Week 1: Course Introduction & Tools Overview

Developer Tools: VSCode & Marp

- **Visual Studio Code (VSCode)** – a lightweight **IDE** with rich features (file explorer, text editor, version control integration, terminal, etc.). It's extensible via plugins and serves as the primary coding environment.
    - **2025 Stats:** Most popular IDE with 60,000+ extensions available
    - Supports all major programming languages with IntelliSense
- **Marp** – a toolchain to create slides with Markdown. *Write* content in a Markdown file and Marp **compiles it into presentation formats** (PDF, PPT, HTML). Using the Marp VSCode extension, you can preview slides and export them easily.
- These tools streamline our workflow: VSCode for all-in-one development and Marp for documentation and presentations.

Git & GitHub Basics

- **Git** is a distributed version control system for tracking code changes. It runs locally and lets you save versions of your code over time.
    - **2025 Adoption:** 93.87% of developers use Git (up from 87.1% in 2016)
- **GitHub** is an **online platform for Git repositories**, providing cloud storage for code and facilitating collaboration. It acts as a portfolio for projects and a central hub to share code with teammates.
- In this course, you'll use GitHub to access course materials and submit assignments. Common ASE course material is available in the public GitHub repo (e.g. the `nkuase/ASE` repo) for reference.

Course Overview & Policies

- **Course Introduction:** The course covers *high-quality software engineering* practices, including development processes, design principles, and team collaboration. We will manage a project through all stages from requirements to testing.
- **Course Policies:** Attendance and participation are crucial. Assignments must be submitted on time; late work may receive little or no credit. **Academic integrity** is emphasized – all code must be your own work (plagiarism or unauthorized collaboration is prohibited).
- **Tools & Workflow:** We will use VSCode, Git/GitHub for version control, and Marp for documentation. Use these consistently to document your progress and collaborate.

High-Quality Software: Storytelling & Assignments

- **Software as Storytelling:** Writing code and documentation is akin to storytelling. Your code should be clear and well-structured so that other developers (and your future self) can follow the "story" easily. Good software design tells a coherent story about the problem and solution.
- **Communication:** Throughout the course, you'll practice explaining your design decisions. This includes writing user stories, use cases, and documentation that communicate the purpose and usage of your software.
- **Assignments Overview:** The course assignments are structured to apply concepts in practice. Early assignments focus on individual skills (tools setup, simple design tasks). Later assignments build toward a **capstone project**, where you'll work in teams to deliver a software product incrementally. Each assignment reinforces topics from lectures (e.g. version control, requirements, design, testing).

Week 2: Software Process Overview

From Waterfall to Agile

- **Waterfall Model:** A linear, phase-driven software process. Each phase (requirements → design → implementation → testing → maintenance) is completed before the next begins. Changes are costly once a phase is finished, and testing is done only after development is complete.
- **Agile Methods:** An iterative, incremental approach that welcomes changing requirements even late in development. Work is done in short cycles (sprints) that produce working software frequently. Testing and integration happen continuously alongside development.
    - **2025 Stats:** 61% of organizations have used Agile for 5+ years
- **Comparison:** *Waterfall* emphasizes upfront planning and heavy documentation, suitable when requirements are well-understood and unlikely to change. *Agile* emphasizes adaptability and customer feedback, delivering value faster and adjusting as needs evolve.

Scrum Framework

- **Scrum** is an Agile framework for managing work in iterations (sprints). It breaks projects into small increments to be completed in fixed-length sprints (often 1–4 weeks). The focus is on continuous improvement and delivering potentially shippable software each sprint.
    - **2025 Stats:** 87% of Agile organizations use Scrum as their preferred framework
- **Roles:** In Scrum, team members fill one of three roles – **Product Owner**, **Scrum Master**, or **Development Team**. The Product Owner manages the backlog and priorities (representing the customer's interests). The Scrum Master facilitates the process and removes impediments. The Development Team (Scrum Team) is self-organizing and builds the product increment.
- **Events & Artifacts:** Key Scrum events include **Daily Stand-ups** (15-min daily team sync), **Sprint Planning**, **Sprint Review** (demonstrate increment), and **Sprint Retrospective** (reflect and improve). Scrum artifacts include the **Product Backlog**

