

# Module 6: Asynchronous JavaScript

Weeks 10-11

---

## Learning Objectives

By the end of this module, you will:

- Understand the JavaScript event loop and call stack
  - Master callbacks and avoid callback hell
  - Work effectively with Promises and Promise chains
  - Use `async/await` for cleaner asynchronous code
  - Fetch data from REST APIs using the Fetch API
  - Handle errors in asynchronous operations
  - Implement parallel and sequential `async` operations
- 

## Understanding Asynchronous Programming

### Synchronous vs Asynchronous:

```
javascript

// Synchronous code - blocking
console.log('First');
console.log('Second');
console.log('Third');
// Output: First, Second, Third (in order)

// Asynchronous code - non-blocking
console.log('First');
setTimeout(() => {
  console.log('End');
})

asyncExample();
// Output: Start, End, Promise callback, Timeout callback
// Promises have higher priority than setTimeout in the event loop
```

---

# Callbacks

## Basic Callback Pattern:

```
javascript

// Simple callback function
function fetchStudentData(studentId, callback) {
  console.log('Fetching student data...');

  // Simulate network delay
  setTimeout(() => {
    const student = {
      id: studentId,
      name: 'Alice Johnson',
      major: 'Computer Science',
      gpa: 3.85
    };

    callback(null, student); // First parameter is error, second is data
  }, 1500);
}

// Using the callback
fetchStudentData('STU001', (error, student) => {
  if (error) {
    console.error('Error fetching student:', error);
    return;
  }

  console.log('Student data received:', student);
});
```

## Error-First Callbacks (Node.js Convention):

```
javascript
```

```
function validateAndSaveStudent(studentData, callback) {  
  // Validation  
  if (!studentData.name) {  
    callback(new Error('Student name is required'));  
    return;  
  }  
  
  if (!studentData.email) {  
    callback(new Error('Student email is required'));  
    return;  
  }  
  
  // Simulate saving to database  
  setTimeout(() => {  
    if (Math.random() > 0.8) {  
      // Simulate random error  
      callback(new Error('Database connection failed'));  
    } else {  
      const savedStudent = {  
        ...studentData,  
        id: Math.floor(Math.random() * 10000),  
        createdAt: new Date()  
      };  
      callback(null, savedStudent);  
    }  
  }, 1000);  
}  
  
// Usage with proper error handling  
const newStudent = {  
  name: 'Bob Smith',  
  email: 'bob@university.edu',  
  major: 'Mathematics'  
};  
  
validateAndSaveStudent(newStudent, (error, savedStudent) => {  
  if (error) {  
    console.error('Failed to save student:', error.message);  
    return;  
  }  
})
```

```
console.log('Student saved successfully:', savedStudent);  
});
```

## Callback Hell Problem:

javascript

*// This becomes difficult to read and maintain*

```
fetchStudentData(studentId, (error, student) => {
  if (error) {
    console.error('Error:', error);
    return;
  }

  fetchStudentCourses(student.id, (error, courses) => {
    if (error) {
      console.error('Error:', error);
      return;
    }

    fetchCourseGrades(courses, (error, grades) => {
      if (error) {
        console.error('Error:', error);
        return;
      }

      calculateGPA(grades, (error, gpa) => {
        if (error) {
          console.error('Error:', error);
          return;
        }

        updateStudentRecord(student.id, gpa, (error, result) => {
          if (error) {
            console.error('Error:', error);
            return;
          }

          console.log('Student GPA updated successfully');
        });
      });
    });
  });
});
```

# Promises

## Creating Promises:

javascript

*// Basic Promise creation*

```
function fetchStudentPromise(studentId) {  
  return new Promise((resolve, reject) => {  
    console.log('Fetching student data...');  
  
    setTimeout(() => {  
      if (!studentId) {  
        reject(new Error('Student ID is required'));  
        return;  
      }  
  
      if (Math.random() > 0.9) {  
        reject(new Error('Student not found'));  
        return;  
      }  
  
      const student = {  
        id: studentId,  
        name: 'Alice Johnson',  
        major: 'Computer Science',  
        gpa: 3.85,  
        courses: ['CS101', 'CS201', 'MATH101']  
      };  
  
      resolve(student);  
    }, 1000);  
  });  
}
```

