។

មេរៀនទី ៤៖ ការកែកំហុស (Debugging)

Debugging គឺជាជំនាញសំខាន់សម្រាប់អ្នកអភិវឌ្ឍន៍ JavaScript ណាមួយ។ វាពាក់ព័ន្ធនឹងការកំណត់អត្តសញ្ញាណ និងជួសជុលកំហុសនៅក្នុង Code របស់អ្នក ដើម្បីធានាថាវាដំណើរការយ៉ាងរលូន និងដំណើរការដូចការរំពឹងទុក។ មេរៀននេះស្វែងយល់ពីបច្ចេកទេសបំបាត់កំហុសផ្សេងៗរួមទាំងការប្រើប្រាស់ try...catch statements និង error objects។

៤.១ ការកំណត់ និងកែកំហុសក្នុងក្ងៃ JavaScript

៤.១.១ ប្រភេទនៃកំហុស

កំហុស JavaScript អាចត្រូវបានបែងចែកយ៉ាងទូលំទូលាយជាបីប្រភេទ៖

រូបមន្ត Errors ៖ ទាំងនេះកើតឡើងនៅពេលដែលម៉ាស៊ីន JavaScript ជួបប្រទះ Code ដែលមិនអនុលោមតាមច្បាប់ language's រូបមន្ត។ ឧទាហរណ៍ បាត់វង់ក្រចក ឬពាក្យគន្លឹះអក្ខរាវិរុទ្ធខុស។

Runtime Errors ៖ កំហុសទាំងនេះកើតឡើងកំឡុងពេលដំណើរការ script ។ ឧទាហរណ៍រួមមានការព្យាយាមចូលប្រើអថេរដែលមិនបានកំណត់ ឬហៅវិធីសាស្ត្រនៅលើ null ឬ undefined ។

Logical Errors ៖ ទាំងនេះកើតឡើងនៅពេលដែល Code ដំណើរការដោយគ្មានកំហុសឆ្គង ប៉ុន្តែបង្កើតលទ្ធផលមិនត្រឹមត្រូវដោយសារតែកំហុសនៅក្នុងតក្កវិជ្ជា។

៤.១.២ បច្ចេកទេសកែកំហុសឆ្ងៅ

- Using Console Statements ៖ console.log( ) គឺជាវិធីត្រង់មួយដើម្បីបញ្ចេញតម្លៃ និងពិនិត្យមើលលំហូរនៃការប្រតិបត្តិ។ អ្នកអាចកត់ត្រាអថេរ function calls និងសារដើម្បីតាមដាន the code's behavior ។

ឧទាហរណ៍៖

```
let x = 5;
console.log("Value of x:", x);
```

- Using Breakpoints ៖ modern browsers ភាគច្រើនផ្តល់ជូននូវឧបករណ៍អ្នកអភិវឌ្ឍន៍ដែលអនុញ្ញាតឱ្យអ្នកកំណត់ចំណុចបំបែក។ ចំណុចបំបែកផ្អាកដំណើរការ Code នៅបន្ទាត់ជាក់លាក់មួយ អនុញ្ញាតឱ្យអ្នកត្រួតពិនិត្យអថេរ និង call stack ។
- Stack Traces ៖ នៅពេលដែលមានកំហុសកើតឡើង JavaScript ផ្តល់នូវដាន stack ដែលបង្ហាញពីលំដាប់នៃការហៅមុខងារដែលនាំទៅដល់កំហុស។ នេះអាចជួយកំណត់កន្លែងដែលអ្វីៗខុស។

- Debugging Tools ៖ Browsers ដូចជា Chrome និង Firefox ផ្តល់ជូននូវឧបករណ៍បំបាត់កំហុសដ៏ស្មុគ្រស្មាញ រួមទាំងសមត្ថភាពក្នុងការបោះជំហានឆ្លងកាត់ Code តាមបន្ទាត់មួយ វាយតម្លៃកន្សោម និងពិនិត្យមើលស្ថានភាពបច្ចុប្បន្ននៃកម្មវិធី។

៤.២ try...catch Statements

try...catch statement គឺជាយន្តការដ៏រឹងមាំមួយសម្រាប់ដោះស្រាយការលើកលែងនៅក្នុង JavaScript ។ វាអនុញ្ញាតឱ្យអ្នកអភិវឌ្ឍន៍សាកល្បង Code សម្រាប់កំហុស និងដោះស្រាយពួកវាដោយសុភាពរាបសារ។

៤.២.១ រូបមន្តមូលដ្ឋាន

The basic structure នៃ try...catch statement មានដូចខាងក្រោម៖

ឧទាហរណ៍៖

```
try {
  // Code that may throw an error
} catch (error) {
  // Code to handle the error
}
```

- try Block ៖ មានលេខ Code ដែលអាចបោះកំហុស។
- catch Block ៖មានលេខ Code ដែលប្រតិបត្តិប្រសិនបើកំហុសត្រូវបានបោះចោលក្នុងប្លុកសាកល្បង។ ប៉ារ៉ាម៉ែត្រកំហុសផ្ទុក object ករណីលើកលែង។

ឧទាហរណ៍៖

```
try {
  let result = riskyFunction();
  console.log(result);
} catch (error) {
  console.error("An error occurred:", error.message);
}
```

ក្នុងឧទាហរណ៍នេះ ប្រសិនបើ riskyFunction( ) បោះកំហុស catch block នឹងដោះស្រាយវា ដោយកត់ត្រាសារកំហុសទៅកាន់ console ។

៤.២.២ finally Block

finally block អាចត្រូវបានប្រើជាមួយ try...catch ដើម្បីប្រតិបត្តិ Code ដែលគួរតែដំណើរការ ដោយមិនគិតពីថាតើកំហុសត្រូវបានបោះចោលឬអត់។

ឧទាហរណ៍៖

```
try {
  // Code that may throw an error
} catch (error) {
  // Code to handle the error
} finally {
  // Code that will run no matter what
}
```

៤.៣ Error Objects

JavaScript ផ្តល់នូវ error objects ដែលមានស្រាប់ជាច្រើនដែលអាចត្រូវបានប្រើដើម្បីតំណាងឱ្យប្រភេទកំហុសផ្សេងៗ។

៤.៣.១ Error Object

The base Error object គឺជាប្រភេទទូទៅដែលប្រភេទកំហុសផ្សេងទៀត inherit ។ ឧទាហរណ៍

```
let error = new Error("Something went wrong");
console.error(error.message);
```

៤.៣.២ TypeError

កើតឡើងនៅពេលដែលតម្លៃមិនមែនជាប្រភេទដែលរំពឹងទុក។ ឧទាហរណ៍៖

```
let num = "not a number";
try {
  num.toFixed(2);
```

```
} catch (error) {
  console.error("TypeError:", error.message);
}
```

៤.៣.៣ ReferenceError

កើតឡើងនៅពេលដែលអថេរដែលមិនមាន referenced។ ឧទាហរណ៍៖

```
try {
  console.log(nonExistentVariable);
} catch (error) {
  console.error("ReferenceError:", error.message);
}
```

៤.៣.៤ រូបមន្ត Error

កើតឡើងនៅពេលដែល JavaScript engine ជួបប្រទះ invalid រូបមន្ត។ ឧទាហរណ៍៖

```
try {
  eval("var a = ;");
} catch (error) {
  console.error("រូបមន្តError:", error.message);
}
```

៤.៣.៥ RangeError

កើតឡើងនៅពេលដែលតម្លៃមួយមិនស្ថិតនៅក្នុងជួរនៃតម្លៃដែលអាចទទួលយកបាន។ ឧទាហរណ៍៖

```
try {
  let arr = new Array(-1);
} catch (error) {
  console.error("RangeError:", error.message);
```

