โปรแกรมเชิงโครงสร้าง

Structured program

ผู้ช่วยศาสตราจารย์สมเกียรติ ช่อเหมือน (tko@webmail.npru.ac.th)

สาขาวิชาวิศวกรรมซอฟต์แวร์ คณะวิทยาศาสตร์และเทคโนโลยี

บทนำ

- การใช้โครงสร้างการควบคุมการไหลของโปรแกรมอย่างเป็นระบบ
  - การใช้ลูป (loops)
  - การตัดสินใจ (decisions)
  - โปรแกรมย่อย (subroutines)
- จัดการกับความซับซ้อนของโค้ด
- ทำให้โปรแกรมมีโครงสร้างที่ดี

imageURL

การเขียนโปรแกรมแบบมีโครงสร้าง

- การเข้าใจลักษณะของโครงสร้างการควบคุม **control flow**
- การใช้งานฟังก์ชันและโพรซีเยอร์ **Procedure**
- การจัดการข้อผิดพลาด **ERROR**
- การจัดการกับข้อมูล **Data Structure**
- การเขียนโค้ดที่มีโครงสร้าง **Structure Programming**

if, else, switch, for, while, do-while

ใช้ซ้ำได้

ข้อผิดพลาด

โครงสร้างข้อมูล

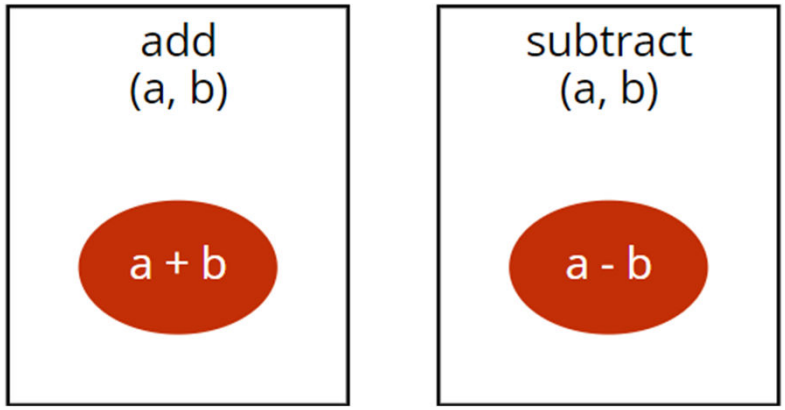
จัดโครงสร้างชัดเจน

โปรแกรมแบบมีโครงสร้าง

- การออกแบบโปรแกรมในลักษณะที่มีโครงสร้างชัดเจน
- การแบ่งโค้ดออกเป็นฟังก์ชันตามลักษณะของการทำงาน

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

function subtract(a, b) {
    return a - b;
}
```

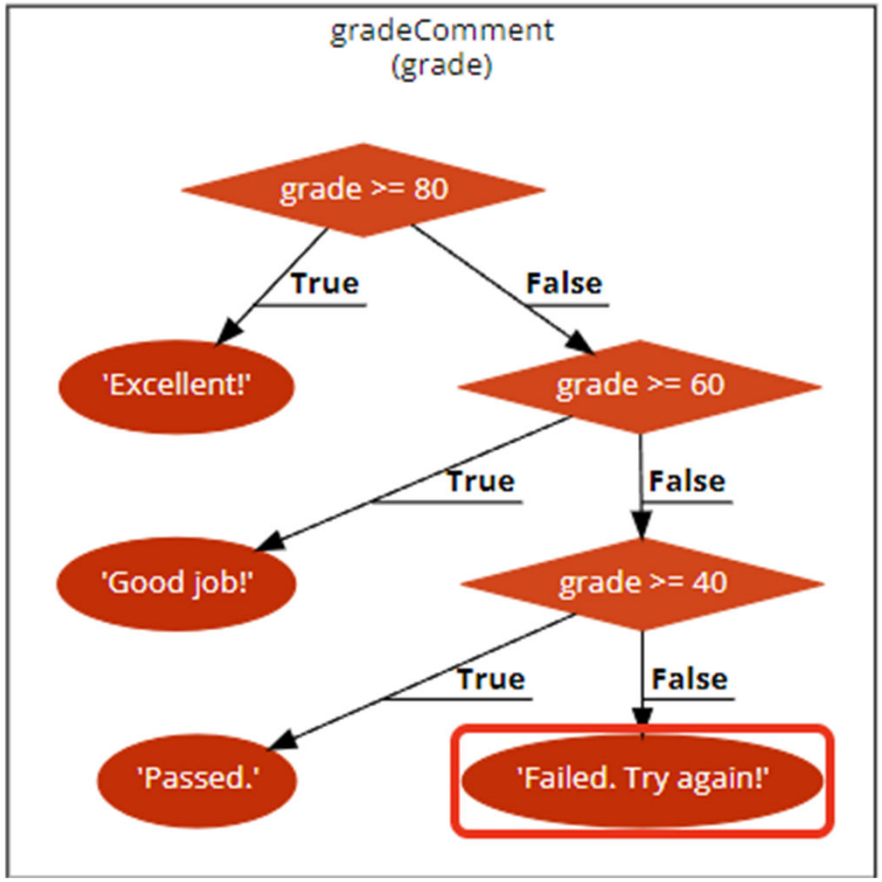
```
let sum = add(5, 3);
let difference = subtract(10, 6);
console.log(sum);        // Output: 8
console.log(difference); // Output: 4
```



โครงสร้างการควบคุม

- การใช้ if/else ในการตัดสินใจ