*// Using the Promise*

```
fetchStudentPromise('STU001')  
  .then(student => {  
    console.log('Student found:', student);  
    return student; // Pass data to next .then()  
  })  
  .then(student => {  
    console.log(`${student.name} is majoring in ${student.major}`);  
    return student.gpa;  
  })  
  .then(gpa => {  
    console.log('Student GPA:', gpa);  
  })  
  .catch(error => {
```

```
    console.error('Error:', error.message);
  })
  .finally(() => {
    console.log('Promise chain completed');
  });
```

## Promise States:

javascript

```
// Demonstrate Promise states
function demonstratePromiseStates() {
  // Pending Promise
  const pendingPromise = new Promise((resolve, reject) => {
    // Never resolves - stays pending
  });
  console.log('Pending promise:', pendingPromise);

  // Fulfilled Promise
  const fulfilledPromise = Promise.resolve('Success!');
  console.log('Fulfilled promise:', fulfilledPromise);

  // Rejected Promise
  const rejectedPromise = Promise.reject(new Error('Failed!'));
  console.log('Rejected promise:', rejectedPromise);

  // Handle rejected promise to avoid uncaught error
  rejectedPromise.catch(error => {
    console.log('Caught rejection:', error.message);
  });
}

demonstratePromiseStates();
```

## Promise Chaining:

javascript



```
class StudentAPI {
  static fetchStudent(id) {
    return new Promise((resolve, reject) => {
      setTimeout(() => {
        if (id === 'STU001') {
          resolve({ id, name: 'Alice Johnson', major: 'CS' });
        } else {
          reject(new Error('Student not found'));
        }
      }, 500);
    });
  }

  static fetchCourses(studentId) {
    return new Promise((resolve) => {
      setTimeout(() => {
        resolve([
          { id: 'CS101', name: 'Programming Fundamentals', grade: 'A' },
          { id: 'CS201', name: 'Data Structures', grade: 'B+' },
          { id: 'MATH101', name: 'Calculus I', grade: 'A-' }
        ]);
      }, 300);
    });
  }

  static calculateGPA(courses) {
    return new Promise((resolve) => {
      setTimeout(() => {
        const gradePoints = {
          'A': 4.0, 'A-': 3.7, 'B+': 3.3, 'B': 3.0,
          'B-': 2.7, 'C+': 2.3, 'C': 2.0, 'D': 1.0, 'F': 0.0
        };

        const totalPoints = courses.reduce((sum, course) =>
          sum + gradePoints[course.grade], 0);
        const gpa = totalPoints / courses.length;

        resolve(gpa);
      }, 200);
    });
  }
}
```

*// Clean Promise chain*

```
StudentAPI.fetchStudent('STU001')
  .then(student => {
    console.log('Student:', student);
    return StudentAPI.fetchCourses(student.id);
  })
  .then(courses => {
    console.log('Courses:', courses);
    return StudentAPI.calculateGPA(courses);
  })
  .then(gpa => {
    console.log('Calculated GPA:', gpa.toFixed(2));
  })
  .catch(error => {
    console.error('Error in chain:', error.message);
  });
```

---

## Promise Utilities

### Promise.all() - All or Nothing:

javascript

```

// Wait for all promises to resolve
function fetchAllStudentData() {
  const studentIds = ['STU001', 'STU002', 'STU003'];

  const promises = studentIds.map(id =>
    StudentAPI.fetchStudent(id).catch(error => ({
      id,
      error: error.message
    })))
  );

  return Promise.all(promises);
}

fetchAllStudentData()
  .then(results => {
    console.log('All students fetched:', results);
  })
  .catch(error => {
    console.error('One or more requests failed:', error);
  });

// Promise.all fails fast - if any promise rejects, the whole thing rejects
const mixedPromises = [
  Promise.resolve('Success 1'),
  Promise.resolve('Success 2'),
  Promise.reject(new Error('Failure')),
  Promise.resolve('Success 3')
];

Promise.all(mixedPromises)
  .then(results => {
    console.log('All succeeded:', results); // Won't execute
  })
  .catch(error => {
    console.log('At least one failed:', error.message); // Will execute
  });

```

## Promise.allSettled() - Wait for All (Success or Failure):

javascript

```
// Wait for all promises regardless of outcome
Promise.allSettled([
  StudentAPI.fetchStudent('STU001'),
  StudentAPI.fetchStudent('INVALID'),
  StudentAPI.fetchStudent('STU003')
])
.then(results => {
  results.forEach((result, index) => {
    if (result.status === 'fulfilled') {
      console.log(`Student ${index + 1}:`, result.value);
    } else {
      console.log(`Student ${index + 1} failed:`, result.reason.message);
    }
  });
});
```