៤.៤ លំហាត់

1. តើគោលបំណងចម្បងនៃប្រយោគ try...catch នៅក្នុង JavaScript គឺអ្វី?
2. ពន្យល់ការប្រៀបធៀបរវាងកំហុសសាំងតំរ (រូបមន្ត error), កំហុសការប្រតិបត្តិការនៅពេលរត់ (runtime error), និងកំហុសត្រឹមត្រូវ (logical error) ក្នុង JavaScript។
3. អ្វីដែលប្លុក finally ធ្វើនៅក្នុងប្រយោគ try...catch?.
4. ផ្តល់ឧទាហរណ៍ដែលអាចធ្វើឲ្យកើតកំហុស TypeError និងពន្យល់មូលហេតុដែលវាកើតឡើង។
5. តើមានការប្រៀបធៀបរវាងObject Error និងObject ReferenceError នៅក្នុង JavaScript យ៉ាងដូចម្តេច?
6. តើអ្នកអាចប្រើ console.log( ) ដូចម្តេចដើម្បីជួយកំណត់មូលហេតុនៃកំហុសនៅក្នុង Code JavaScript របស់អ្នក?
7. តួនាទីនៃObject error នៅក្នុងប្លុក catch នៃប្រយោគ try...catch គឺអ្វី?
8. ពន្យល់ពីរបៀបដែលអ្នកនឹងដោះសោភាពកំហុសជាច្រើននៅក្នុងប្លុក try...catch មួយ។
9. តើអ្នកអាចបំបែកការប្រៀបធៀបរវាង រូបមន្តError និង ReferenceError នៅពេលកំពុងដោះសោភាពកំហុស JavaScript ដូចម្តេច?
10. អត្ថប្រយោជន៍នៃការប្រើ throw ដើម្បីបង្កើតកំហុសផ្ទាល់ខ្លួននៅក្នុង JavaScript គឺអ្វី?
11. តើអ្វីគ្រោះClassដែលអាចកើតមានបើអ្នកមិនបញ្ចូលប្លុក catch បន្ទាប់ពីប្លុក try ដែលមានកំហុស?
12. ពន្យល់ពីរបៀបប្រើគុណលក្ខណៈ stack នៃObjectកំហុស ដើម្បីដោះសោភាពបញ្ហានៅក្នុង Code JavaScript របស់អ្នក។
13. បង្កើតប្រយោគ try...catch ដែលដោះសោភាពកំហុស RangeError នៅពេលព្យាយាមបង្កើតអារេដោយមានទំហំអនិច្ចតិច។
14. សរសេរការផ្តល់កំហុស TypeError និងtry...catch statement។
15. សរសេរកម្មវិធីដែលកត់សៀវភៅសារទៅកាន់កុងសូលដោយប្រើ console.log និងចាប់កំហុសដែលអាចកើតឡើងដោយប្រើ try...catch។
16. រៀបចំស្ថានការណ៍ដែលអ្នកប្រើ finally ដើម្បីធានាថាការបញ្ចប់ការងារត្រូវបានអនុវត្តដោយមិនគិតថាតើមានកំហុសកើតឡើងឬទេ។
17. កែប្រែ Script មួយដែលបញ្ចេញកំហុស និងប្រើប្រយោគ throw ដើម្បីបង្កើតកំហុសផ្ទាល់ខ្លួន។ ដោះសោភាពកំហុសផ្ទាល់ខ្លួនដោយប្រើ try...catch។

18. សរសេរការប្រើប្រាស់ប្រយោគ try...catch ដើម្បីដោះសោភាពកំហុសច្រើនប្រភេទ (ឧ. TypeError, RangeError, រូបមន្តError) ។
19. បង្កើតScriptដែលបង្ហាញកំហុសនៅក្នុងមុខងារ។ បន្ទាប់មកប្រើ console.trace( ) ដើម្បីជួយដោះសោភាពកន្លែងដែលកំហុសកើតឡើងនៅក្នុង function call stack។
20. រៀបចំមុខងារដែលបង្ហាញកំហុសប្រសិនបើលក្ខខណ្ឌជាក់លាក់មួយមិនត្រូវបានបំពេញ។ ដោះសោភាពកំហុសនៅក្នុងប្លុក try...catch ហើយកត់សៀវភៅសារកំហុស
21. សរសេរប្លុក try...catch ដែលដោះសោភាពកំហុសអាស៊ីណោដោយប្រើ Promise។
22. បង្កើតស្ថានការណ៍ដែលប្រើប្លុក finally ដើម្បីលុបធនធានដូចជាការបិទឯកសារ ឬការដកចេញពីមូលដ្ឋានទិន្នន័យដោយមិនគិតថាតើមានកំហុសកើតឡើងឬទេ។

សេចក្តីសន្និដ្ឋានជំពូកទី ៥៖ JavaScript ក្រមិតអ្នកជំនាញ

មេរៀនទី ១៧៖ Generators, Advanced Iteration

មេរៀននេះពន្យល់ពីរបៀបដែល Generators អនុញ្ញាតឱ្យមុខងារផ្អាក និងបន្តដំណើរការ ដោយបង្កើតតម្លៃជាបន្តបន្ទាប់។

- Generators (Synchronous)
  - Generator function ត្រូវបានបង្កើតដោយប្រើ function។ ការហៅមុខងារនេះត្រឡប់ generator object។
  - yield៖ សេចក្តីថ្លែងការណ៍នេះផ្អាកការប្រតិបត្តិ និងបញ្ជូនតម្លៃមួយទៅកាន់កូដខាងក្រៅ។
  - .next( )៖ បន្តការប្រតិបត្តិរហូតដល់ yield បន្ទាប់ ហើយត្រឡប់ object មួយដែលមាន value និង done។
  - សមាសភាព Generator (yield): អនុញ្ញាតឱ្យបញ្ចូល Generator មួយទៅក្នុង Generator មួយទៀត។
- Async Iteration និង Async Generators

  Async Iteration ត្រូវបានប្រើដើម្បីដោះស្រាយទិន្នន័យដែលមកដល់ដោយអសមកាល។
  - Async Generators: ត្រូវបានបង្កើតដោយប្រើ async function ហើយអនុញ្ញាតឱ្យប្រើ await នៅខាងក្នុង។
  - For loop await..of: ប្រើដើម្បីរំលឹកឡើងវិញលើ Async iterable។

មេរៀនទី ១៨៖ Modules

មេរៀននេះពន្យល់អំពី Modules នៅក្នុង JavaScript ដែលជាបំណែកនៃកូដដែលអាចប្រើឡើងវិញបាន និងមាន Scope ឯកជន។

- ការនាំចេញ និងការនាំចូល (Exporting and Importing)

  ដើម្បីចែករំលែកកូដរវាង Modules គេប្រើ export និង import។
  - Named Exports (ការនាំចេញដែលមានឈ្មោះ): អនុញ្ញាតឱ្យនាំចេញជាតុជាច្រើន (ឧទាហរណ៍៖ export const add = ...;)។ ត្រូវបាននាំចូលដោយប្រើដង្កៀបអង្កាញ់ {}។
  - Default Exports (ការនាំចេញលំនាំដើម): ម៉ូឌុលនីមួយៗអាចមាន default export តែមួយគត់។ ត្រូវបាននាំចូលដោយគ្មានដង្កៀបអង្កាញ់។
- អត្ថប្រយោជន៍ និងលក្ខណៈពិសេស

ការប្រើប្រាស់ Modules ជួយបង្កើន Encapsulation (ការលាក់ព័ត៌មាន), Reusability (ការប្រើឡើងវិញ), និង Maintainability (ការថែទាំ)។

- Dynamic Imports (ការនាំចូលឌីណាមិក): អនុញ្ញាតឱ្យផ្ទុកម៉ូឌុលតាមតម្រូវការ (on-demand) ដោយប្រើ import('./module.js')។

មេរៀនទី ១៩៖ តេស្ត (Testing)

Testing គឺជាផ្នែកសំខាន់មួយនៃការអភិវឌ្ឍន៍កម្មវិធី ដើម្បីធានាថា កូដដំណើរការត្រឹមត្រូវ និងចាប់កំហុសនៅដំណាក់កាលដំបូង។

