GitHub Copilot Interface and Basic Usage 🎯

This guide provides hands-on exercises demonstrating GitHub Copilot's core interface features using Java examples. Perfect for understanding how to effectively interact with Copilot's suggestions and leverage context-aware code completion.

🎓 Learning Objectives

By completing these exercises, you will understand:

- How to recognize and interpret Copilot suggestions
- Techniques for accepting, rejecting, and cycling through suggestions
- How Copilot uses context to provide relevant completions
- Best practices for maximizing suggestion quality

🔧 Setup Requirements

- VS Code with GitHub Copilot extension enabled
- Java Development Kit (JDK 21 or higher)
- Maven build tool
- Access to the task-manager project in this repository
- **Foundation classes are pre-created** for immediate hands-on practice

📋 Pre-Exercise Verification

Before starting the exercises, verify your setup by running this test:

```
// Create this file to verify your setup: src/main/java/com/taskmanager/
app/TestSetup.java
package com.taskmanager.app;

import com.taskmanager.app.entity.User;
import com.taskmanager.app.entity.Task;
import com.taskmanager.app.entity.TaskStatus;
import com.taskmanager.app.repository.UserRepository;
import com.taskmanager.app.repository.TaskRepository;
import com.taskmanager.app.service.EmailService;
```

```
import com.taskmanager.app.service.PasswordEncoder;
import com.taskmanager.app.dto.CreateUserRequest;

public class TestSetup {
    public static void main(String[] args) {
        System.out.println("✅ All foundation classes are available!");
        System.out.println("✅ Ready for Copilot exercises!");

        // If this compiles without errors, you're ready to start
        User user = new User("John", "Doe", "john@example.com");
        Task task = new Task();
        CreateUserRequest request = new CreateUserRequest();

        System.out.println("Foundation setup verification complete!");
    }
}
```

Expected Result: This should compile without errors.

📚 Exercise 1: Understanding Copilot Suggestions

Objective: Learn to recognize and interpret different types of Copilot suggestions

Step 1: Basic Method Completion

Open the task-manager project and navigate to the `UserService.java` file (which should be empty). Let's start creating methods and observe Copilot's suggestions:

```
package com.taskmanager.app.service;

import com.taskmanager.app.entity.User;
import com.taskmanager.app.repository.UserRepository;
import com.taskmanager.app.service.EmailService;
import com.taskmanager.app.service.PasswordEncoder;
import java.util.List;
import java.util.Optional;

public class UserService {

    // Notice how these classes are now available with imports!
    // Type this comment and pause to see Copilot's suggestion
    // Create a method to find user by email
```

```
}
```

What to Observe:

- Copilot suggests a complete method signature and implementation
- Notice the ghost text (grayed-out suggestion)
- Suggestion includes proper return type, parameter, and logic using UserRepository
- The suggestion leverages the existing User entity and UserRepository interface

Expected Copilot Suggestion:

```
public Optional<User> findUserByEmail(String email) {
    return userRepository.findByEmail(email);
}
```

Why this works well now: Copilot can see the User entity, UserRepository interface, and their methods, providing contextually accurate suggestions.

Step 2: Context-Aware Field Completion

Add class fields and observe how Copilot adapts:

```
public class UserService {

    private final UserRepository userRepository;
    private final PasswordEncoder passwordEncoder;
    private final EmailService emailService;

    // Type: "public UserService(" and pause

}
```

What to Observe:

- Copilot suggests constructor with all declared fields
- Parameter names match field names
- Assignment statements are automatically generated

Expected Copilot Suggestion:

```
public UserService(UserRepository userRepository,
                   PasswordEncoder passwordEncoder,
                   EmailService emailService) {
    this.userRepository = userRepository;
    this.passwordEncoder = passwordEncoder;
    this.emailService = emailService;
}
```

⚡ Exercise 2: Accepting, Rejecting, and Cycling Through Suggestions

Objective: Master the keyboard shortcuts and techniques for managing suggestions

Step 1: Keyboard Shortcuts Practice

Use this method template to practice suggestion management:

```
public class TaskService {

    private final TaskRepository taskRepository;

    // Practice area - type the comment and method signature slowly
    // Create a method to update task status
    public void updateTaskStatus
}
```

Key Shortcuts to Practice:

Action	Shortcut	When to Use
Accept Suggestion	`Tab` or `→`	When suggestion looks correct
Reject Suggestion	`Esc`	When suggestion is wrong
Next Suggestion	`Alt + ]` (Windows/Linux) `Option + ]` (Mac)	To see alternative approaches

Action	Shortcut	When to Use
Previous Suggestion	`Alt + [` (Windows/Linux) `Option + [` (Mac)	To go back to previous option
Accept Word	`Ctrl + →` (Windows/Linux) `Cmd + →` (Mac)	Accept only part of suggestion

Step 2: Cycling Through Alternatives

Type this method signature and practice cycling:

```
// Create a method to validate task data
public boolean validateTask
```

Practice Sequence:

1. **Type** the signature slowly and **pause**
2. **Observe** the first suggestion
3. **Press** `Alt + ]` (or `Option + ]` on Mac) to see alternatives
4. **Cycle** through 3-4 different suggestions
5. **Accept** the most appropriate one with `Tab`

Common Alternative Suggestions You Might See:

```
// Option 1: Basic validation
public boolean validateTask(Task task) {
    return task != null && task.getTitle() != null && !
task.getTitle().isEmpty();
}

// Option 2: Comprehensive validation
public boolean validateTask(Task task) {
    if (task == null) return false;
    if (task.getTitle() == null || task.getTitle().trim().isEmpty())
return false;
    if (task.getAssignee() == null) return false;
    return true;
}

// Option 3: Exception-based validation
public boolean validateTask(Task task) throws ValidationException {
```

```
    Objects.requireNonNull(task, "Task cannot be null");
    if (task.getTitle() == null || task.getTitle().trim().isEmpty()) {
        throw new ValidationException("Task title is required");
    }
    return true;
}
```

🎯 Exercise 3: Context-Aware Code Completion

Objective: Understand how Copilot uses surrounding code context to provide relevant suggestions

Step 1: Method Context Awareness

Create a REST controller and observe context-aware suggestions:

```
package com.taskmanager.app.controller;

import org.springframework.web.bind.annotation.*;
import org.springframework.http.ResponseEntity;
import com.taskmanager.app.service.TaskService;
import com.taskmanager.app.entity.Task;

@RestController
@RequestMapping("/api/tasks")
public class TaskController {

    private final TaskService taskService;

    public TaskController(TaskService taskService) {
        this.taskService = taskService;
    }

    @GetMapping
    public ResponseEntity<List<Task>> getAllTasks() {
        List<Task> tasks = taskService.getAllTasks();
        return ResponseEntity.ok(tasks);
    }

    // Type this comment and observe context-aware suggestion
    // Create POST endpoint to create a new task
    @PostMapping
```

```
}
```

What to Observe:

- Copilot recognizes this is a REST controller pattern
- Suggests consistent method signature with existing patterns
- Includes proper annotations (@PostMapping, @RequestBody)
- Follows Spring Boot conventions

Expected Context-Aware Suggestion:

```
@PostMapping
public ResponseEntity<Task> createTask(@RequestBody Task task) {
    Task savedTask = taskService.saveTask(task);
    return ResponseEntity.ok(savedTask);
}
```

Step 2: Exception Handling Context

Add exception handling and see how context influences suggestions:

```
@PostMapping
public ResponseEntity<Task> createTask(@RequestBody Task task) {
    try {
        Task savedTask = taskService.saveTask(task);
        return ResponseEntity.ok(savedTask);
    }
    // Type "catch" and observe contextual suggestions

}
```

Context-Aware Suggestions You Might See:

```
// Option 1: Generic exception handling
catch (Exception e) {
    return ResponseEntity.internalServerError().build();
}

// Option 2: Specific exception handling
catch (ValidationException e) {
    return ResponseEntity.badRequest().body(null);
} catch (DataAccessException e) {
```

```
    return ResponseEntity.internalServerError().build();
}

// Option 3: Detailed error response
catch (Exception e) {
    logger.error("Error creating task", e);
    return ResponseEntity.internalServerError()
        .body(new ErrorResponse("Failed to create task: " +
e.getMessage()));
}
```

🔄 Exercise 4: Advanced Context Usage

Objective: Leverage file context and project patterns for better suggestions

Step 1: Using Existing Code Patterns

Open the existing `User.java` entity in the project, then create a new `Task.java` entity:

```
package com.taskmanager.app.entity;

import javax.persistence.*;
import java.time.LocalDateTime;

// Type this comment and let Copilot suggest based on existing User entity
pattern
// Create Task entity with JPA annotations
@Entity
@Table(name = "tasks")
public class Task {

    // Copilot will suggest fields similar to User entity structure

}
```

What to Observe:

- Copilot analyzes existing entity patterns in your project
- Suggests similar field structures and annotations
- Follows established naming conventions
- Includes appropriate data types and relationships

Expected Pattern-Based Suggestion:

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long id;

@Column(nullable = false)
private String title;

@Column(length = 1000)
private String description;

@Enumerated(EnumType.STRING)
private TaskStatus status;

@ManyToOne(fetch = FetchType.LAZY)
@JoinColumn(name = "assignee_id")
private User assignee;

@Column(name = "created_date")
private LocalDateTime createdDate;

@Column(name = "due_date")
private LocalDateTime dueDate;
```

Step 2: Test-Driven Development Context

Create a test class and observe test-specific suggestions:

```
package com.taskmanager.app.service;

import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.BeforeEach;
import org.mockito.Mock;
import org.mockito.MockitoAnnotations;
import static org.junit.jupiter.api.Assertions.*;
import static org.mockito.Mockito.*;

public class TaskServiceTest {

    @Mock
    private TaskRepository taskRepository;

    private TaskService taskService;

    @BeforeEach
```

```
    void setUp() {
        MockitoAnnotations.openMocks(this);
        taskService = new TaskService(taskRepository);
    }

