## **Bug Hunt Game Solutions**

Below are the correct code snippets for each bug, categorized by language and difficulty level.

### **Python**

#### **Easy Difficulty**

* **Q1: Incorrect indentation in function**  
  def greet\_user(name):  
   print(f"Hello, {name}!")  
   print("Welcome to our program!")
* **Q2: Wrong comparison operator**  
  def can\_vote(age):  
   if age >= 18:  
   return True  
   return False
* **Q3: Missing colon after if statement**  
  def check\_positive(num):  
   if num > 0:  
   return "Positive"  
   else:  
   return "Not positive"
* **Q4: Concatenating string with integer**  
  age = 30  
  message = "I am " + str(age) + " years old."
* **Q5: Dangerous mutable default parameter**  
  def add\_item(item, shopping\_list=None):  
   if shopping\_list is None:  
   shopping\_list = []  
   shopping\_list.append(item)  
   return shopping\_list
* **Q6: Incorrect indentation in function**  
  def greet\_user(name):  
   print(f"Hello, {name}!")
* **Q7: Wrong comparison operator**  
  def can\_vote(age):  
   if age >= 18:  
   return True  
   return False
* **Q8: Missing colon after if statement**  
  def check\_positive(num):  
   if num > 0:  
   return "Positive"
* **Q9: Concatenating string with integer**  
  value = 10  
  text = "Number: " + str(value)
* **Q10: Dangerous mutable default parameter**  
  def log\_message(msg, logs=None):  
   if logs is None:  
   logs = []  
   logs.append(msg)  
   return logs
* **Q11: Missing parenthesis in print**  
  print("Hello Python")
* **Q12: Incorrect loop range**  
  for i in range(1, 6):  
   print(i)
* **Q13: Missing indentation for class method**  
  class MyClass:  
   def my\_method():  
   pass
* **Q14: Incorrect dictionary key access**  
  data = {'name': 'Alice'}  
  print(data['name'])
* **Q15: Using list comprehension for mutable default**  
  def process\_data(data, config=None):  
   if config is None:  
   config = {}  
   config['processed'] = True  
   return config
* **Q16: Unmatched quote**  
  print('This is a string')
* **Q17: Incorrect boolean logic**  
  is\_admin = False  
  is\_editor = True  
  if is\_admin or (is\_editor and not is\_admin):  
   print('Access granted')
* **Q18: Incorrect indentation after function call**  
  if True:  
   print('Inside if')  
  print('Outside if')
* **Q19: Attempting to modify a tuple**  
  my\_list = [1, 2, 3]  
  my\_list[0] = 4
* **Q20: Replicated Mutable Default Argument**  
  def process\_data(data, config=None):  
   if config is None:  
   config = {}  
   config['processed'] = True  
   return config