- Unit Testing (ការធ្វើតេស្តឯកតា)

Unit testing ផ្តោតលើការសាកល្បងមុខងារ ឬសមាសធាតុនីមួយៗដាច់ដោយឡែកពីគ្នា (in isolation)។ វាជួយ ចាប់បញ្ហាបានឆាប់រហ័ស ធ្វើឱ្យ Debugging ងាយស្រួល និងជួយសម្រួលដល់ Refactoring។

- Integration Testing (ការធ្វើតេស្តរួមបញ្ចូល)

Integration testing ផ្តោតលើការផ្ទៀងផ្ទាត់ថាសមាសធាតុ ឬប្រព័ន្ធផ្សេងៗដំណើរការជាមួយគ្នាបានត្រឹមត្រូវ និងទំនាក់ទំនងគ្នាបានត្រឹមត្រូវ។

- Testing Frameworks

Jest និង Mocha គឺជា frameworks ពេញនិយម៖

- Jest: ជាដំណោះស្រាយ all-in-one ដែលមាន test runner, assertion library, និង mocking capabilities ភ្ជាប់មកជាមួយ។
- Mocha: ជា test runner ដែលអាចបត់បែនបាន ហើយធ្វើការជាមួយ libraries ផ្សេងៗដូចជា Chai។

មេរៀនទី ២០៖ ការកែកំហុស (Debugging)

Debugging គឺជាជំនាញសំខាន់មួយក្នុងការកំណត់អត្តសញ្ញាណ និងជួសជុលកំហុសនៅក្នុងកូដ

- ប្រភេទនៃកំហុស JavaScript
  - រូបមន្ត Errors: កំហុសដែលកើតឡើងនៅពេលកូដមិនគោរពតាមច្បាប់ រូបមន្ត (ឧ. បាត់វង់ក្រចក)។
  - Runtime Errors: កំហុសដែលកើតឡើងកំឡុងពេល Script កំពុងដំណើរការ (ឧ. ចូលប្រើអថេរដែលមិនត្រូវបានកំណត់)។

- Logical Errors: កូដដំណើរការធម្មតា តែលទ្ធផលមិនត្រឹមត្រូវ ដោយសារតែកំហុសនៅក្នុងតក្កវិជ្ជា។

- បច្ចេកទេសកែកំហុសទូទៅ
  - console.log( )៖ វិធីសាមញ្ញដើម្បីតាមដានតម្លៃអថេរ ឬលំហូរនៃការប្រតិបត្តិ។
  - Breakpoints: នៅក្នុង Developer Tools របស់ Browser អនុញ្ញាតឱ្យផ្អាកការប្រតិបត្តិនៅបន្ទាត់ជាក់លាក់មួយ។
  - Stack Traces: បង្ហាញពីលំដាប់នៃការហៅមុខងារដែលនាំឱ្យមានកំហុស។

- ការគ្រប់គ្រងកំហុសជាមួយ try...catch

try...catch អនុញ្ញាតឱ្យអ្នកសាកល្បងកូដសម្រាប់កំហុស និងដោះស្រាយពួកវាដោយសុភាពរាបសារ។ ប្លុក finally ត្រូវបានប្រើដើម្បីប្រតិបត្តិកូដដែលគួរដំណើរការដោយមិនគិតពីថាតើមានកំហុសកើតឡើងឬអត់ (សម្រាប់ cleanup)។

- Error Objects

JavaScript មាន Error objects ជាច្រើនដែលតំណាងឱ្យប្រភេទកំហុសផ្សេងៗគ្នាដូចជា TypeError, ReferenceError, និង រូបមន្តError។

ជំពូកទី ៦៖ សេចក្តីសន្និដ្ឋាន

១.១ សេចក្តីសន្និដ្ឋាន

សៀវភៅ «មូលដ្ឋានគ្រឹះនៃភាសា JavaScript (ES2024)» ដែលបានបង្ហាញអំពីសារសំខាន់ និងទិសដៅថ្មីៗសម្រាប់អ្នកអភិវឌ្ឍកម្មវិធី ទាំងអ្នកចាប់ផ្តើម និងអ្នកមានបទពិសោធន៍។ ជាពិសេសទៅ JavaScript មិនមែនជាភាសាធម្មតាទៀតទេ តែវាបានក្លាយទៅជាមូលដ្ឋាននៃការអភិវឌ្ឍវេបសាយ កម្មវិធីចល័ត និងប្រព័ន្ធផ្សេងៗទៀត ដែលមានការផ្លាស់ប្តូរយ៉ាងឆាប់រហ័ស និងមានការអភិវឌ្ឍន៍ឥតឈប់ឈរនាពេលបច្ចុប្បន្ននេះ។

សៀវភៅនេះមានការណែនាំលម្អិត និងបានរួមបញ្ចូលនូវ ឧទាហរណ៍ជាក់ស្តែង និងអនុវត្តលំហាត់ជាច្រើនដែលធ្វើឱ្យអ្នកសិក្សា, អ្នកអភិវឌ្ឍពង្រឹងចំណេះដឹងកាន់តែប្រសើឡើង។ ហើយភាសា JavaScript មានសមត្ថភាពទូលំទូលាយ ចាប់ពីការគ្រប់គ្រង DOM ការគ្រប់គ្រងព័ត៌មានទិន្នន័យ ការបង្កើត UI Interaction ការដោះស្រាយ Asynchronous Programming ដល់ការគ្រប់គ្រង Modules និង Testing ដែលជាគ្រឹះសំខាន់សម្រាប់កម្មវិធីទំនើបនាពេលបច្ចុប្បន្ន។

ជាក់ស្តែងអ្នកសិក្សាបានអាចសម្របខ្លួននឹង រូបមន្ត ថ្មីៗ ដូចជា Optional Chaining, Nullish Coalescing Operator, Class Fields និងការគ្រប់គ្រង Private Methods ក្នុង Classes ដែលធ្វើឱ្យការសរសេរកូដមានភាពស្រស់ស្អាត, កាត់បន្ថយBug និងងាយស្រួលក្នុងការថែទាំ។ វាក៏បានបង្ហាញថា JavaScript មិនត្រឹមតែទទួលបានចំណេះដឹងផ្នែក Code តែប៉ុណ្ណោះទេ ប៉ុន្តែត្រូវអនុវត្តន៍កិច្ចការរបស់អ្នកអភិវឌ្ឍដែលត្រូវរួមបញ្ចូលការគិតវិភាគ ការដោះស្រាយបញ្ហា និងការចូលរួមសហគមន៍ផងដែរ។

JavaScript មិនត្រឹមតែគាំទ្រ Client-Side តែប៉ុណ្ណោះទេ មានទាំង Node.js និងស្ថាបត្យកម្មមូលដ្ឋាននានាផងដែរ បានផ្តល់ណែនាំដល់អ្នកអភិវឌ្ឍឱ្យពង្រឹងចំណេះដឹងចម្រុះរវាង Front-End និង Back-End។ នេះគឺជាចំណុចសំខាន់ណាស់សម្រាប់អ្នកដែលចង់ក្លាយជា Full-Stack Developer។

ក្រៅពីនេះ សៀវភៅ JavaScript (ES2024) ក៏បានបង្ហាញថា ការយល់ពីមូលដ្ឋានគ្រឹះ Debugging, Testing និង Version Control ជារឿងចាំបាច់។ ក្នុងនាអនាគត ការយល់ដឹងច្បាស់លាស់អំពី JavaScript (ES2024) និងស្តង់ដាថ្មីៗនឹងជួយអ្នកអភិវឌ្ឍទទួលបានចំណេះដឹងទំនើបទាន់សម័យ ហើយបន្ថែមឱកាសក្នុងការរកការងារកាន់តែមានទំនុកចិត្តគ្រប់ស្ថាប័ននានា។