DevOps Culture

- **DevOps** combines software **Dev**elopment and IT **Ops** (operations) to streamline the path from code to deployment. It is a set of practices and a culture that emphasize close collaboration between developers and operations teams.
- **Continuous Integration & Delivery:** DevOps teams use automation for building, testing, and deploying code (CI/CD pipelines). This enables rapid, reliable releases – code changes can go live faster and with fewer errors.
    - **2025 Trend:** 60% of businesses adopting GitOps for deployment automation (Forrester)
    - AI/ML integration in DevOps workflows for predictive monitoring and anomaly detection
- **Key Principles:** DevOps focuses on shared responsibility for outcomes, automation of repetitive tasks, and continuous feedback. By breaking down silos between dev and ops, DevOps achieves faster delivery **without sacrificing reliability**. Teams monitor

Week 3: Python, OOP, and UML Fundamentals

Python for Software Engineering

- **Python** – an interpreted, high-level, general-purpose programming language known for its readable syntax. Python's design philosophy emphasizes code clarity and a simple syntax that helps developers write clear, logical code.
- Python supports multiple paradigms (procedural, object-oriented, functional). In this course we use Python to illustrate object-oriented programming (OOP) concepts because of its simplicity and widespread use.
- Key Python features include dynamic typing (you don't declare variable types), a rich standard library, and cross-platform portability. This allows rapid prototyping and experimentation while learning design principles.

Object-Oriented Programming Basics

- **Object-Oriented Programming (OOP)** is a paradigm centered on *objects* (instances of classes) that encapsulate data and behavior. OOP promotes organizing software as a collection of interacting objects.
- **Four Pillars of OOP:**
    - *Abstraction* – modeling real-world entities by focusing on relevant features while ignoring unnecessary details.
    - *Encapsulation* – hiding internal state and implementation details of an object behind a public interface.
    - *Inheritance* – creating new classes by extending existing ones, enabling reuse of code and establishing hierarchies.
    - *Polymorphism* – treating objects of different subclasses through a common interface, and automatically calling the correct overridden behaviors at runtime.

Abstraction & Inheritance in Practice

Abstraction Example:

```python
class User:
    def __init__(self, name, email):
        self.name = name
        self.email = email

    def send_notification(self):
        # Abstract the complex email sending logic
        pass
```

Inheritance Example:

```python
class Employee:
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary

class Manager(Employee):  # Inherits from Employee
```

Abstraction & Inheritance Concepts

- **Abstraction:** In design, identify the essential attributes of an entity and model a class for it, omitting extraneous details. For instance, a "User" class might abstract common user properties (name, email) relevant to the system, leaving out personal details not needed. Abstraction allows focusing on *what* an object does instead of *how* it does it.
- **Inheritance:** Establishes an "is-a" relationship between classes. A base (parent) class defines general attributes and methods, and a derived (child) class inherits those and extends or overrides behavior. Inheritance promotes code reuse – new classes reuse existing code instead of writing from scratch.
- Using inheritance wisely can reduce duplication, but be cautious: subclasses should truly satisfy the relationship (Liskov Substitution Principle). If a subclass violates expectations of the parent class, the design can break. Prefer composition over inheritance when appropriate to avoid rigid hierarchies.

UML Diagrams for Design

- **Unified Modeling Language (UML)** provides a standard way to visualize system design. UML diagrams are blueprints for understanding software structure and behavior without diving into code.
- **Structure Diagrams:** e.g. *Class Diagrams* show classes, their attributes and methods, and relationships (associations, inheritance, composition). This helps in designing the static architecture of the system.
- **Behavior Diagrams:** e.g. *Sequence Diagrams* illustrate interactions over time between objects (method call flows), and *State Diagrams* show state transitions of an object. *Use Case Diagrams* (covered next week) depict the functional requirements by showing actors and their interactions with the system.
- By using UML, we can communicate designs clearly. For instance, a class diagram can clarify how classes relate (a **composition** is depicted with a filled diamond, meaning a strong "has-a" relationship). UML is a common language for developers to collaborate

Software Requirements Fundamentals

- **Requirements** capture what the software should *do* (functional requirements) and how it should *be* (non-functional requirements). A functional requirement might be "the system shall **send a password reset email** to the user upon request," whereas a non-functional requirement could be "the email is sent **within 5 seconds** of request (performance)".
- *Functional vs Non-Functional:* Broadly, functional requirements define **what** a system must accomplish (features and behaviors), and non-functional requirements define **how** the system performs or the constraints on it (quality attributes like reliability, usability, security). For example, functional requirements are expressed as specific actions or outputs ("system shall do X"), while non-functional are expressed as properties or criteria ("system shall be Y" in terms of speed, security, etc.).
- Gathering clear requirements is critical to project success. Techniques include stakeholder interviews, user stories, and use cases. A well-written requirement is

Week 4: Advanced OOP, Design Principles, and Patterns

The APIEC Framework

A unified framework for object-oriented design principles:

- **A**bstraction – Focus on essential features, hide complexity
- **P**olymorphism – Many forms, uniform interfaces
- **I**nheritance – Code reuse through hierarchies
- **E**ncapsulation – Hide internal state, expose interfaces
- **C**omposition – Build complex objects from simpler ones

Key Principle: Favor composition over inheritance for flexibility and reduced coupling.