```
function gradeComment(grade) {
  if (grade >= 80) {
    return 'Excellent!';
  } else if (grade >= 60) {
    return 'Good job!';
  } else if (grade >= 40) {
    return 'Passed.';
  } else {
    return 'Failed. Try again!';
  }
}
```

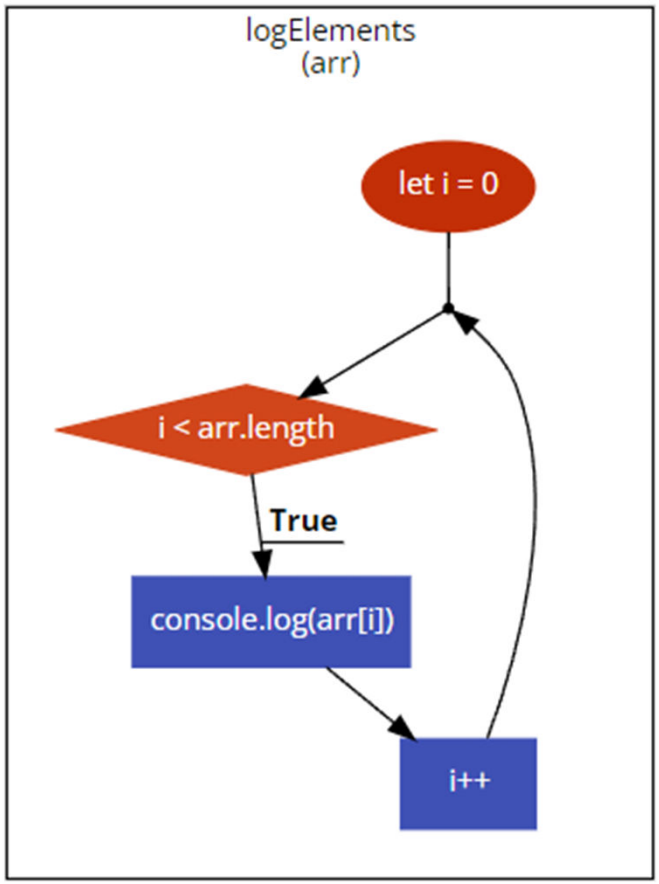



```
console.log(gradeComment(85)); // 'Excellent!'
```

โครงสร้างการควบคุม

- การใช้ for loop เพื่อวนซ้ำผ่านอาร์เรย์

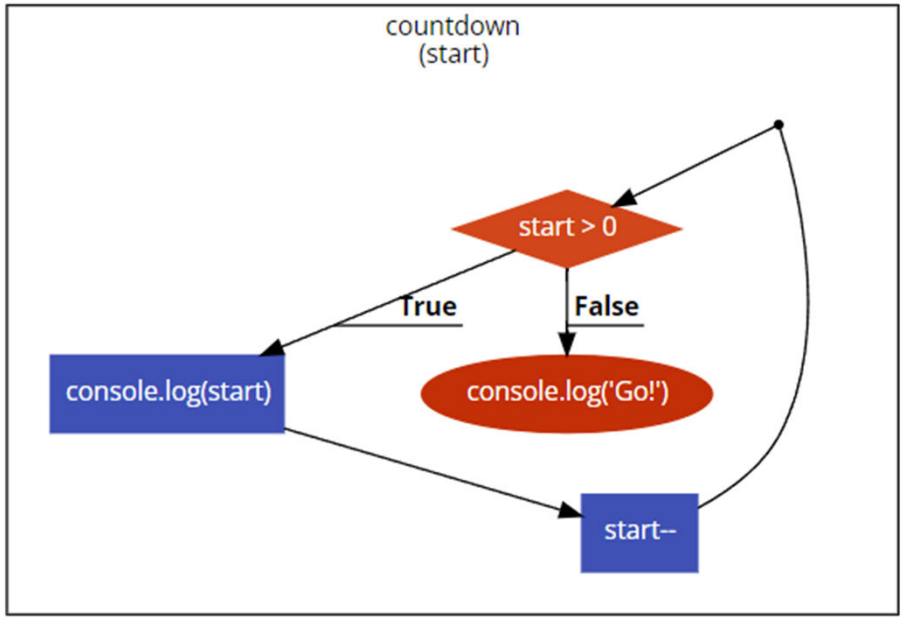
```
function logElements(arr) {
  for (let i = 0; i < arr.length; i++) {
    console.log(arr[i]);
  }
}