## Promise.race() - First to Complete:

```
javascript

// Return the first promise to complete (resolve or reject)
const fastServer = new Promise(resolve =>
  setTimeout(() => resolve('Fast server response'), 100));
const slowServer = new Promise(resolve =>
  setTimeout(() => resolve('Slow server response'), 1000));
const timeout = new Promise(_ reject =>
  setTimeout(() => reject(new Error('Request timeout')), 500));

Promise.race([fastServer, slowServer, timeout])
  .then(result => {
    console.log('Winner:', result); // 'Fast server response'
  })
  .catch(error => {
    console.error('Error:', error.message);
  });
```

## Promise.any() - First Successful:

```
javascript
```

```
// Return the first promise to resolve (ignore rejections unless all fail)
const servers = [
  new Promise(⌵, reject) => setTimeout(() => reject(new Error('Server 1 down')), 100)),
  new Promise(resolve => setTimeout(() => resolve('Server 2 success'), 200)),
  new Promise(resolve => setTimeout(() => resolve('Server 3 success'), 150))
];

Promise.any(servers)
  .then(result => {
    console.log('First success:', result); // 'Server 3 success'
  })
  .catch(error => {
    console.error('All servers failed:', error);
  });
```

---

## Async/Await

### Basic Async/Await Usage:

```
javascript
```

```

// Convert Promise chain to async/await
async function fetchStudentReport(studentId) {
  try {
    console.log('Starting to fetch student report...');

    // These run sequentially
    const student = await StudentAPI.fetchStudent(studentId);
    console.log('Student fetched:', student.name);

    const courses = await StudentAPI.fetchCourses(studentId);
    console.log('Courses fetched:', courses.length);

    const gpa = await StudentAPI.calculateGPA(courses);
    console.log('GPA calculated:', gpa.toFixed(2));

    return {
      student,
      courses,
      gpa,
      generatedAt: new Date()
    };
  } catch (error) {
    console.error('Error generating report:', error.message);
    throw error; // Re-throw to let caller handle
  }
}

// Using async function
fetchStudentReport('STU001')
  .then(report => {
    console.log('Report generated:', report);
  })
  .catch(error => {
    console.error('Report generation failed:', error.message);
  });

```

## Parallel vs Sequential Execution:

javascript

*// Sequential execution (slower)*

```
async function fetchStudentsSequential(ids) {  
  const students = [];  
  
  for (const id of ids) {  
    try {  
      const student = await StudentAPI.fetchStudent(id);  
      students.push(student);  
    } catch (error) {  
      console.error(`Failed to fetch student ${id}:`, error.message);  
    }  
  }  
  
  return students;  
}
```

*// Parallel execution (faster)*

```
async function fetchStudentsParallel(ids) {  
  const promises = ids.map(async (id) => {  
    try {  
      return await StudentAPI.fetchStudent(id);  
    } catch (error) {  
      console.error(`Failed to fetch student ${id}:`, error.message);  
      return null; // Return null instead of throwing  
    }  
  });  
  
  const results = await Promise.all(promises);  
  return results.filter(student => student !== null);  
}
```

*// Comparison*

```
async function compareExecutionTimes() {  
  const studentIds = ['STU001', 'STU002', 'STU003', 'STU004'];  
  
  console.time('Sequential');  
  const sequentialResults = await fetchStudentsSequential(studentIds);  
  console.timeEnd('Sequential');  
  
  console.time('Parallel');  
  const parallelResults = await fetchStudentsParallel(studentIds);  
  console.timeEnd('Parallel');
```

```
console.log('Sequential count:', sequentialResults.length);
console.log('Parallel count:', parallelResults.length);
}

compareExecutionTimes();
```