APIEC principles work together to create maintainable, extensible software designs.

Encapsulation & Dependency Injection

Encapsulation Example:

```
class BankAccount:
    def __init__(self, balance):
        self.__balance = balance  # Private attribute

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount

    def get_balance(self):  # Controlled access
        return self.__balance
```

Dependency Injection Example:

```
class EmailService:
    def send(self, message):
        pass
```

Encapsulation & Dependency Injection Concepts

- **Encapsulation:** Encapsulation means keeping an object's internal data private and exposing only what's necessary through methods or an interface. This protects the integrity of the object's state. For example, rather than allowing direct access to a class's fields, we provide getter/setter methods or use properties. Encapsulation promotes modularity and makes it easier to change implementation without affecting other parts of the code.
- **Dependency Injection (DI):** DI is a design technique where an object's dependencies (other objects it needs to function) are provided from the outside rather than created internally. In practice, this often means **passing required components via constructors or setters**. DI supports the *Inversion of Control* principle – instead of a class controlling its dependencies, the control is inverted and handed to an external entity (like a framework or a factory). This leads to more testable and flexible code, since you can swap out implementations (e.g., inject a mock object in tests or a

Polymorphism & Composition

Polymorphism Example:

```python
class Shape:
    def draw(self):
        pass

class Circle(Shape):
    def draw(self):
        print("Drawing circle")

class Square(Shape):
    def draw(self):
        print("Drawing square")

shapes = [Circle(), Square()]
for shape in shapes:
    shape.draw()  # Polymorphic behavior
```

Composition Example

Composition (favor over inheritance):

```python
class Engine:
    def start(self):
        print("Engine starting")

class Wheels:
    def rotate(self):
        print("Wheels rotating")

class Car:  # Composition: Car HAS-A Engine and Wheels
    def __init__(self):
        self.engine = Engine()
        self.wheels = Wheels()

    def drive(self):
        self.engine.start()
        self.wheels.rotate()
```

Polymorphism & Composition Concepts

- **Polymorphism:** Literally "many forms," polymorphism in OOP allows treating objects of different classes through a uniform interface. For example, if `Shape` is a base class with a method `draw()`, polymorphism lets us call `draw()` on any subclass (`Circle`, `Square`, etc.) and get subclass-specific behavior. At runtime, the actual object's implementation is invoked. Polymorphism improves extensibility – new subclasses can be introduced without changing code that uses the base class interface.
- **Composition:** Composition is a design technique where a class is composed of one or more other classes, implying a strong *has-a* relationship. Instead of using inheritance to reuse code, a class can have instances of other classes as members. For example, a `Car` class might *compose* a `Engine` class and `Wheel` classes, rather than inheriting from them, since a car is not a type of engine but has an engine. Composition is often favored over inheritance for flexibility – you can change composed parts without

Use Case Diagrams

Example UML use case diagram (restaurant scenario): Each **actor** (user role) is connected to the **use cases** (ovals) they participate in.

- **Use Case Diagram** – a UML diagram to model **functional requirements** of the system. It shows *actors* (users or external systems) and *use cases* (services or functions the system provides to those actors).
- Use case diagrams provide a high-level overview of **who** uses the system and **what** they can do. For example, in an online shopping system, actors might be *Customer* and *Admin*, and use cases could include *Browse Products*, *Place Order*, *Manage Inventory*, etc.
- Each use case represents a distinct functionality or goal from the actor's perspective. Relationships in use case diagrams can include **<<extends>>** or **<<includes>>** (to show optional or common sub-flows). This diagram is a communication tool between

Software Testing

- **Software Testing** is the process of evaluating a software item to detect differences between **expected output and actual output** (i.e., finding defects). In practice, testing involves executing the software with sample inputs and verifying that outputs and behaviors match the requirements.
    - **2025 Market:** Software testing market valued at $87.42B in 2024, projected to reach $512.3B by 2033 (CAGR 21.71%)
- **Testing Levels:**
    - *Unit Testing* – testing individual components or functions in isolation for correct behavior.
    - *Integration Testing* – checking that different modules or services work together properly.
    - *System Testing* – validating the entire integrated system against the requirements