logElements([1, 2, 3, 4, 5]); // Logs each element
```



โครงสร้างการควบคุม

- การใช้ while loop เพื่อทำซ้ำจนกว่าเงื่อนไขจะเป็นเท็จ

```
function countdown(start) {
  while (start > 0) {
    console.log(start);
    start--;
  }
  console.log('Go!');
}
countdown(5); // Logs countdown from 5 to 'Go!'
```

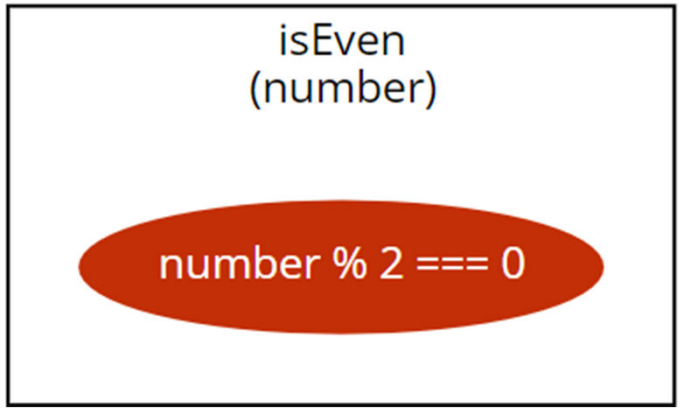


โครงสร้างการควบคุม

- การใช้ฟังก์ชันในการแยก logic จากโค้ดหลัก

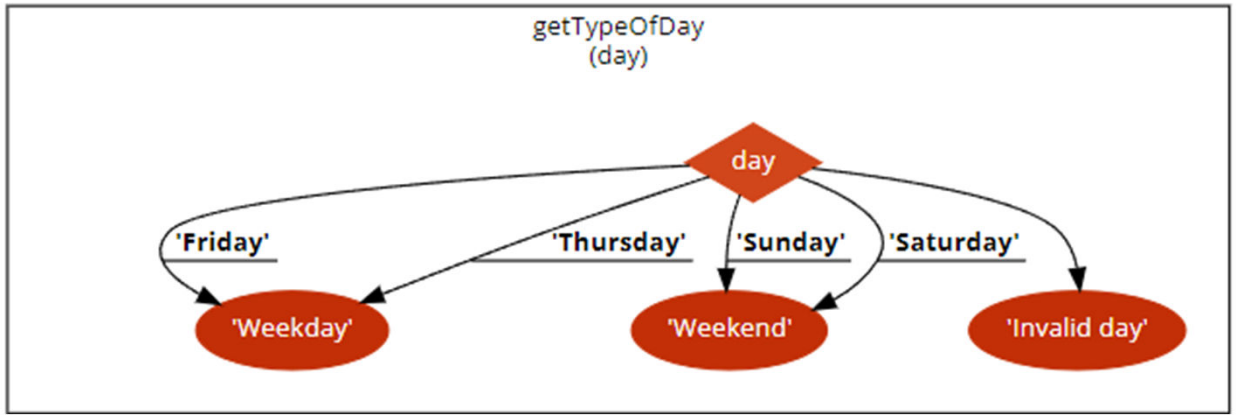
```
function isEven(number) {
  return number % 2 === 0;
}
```

```
console.log(isEven(10)); // true
```



โครงสร้างการควบคุม

- การใช้ switch ในการจัดการหลายเงื่อนไข