## Advanced Async Patterns:

javascript



```
class AsyncStudentProcessor {
  constructor() {
    this.processingQueue = [];
    this.isProcessing = false;
  }

  // Process items one at a time
  async processQueue() {
    if (this.isProcessing) return;

    this.isProcessing = true;

    while (this.processingQueue.length > 0) {
      const task = this.processingQueue.shift();
      try {
        await this.processStudent(task.studentId);
        task.resolve();
      } catch (error) {
        task.reject(error);
      }
    }

    this.isProcessing = false;
  }

  // Add student to processing queue
  queueStudent(studentId) {
    return new Promise((resolve, reject) => {
      this.processingQueue.push({ studentId, resolve, reject });
      this.processQueue(); // Start processing if not already running
    });
  }

  async processStudent(studentId) {
    console.log(`Processing student ${studentId}...`);

    // Simulate complex processing
    await new Promise(resolve => setTimeout(resolve, 1000));

    console.log(`Student ${studentId} processed successfully`);
  }

  // Retry with exponential backoff
}
```

```

async retryOperation(operation, maxRetries = 3, baseDelay = 1000) {
  let lastError;

  for (let attempt = 1; attempt <= maxRetries; attempt++) {
    try {
      return await operation();
    } catch (error) {
      lastError = error;

      if (attempt === maxRetries) {
        break; // Don't wait after last attempt
      }

      const delay = baseDelay * Math.pow(2, attempt - 1);
      console.log(`Attempt ${attempt} failed, retrying in ${delay}ms...`);
      await new Promise(resolve => setTimeout(resolve, delay));
    }
  }

  throw new Error(`Operation failed after ${maxRetries} attempts: ${lastError.message}`);
}

// Batch processing with concurrency limit
async processBatch(studentIds, concurrency = 3) {
  const results = [];

  for (let i = 0; i < studentIds.length; i += concurrency) {
    const batch = studentIds.slice(i, i + concurrency);
    const batchPromises = batch.map(id =>
      this.retryOperation(() => StudentAPI.fetchStudent(id))
    );

    const batchResults = await Promise.allSettled(batchPromises);
    results.push(...batchResults);
  }

  return results;
}

// Usage
const processor = new AsyncStudentProcessor();

// Queue multiple students

```

```
processor.queueStudent('STU001');  
processor.queueStudent('STU002');  
processor.queueStudent('STU003');
```

---

## Working with APIs

### Fetch API Basics:

```
javascript
```

*// Basic GET request*

```
async function fetchStudent(id) {
  try {
    const response = await fetch(`/api/students/${id}`);

    if (!response.ok) {
      throw new Error(`HTTP error! status: ${response.status}`);
    }

    const student = await response.json();
    return student;

  } catch (error) {
    console.error('Fetch error:', error);
    throw error;
  }
}
```

*// POST request with data*

```
async function createStudent(studentData) {
  try {
    const response = await fetch('/api/students', {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json',
        'Authorization': 'Bearer your-token-here'
      },
      body: JSON.stringify(studentData)
    });

    if (!response.ok) {
      const errorData = await response.json();
      throw new Error(errorData.message || 'Failed to create student');
    }

    const createdStudent = await response.json();
    return createdStudent;

  } catch (error) {
    console.error('Create student error:', error);
    throw error;
  }
}
```

```
// Usage
const newStudent = {
  name: 'Charlie Brown',
  email: 'charlie@university.edu',
  major: 'Physics'
};

createStudent(newStudent)
  .then(student => {
    console.log('Student created:', student);
  })
  .catch(error => {
    console.error('Failed to create student:', error.message);
  });
```

## Advanced API Client:

javascript

```
class UniversityAPI {
  constructor(baseUrl, apiKey) {
    this.baseUrl = baseUrl;
    this.apiKey = apiKey;
  }

  async request(endpoint, options = {}) {
    const url = `${this.baseUrl}${endpoint}`;
    const config = {
      ...options,
      headers: {
        'Content-Type': 'application/json',
        'Authorization': `Bearer ${this.apiKey}`,
        ...options.headers
      }
    };

    try {
      const response = await fetch(url, config);

      // Handle different status codes
      if (response.status === 401) {
        throw new Error('Authentication failed');
      }

      if (response.status === 403) {
        throw new Error('Access forbidden');
      }

      if (response.status === 404) {
        throw new Error('Resource not found');
      }

      if (!response.ok) {
        const errorText = await response.text();
        throw new Error(`HTTP ${response.status}: ${errorText}`);
      }

      // Handle empty responses
      const contentType = response.headers.get('Content-Type');
      if (contentType && contentType.includes('application/json')) {
        return await response.json();
      } else {