Testing Example & TDD

Unit Test Example (pytest):

```python
def calculate_total(price, quantity):
    return price * quantity

def test_calculate_total():
    assert calculate_total(10, 5) == 50
    assert calculate_total(0, 10) == 0
    assert calculate_total(7.5, 2) == 15.0
```

Approach: We follow good testing practices like writing test cases for both normal and edge conditions. Automated tests (using frameworks like `unittest` or `pytest` in Python) help catch regressions quickly. A solid test suite gives confidence to refactor code since you can verify nothing broke. In fact, one of the Agile practices is **Test-Driven Development (TDD)** – writing tests before implementing code to drive correct behavior and design.

SOLID Principles

- **SOLID Principles:** Five foundational design principles for maintainable OOP code:
    1. **S**ingle Responsibility Principle – a class should have only one reason to change (one responsibility).
    2. **O**pen/Closed Principle – software entities should be open for extension but closed for modification.
    3. **L**iskov Substitution Principle – objects of a superclass should be replaceable with objects of subclasses without breaking the system.
    4. **I**nterface Segregation Principle – no client should be forced to depend on methods it doesn't use (split large interfaces into smaller, specific ones).
    5. **D**ependency Inversion Principle – high-level modules should not depend on low-level modules; both should depend on abstractions.
- Applying SOLID leads to cleaner, more flexible designs. For instance, adhering to SRP

SOLID Example: Single Responsibility

Violation (Multiple Responsibilities):

```python
class User:
    def save_to_database(self):
        # Database logic here
        pass

    def send_email(self):
        # Email logic here
        pass
```

Following SRP (Single Responsibility):

```python
class User:
    def __init__(self, name, email):
        self.name = name
        self.email = email

class UserRepository:  # Handles DB operations
```

Design Patterns

- **Design Patterns:** A design pattern is a general, reusable solution to a commonly occurring problem in software design. It's a template for how to solve a problem that can be adapted to different situations, rather than a direct piece of code. Classic patterns (from the "Gang of Four" book) include creational patterns (e.g. Factory), structural patterns (e.g. Adapter), and behavioral patterns (e.g. Observer).
  - **2025 Reality:** Many patterns now built into modern languages, but remain essential as conceptual tools
  - 23 Gang of Four patterns still relevant as shared vocabulary among developers

Iterator Pattern Example

Iterator Pattern in Python:

```python
class BookCollection:
    def __init__(self):
        self.books = []

    def add_book(self, book):
        self.books.append(book)

    def __iter__(self):  # Iterator protocol
        return iter(self.books)

    def __next__(self):
        # Custom iteration logic if needed
        pass

# Usage
collection = BookCollection()
for book in collection:  # Clean, standard iteration
```

Iterator Pattern Concepts

- *Iterator Pattern:* The Iterator is a behavioral pattern that **provides a standard way to access elements of a collection sequentially without exposing the collection's internal structure**. For example, Python's iterator protocol (`__iter__` and `__next__`) allows you to loop over objects like lists and custom collections in a uniform way. Design patterns like this improve interoperability and code clarity. Using known patterns can also make your design more communicative – other developers recognize the pattern and understand the solution approach more readily.

Homework 3: High-Level Programming Languages

Building Todo Apps in JavaScript, TypeScript, and React

HW3: Project Overview

Built three implementations of the same Todo application to demonstrate the evolution and benefits of high-level programming languages:

1. **JavaScript Todo App** – Vanilla JavaScript with DOM manipulation
2. **TypeScript Todo App** – Type-safe version with strict typing
3. **React Todo App** – Component-based architecture with hooks

Goal: Understand how high-level languages and frameworks help create high-quality software effectively.