```
console.log(getTypeOfDay('Saturday')); // 'Weekend'
```

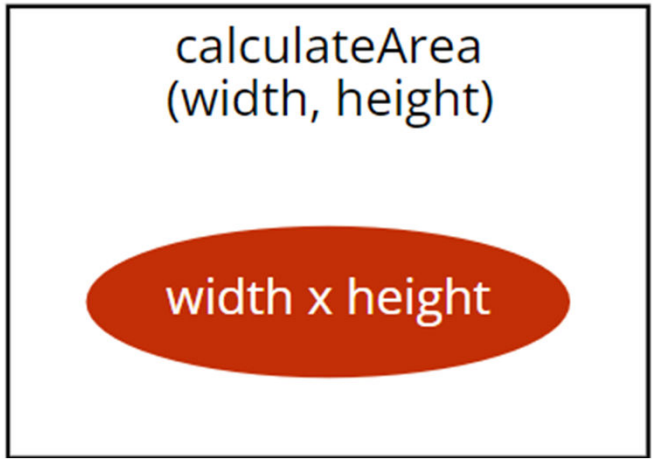
```
function getTypeOfDay(day) {
  switch (day) {
    case 'Monday':
    case 'Tuesday':
    case 'Wednesday':
    case 'Thursday':
    case 'Friday':
      return 'Weekday';
    case 'Saturday':
    case 'Sunday':
      return 'Weekend';
    default:
      return 'Invalid day';
  }
}
```

การใช้งานฟังก์ชันและโพรซีเดอร์

- ฟังก์ชันที่รับค่าอินพุตและคืนค่าผลลัพธ์

```
function calculateArea(width, height) {
  return width * height;
}
```

```
const area = calculateArea(5, 3); // คืนค่า 15
```

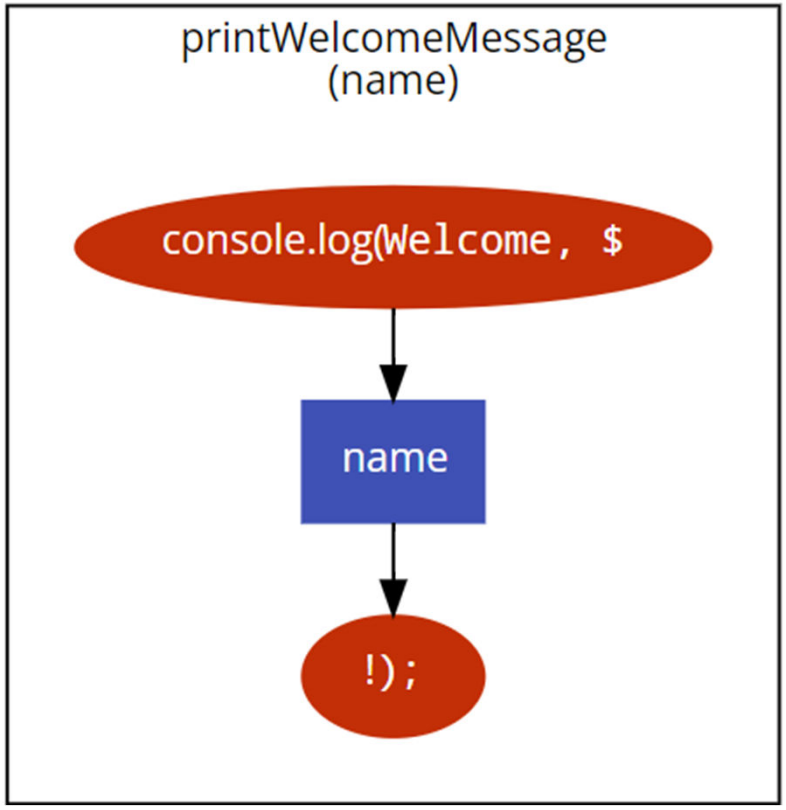


การใช้งานฟังก์ชันและโพรซีเดอร์

- โพรซีเดอร์ที่ทำการดำเนินการแต่ไม่คืนค่าผลลัพธ์

```
function printWelcomeMessage(name) {
  console.log(`Welcome, ${name}!`);
}
```

```
printWelcomeMessage('Alice'); // แสดง "Welcome, Alice!"
```



การใช้งานฟังก์ชันและโพรซีเดอร์

- ฟังก์ชันที่แสดงการแยกส่วนของความรับผิดชอบ

```
function convertTemperature(celsius) {
  const fahrenheit = (celsius * 9/5) + 32;
  return fahrenheit;
}

displayConvertedTemperature(30); // แสดง "30°C is 86°F"

function displayConvertedTemperature(celsius) {
  const fahrenheit = convertTemperature(celsius);
  console.log(`${celsius}°C is ${fahrenheit}°F`);
}
```

การใช้งานฟังก์ชันและโพรซีเดอร์

- ฟังก์ชันที่แสดงการใช้การ return แบบเงื่อนไข