```

```
        return await response.text();
    }

    } catch (error) {
        if (error.name === 'TypeError') {
            throw new Error('Network error - please check your connection');
        }
        throw error;
    }
}

// GET request
async get(endpoint) {
    return this.request(endpoint, { method: 'GET' });
}

// POST request
async post(endpoint, data) {
    return this.request(endpoint, {
        method: 'POST',
        body: JSON.stringify(data)
    });
}

// PUT request
async put(endpoint, data) {
    return this.request(endpoint, {
        method: 'PUT',
        body: JSON.stringify(data)
    });
}

// DELETE request
async delete(endpoint) {
    return this.request(endpoint, { method: 'DELETE' });
}

// Student-specific methods
async getStudents(page = 1, limit = 10) {
    return this.get(`/students?page=${page}&limit=${limit}`);
}

async getStudent(id) {
    return this.get(`/students/${id}`);
}
```

```

    }

    async createStudent(studentData) {
        return this.post('/students', studentData);
    }

    async updateStudent(id, updates) {
        return this.put(`/students/${id}`, updates);
    }

    async deleteStudent(id) {
        return this.delete(`/students/${id}`);
    }

    // Batch operations
    async getMultipleStudents(ids) {
        const promises = ids.map(id =>
            this.getStudent(id).catch(error => ({
                id,
                error: error.message
            })))
        );

        return Promise.all(promises);
    }
}

// Usage
const api = new UniversityAPI('https://api.university.edu', 'your-api-key');

// Example operations
async function demonstrateAPI() {
    try {
        // Get all students
        const students = await api.getStudents(1, 20);
        console.log('Students:', students);

        // Get specific student
        const student = await api.getStudent('STU001');
        console.log('Student details:', student);

        // Create new student
        const newStudent = await api.createStudent({
            name: 'Diana Prince',

```



```
    email: 'diana@university.edu',
    major: 'Psychology'
  });
  console.log('New student created:', newStudent);

  // Update student
  const updatedStudent = await api.updateStudent(newStudent.id, {
    gpa: 3.9
  });
  console.log('Student updated:', updatedStudent);

} catch (error) {
  console.error('API operation failed:', error.message);
}

}

demonstrateAPI();
```

---

## Error Handling in Async Code

### Try-Catch with Async/Await:

javascript

```
async function robustStudentOperations() {
  try {
    // Multiple operations that might fail
    const student = await api.getStudent('STU001');
    console.log('Student fetched:', student.name);

    const courses = await api.getStudentCourses(student.id);
    console.log('Courses fetched:', courses.length);

    const grades = await api.getStudentGrades(student.id);
    console.log('Grades fetched:', grades.length);

    return { student, courses, grades };
  } catch (error) {
    // Handle different types of errors
    if (error.message.includes('Network error')) {
      console.error('Connection problem - please try again later');
    } else if (error.message.includes('not found')) {
      console.error('Student not found in database');
    } else if (error.message.includes('Authentication')) {
      console.error('Please log in again');
    } else {
      console.error('Unexpected error:', error.message);
    }

    // Re-throw if needed
    throw error;
  }
}
```

## Custom Error Classes:

javascript

```
class APIError extends Error {
  constructor(message, status, endpoint) {
    super(message);
    this.name = 'APIError';
    this.status = status;
    this.endpoint = endpoint;
  }
}
```

```
class ValidationError extends Error {
  constructor(message, field) {
    super(message);
    this.name = 'ValidationError';
    this.field = field;
  }
}
```

```
class NetworkError extends Error {
  constructor(message, originalError) {
    super(message);
    this.name = 'NetworkError';
    this.originalError = originalError;
  }
}
```

*// Enhanced API client with custom errors*

```
class EnhancedAPI extends UniversityAPI {
  async request(endpoint, options = {}) {
    try {
      return await super.request(endpoint, options);
    } catch (error) {
      if (error.message.includes('Network error')) {
        throw new NetworkError('Unable to connect to server', error);
      }

      if (error.message.includes('HTTP 4')) {
        throw new APIError(error.message, 400, endpoint);
      }

      throw error;
    }
  }
}
```

```
async createStudentWithValidation(studentData) {  
  // Client-side validation  
  if (!studentData.name || studentData.name.trim().length < 2) {  
    throw new ValidationError('Name must be at least 2 characters', 'name');  
  }  
  
  if (!studentData.email || !studentData.email.includes('@')) {  
    throw new ValidationError('Valid email is required', 'email');  
  }  
  
  return this.createStudent(studentData);  
}  
}  
  
// Usage with specific error handling  
async function handleSpecificErrors() {  
  const api = new EnhancedAPI('https://api.university.edu', 'api-key');  
  
  try {  
    const student = await api.createStudentWithValidation({  
      name: 'X', // Too short  
      email: 'invalid-email'  
    });  
  
  } catch (error) {  
    if (error instanceof ValidationError) {  
      console.error(`Validation error in ${error.field}: ${error.message}`);  
    } else if (error instanceof APIError) {  
      console.error(`API error (${error.status}) on ${error.endpoint}: ${error.message}`);  
    } else if (error instanceof NetworkError) {  
      console.error('Network problem:', error.message);  
      // Could implement retry logic here  
    } else {  
      console.error('Unexpected error:', error.message);  
    }  
  }  
}
```