សេចក្តីសន្និដ្ឋាន សៀវភៅ «មូលដ្ឋានគ្រឹះនៃភាសា JavaScript (ES2024)» ជារូបមន្តចាប់ផ្តើមល្អបំផុតសម្រាប់អ្នកចង់ចូលមុខវិស័យ IT ហើយវាអាចប្រើបានជាឯកសារយោង សម្រាប់ការសិក្សា និងសិក្សាបន្ថែមផងដែរ។

១.២ អនុសាសន៍

ដើម្បីធានាថាការសិក្សា និងការអនុវត្ត JavaScript របស់អ្នកមានប្រសិទ្ធភាពខ្ពស់ និងអាចជួយអ្នកឱ្យក្លាយជាអ្នកជំនាញពិតប្រាកដ វាជាការចាំបាច់ដែលអ្នកត្រូវ ចូលរួមយ៉ាងសកម្មក្នុងសហគមន៍ JavaScript ដូចជា Stack Overflow, GitHub, និងបណ្តាញសង្គមផ្សេងៗ។ ការចូលរួមនេះមិនត្រឹមតែជួយអ្នកចែករំលែកចំណេះដឹង សួរសំណួរ និងស្វែងរកដំណោះស្រាយប៉ុណ្ណោះទេ តែថែមទាំងជួយអ្នកទទួលបានបទពិសោធន៍ថ្មីៗ និងពង្រីកទំនាក់ទំនងអាជីពរបស់អ្នកទៀតផង។ ក្រៅពីនេះ អ្នកគួរតែចំណាយពេលធ្វើ Mini Projects ជាប្រចាំ ដូចជា Weather App, To-Do List, ឬ Portfolio Website ហើយត្រូវ ទាញយកមតិកែលម្អ ពីអ្នកជំនាញ ឬសមាជិកសហគមន៍ជានិច្ច ដើម្បីកែលម្អជំនាញ Coding និងការរចនាប្រព័ន្ធរបស់អ្នក។ ការសិក្សា និងការអនុវត្ត Frameworks និង Libraries ទំនើបៗ ដូចជា React, Vue, Angular សម្រាប់ Front-End និង Node.js, Express.js សម្រាប់ Back-End ក៏មានសារៈសំខាន់ខ្លាំងណាស់ដែរ ព្រោះការយល់ដឹងស៊ីជម្រៅពីស្ថាបត្យកម្មទាំងនេះនឹងបង្កើនតម្លៃ និងឱកាសការងាររបស់អ្នកក្នុងទីផ្សារ។

ទន្ទឹមនឹងនេះ ការអនុវត្តការប្រើប្រាស់ Git សម្រាប់ Version Control គឺជាជំនាញមិនអាចខ្វះបានសម្រាប់ការសហការជាក្រុម និងការគ្រប់គ្រងកូដប្រកបដោយប្រសិទ្ធភាព ហើយការ អនុវត្ត Debugging ជាប្រចាំនឹងជួយអ្នកក្នុងការស្វែងរក និងដោះស្រាយបញ្ហាក្នុងកូដបានយ៉ាងឆាប់រហ័ស។ អ្នកក៏គួរតែ តាមដាន និងទទួលស្គាល់ការកែប្រែស្តង់ដារ ECMAScript (ដូចជា ES6+) និងសិក្សា រូបមន្ត ថ្មីៗ ដើម្បីធានាថាចំណេះដឹង JavaScript របស់អ្នកនៅតែទាន់សម័យ និងស្របតាមការវិវឌ្ឍន៍ចុងក្រោយ។ ការ អនុវត្តការសរសេរកូដដែលមានស្តង់ដារ (Clean Code) និងការ ចងក្រងឯកសារយោង (Documentation) ឱ្យបានល្អ ក៏ជាចំណុចគន្លឹះដែលធ្វើឱ្យកូដរបស់អ្នកងាយស្រួលអាន ថែទាំ និងចែករំលែកទៅអ្នកដទៃ។ ដើម្បីងាយស្រួលក្នុងការចូលប្រើប្រាស់ចំណេះដឹង និង Resources សំខាន់ៗ អ្នកគួរតែ រៀបចំ និងគ្រប់គ្រង Resources ផ្ទាល់ខ្លួន ឱ្យមានរបៀបរៀបរយ ដោយរក្សាទុកវាក្នុង Notion, GitHub Repo, ឬជា Cheat Sheets ផ្ទាល់ខ្លួន។ ជាចុងក្រោយ ការ រៀបចំផែនការរយៈពេលវែង សម្រាប់ផ្លូវសិក្សា JavaScript របស់អ្នក ដោយមាន Roadmap ច្បាស់លាស់ នឹងជួយអ្នកកំណត់ទិសដៅ និងសម្រេចបាននូវគោលដៅចុងក្រោយរបស់អ្នកក្នុងវិស័យនេះប្រកបដោយជោគជ័យ។

១.៣ Weather App Project

១.៣.១ សេចក្តីផ្តើម

Weather App នេះគឺជាកម្មវិធីអេឡិចត្រូនិចសម្រាប់មើលព័ត៌មានអាកាសធាតុដែលបង្កើតឡើងដោយប្រើភាសា HTML, CSS និង JavaScript។ កម្មវិធីនេះអាច៖

- ស្វែងរកព័ត៌មានអាកាសធាតុតាមទីក្រុង
- ប្រើទីតាំងបច្ចុប្បន្នដើម្បីមើលអាកាសធាតុ
- បង្ហាញព័ត៌មានអាកាសធាតុបច្ចុប្បន្ន និងទស្សន៍ទាយ 5 ថ្ងៃទៅមុខ
- រក្សាទុកទីក្រុងដែលចូលចិត្ត
- ផ្លាស់ប្តូរឯកតាសីតុណ្ហភាព (°C/°F)
- ផ្លាស់ប្តូររបៀបពណ៌ (ពណ៌ខ្មៅ/ស)

១.៣.២ រចនាសម្ព័ន្ធកូដ

- index.html file

  ឯកសារនេះគឺជារចនាសម្ព័ន្ធមូលដ្ឋានរបស់កម្មវិធី។ ក្នុងនេះមាន៖
  - ប្រអប់ស្វែងរកទីក្រុង
  - ប៊ូតុងប្រើទីតាំងបច្ចុប្បន្ន
  - ផ្នែកបង្ហាញព័ត៌មានអាកាសធាតុបច្ចុប្បន្ន
  - ផ្នែកទស្សន៍ទាយ 5 ថ្ងៃ
  - បញ្ជីទីក្រុងដែលអ្នកចូលចិត្ត
- main.css file

  ឯកសារនេះគ្រប់គ្រងរចនាប័ទ្មនៃកម្មវិធី។ ក្នុងនេះមាន៖
  - រចនាប័ទ្មទូទៅសម្រាប់ទម្រង់ពណ៌ផ្សេងៗ
  - ការបង្កើនភាពស្រស់ស្អាតនិងសម្រាប់ផ្លាស់ប្តូរពណ៌ខ្មៅ/ស
  - ផ្នែខាងក្រោយដែលផ្លាស់ប្តូរដោយអាស្រ័យលើលក្ខណៈអាកាសធាតុ
  - ការធ្វើឱ្យទាន់សម័យនៅពេលប្រើឧបករណ៍ផ្សេងៗ
- api.js file

  ជាម៉ូឌុលសម្រាប់ទាក់ទងជាមួយ OpenWeatherMap API។ ក្នុងនេះមាន៖
  - មុខងារដើម្បីទាញយកទិន្នន័យអាកាសធាតុបច្ចុប្បន្ន
  - មុខងារដើម្បីទាញយកទិន្នន័យទស្សន៍ទាយ
  - ដំណើរការទិន្នន័យទស្សន៍ទាយ

- app.js file

  ជាម៉ូឌុលសំខាន់ដែលគ្រប់គ្រងតក្កវិធីកម្មវិធី។ ក្នុងនេះមាន៖

  - ការគ្រប់គ្រង state របស់កម្មវិធី
  - ការដំឡើង event listeners
  - ការគ្រប់គ្រងអ្នកប្រើប្រាស់
  - ការហៅ API ដើម្បីទទួលទិន្នន័យអាកាសធាតុ