Application Structure Comparison

JavaScript

```
javascript-todo/
├── app.js              # Business logic and DOM manipulation
└── index.html          # Structure and styles
```

TypeScript

```
typescript-todo/
├── src/
│   ├── app.ts          # Type-safe source code
│   ├── index.html      # HTML structure
│   └── tsconfig.json   # TypeScript configuration
└── app.js              # Compiled JavaScript
```

React

How High-Level Languages Improve Software Quality

1. Abstraction - Focus on "What" Not "How"

JavaScript (Low-Level DOM Manipulation):

```
const li = document.createElement('li');
const checkbox = document.createElement('input');
checkbox.type = 'checkbox';
li.appendChild(checkbox);
// ... manual DOM construction
```

React (High-Level Declarative):

```
<li className={todo.completed ? 'completed' : ''}>
  <input type="checkbox" checked={todo.completed} />
  <span>{todo.text}</span>
</li>
```

Benefit: React abstracts away DOM manipulation, letting developers focus on UI logic

2. Type Safety - Catching Errors Early

JavaScript (Runtime Errors):

```javascript
function deleteTodo(id) {
  todos = todos.filter(t => t.id !== id);  // No type checking
}
```

TypeScript (Compile-Time Safety):

```typescript
private deleteTodo(id: number): void {
  const todo: Todo | undefined = this.todos.find(
    (t: Todo): boolean => t.id === id
  );
  // TypeScript ensures type correctness
}
```

Benefit: TypeScript catches type errors during development, preventing runtime bugs and providing better IDE support.

3. Component Reusability

JavaScript (Duplicate Code):

```
// Must repeat similar logic for each todo item
todos.forEach(todo => {
  const li = document.createElement('li');
  const checkbox = document.createElement('input');
  // ... repeat for every todo
});
```

React (Reusable Components):

```
const TodoItem: React.FC<TodoItemProps> = ({ todo, onToggle, onDelete }) => (
  <li className={todo.completed ? 'completed' : ''}>
    <input type="checkbox" onChange={() => onToggle(todo.id)} />
    <span>{todo.text}</span>
    <button onClick={() => onDelete(todo.id)}>Delete</button>
  </li>
);
```

4. State Management

JavaScript (Manual State Tracking):

```
let todos = [];

function addTodo() {
  todos.push(newTodo);
  displayTodos();  // Manually update UI
}
```

React (Automatic UI Updates):

```
const [todos, setTodos] = useState<Todo[]>([]);

const addTodo = (text: string): void => {
  setTodos([...todos, newTodo]);  // UI updates automatically
};
```

Benefit: React automatically re-renders when state changes, eliminating manual DOM

5. Maintainability & Scalability

Code Organization Comparison

Aspect	JavaScript	TypeScript	React
Structure	Functions	Classes	Components
Type Safety	None	Full	Full
Reusability	Limited	Medium	High
Testing	Difficult	Easier	Easiest
Team Collaboration	Hard	Better	Best
Scalability	Poor	Good	Excellent

Real-World Impact: Lines of Code

Same functionality, different complexity:

- **JavaScript:** ~140 lines of imperative code
- **TypeScript:** ~160 lines (extra type annotations pay off in safety)
- **React:** ~180 lines split into 4 reusable components

But consider:

- React code is more maintainable
- TypeScript prevents entire classes of bugs
- Components can be reused across projects

Developer Experience Benefits

JavaScript

- ❌ No autocomplete for object properties
- ❌ Runtime type errors
- ❌ Manual DOM updates prone to bugs
- ✅ Quick to start, no build step

TypeScript

- ✅ Full IDE autocomplete and IntelliSense
- ✅ Compile-time error detection
- ✅ Self-documenting interfaces
- ⚠️ Requires build step

Key Features Implemented

All three apps implement identical functionality:

1. ✅ **Add todos** - Create new todo items
2. ✅ **Toggle completion** - Mark as complete/incomplete
3. ✅ **Delete todos** - Remove individual items (with confirmation for uncompleted)
4. ✅ **Clear completed** - Bulk remove finished tasks
5. ✅ **Clear all** - Delete all todos (with confirmation)
6. ✅ **Persistent storage** (React only) - Uses localStorage

Lessons Learned: Language Evolution

JavaScript (1995)

- Foundation of web interactivity
- Flexible but error-prone
- Manual everything

TypeScript (2012)

- JavaScript + Type Safety
- Better tooling and refactoring
- Catches bugs before runtime

React (2013)

- Declarative UI paradigm

How High-Level Languages Enable Quality Software

1. **Faster Development**
    - Less boilerplate code
    - Built-in patterns and best practices
    - Rich ecosystems and libraries
2. **Fewer Bugs**
    - Type safety catches errors early
    - Automatic state management
    - Framework handles edge cases
3. **Better Collaboration**
    - Clear interfaces and contracts

Practical Application to Course Principles

APIEC Framework Applied:

- **Abstraction:** React abstracts DOM manipulation
- **Polymorphism:** Components accept different data through props
- **Inheritance:** TypeScript class hierarchy (TodoApp extends base functionality)
- **Encapsulation:** Private class members, component state
- **Composition:** React components composed of smaller components