# Assignment 6: Student Portal with API Integration

## Requirements:

### Part 1: API Integration

1. Create a comprehensive API client for student management
2. Implement proper error handling and retry logic
3. Add loading states and progress indicators
4. Handle network failures gracefully

### Part 2: Asynchronous Operations

1. Build concurrent data fetching for better performance
2. Implement caching to reduce API calls
3. Add real-time search with debounced requests
4. Create batch operations for multiple students

### Part 3: User Experience

1. Show loading spinners during async operations
2. Implement optimistic updates (update UI before API confirms)
3. Add offline support with queued operations
4. Create comprehensive error messages for users

## Code Structure Template:

```
javascript
```

```
class StudentPortalApp {
  constructor() {
    this.api = new UniversityAPI('your-api-url', 'your-key');
    this.cache = new Map();
    this.loadingStates = new Set();
    this.offlineQueue = [];
  }

  async loadStudents(page = 1) {
    // Implementation with caching and error handling
  }

  async createStudent(studentData) {
    // Implementation with validation and optimistic updates
  }

  async handleOfflineMode() {
    // Queue operations when offline
  }

  debounce(func, delay) {
    // Debounce utility for search
  }
}
```

## Best Practices Summary

### Async/Await Best Practices:

1. **Always use try-catch:** Handle errors properly in async functions
2. **Prefer async/await over Promises:** More readable for complex operations
3. **Use Promise.all for parallelism:** When operations can run concurrently
4. **Handle loading states:** Show users that operations are in progress

### API Integration:

1. **Validate before sending:** Check data on client before API calls
2. **Handle all HTTP status codes:** Don't just check for 200
3. **Implement retry logic:** For transient failures
4. **Use proper error messages:** Help users understand what went wrong

## Performance:

1. **Cache API responses:** Reduce unnecessary network requests
  2. **Debounce user input:** For search and real-time features
  3. **Use pagination:** Don't load all data at once
  4. **Implement progressive loading:** Show partial results while loading more
- 

## Next Module Preview

### Module 7: Modern JavaScript Features

- ES6+ syntax and features
- Destructuring and spread operators
- Modules and imports
- Advanced object features
- Symbols, iterators, and generators

## Preparation:

- Practice async/await patterns
  - Build API-integrated applications
  - Master error handling techniques
  - Understand Promise utilities
- 

## Questions for Review

1. When should you use Promise.all vs Promise.allSettled?
2. How do you handle errors in async/await functions?
3. What's the difference between sequential and parallel async operations?
4. How can you implement retry logic for failed API calls?
5. What are the best practices for API client architecture?

## Practice Exercises:

- Build a real-time search interface
- Create an offline-capable application

- Implement a file upload system with progress tracking
- Design a batch processing system for large datasets('Second (delayed)'); }, 1000); console.log('Third');  
// Output: First, Third, Second (delayed)

### The Event Loop:

```
```javascript
```

```
// Call stack demonstration
```

```
function first() {  
  console.log('First function');  
  second();  
  console.log('First function end');  
}  
  
function second() {  
  console.log('Second function');  
  third();  
  console.log('Second function end');  
}  
  
function third() {  
  console.log('Third function');  
}
```

```
first();
```

```
// Call stack: first() -> second() -> third() -> back to second() -> back to first()
```

```
// With asynchronous code
```

```
function asyncExample() {  
  console.log('Start');  
  
  setTimeout(() => {  
    console.log('Timeout callback');  
  }, 0);  
  
  Promise.resolve().then(() => {  
    console.log('Promise callback');  
  });  
  
  console.log
```