```
function isAdult(age) {
  return age >= 18;
}

function checkAgeAndNotify(age) {
  if (isAdult(age)) {
    console.log('You are an adult.');
  } else {
    console.log('You are not an adult.');
  }
}
```

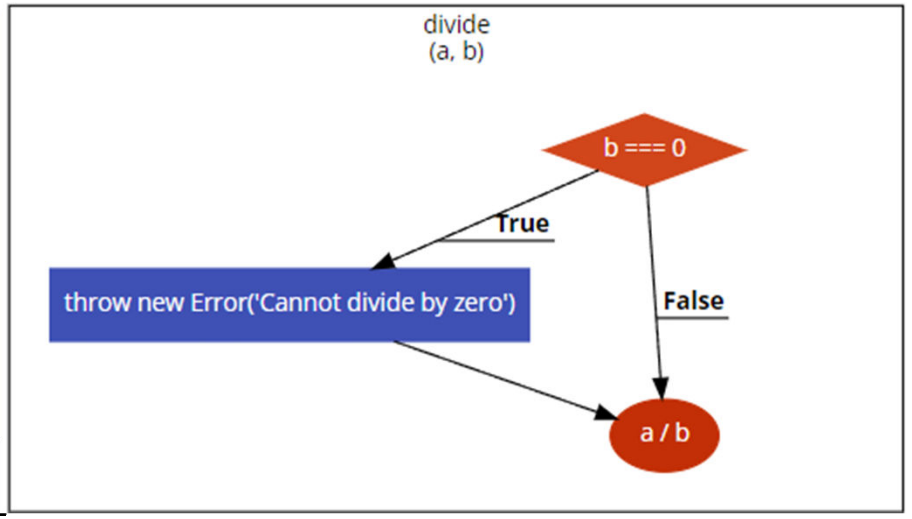
```
checkAgeAndNotify(20); // แสดง "You are an adult."
```

การจัดการข้อผิดพลาด

- ฟังก์ชันที่ดำเนินการทางคณิตศาสตร์

```
function divide(a, b) {
  if (b === 0) {
    throw new Error('Cannot divide by zero');
  }
  return a / b;
}
```

```
safeDivide(10, 2); // Result: 5
```

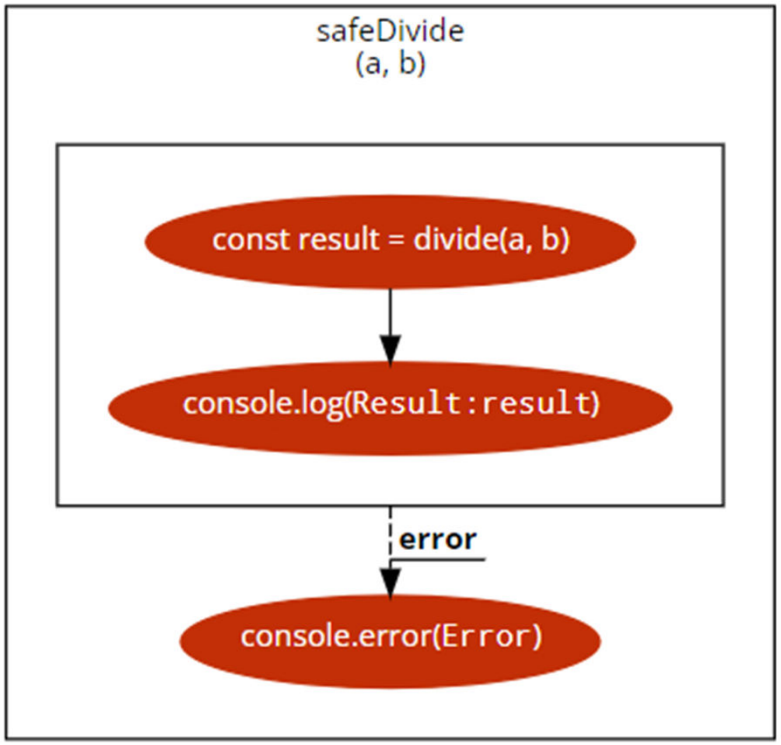


การจัดการข้อผิดพลาด

- การใช้ try...catch ในการจัดการข้อผิดพลาด

```
function safeDivide(a, b) {
  try {
    const result = divide(a, b);
    console.log(`Result: ${result}`);
  } catch (error) {
    console.error(`Error: ${error.message}`);
  }
}
```

```
safeDivide(10, 0); // Error: Cannot divide by zero
```



การจัดการข้อผิดพลาด

- การใช้ finally ในการทำความสะอาดหรือปิดทรัพยากร