#### **Medium Difficulty**

* **Q1: Dangerous mutable default parameter**  
  def add\_item(item, shopping\_list=None):  
   if shopping\_list is None:  
   shopping\_list = []  
   shopping\_list.append(item)  
   return shopping\_list
* **Q2: Late binding in loops with closures**  
  funcs = []  
  for i in range(3):  
   def f(j=i):  
   print(j)  
   funcs.append(f)  
  for f in funcs:  
   f()
* **Q3: Modifying a shallow copy**  
  import copy  
  original = [[1, 2], [3, 4]]  
  copy\_deep = copy.deepcopy(original)  
  copy\_deep[0][0] = 99
* **Q4: Dangerous mutable default parameter**  
  def cache\_result(arg, cache=None):  
   if cache is None:  
   cache = {}  
   if arg in cache:  
   return cache[arg]  
   result = arg \* 2  
   cache[arg] = result  
   return result
* **Q5: Late binding in loops with closures**  
  actions = []  
  for x in [1, 2, 3]:  
   actions.append(lambda val=x: val \* 2)  
  print([action() for action in actions])
* **Q6: Modifying a shallow copy**  
  import copy  
  list\_of\_lists = [[1], [2], [3]]  
  deep\_copy\_list = copy.deepcopy(list\_of\_lists)  
  deep\_copy\_list[0].append(9)
* **Q7: Dangerous mutable default parameter**  
  def process\_names(name, processed\_names=None):  
   if processed\_names is None:  
   processed\_names = []  
   processed\_names.append(name.upper())  
   return processed\_names
* **Q8: Late binding in loops with closures**  
  def make\_multipliers():  
   return [lambda x, i=i: i \* x for i in range(4)]  
  multipliers = make\_multipliers()  
  print([m(2) for m in multipliers])
* **Q9: Modifying a shallow copy**  
  import copy  
  dict1 = {'a': [1,2]}  
  dict2 = copy.deepcopy(dict1)  
  dict2['a'].append(3)
* **Q10: Dangerous mutable default parameter**  
  def create\_object(name, attributes=None):  
   if attributes is None:  
   attributes = {}  
   attributes['timestamp'] = 'now'  
   return attributes
* **Q11: Late binding in loops with closures**  
  def factory():  
   functions = []  
   for j in range(3):  
   functions.append(lambda val=j: val)  
   return functions  
  funcs = factory()  
  print([f() for f in funcs])
* **Q12: Modifying a shallow copy**  
  import copy  
  matrix = [[1, 2], [3, 4]]  
  new\_matrix = copy.deepcopy(matrix)  
  new\_matrix[0][0] = 0
* **Q13: Dangerous mutable default parameter**  
  def add\_log\_entry(entry, log\_entries=None):  
   if log\_entries is None:  
   log\_entries = []  
   log\_entries.append(entry)  
   return log\_entries
* **Q14: Late binding in loops with closures**  
  callbacks = []  
  for k in range(5):  
   callbacks.append(lambda val=k: val\*val)  
  print([cb() for cb in callbacks])
* **Q15: Modifying a shallow copy**  
  import copy  
  nested\_dict = {'data': {'value': 10}}  
  copy\_dict = copy.deepcopy(nested\_dict)  
  copy\_dict['data']['value'] = 20
* **Q16: Dangerous mutable default parameter**  
  def configure\_app(settings=None):  
   if settings is None:  
   settings = {}  
   settings['configured'] = True  
   return settings
* **Q17: Late binding in loops with closures**  
  def get\_multipliers():  
   multipliers = []  
   for factor in range(1, 4):  
   multipliers.append(lambda x, f=factor: x \* f)  
   return multipliers  
  funcs = get\_multipliers()  
  print([f(5) for f in funcs])
* **Q18: Modifying a shallow copy**  
  import copy  
  set\_of\_lists = [{1}, {2}, {3}]  
  copy\_set = copy.deepcopy(set\_of\_lists)  
  list(copy\_set)[0].add(4)
* **Q19: Dangerous mutable default parameter**  
  def add\_to\_set(element, current\_set=None):  
   if current\_set is None:  
   current\_set = set()  
   current\_set.add(element)  
   return current\_set
* **Q20: Late binding in loops with closures**  
  def generate\_greeters():  
   greeters = []  
   for name in ['Alice', 'Bob']:  
   greeters.append(lambda n=name: f'Hello, {n}')  
   return greeters  
  funcs = generate\_greeters()  
  print([f() for f in funcs])

#### **Hard Difficulty**

* **Q1: Race condition in thread unsafe counter**  
  import threading  
    
  counter = 0  
  lock = threading.Lock()  
  def increment():  
   global counter  
   for \_ in range(100000):  
   with lock:  
   counter += 1  
    
  threads = [threading.Thread(target=increment) for \_ in range(5)]  
  for t in threads: t.start()  
  for t in threads: t.join()  
  print(f"Final counter: {counter}")
* **Q2: Circular reference preventing garbage collection**  
  import weakref  
    
  class Node:  
   def \_\_init\_\_(self, value):  
   self.value = value  
   self.next = None  
   self.prev = None  
    
   def \_\_del\_\_(self):  
   print(f"Deleting Node {self.value}")  
    