SOLID Principles:

- **Single Responsibility:** Each React component has one job
- **Open/Closed:** Components extensible via props
- **Dependency Inversion:** React uses dependency injection via props

Conclusion: Why High-Level Languages Matter

The Evolution Path:

1. Machine Code → Assembly → C (increasing abstraction)
2. C → JavaScript (high-level, garbage collected)
3. JavaScript → TypeScript (type safety)
4. TypeScript → React (framework abstractions)

Each step trades some control for:

- ✅ Productivity gains
- ✅ Reduced bug rates
- ✅ Better maintainability
- ✅ Easier collaboration

Result: High-level languages and frameworks enable developers to build high-quality

References

1. M. Fowler, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.
2. E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.
3. Northern Kentucky University, CSC 640 Course Materials (Weeks 1–4 slides) (accessed 2025).
4. Industry Statistics: Git (2025), Agile/Scrum Surveys (2025), DevOps Reports (Forrester 2025), Software Testing Market Analysis (2024-2025).
5. HW3 Todo Applications - JavaScript, TypeScript, and React implementations (2025).

Week 5: High-Level Programming with JavaScript

The Journey from Low-Level to High-Level

The Problem with Low-Level Languages:

- C requires 500+ lines for a simple web server
- Manual memory management prone to errors
- Complex string handling and buffer management
- Takes weeks to build basic functionality

High-Level Languages Solve This:

- JavaScript: 20 lines for the same web server
- Automatic memory management (garbage collection)
- Built-in data structures and APIs
- Hours instead of weeks for development

Key Insight: High-level languages manage complexity so developers can focus on

What is JavaScript?

JavaScript is a high-level, interpreted programming language that runs in browsers and on servers (Node.js). Created in 1995 by Brendan Eich at Netscape, it has become the lingua franca of the web.

Key Characteristics:

- **Interpreted:** No compilation step required
- **Dynamic typing:** Variables can hold any type
- **First-class functions:** Functions are values that can be passed around
- **Prototype-based:** Object inheritance through prototypes
- **Event-driven:** Designed for asynchronous operations
- **Multi-paradigm:** Supports procedural, object-oriented, and functional programming

JavaScript's Role in Web Development

The Web Technology Stack:

- **HTML:** Structure and content
- **CSS:** Styling and layout
- **JavaScript:** Behavior and interactivity

JavaScript's Unique Position:

- Only language that runs natively in all browsers
- Both client-side (browser) and server-side (Node.js)
- Powers interactive web applications
- Enables Single Page Applications (SPAs)

Market Reality (2025):

JavaScript Basics: Variables and Types

Dynamic Typing:

```
let message = "Hello";          // String
let count = 42;                 // Number
let isActive = true;            // Boolean
let items = [1, 2, 3];          // Array
let user = {name: "Sarah"};     // Object
let nothing = null;             // Null
let notDefined;                 // Undefined
```

Variable Declarations:

- `var` - Function-scoped (old style, avoid)
- `let` - Block-scoped, can be reassigned
- `const` - Block-scoped, cannot be reassigned

Type Coercion:

Functions in JavaScript

Function Declarations:

```
function greet(name) {
    return "Hello, " + name;
}

// Arrow function (ES6+)
const greet = (name) => "Hello, " + name;

// Function as value
const sayHello = greet;
sayHello("World");  // "Hello, World"
```

Higher-Order Functions:

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

// Map: transform each element
const doubled = numbers.map(n => n * 2);  // [2, 4, 6, 8, 10]
```

Objects and Arrays

Objects (Key-Value Pairs):

```
const person = {
    name: "Sarah",
    age: 25,
    skills: ["JavaScript", "Python"],

    // Method
    introduce() {
        return `Hi, I'm ${this.name}`;
    }
};

// Access properties
console.log(person.name);          // "Sarah"
console.log(person["age"]);        // 25
console.log(person.introduce()); // "Hi, I'm Sarah"
```

Destructuring:

Asynchronous JavaScript: Callbacks

The Problem:
JavaScript is single-threaded, but operations like network requests take time. We need to handle operations that complete later without blocking execution.

Callback Pattern:

```javascript
function fetchData(url, callback) {
    // Simulate network request
    setTimeout(() => {
        const data = {id: 1, name: "Sarah"};
        callback(data);
    }, 1000);
}

// Usage
fetchData("/api/user", (user) => {
    console.log(user.name);  // "Sarah" (after 1 second)
});
```

Asynchronous JavaScript: Promises

Promises solve callback hell:

```
function fetchData(url) {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            if (url) {
                resolve({id: 1, name: "Sarah"});
            } else {
                reject(new Error("Invalid URL"));
            }
        }, 1000);
    });
}

// Chaining promises
fetchData("/api/user")
    .then(user => fetchPosts(user.id))
    .then(posts => fetchComments(posts[0].id))
    .then(comments => console.log(comments))
    .catch(error => console.error(error));
```

DOM Manipulation

The Document Object Model (DOM):

JavaScript's primary use in browsers is manipulating the DOM - the tree structure representing HTML.

Common DOM Operations:

```javascript
// Select elements
const button = document.getElementById("myButton");
const items = document.querySelectorAll(".item");

// Modify content
button.textContent = "Click Me";
button.innerHTML = "<strong>Click</strong> Me";

// Modify attributes
button.setAttribute("disabled", true);
button.classList.add("active");
```

JavaScript Ecosystem

Package Management:

- **npm** (Node Package Manager): 2+ million packages
- `package.json` - Project dependencies and scripts
- Install packages: `npm install express`

Popular Libraries/Frameworks:

- **React, Vue, Angular** - UI frameworks
- **Express** - Web server framework
- **Lodash** - Utility functions
- **Axios** - HTTP client
- **Jest** - Testing framework

Build Tools:

The Impact of JavaScript on Development

Productivity Gains:

- **100x faster development** compared to C for web tasks
- Hours instead of weeks for basic functionality
- Rich ecosystem reduces "reinventing the wheel"

Challenges JavaScript Introduces:

- **No type safety** - bugs caught at runtime, not compile time
- **Loose equality** - unexpected type coercion
- **Callback hell** - complex async code (solved by promises/async-await)
- **Growing codebases** - harder to maintain at scale

This leads us to Week 6: TypeScript addresses JavaScript's type safety issues while preserving its productivity benefits.

Week 6: TypeScript & React

From JavaScript to TypeScript: The Type Safety Evolution

JavaScript's Growing Pains:

- Fast development but runtime errors
- No compile-time type checking
- Refactoring is risky in large codebases
- Hard to maintain 5000+ line projects
- 90% of production bugs are type-related

TypeScript's Solution:

- Superset of JavaScript (all JS is valid TS)
- Static type checking at compile time
- Catches errors before code runs
- Better IDE support (autocomplete, refactoring)

What is TypeScript?

TypeScript is a strongly-typed programming language developed by Microsoft (2012) that builds on JavaScript by adding static type definitions.

Key Characteristics:

- **Transpiles to JavaScript:** TypeScript code compiles to plain JavaScript
- **Gradual typing:** Can adopt types incrementally
- **Type inference:** Types often inferred automatically
- **Modern features:** Includes latest ECMAScript features
- **Tooling support:** Excellent IDE integration

Market Reality (2025):

- Used by 78% of professional JavaScript developers
- Powers major projects: VSCode, Slack, Airbnb

TypeScript Basics: Type Annotations

Basic Types:

```typescript
let username: string = "Sarah";
let age: number = 25;
let isActive: boolean = true;
let values: number[] = [1, 2, 3];
let tuple: [string, number] = ["Sarah", 25];

// Type inference (TypeScript guesses the type)
let message = "Hello";  // inferred as string
```

Compare to JavaScript:

```typescript
// JavaScript - no type safety
let age = 25;
age = "twenty-five";  // No error, but likely a bug

// TypeScript - compile error
let age: number = 25;
```

Interfaces: Defining Object Shapes

Interface definition:

```
interface User {
    id: number;
    name: string;
    email: string;
    age?: number;            // Optional property
    readonly created: Date; // Cannot be modified
}

function createUser(data: User): void {
    console.log(`Creating user: ${data.name}`);
}

const newUser: User = {
    id: 1,
    name: "Sarah",
    email: "sarah@example.com",
    created: new Date()
```

Generics: Reusable Type-Safe Code

Generic functions:

```typescript
function identity<T>(value: T): T {
    return value;
}

const num = identity<number>(42);     // Returns number
const str = identity<string>("hello"); // Returns string