```
function readFile(path) {
  let file;
  try {
    file = openFile(path); // สมมติว่าเป็นฟังก์ชันที่เปิดไฟล์
    const content = readContent(file); // สมมติว่าเป็นฟังก์ชันที่อ่านเนื้อหา
    return content;
  } catch (error) {
    console.error(`Error: ${error.message}`);
    // จัดการข้อผิดพลาด เช่น แสดงข้อความแจ้งผู้ใช้
  } finally {
    if (file) {
      closeFile(file); // สมมติว่าเป็นฟังก์ชันที่ปิดไฟล์
    }
  }
}
```

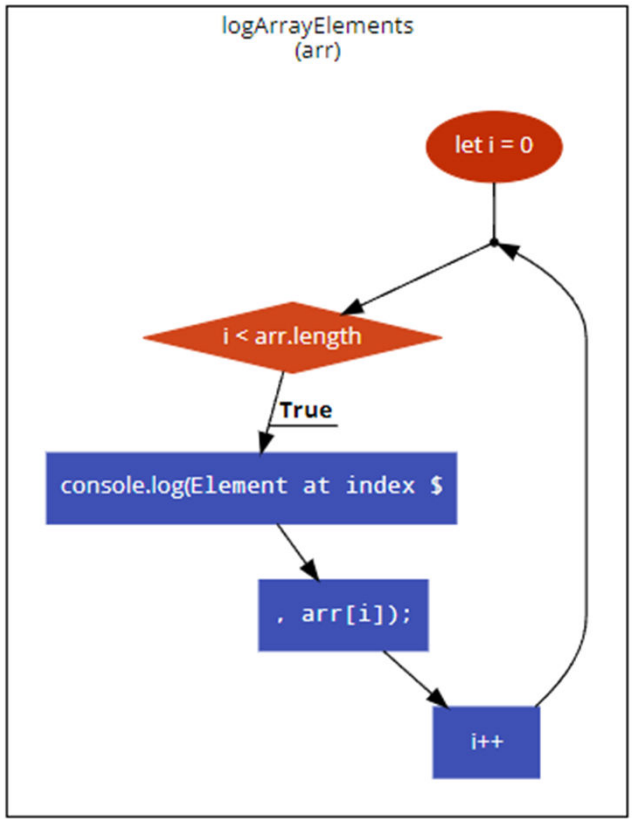
```
const content = readFile('data.txt'); // พยายามอ่านไฟล์ และจัดการข้อผิดพลาด
```

การจัดการกับข้อมูล

- การใช้งานอาร์เรย์

```
function logArrayElements(arr) {
  for (let i = 0; i < arr.length; i++) {
    console.log(`Element at index ${i}:`, arr[i]);
  }
}
```

```
logArrayElements([1, 2, 3, 4]);
```



การจัดการกับข้อมูล

- การใช้งานออบเจ็กต์

```
function updateProfile(profile, newEmail, newName) {
  // เปลี่ยนแปลงอีเมล์และชื่อในโปรไฟล์
  profile.email = newEmail;
  profile.name = newName;
  return profile;
}
```

```
const profile = updateProfile({ email: 'old@example.com', name: 'Old Name' }, 'new@example.com', 'New Name');
```

การจัดการกับข้อมูล

- การใช้งาน Map สำหรับการเก็บคู่ของ key-value

```
function getAndSetUserPreferences(preferences, key, value) {
  if (preferences.has(key)) {
    console.log(`Updating ${key}`);
  } else {
    console.log(`Setting new preference ${key}`);
  }
  preferences.set(key, value);
  return preferences.get(key);
}
```

```
const userPreferences = new Map(); getAndSetUserPreferences(userPreferences, 'theme', 'dark');
```

การจัดการกับข้อมูล

- การใช้งาน Set สำหรับการเก็บข้อมูลที่ไม่ซ้ำกัน

```
function logUniqueValues(values) {
  const uniqueValues = new Set(values);
  uniqueValues.forEach(value => {
    console.log(value);
  });
}
```

```
logUniqueValues([1, 2, 2, 3, 4, 4, 5]);
```

การจัดการกับข้อมูล

- การใช้งานโครงสร้างข้อมูลแบบซับซ้อน เช่น อาร์เรย์ของออบเจ็กต์

```
function findUserById(users, id) {
  return users.find(user => user.id === id);
}
```