- storage.js file

  ជាម៉ូឌុលសម្រាប់គ្រប់គ្រងទិន្នន័យក្នុង localStorage។ វាមាន:

  - មុខងារដើម្បីទទួលទីក្រុងដែលអ្នកចូលចិត្ត
  - មុខងារដើម្បីបន្ថែមទីក្រុងដែលអ្នកចូលចិត្ត
  - មុខងារដើម្បីដកចេញទីក្រុងដែលអ្នកចូលចិត្ត

- ui.js file

  ជាម៉ូឌុលសម្រាប់ធ្វើបច្ចុប្បន្នភាព UI។ ក្នុងនេះមាន៖

  - មុខងារដើម្បីធ្វើបច្ចុប្បន្នភាពព័ត៌មានអាកាសធាតុបច្ចុប្បន្ន
  - មុខងារដើម្បីធ្វើបច្ចុប្បន្នភាពទស្សន៍ទាយ
  - មុខងារដើម្បីកំណត់ផ្ទៃខាងក្រោយតាមលក្ខណៈអាកាសធាតុ
  - មុខងារដើម្បីបង្ហាញនិងលុបកំហុស

- utils.js file

  ជាម៉ូឌុលជំនួយដែលមានមុខងារបំលែង៖

  - បំលែងកាលបរិច្ឆេទ
  - បំលែងឯកតាសីតុណ្ហភាពពី Kelvin ទៅ Celsius ឬ Fahrenheit

១.៣.៣ លក្ខណៈពិសេសនៃកម្មវិធី

- ការស្វែងរកអាកាសធាតុ៖
  - អ្នកប្រើប្រាស់អាចស្វែងរកអាកាសធាតុតាមឈ្មោះទីក្រុង
  - ឬប្រើទីតាំងបច្ចុប្បន្នរបស់ពួកគេ
- ទិន្នន័យអាកាសធាតុ៖
  - សីតុណ្ហភាពបច្ចុប្បន្ន
  - អាកាសធាតុសរុប
  - សីតុណ្ហភាពមានអារម្មណ៍ថា
  - សំណើម

- ល្បឿនខ្យល់

- ទស្សន៍ទាយ 5 ថ្ងៃ៖
  - បង្ហាញទិន្នន័យទស្សន៍ទាយសម្រាប់ 5 ថ្ងៃបន្ទាប់
  - រួមមានរូបតំណាងអាកាសធាតុ និងសីតុណ្ហភាព
- ទីក្រុងដែលចូលចិត្ត៖
  - អ្នកប្រើប្រាស់អាចរក្សាទុកទីក្រុងដែលពួកគេចូលចិត្ត
  - ចុចលើទីក្រុងដើម្បីមើលអាកាសធាតុឡើងវិញ
- ការផ្លាស់ប្តូរឯកតាសីតុណ្ហភាព៖
  - ប្តូររវាង Celsius និង Fahrenheit ដោយសាមញ្ញ
- ការផ្លាស់ប្តូរពណ៌ខ្មៅ/ស៖
  - អាចប្តូររវាងពណ៌ខ្មៅនិងស
  - រក្សាការចំណាំជាប់លាប់នៅក្នុង localStorage
- ផ្ទៃខាងក្រោយដែលផ្លាស់ប្តូរដោយអាស្រ័យលើអាកាសធាតុ៖
  - ផ្ទៃខាងក្រោយផ្លាស់ប្តូរតាមលក្ខណៈអាកាសធាតុ (ពន្លឺថ្ងៃ, ពន្លឺយប់, ភ្លៀង, ពពក, ព្រិល)

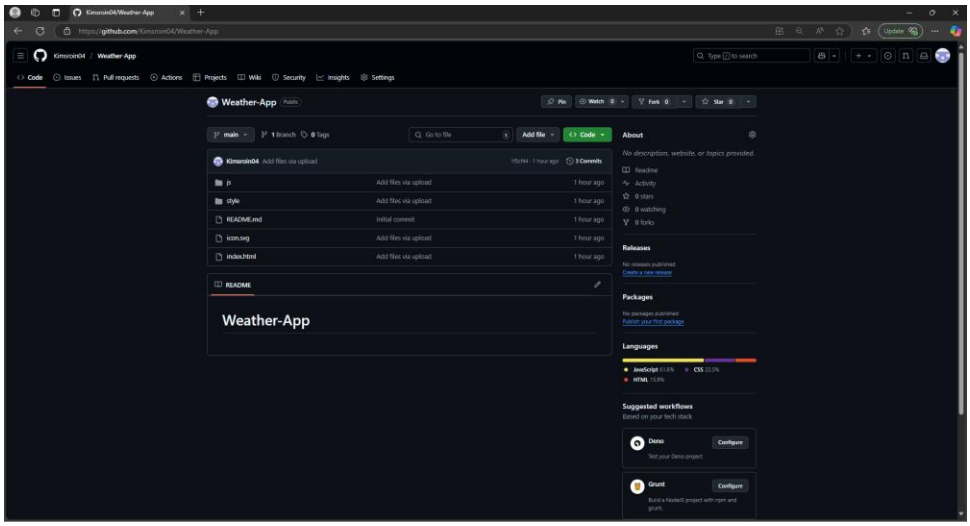
១.៣.៤ ដំណើរការ

- នៅពេលអ្នកប្រើប្រាស់បញ្ចូលទីក្រុង ឬប្រើទីតាំងបច្ចុប្បន្ន៖
  - កម្មវិធីហៅ API របស់ OpenWeatherMap
  - ទទួលទិន្នន័យអាកាសធាតុបច្ចុប្បន្ន
  - ទទួលទិន្នន័យទស្សន៍ទាយ 5 ថ្ងៃ
- ទិន្នន័យត្រូវបានដំណើរការ និងបង្ហាញនៅលើ UI៖
  - ព័ត៌មានអាកាសធាតុបច្ចុប្បន្នត្រូវបានបង្ហាញនៅផ្នែកខាងលើ
  - ទស្សន៍ទាយ 5 ថ្ងៃត្រូវបានបង្ហាញនៅផ្នែកខាងក្រោម
- អ្នកប្រើប្រាស់អាច៖
  - បន្ថែមទីក្រុងទៅបញ្ជីចូលចិត្ត
  - ផ្លាស់ប្តូរឯកតាសីតុណ្ហភាព
  - ផ្លាស់ប្តូររបៀបពណ៌