  n1 = Node(1)  
  n2 = Node(2)  
  n1.next = n2  
  n2.prev = weakref.ref(n1) # Use weak reference for back-pointer  
    
  del n1  
  del n2  
  import gc  
  gc.collect()  
  print(len(gc.get\_objects()))
* **Q3: Race condition in thread unsafe counter**  
  import threading  
    
  count = {'value': 0}  
  lock = threading.Lock()  
  def add\_to\_count():  
   for \_ in range(100000):  
   with lock:  
   count['value'] += 1  
    
  threads = [threading.Thread(target=add\_to\_count) for \_ in range(5)]  
  for t in threads: t.start()  
  for t in threads: t.join()  
  print(f"Final count: {count['value']}")
* **Q4: Circular reference in linked list**  
  import weakref  
    
  class LinkedListNode:  
   def \_\_init\_\_(self, data):  
   self.data = data  
   self.next = None  
    
   def set\_next(self, node):  
   self.next = node  
    
  node1 = LinkedListNode(1)  
  node2 = LinkedListNode(2)  
  node1.set\_next(node2)  
  # node2.set\_next(weakref.ref(node1)) # No direct circular ref for simple lists  
    
  del node1  
  del node2  
  import gc  
  gc.collect()
* **Q5: Race condition in thread unsafe counter**  
  import threading  
    
  shared\_data = [0]  
  lock = threading.Lock()  
  def modify\_data():  
   for \_ in range(100000):  
   with lock:  
   shared\_data[0] += 1  
    
  threads = [threading.Thread(target=modify\_data) for \_ in range(5)]  
  for t in threads: t.start()  
  for t in threads: t.join()  
  print(f"Final data: {shared\_data[0]}")
* **Q6: Circular reference preventing garbage collection**  
  import weakref  
    
  class A:  
   def \_\_init\_\_(self):  
   self.b = None  
  class B:  
   def \_\_init\_\_(self):  
   self.a = None  
    
  a = A()  
  b = B()  
  a.b = b  
  b.a = weakref.ref(a) # Break circular reference  
    
  del a  
  del b  
  import gc  
  gc.collect()
* **Q7: Race condition in thread unsafe counter**  
  import threading  
    
  value = 0  
  lock = threading.Lock()  
    
  def increment\_value():  
   global value  
   for \_ in range(100000):  
   with lock:  
   temp = value  
   value = temp + 1  
    
  threads = [threading.Thread(target=increment\_value) for \_ in range(5)]  
  for t in threads: t.start()  
  for t in threads: t.join()  
  print(f"Final value: {value}")
* **Q8: Circular reference preventing garbage collection**  
  import weakref  
    
  class Parent:  
   def \_\_init\_\_(self, child):  
   self.child = child  
  class Child:  
   def \_\_init\_\_(self, parent\_ref):  
   self.parent = parent\_ref  
    
  p = Parent(None)  
  c = Child(weakref.ref(p))  
  p.child = c  
    
  del p  
  del c  
  import gc  
  gc.collect()
* **Q9: Race condition in thread unsafe counter**  
  import threading  
    
  data\_list = []  
  lock = threading.Lock()  
  def append\_data():  
   for i in range(10000):  
   with lock:  
   data\_list.append(i)  
    
  threads = [threading.Thread(target=append\_data) for \_ in range(10)]  
  for t in threads: t.start()  
  for t in threads: t.join()  
  print(f"List length: {len(data\_list)}")
* **Q10: Circular reference preventing garbage collection**  
  import weakref  
    
  class ClassA:  
   def \_\_init\_\_(self):  
   self.connected\_obj = None  
  class ClassB:  
   def \_\_init\_\_(self):  
   self.connected\_obj = None  
    
  o1 = ClassA()  
  o2 = ClassB()  
  o1.connected\_obj = o2  
  o2.connected\_obj = weakref.ref(o1)  
    
  del o1  
  del o2  
  import gc  
  gc.collect()
* **Q11: Race condition in thread unsafe counter**  
  import threading  
    