```
const users = [{ id: 1, name: 'Alice' }, { id: 2, name: 'Bob' }];
const user = findUserById(users, 2); console.log(user); // { id: 2, name: 'Bob' }
```

การเขียนโค้ดที่มีโครงสร้าง

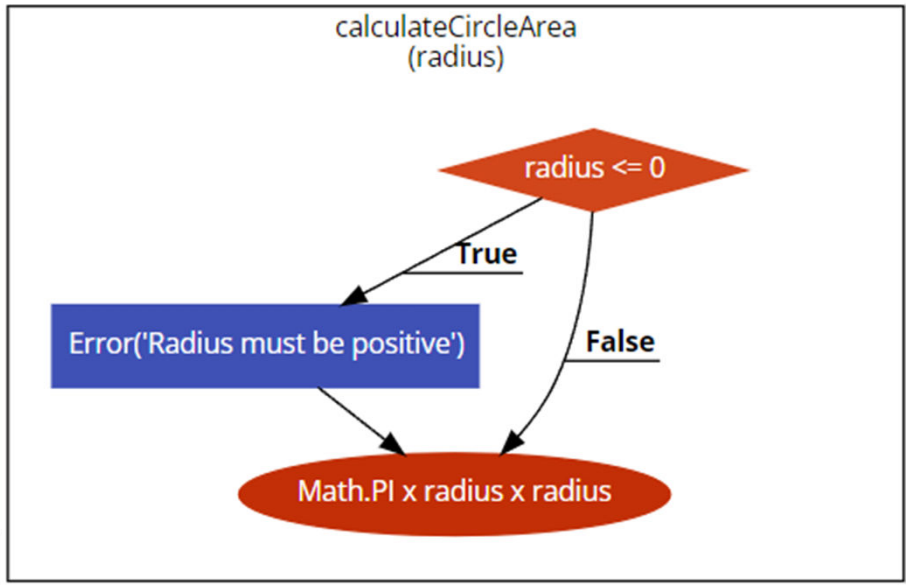
- หลักการของการเขียนโค้ดที่สะอาด (clean code)
- การใช้ชื่อตัวแปรและฟังก์ชันที่เข้าใจง่าย
- การจัดแบ่งโค้ดออกเป็นฟังก์ชันเล็ก ๆ ที่มีความรับผิดชอบเฉพาะ
- การเขียนคอมเมนต์เพื่ออธิบายความตั้งใจของโค้ด

การเขียนโค้ดที่มีโครงสร้าง

- ฟังก์ชันที่คำนวณและคืนค่าพื้นที่ของวงกลม

```
function calculateCircleArea(radius) {
  if (radius <= 0) {
    throw new Error('Radius must be positive');
  }
  return Math.PI * radius * radius;
}
```

```
try { const area = calculateCircleArea(5);
console.log(`Area of the circle: ${area}`); }
catch (error) {
console.error(error.message); }
```



การเขียนโค้ดที่มีโครงสร้าง

- ฟังก์ชันที่เปลี่ยนแปลงข้อมูลผู้ใช้ในระบบ

```
function updateUserProfile(userId, newProfileData) {
  const user = findUserById(userId); // สมมติว่าฟังก์ชันนี้ทำการค้นหาผู้ใช้จาก ID
  if (!user) {
    throw new Error(`User with ID ${userId} not found`);
  }

  // อัพเดตข้อมูลโปรไฟล์
  Object.assign(user.profile, newProfileData);
  saveUserProfile(user); // สมมติว่าฟังก์ชันนี้ทำการบันทึกข้อมูลโปรไฟล์
}
```

```
try { updateUserProfile(1, { email: 'newemail@example.com', name:
'New Name' }); } catch (error) { console.error(error.message); }
```

การเขียนโค้ดที่มีโครงสร้าง

- ฟังก์ชันที่แสดงข้อมูลผู้ใช้

```
function displayUserDetails(userId) {
  try {
    const user = findUserById(userId);
    console.log(`User ID: ${user.id}`);
    console.log(`Name: ${user.name}`);
    console.log(`Email: ${user.email}`);
  } catch (error) {
    console.error(`An error occurred: ${error.message}`);
  }
}
```

```
displayUserDetails(1);
```

การเขียนโค้ดที่มีโครงสร้าง

- ฟังก์ชันสำหรับการค้นหาผู้ใช้และจัดการข้อผิดพลาด