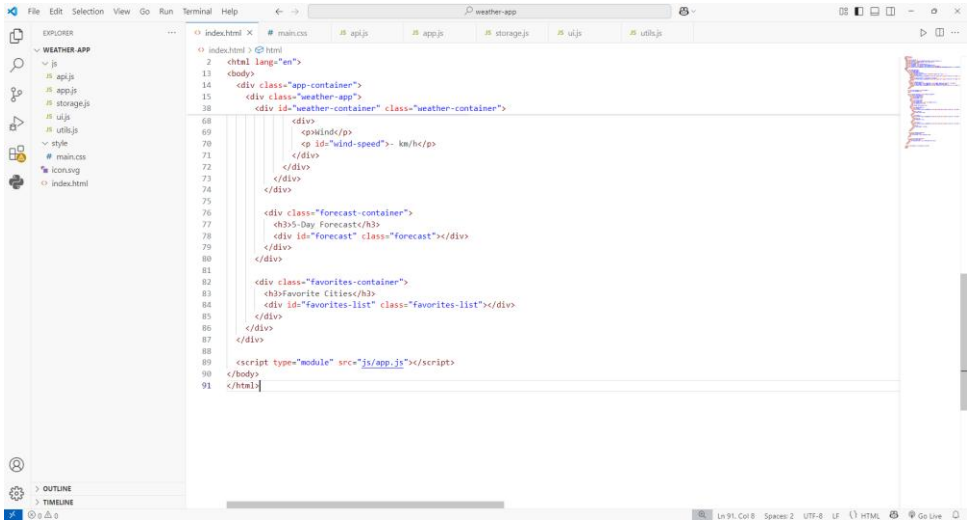
១.៤ ទាញយកឯកសារ

លោកអ្នកអាចទាញយកឯកសារអំពី Project នេះបានតាមតំណភ្ជាប់ (Link) ខាងក្រោម៖

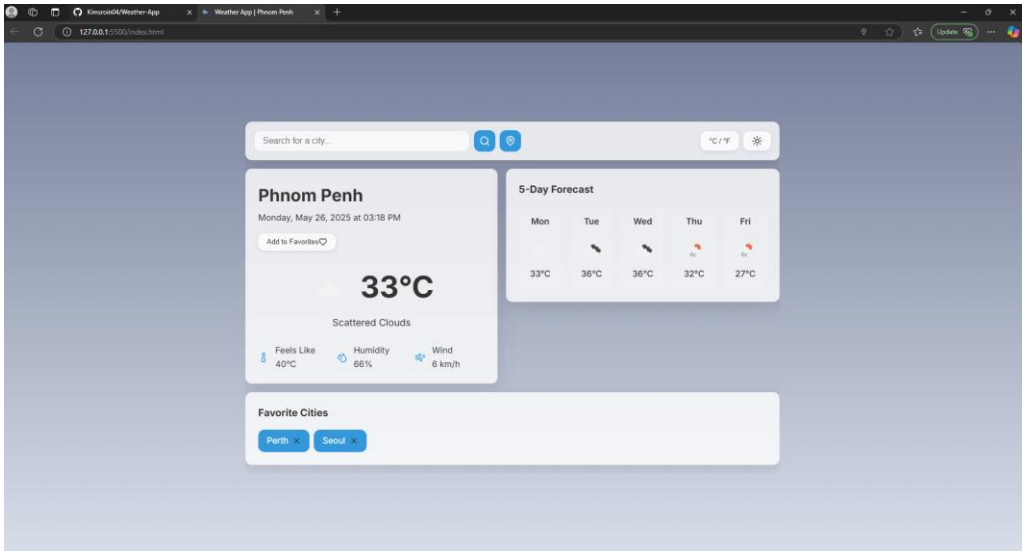
https://github.com/Kimsroin04/Weather-App



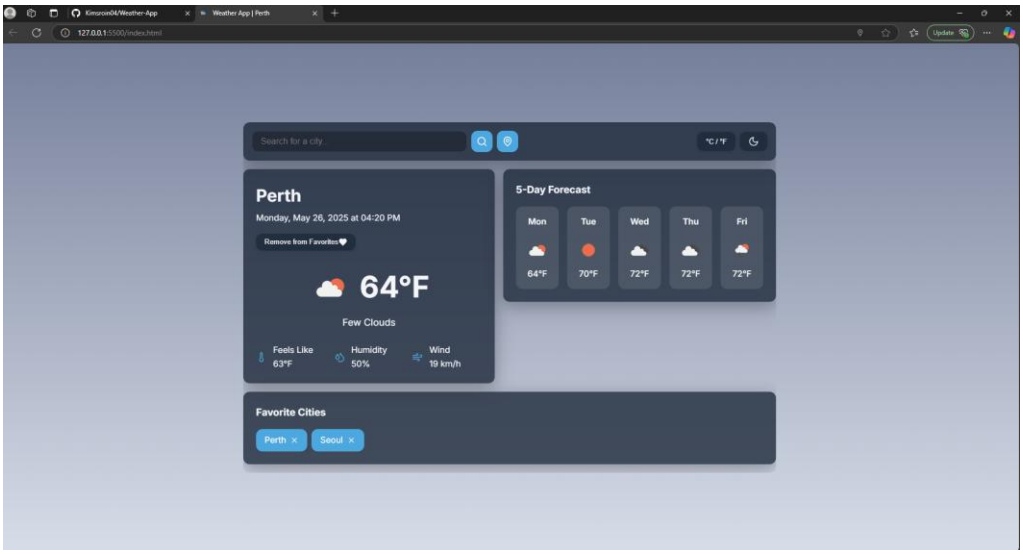
រូបភាព 20៖ Weather-App project



រូបភាព 21៖ Run Project



រូបភាព 22៖ Phnom Penh



រូបភាព 23៖ Perth

ឯកសារយោង

ECMA International. (2024). ECMAScript® 2024 language specification (ECMA-262). https://www.ecma-international.org/publications-and-standards/standards/ecma-262/

Resig, J., & Bibeault, B. (2013). Secrets of the JavaScript ninja (2nd ed.). Shelter Island, NY: Manning Publications.

Freeman, E., & Robson, E. (2014). Head first JavaScript programming: A brain-friendly guide. Sebastopol, CA: O'Reilly Media.

Zakas, N. C. (2016). Understanding ECMAScript 6: The definitive guide for JavaScript developers. Shelter Island, NY: No Starch Press.

Osmani, A. (2018). Learning JavaScript design patterns (2nd ed.). Sebastopol, CA: O'Reilly Media.

Shah, A. (2021). Modern JavaScript for beginners: Learn JavaScript fundamentals through hands-on projects. Independently published.

Berry, J. (2022). JavaScript from beginner to professional. Independently published.

Mozilla Contributors. (2023). JavaScript guide. Mozilla Developer Network. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide

Wieruch, R. (2018). The road to learn React. Independently published.

Duffy, K. (2020). Modern JavaScript: Develop and design. Boston, MA: Addison-Wesley.

Simpson, K. (2020). You don't know JS yet: Get started. Independently published.

Warwickshire, UK: In Easy Steps Limited.

Beighley, L., & Morrison, M. (2014). Head first HTML and CSS (2nd ed.). Sebastopol, CA: O'Reilly Media.

Duckett, J. (2014). JavaScript and jQuery: Interactive front-end web development. Indianapolis, IN: Wiley.

Meloni, J. C., & Kyrnin, J. (2018). HTML, CSS, and JavaScript all in one (2nd ed.). Indianapolis, IN: Sams Publishing.

Zakas, N. C. (2021). Principles of object-oriented JavaScript. Sebastopol, CA: O'Reilly Media.

Freeman, A. (2021). Pro JavaScript for web apps: Modern techniques for real-world JavaScript development. Berkeley, CA: Apress.

Flanagan, D. (2020). JavaScript: The definitive guide (7th ed.). Sebastopol, CA: O'Reilly Media.

ឧបសម្ព័ន្ធ

ព្រះរាជាណាចក្រកម្ពុជា
ជាតិ សាសនា ព្រះមហាក្សត្រ

វិទ្យាស្ថានជាតិអប់រំ

សូមគោរពជូន
ឯកឧត្តមនាយកវិទ្យាស្ថានជាតិអប់រំ

កម្មវត្ថុ៖ សំណើសុំសរសេរសៀវភៅនិក្ខេបបទ ដើម្បីបំពេញលក្ខខណ្ឌបញ្ចប់ការសិក្សាបណ្តុះបណ្តាលបរិញ្ញាបត្រជាន់ខ្ពស់អប់រំថ្នាក់ជីវៈគ្រូបង្រៀនកម្រិតឧត្តម(បរិញ្ញាបត្រ+២) ជំនាន់ទី២។