// Array utilities
function first<T>(arr: T[]): T | undefined {
    return arr[0];
}

const firstNum = first([1, 2, 3]);      // number | undefined
const firstStr = first(["a", "b", "c"]); // string | undefined
```

Introduction to React

What is React?

React is a JavaScript library for building user interfaces, created by Facebook (2013). It focuses on building reusable UI components with declarative syntax.

Core Concepts:

- **Components:** Reusable UI pieces
- **JSX:** HTML-like syntax in JavaScript
- **State:** Component data that changes
- **Props:** Data passed to components
- **Virtual DOM:** Efficient rendering

Why React?

- **Component-based:** Build complex UIs from simple pieces

React Components

Functional Components:

```
import React from 'react';

interface GreetingProps {
    name: string;
    age?: number;
}

function Greeting({name, age}: GreetingProps) {
    return (
        <div>
            <h1>Hello, {name}!</h1>
            {age && <p>Age: {age}</p>}
        </div>
    );
}

// Usage
<Greeting name="Sarah" age={25} />
```

React State with useState

State management:

```
import React, {useState} from 'react';

function Counter() {
    const [count, setCount] = useState<number>(0);

    return (
        <div>
            <p>Count: {count}</p>
            <button onClick={() => setCount(count + 1)}>
                Increment
            </button>
        </div>
    );
}
```

State rules:

React Effects with useEffect

Side effects:

```
import React, {useState, useEffect} from 'react';

function UserProfile({userId}: {userId: number}) {
    const [user, setUser] = useState<User | null>(null);
    const [loading, setLoading] = useState(true);

    useEffect(() => {
        // Runs after component mounts and when userId changes
        fetch(`/api/users/${userId}`)
            .then(res => res.json())
            .then(data => {
                setUser(data);
                setLoading(false);
            });
    }, [userId]);  // Dependency array

    if (loading) return <div>Loading...</div>;
    if (!user) return <div>User not found</div>;

    return <div>{user.name}</div>;
```

The Virtual DOM

How React Optimizes Performance:

Traditional DOM Updates:

1. User action triggers change
2. Directly manipulate DOM (expensive)
3. Browser reflows/repaints entire page

React's Virtual DOM:

1. User action triggers state change
2. React creates new virtual DOM tree (fast, in memory)
3. React diffs old and new virtual trees
4. React updates only changed parts of real DOM

Declarative vs Imperative UI

Imperative (jQuery style):

```
// Tell the browser HOW to do it
const button = document.getElementById("myButton");
button.addEventListener("click", () => {
    const counter = document.getElementById("counter");
    const count = parseInt(counter.textContent);
    counter.textContent = count + 1;
    if (count + 1 >= 10) {
        button.disabled = true;
    }
});
```

Declarative (React style):

```
// Tell React WHAT you want
function Counter() {
    const [count, setCount] = useState(0);
```

The Three Pillars of Modern Web Development

1. **High-Level Language (JavaScript)**
    - Productivity: 100x faster than C
    - Rich ecosystem and tooling
    - But: No type safety
2. **Type System (TypeScript)**
    - Static type checking
    - 90% fewer bugs
    - Better IDE support
    - But: Still needs better UI management
3. **Framework (React)**

From Week 5 to Week 6: The Evolution

Week 5 (JavaScript):

- Solved complexity problem from C
- Fast development, flexible
- But: Type errors, hard to maintain

Week 6 (TypeScript + React):

- TypeScript: Added type safety to JavaScript
- React: Solved UI state management
- Result: Fast, safe, maintainable

The Journey:

1. C (500 lines, 1 week) → JavaScript (20 lines, 2 hours)

Key Takeaways: TypeScript & React

TypeScript:

- Static typing prevents runtime errors
- Gradual adoption - use as much or as little as needed
- Essential for large codebases
- Better tooling and refactoring support

React:

- Component-based architecture promotes reusability
- Declarative syntax simplifies complex UIs
- Virtual DOM provides automatic optimization
- Large ecosystem and community

Together:

Key Takeaways

- **Week 1:** Foundation tools (VSCode, Git/GitHub, Marp) and course philosophy
- **Week 2:** Software processes (Waterfall → Agile → Scrum → DevOps)
- **Week 3:** Python, OOP fundamentals, UML, and requirements
- **Week 4:** APIEC framework, SOLID principles, design patterns, and testing
- **Week 5:** High-level programming with JavaScript - productivity gains, async programming, DOM manipulation, and ecosystem
- **Week 6:** TypeScript & React - type safety, component architecture, Virtual DOM, and declarative UI
- **HW3:** High-level programming languages (JavaScript → TypeScript → React) demonstrate how abstraction, type safety, and framework support enable developers to build high-quality software more effectively

Remember: High-quality software requires both technical excellence (SOLID, patterns)