```
function findUserById(userId) {
  const user = database.users.find(user => user.id === userId);
  if (!user) {
    throw new Error(`User with ID ${userId} not found`);
  }
  return user;
}
```

การเขียนโค้ดที่มีโครงสร้าง

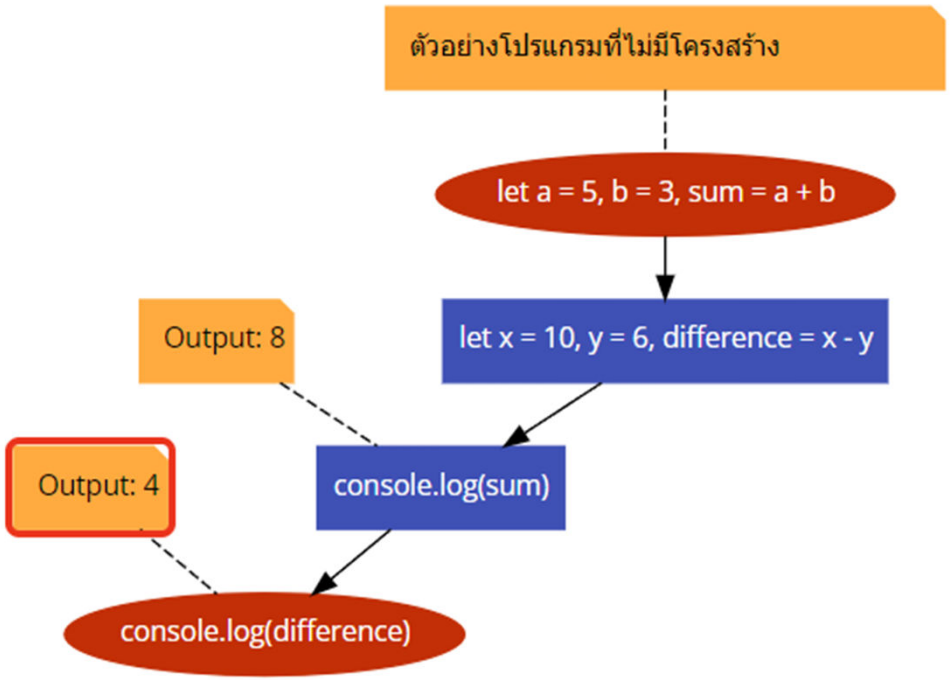
- ฟังก์ชันที่แสดงข้อมูลผู้ใช้

```
function displayUserDetails(userId) {
  try {
    const user = findUserById(userId);
    console.log(`User ID: ${user.id}`);
    console.log(`Name: ${user.name}`);
    console.log(`Email: ${user.email}`);
  } catch (error) {
    console.error(`An error occurred: ${error.message}`);
  }
}
```

การเขียนโปรแกรมแบบไม่มีโครงสร้าง

- การแบ่งเป็นหน่วยย่อยหรือฟังก์ชัน
- โปรแกรมมีโครงสร้างที่ชัดเจนและง่ายต่อการดูแลรักษา.
- "spaghetti code" การกระโดดหน้าไปหลังและการควบคุมที่ซับซ้อน

```
// ตัวอย่างโปรแกรมที่ไม่มีโครงสร้าง
let a = 5, b = 3, sum = a + b;
let x = 10, y = 6, difference = x - y;
console.log(sum);        // Output: 8
console.log(difference); // Output: 4
```

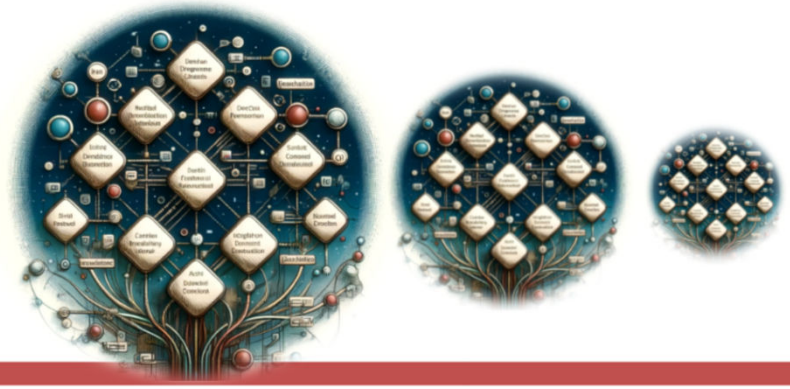


การเขียนโปรแกรมแบบมีโครงสร้าง

- การอ่านและเข้าใจ โปรแกรมที่มีโครงสร้างชัดเจนจะทำให้ง่ายต่อการอ่านและเข้าใจ.
- การดูแลรักษา (Maintainability) ง่ายต่อการปรับปรุงหรือแก้ไขโปรแกรมในอนาคต.
- การทดสอบและการตรวจจับข้อผิดพลาด (Debugging) ง่ายต่อการทดสอบและตรวจจับข้อผิดพลาดในโปรแกรม.
- การเรียบเรียง สามารถแบ่งปัญหาใหญ่ๆ ออกเป็นปัญหาย่อย ๆ และแก้ไขได้เป็นลำดับ.

สรุปท้ายบท

- โครงสร้างการควบคุม ทำให้โปรแกรมทำงานได้อย่างมีระบบตามเงื่อนไข
- การแบ่งโค้ดออกเป็นโปรแกรมย่อยหรือฟังก์ชันช่วยให้จัดการกับความซับซ้อน ลดการซ้ำซ้อนของโค้ด และทำให้โปรแกรมมีโครงสร้างที่ชัดเจน
- การสร้างโปรแกรมย่อย ทำให้ง่ายต่อการทดสอบและการบำรุงรักษา
- โครงสร้างที่ดี ควรใช้โครงสร้างการควบคุมอย่างเหมาะสม



ถาม-ตอบ