  counter = 0  
  lock = threading.Lock()  
  def increment():  
   global counter  
   for \_ in range(100000):  
   with lock:  
   counter += 1  
    
  threads = [threading.Thread(target=increment) for \_ in range(5)]  
  for t in threads: t.start()  
  for t in threads: t.join()  
  print(f"Final counter: {counter}")
* **Q12: Circular reference preventing garbage collection**  
  import weakref  
    
  class Node:  
   def \_\_init\_\_(self, value):  
   self.value = value  
   self.next = None  
   self.prev = None  
    
   def \_\_del\_\_(self):  
   print(f"Deleting Node {self.value}")  
    
  n1 = Node(1)  
  n2 = Node(2)  
  n1.next = n2  
  n2.prev = weakref.ref(n1)  
    
  del n1  
  del n2  
  import gc  
  gc.collect()  
  print(len(gc.get\_objects()))
* **Q13: Race condition in thread unsafe counter**  
  import threading  
    
  count = {'value': 0}  
  lock = threading.Lock()  
  def add\_to\_count():  
   for \_ in range(100000):  
   with lock:  
   count['value'] += 1  
    
  threads = [threading.Thread(target=add\_to\_count) for \_ in range(5)]  
  for t in threads: t.start()  
  for t in threads: t.join()  
  print(f"Final count: {count['value']}")
* **Q14: Circular reference in linked list**  
  import weakref  
    
  class LinkedListNode:  
   def \_\_init\_\_(self, data):  
   self.data = data  
   self.next = None  
    
   def set\_next(self, node):  
   self.next = node  
    
  node1 = LinkedListNode(1)  
  node2 = LinkedListNode(2)  
  node1.set\_next(node2)  
  # node2.set\_next(weakref.ref(node1))  
    
  del node1  
  del node2  
  import gc  
  gc.collect()
* **Q15: Race condition in thread unsafe counter**  
  import threading  
    
  shared\_data = [0]  
  lock = threading.Lock()  
  def modify\_data():  
   for \_ in range(100000):  
   with lock:  
   shared\_data[0] += 1  
    
  threads = [threading.Thread(target=modify\_data) for \_ in range(5)]  
  for t in threads: t.start()  
  for t in threads: t.join()  
  print(f"Final data: {shared\_data[0]}")
* **Q16: Circular reference preventing garbage collection**  
  import weakref  
    
  class A:  
   def \_\_init\_\_(self):  
   self.b = None  
  class B:  
   def \_\_init\_\_(self):  
   self.a = None  
    
  a = A()  
  b = B()  
  a.b = b  
  b.a = weakref.ref(a)  
    
  del a  
  del b  
  import gc  
  gc.collect()
* **Q17: Race condition in thread unsafe counter**  
  import threading  
    
  value = 0  
  lock = threading.Lock()  
    
  def increment\_value():  
   global value  
   for \_ in range(100000):  
   with lock:  
   temp = value  
   value = temp + 1  
    
  threads = [threading.Thread(target=increment\_value) for \_ in range(5)]  
  for t in threads: t.start()  
  for t in threads: t.join()  
  print(f"Final value: {value}")
* **Q18: Circular reference preventing garbage collection**  
  import weakref  
    
  class Parent:  
   def \_\_init\_\_(self, child):  
   self.child = child  
  class Child:  
   def \_\_init\_\_(self, parent\_ref):  
   self.parent = parent\_ref  
    
  p = Parent(None)  
  c = Child(weakref.ref(p))  
  p.child = c  
    
  del p  
  del c  
  import gc  
  gc.collect()
* **Q19: Race condition in thread unsafe counter**  
  import threading  
    
  data\_list = []  
  lock = threading.Lock()  
  def append\_data():  
   for i in range(10000):  
   with lock:  
   data\_list.append(i)  
    
  threads = [threading.Thread(target=append\_data) for \_ in range(10)]  
  for t in threads: t.start()  
  for t in threads: t.join()  
  print(f"List length: {len(data\_list)}")
* **Q20: Circular reference preventing garbage collection**  
  import weakref  
    
  class ClassA:  
   def \_\_init\_\_(self):  
   self.connected\_obj = None  
  class ClassB:  
   def \_\_init\_\_(self):  
   self.connected\_obj = None  
    
  o1 = ClassA()  
  o2 = ClassB()  
  o1.connected\_obj = o2  
  o2.connected\_obj = weakref.ref(o1)  
    
  del o1  
  del o2  
  import gc  
  gc.collect()