ខ្ញុំបាទឈ្មោះ: **សារឿន គីមស្រ៊ាន** ជានិស្សិតកម្រិតឧត្តម (បរិញ្ញាបត្រ+២) ជំនាន់ទី២ មុខវិជ្ជាឯកទេសព័ត៌មានវិទ្យា ក្រុមទី២ បានទទួលការឯកភាពពីគ្រូណែនាំគោលឈ្មោះ: **ឈឹម គីមឃុន** ជាបុគ្គលិកអប់រំ/គ្រូឧទ្ទេសមុខវិជ្ជា ព័ត៌មានវិទ្យា និងគ្រូណែនាំរងឈ្មោះ: **រុយ សេរីសោភា** ជាបុគ្គលិកអប់រំ/គ្រូឧទ្ទេសមុខវិជ្ជា ព័ត៌មានវិទ្យា ដើម្បីសរសេរសៀវភៅនិក្ខេបបទ ក្រោមប្រធានបទ "**មូលដ្ឋានគ្រឹះនៃភាសា JavaScript (ES2024)**" ក្នុងគោលបំណងបំពេញលក្ខខណ្ឌបញ្ចប់ការសិក្សាបណ្តុះបណ្តាលបរិញ្ញាបត្រជាន់ខ្ពស់អប់រំ ជ្នាជីវៈគ្រូបង្រៀនកម្រិតឧត្តម(បរិញ្ញាបត្រ+២) ជំនាន់ទី២។

អាស្រ័យដូចបានគោរពខាងលើ សូម **ឯកឧត្តមនាយកវិទ្យាស្ថានជាតិអប់រំ** មេត្តាពិនិត្យ និងសម្រេចដោយសេចក្តីអនុគ្រោះ។

សូម **ឯកឧត្តមនាយកវិទ្យាស្ថានជាតិអប់រំ** មេត្តាទទួលនូវការគោរពដ៏ខ្ពង់ខ្ពស់ពីខ្ញុំ

ថ្ងៃព្រហស្បតិ៍ ៥កើត ខែមិគសិរ ឆ្នាំរោង ឆស័ក ព.ស.២៥៦៨
រាជធានីភ្នំពេញ ថ្ងៃទី ០៥ ខែ ធ្នូ ឆ្នាំ២០២៥
ហត្ថលេខា

សារឿន គីមស្រ៊ាន

បានឃើញ និងគោរពជូន	បានឃើញ និងគោរពជូន
ឯកឧត្តមនាយកវិទ្យាស្ថានជាតិអប់រំ ពិនិត្យ និងសម្រេច	ឯកឧត្តមនាយកវិទ្យាស្ថានជាតិអប់រំ ពិនិត្យ និងសម្រេច
គ្រូណែនាំគោល	**គ្រូណែនាំរង**
ឈឹម គីមឃុន	**រុយ សេរីសោភា**

ការិយាល័យស្រាវជ្រាវ

បានឃើញ និងឯកភាព
នាយកវិទ្យាស្ថានជាតិអប់រំ

ព្រះរាជាណាចក្រកម្ពុជា
ជាតិ សាសនា ព្រះមហាក្សត្រ

វិទ្យាស្ថានជាតិអប់រំ
លេខ :..៣៣៧/២..ឧជអ. ឋ.សាន

សេចក្តីជូនដំណឹង

យោង៖ -ផែនការអនុវត្តកម្មវិធីបណ្តុះបណ្តាលគរុនិស្សិតកម្រិតឧត្តម (បរិញ្ញាបត្រ+២) ជំនាន់ទី២ ឆ្នាំសិក្សា ២០២៣-២០២៤ និង២០២៤-២០២៥

-លិខិតលេខ៦១៣ វជអ.សជ ស្តីពីការការពារនិក្ខេបបទ សៀវភៅ និង/ឬអត្ថបទស្រាវជ្រាវ ចុះថ្ងៃទី២៧ ខែវិច្ឆិកា ឆ្នាំ២០២៤។

វិទ្យាស្ថានជាតិអប់រំ សូមជូនដំណឹងដល់គណៈគ្រប់គ្រង បុគ្គលិកអប់រំ គ្រូឧទ្ទេស និងគរុនិស្សិតកម្រិតឧត្តម(បរិញ្ញាបត្រ+២)ជំនាន់ទី២ ជ្រាបថា៖ ការការពារបញ្ចប់និក្ខេបបទស្រាវជ្រាវ ដែលត្រូវការពារបញ្ចប់នៅថ្ងៃទី១៤-១៨ ខែកក្កដា ឆ្នាំ២០២៥ ត្រូវបានលើកទៅថ្ងៃទី២១-២៣ ខែកក្កដា ឆ្នាំ២០២៥វិញ។ ដោយមូលហេតុគ្រូឧទ្ទេស និស្សិត គរុនិស្សិត និងគរុសិស្សត្រូវចូលរួមសិក្ខាសាលាស្តីពី "ការបំប៉នសមត្ថភាពបុគ្គលិក គ្រូឧទ្ទេស និស្សិត គរុនិស្សិត និងគរុសិស្ស"នៅវិទ្យាស្ថានជាតិអប់រំ។

អាស្រ័យហេតុនេះ សូម លោក/លោកស្រីជាគណៈគ្រប់គ្រង បុគ្គលិកអប់រំ គ្រូឧទ្ទេស និង គរុនិស្សិត ជ្រាបជាព័ត៌មាន។

ថ្ងៃព្រហស្បតិ៍ [illegible] ខែ[illegible] ឆ្នាំម្សាញ់ សប្តស័ក ព.ស.២៥៦៩
រាជធានីភ្នំពេញ ថ្ងៃទី១០ ខែកក្កដា ឆ្នាំ២០២៥
នាយកវិទ្យាស្ថានជាតិអប់រំ

បណ្ឌិត.សៀង សុចន្ណា

ទម្រង់វាយតម្លៃ និងឯកសារពាក់ព័ន្ធ

អាសយដ្ឋានលេខ ១២៣ មហាវិថី ព្រះនរោត្តម ភ្នំពេញ កម្ពុជា ទូរស័ព្ទ (៨៥៥-២៣) ២១៧/៨៨៨ គេហទំព័រ www.nie.edu.kh

លិខិតបញ្ជាក់

លោក លឹម គឹមឃុន ជាគ្រូណែនាំគោល
លោក រុយ សិរីសោភា ជាគ្រូណែនាំរង

សូមបញ្ជាក់ និងទទួលស្គាល់ថា

លោក សារឿន គឹមស្រ៊ាន់ ជាគរុនិស្សិតកម្រិតឧត្តម(បរិញ្ញាបត្រ+២) ជំនាន់ទី២ ពិតជាបានសរសេរសៀវភៅក្រោមប្រធានបទ «**មូលដ្ឋានគ្រឹះនៃភាសា JAVASCRIPT(ES2024)**» ពិតប្រាកដមែន។

ថ្ងៃចន្ទ ៣កើត ខែស្រាពណ៍ ឆ្នាំម្សាញ់ សប្តស័ក ព.ស. ២៥៦៩
រាជធានីភ្នំពេញ ថ្ងៃទី២៨ ខែកក្កដា ឆ្នាំ២០២៥

.....	.....
លោក លឹម គឹមឃុន	**លោក រុយ សិរីសោភា**
គ្រូឧទ្ទេសនៃវិទ្យាស្ថានជាតិអប់រំ	គ្រូឧទ្ទេសនៃវិទ្យាស្ថានជាតិអប់រំ

ព្រះរាជាណាចក្រកម្ពុជា
ជាតិ សាសនា ព្រះមហាក្សត្រ

ពាក្យសុំចុះឈ្មោះសរសេរសៀវភៅ

ខ្ញុំបាទឈ្មោះ សារឿន គឹមស្រ៊ាន់ ភេទប្រុស កើតថ្ងៃទី២៨ ខែកញ្ញា ឆ្នាំ១៩៩៩ ជាគរុនិស្សិត កម្រិតឧត្តម(បរិញ្ញាបត្រ+២) ជំនាន់ទី២ ឯកទេសព័ត៌មានវិទ្យា ក្រុម២ នៃវិទ្យាស្ថានជាតិអប់រំ។

សូមគោរពជូន
ឯកឧត្តមនាយកវិទ្យាស្ថានជាតិអប់រំ

កម្មវត្ថុ ៖ សំណើសុំចុះឈ្មោះសរសេរសៀវភៅនវានុវត្តន៍។
តាមរយៈ ៖ ការិយាល័យស្រាវជ្រាវអប់រំ នៃវិទ្យាស្ថានជាតិអប់រំ។