    // Type this comment and observe testing-specific suggestions
    // Test creating a valid task
    @Test
    void
}
```

Testing Context Suggestions:

```
@Test
void testCreateValidTask() {
    // Given
    Task task = new Task();
    task.setTitle("Test Task");
    task.setDescription("Test Description");

    when(taskRepository.save(any(Task.class))).thenReturn(task);

    // When
    Task result = taskService.createTask(task);

    // Then
    assertNotNull(result);
    assertEquals("Test Task", result.getTitle());
    verify(taskRepository, times(1)).save(task);
}
```

📊 Exercise 5: Measuring Suggestion Quality

Objective: Learn to evaluate and improve suggestion quality

Quality Assessment Checklist:

Criteria	Good Suggestion ✅	Poor Suggestion ❌
Syntax	Compiles without errors	Syntax errors present
Conventions	Follows Java naming conventions	Inconsistent naming

Criteria	Good Suggestion ✅	Poor Suggestion ❌
Context	Matches existing code patterns	Ignores project structure
Logic	Implements reasonable logic	Logic errors or gaps
Documentation	Includes relevant comments	Missing or incorrect comments

Practice: Improve This Suggestion

Original prompt:

```
// Create a method to process user registration
public void processUserRegistration
```

Poor Quality Suggestion:

```
public void processUserRegistration(String name, String email) {
    // Save user
    System.out.println("User saved: " + name);
}
```

High Quality Suggestion (what to expect after refinement):

```
public User processUserRegistration(CreateUserRequest request) {
    // Validate input
    validateUserRequest(request);

    // Check if user already exists
    if (userRepository.findByEmail(request.getEmail()).isPresent()) {
        throw new UserAlreadyExistsException("User with email " +
request.getEmail() + " already exists");
    }

    // Create new user
    User user = User.builder()
        .firstName(request.getFirstName())
        .lastName(request.getLastName())
        .email(request.getEmail())
        .password(passwordEncoder.encode(request.getPassword()))
        .createdDate(LocalDateTime.now())
        .build();

    // Save and send welcome email
```

```
    User savedUser = userRepository.save(user);
    emailService.sendWelcomeEmail(savedUser.getEmail());

    return savedUser;
}
```

💡 Best Practices for Interface Usage

Do's ✅

1. **Pause After Comments**: Write descriptive comments and pause to get better suggestions
2. **Review Before Accepting**: Always read the suggestion before pressing Tab
3. **Use Descriptive Names**: Better variable/method names lead to better suggestions
4. **Provide Context**: Include relevant imports and field declarations
5. **Cycle Through Options**: Don't accept the first suggestion automatically

Don'ts ❌

1. **Don't Accept Blindly**: Always understand what the suggestion does
2. **Don't Ignore Errors**: Red squiggly lines indicate problems to fix
3. **Don't Skip Testing**: Test suggested code before committing
4. **Don't Override Good Patterns**: Let Copilot follow existing project conventions
5. **Don't Rush**: Take time to evaluate suggestion quality

🚀 Progressive Learning Path

Beginner

- Practice basic acceptance/rejection of suggestions
- Learn keyboard shortcuts
- Focus on simple method completions

Intermediate

- Master suggestion cycling techniques
- Use context-aware completions effectively

- Create more complex code structures

Advanced

- Leverage project patterns for suggestions
- Combine multiple context sources
- Optimize suggestion quality through better prompts

🔧 Troubleshooting Common Issues

Problem: Suggestions are not appearing

Solutions:

1. Check Copilot status in VS Code status bar
2. Restart VS Code
3. Verify internet connection
4. Check file type is supported (.java)

Problem: Poor quality suggestions

Solutions:

1. Add more context (imports, comments, existing code)
2. Use more descriptive variable/method names
3. Follow consistent coding patterns
4. Break complex logic into smaller methods

Problem: Suggestions don't match project patterns

Solutions:

1. Open related files in the same VS Code session
2. Add relevant imports and dependencies
3. Use consistent naming conventions
4. Reference existing project code in comments

🎯 Assessment Questions

Test your understanding with these questions:

1. **What keyboard shortcut cycles to the next Copilot suggestion?**
2. **How does Copilot use context from open files to improve suggestions?**
3. **When should you reject a Copilot suggestion?**
4. **What makes a high-quality Copilot suggestion?**
5. **How can you improve the quality of suggestions you receive?**

🏁 Conclusion

Mastering GitHub Copilot's interface and basic usage is fundamental to AI-assisted development. The key is to:

- **Understand** how suggestions work and what influences their quality
- **Practice** the keyboard shortcuts until they become natural
- **Evaluate** suggestions critically before accepting them
- **Use context** effectively to get better, more relevant suggestions

Remember: Copilot is a powerful assistant, but you remain the developer making the final decisions. Use these skills to enhance your productivity while maintaining code quality and project consistency.

Next Steps: Practice these exercises with your own code, then move on to Advanced Copilot Chat Features to learn about conversational AI assistance.