### **JavaScript**

#### **Easy Difficulty**

* **Q1: Wrong comparison operator**  
  function canVote(age) {  
   if (age >= 18) {  
   return true;  
   }  
   return false;  
  }
* **Q2: Accessing property of undefined object**  
  let user = { name: "John Doe" };  
  console.log(user.name);
* **Q3: Variable declared with var in a loop**  
  for (let i = 0; i < 3; i++) {  
   setTimeout(function() {  
   console.log(i);  
   }, 1000);  
  }
* **Q4: Missing await in async function**  
  async function fetchUserData(userId) {  
   try {  
   let response = await fetch(`/api/users/${userId}`);  
   let userData = await response.json();  
   return userData.name;  
   } catch (error) {  
   return "Error loading user";  
   }  
  }
* **Q5: Incorrect array length check**  
  function isEmpty(arr) {  
   return arr && arr.length === 0;  
  }
* **Q6: Calling method on null**  
  let data = {};  
  console.log(data.toString());
* **Q7: Variable declared with var in a loop causing unexpected behavior**  
  const buttons = document.querySelectorAll('button');  
  for (let i = 0; i < buttons.length; i++) {  
   buttons[i].onclick = function() {  
   console.log('Button ' + i + ' clicked');  
   };  
  }
* **Q8: Forgetting to await a promise**  
  async function processData() {  
   let result = await Promise.resolve(10);  
   console.log(result);  
  }
* **Q9: Incorrect conditional for string check**  
  function checkString(str) {  
   if (str === 'hello') {  
   return true;  
   }  
   return false;  
  }
* **Q10: Typo in method name**  
  let arr = [1, 2, 3];  
  arr.push(4);
* **Q11: Variable declared with var in a loop**  
  for (let k = 0; k < 5; k++) {  
   setTimeout(() => console.log(k), 0);  
  }
* **Q12: Missing catch for promise rejection**  
  new Promise((resolve, reject) => {  
   reject('Something went wrong!');  
  }).catch(error => console.error(error));
* **Q13: Incorrect logical operator**  
  function isEvenAndPositive(num) {  
   return num % 2 === 0 && num > 0;  
  }
* **Q14: Using const without initialization**  
  const PI = 3.14;
* **Q15: Variable access before declaration (hoisting)**  
  var myVar = 10;  
  console.log(myVar);
* **Q16: Async function not returning a promise**  
  async function fetchData() {  
   return Promise.resolve('Data');  
  }
* **Q17: Incorrect comparison for object equality**  
  let obj1 = {a: 1};  
  let obj2 = {a: 1};  
  console.log(JSON.stringify(obj1) === JSON.stringify(obj2));
* **Q18: Attempting to reassign a const variable**  
  let count = 5;  
  count = 10;
* **Q19: Using 'this' incorrectly in a callback**  
  class MyClass {  
   constructor() {  
   this.value = 10;  
   }  
   logValue() {  
   setTimeout(() => {  
   console.log(this.value);  
   }, 100);  
   }  
  }
* **Q20: Missing await in async function**  
  async function fetchData() {  
   let data = await Promise.resolve('Hello');  
   return data;  
  }