យោងតាមកម្មវិធីបណ្តុះបណ្តាលបរិញ្ញាបត្រជាន់ខ្ពស់អប់រំ វិជ្ជាជីវៈគ្រូបង្រៀនកម្រិតឧត្តម (បរិញ្ញាបត្រ+២) នៅវិទ្យាស្ថានជាតិអប់រំ ខ្ញុំបាទត្រូវសរសេរសៀវភៅដើម្បីបំពេញលក្ខខណ្ឌបញ្ចប់ការបណ្តុះបណ្តាល។

អាស្រ័យហេតុនេះ សូម **ឯកឧត្តមនាយក** មេត្តាអនុញ្ញាតឱ្យខ្ញុំបាទចុះឈ្មោះសរសេរសៀវភៅលើប្រធានបទ **«មូលដ្ឋានគ្រឹះនៃភាសា JAVASCRIPT(ES2024)»** ក្រោមការណែនាំរបស់**លោក លឹម គឹមឃុន** ជាគ្រូណែនាំគោល និង**លោក រុយ សិរីសោភា** ជាគ្រូណែនាំរង ដោយសេចក្តីអនុគ្រោះ ។

សូម **ឯកឧត្តមនាយក** ទទួលនូវការគោរពដ៏ខ្ពង់ខ្ពស់ពីខ្ញុំ

ថ្ងៃពុធ ១១កើត ខែមិគសិរ ឆ្នាំរោង ឆស័ក ព.ស.២៥៦៨
រាជធានីភ្នំពេញ ថ្ងៃទី១១ ខែធ្នូ ឆ្នាំ២០២៤
ហត្ថលេខា

សារឿន គឹមស្រ៊ាន់

បានឃើញ និងឯកភាព
នាយកវិទ្យាស្ថានជាតិអប់រំ

ព្រះរាជាណាចក្រកម្ពុជា
ជាតិ សាសនា ព្រះមហាក្សត្រ

ពាក្យសុំការពារសៀវភៅ

ខ្ញុំបាទឈ្មោះ សារឿន គឹមស្រ៊ាន់ ភេទប្រុស កើតថ្ងៃទី២៨ ខែកញ្ញា ឆ្នាំ១៩៩៩ ជាគរុនិស្សិតកម្រិតឧត្តម(បរិញ្ញាបត្រ+២) ជំនាន់ទី២ ឯកទេសព័ត៌មានវិទ្យា ក្រុម២ នៃវិទ្យាស្ថានជាតិអប់រំ។

សូមគោរពជូន
ឯកឧត្តមនាយកវិទ្យាស្ថានជាតិអប់រំ

កម្មវត្ថុ ៖ សំណើសុំការពារសៀវភៅ។
តាមរយៈ ៖ ការិយាល័យស្រាវជ្រាវអប់រំ នៃវិទ្យាស្ថានជាតិអប់រំ។

យោងតាមកម្មវិធីបណ្តុះបណ្តាលបរិញ្ញាបត្រជាន់ខ្ពស់អប់រំ វិជ្ជាជីវៈគ្រូបង្រៀនកម្រិតឧត្តម(បរិញ្ញាបត្រ+២) នៅវិទ្យាស្ថានជាតិអប់រំ ខ្ញុំបាទស្នើសុំការពារសៀវភៅដើម្បីបំពេញតាមលក្ខខណ្ឌបណ្តុះបណ្តាល មុនប្រឡងបញ្ចប់ការសិក្សា។

អាស្រ័យហេតុនេះ សូម **ឯកឧត្តមនាយក** មេត្តាអនុញ្ញាតឱ្យខ្ញុំបាទការពារសៀវភៅលើប្រធានបទ « **មូលដ្ឋានគ្រឹះនៃភាសា JAVASCRIPT(ES2024)** » ក្រោមការណែនាំរបស់**លោក លឹម គឹមឃុន** ជាគ្រូណែនាំគោល និង**លោក រុយ សិរីសោភា** ជាគ្រូណែនាំរង នៅថ្ងៃទី២១ ដល់ថ្ងៃទី២៣ ខែកក្កដា ឆ្នាំ២០២៥ ដោយសេចក្តីអនុគ្រោះ។

សូម **ឯកឧត្តមនាយក** ទទួលនូវការគោរពដ៏ខ្ពង់ខ្ពស់ពីខ្ញុំ

ថ្ងៃចន្ទ ១៣រោច ខែជេស្ឋ ឆ្នាំម្សាញ់ សប្តស័ក ព.ស.២៥៦៩
រាជធានីភ្នំពេញ ថ្ងៃទី២៣ ខែមិថុនា ឆ្នាំ២០២៥
ហត្ថលេខា

សារឿន គឹមស្រ៊ាន់

បានឃើញ និងឯកភាព
នាយកវិទ្យាស្ថានជាតិអប់រំ

ព្រះរាជាណាចក្រកម្ពុជា
ជាតិ សាសនា ព្រះមហាក្សត្រ

ពាក្យសុំបោះពុម្ពសៀវភៅ

ខ្ញុំបាទឈ្មោះ សារឿន គឹមស្រ៊ាន់ ភេទប្រុស កើតថ្ងៃទី២៨ ខែកញ្ញា ឆ្នាំ១៩៩៩ ជាគរុនិស្សិតកម្រិតឧត្តម(បរិញ្ញាបត្រ+២) ជំនាន់ទី២ ឯកទេសព័ត៌មានវិទ្យា ក្រុម២ នៃវិទ្យាស្ថានជាតិអប់រំ។

សូមគោរពជូន
ឯកឧត្តមនាយកវិទ្យាស្ថានជាតិអប់រំ

កម្មវត្ថុ ៖ សំណើសុំបោះពុម្ពសៀវភៅ។
តាមរយៈ ៖ ការិយាល័យស្រាវជ្រាវអប់រំ នៃវិទ្យាស្ថានជាតិអប់រំ។

យោងតាមកម្មវិធីបណ្តុះបណ្តាលបរិញ្ញាបត្រជាន់ខ្ពស់អប់រំ វិជ្ជាជីវៈគ្រូបង្រៀនកម្រិតឧត្តម(បរិញ្ញាបត្រ+២) នៅវិទ្យាស្ថានជាតិអប់រំ ខ្ញុំបាទស្នើសុំបោះពុម្ពដើម្បីបំពេញលក្ខខណ្ឌបញ្ចប់ការបណ្តុះបណ្តាលបរិញ្ញាបត្រជាន់ខ្ពស់អប់រំ វិជ្ជាជីវៈគ្រូបង្រៀនកម្រិតឧត្តម(បរិញ្ញាបត្រ+២) នៅវិទ្យាស្ថានជាតិអប់រំ។

អាស្រ័យហេតុនេះ សូម **ឯកឧត្តមនាយក** មេត្តាអនុញ្ញាតឱ្យខ្ញុំបាទបោះពុម្ពសៀវភៅលើប្រធានបទ **«មូលដ្ឋានគ្រឹះនៃភាសា JAVASCRIPT(ES2024)»** ក្រោមការណែនាំរបស់**លោក លឹម គឹមឃុន** ជាគ្រូណែនាំគោល និង**លោក រុយ សិរីសោភា** ជាគ្រូណែនាំរង ដោយសេចក្តីអនុគ្រោះ។

សូម **ឯកឧត្តមនាយក** ទទួលនូវការគោរពដ៏ខ្ពង់ខ្ពស់ពីខ្ញុំ

ថ្ងៃចន្ទ ៣កើត ខែស្រាពណ៍ ឆ្នាំម្សាញ់ សប្តស័ក ព.ស. ២៥៦៩
រាជធានីភ្នំពេញ ថ្ងៃទី២៨ ខែកក្កដា ឆ្នាំ២០២៥
ហត្ថលេខា

សារឿន គឹមស្រ៊ាន់

បានឃើញ និងឯកភាព
ឯកឧត្តមនាយកវិទ្យាស្ថានជាតិអប់រំ