#### **Medium Difficulty**

* **Q1: Variable declared with var in a loop**  
  for (let i = 0; i < 3; i++) {  
   setTimeout(function() {  
   console.log(i);  
   }, 1000);  
  }
* **Q2: Missing await in async function**  
  async function fetchUserData(userId) {  
   try {  
   let response = await fetch(`/api/users/${userId}`);  
   let userData = await response.json();  
   return userData.name;  
   } catch (error) {  
   return "Error loading user";  
   }  
  }
* **Q3: Incorrect 'this' context in event handler**  
  class MyButton {  
   constructor(label) {  
   this.label = label;  
   this.element = document.createElement('button');  
   this.element.textContent = label;  
   this.element.addEventListener('click', this.handleClick.bind(this));  
   }  
   handleClick() {  
   console.log(`Button ${this.label} clicked`);  
   }  
  }
* **Q4: Loose equality (==) causing unexpected results**  
  if ('0' === false) {  
   console.log('True');  
  } else {  
   console.log('False');  
  }
* **Q5: Adding method to Array.prototype incorrectly**  
  Object.defineProperty(Array.prototype, 'first', {  
   value: function() { return this[0]; },  
   enumerable: false  
  });  
    
  for (let key in []) {  
   console.log(key);  
  }
* **Q6: Variable declared with var in a loop causing unexpected behavior**  
  const items = ['a', 'b', 'c'];  
  const callbacks = [];  
  for (let i = 0; i < items.length; i++) {  
   callbacks.push(function() { console.log(items[i]); });  
  }  
  callbacks.forEach(cb => cb());
* **Q7: Chaining promises without returning**  
  function step1() { return Promise.resolve(1); }  
  function step2(val) { return Promise.resolve(val + 1); }  
    
  step1().then(result => {  
   return step2(result); // Return the promise  
  }).then(finalResult => {  
   console.log(finalResult);  
  });
* **Q8: Incorrect 'this' context in event handler**  
  document.getElementById('myBtn').addEventListener('click', function() {  
   const self = this; // Capture 'this'  
   console.log(self.id);  
   setTimeout(function() {  
   console.log(self.id);  
   }, 100);  
  });  
  // OR using arrow function:  
  // document.getElementById('myBtn').addEventListener('click', function() {  
  // console.log(this.id);  
  // setTimeout(() => {  
  // console.log(this.id);  
  // }, 100);  
  // });
* **Q9: Loose equality with numbers and strings**  
  console.log(10 === '10');
* **Q10: Incorrectly extending built-in objects**  
  // Avoid extending built-in prototypes directly  
  function reverseString(str) {  
   return str.split('').reverse().join('');  
  }  
    
  let str = 'hello';  
  console.log(reverseString(str));
* **Q11: Variable declared with var in a loop**  
  for (let i = 0; i < 3; i++) {  
   setTimeout(function() {  
   console.log('Value is: ' + i);  
   }, 1000);  
  }
* **Q12: Missing await in async function**  
  async function processRequest() {  
   let data = await fetch('/api/data');  
   let json = await data.json();  
   return json;  
  }
* **Q13: Incorrect 'this' context in callback**  
  const obj = {  
   name: 'Object',  
   greet: function() {  
   setTimeout(() => {  
   console.log(`Hello from ${this.name}`);  
   }, 100);  
   }  
  };
* **Q14: Loose equality with null and undefined**  
  console.log(null === undefined);
* **Q15: Incorrectly using instanceof with objects from different realms**  
  const arr = [1, 2, 3];  
  console.log(Object.prototype.toString.call(arr) === '[object Array]');
* **Q16: Variable declared with var in a loop**  
  var functions = [];  
  for (let i = 0; i < 3; i++) {  
   functions.push(function() { return i; });  
  }  
  console.log(functions[0](), functions[1](), functions[2]());
* **Q17: Not handling promise rejection correctly**  
  new Promise((resolve, reject) => {  
   reject('Failed to load data');  
  }).catch(error => {  
   console.error('Caught error:', error);  
  });
* **Q18: Incorrect 'this' context in forEach callback**  
  const team = {  
   name: 'Warriors',  
   players: ['Curry', 'Thompson'],  
   listPlayers: function() {  
   this.players.forEach((player) => {  
   console.log(`${this.name}: ${player}`); // 'this' is lexically bound  
   });  
   }  
  };
* **Q19: Adding string to number**  
  let result = parseInt('5') + 5;  
  console.log(result);
* **Q20: Incorrectly checking for own property**  
  const obj = { a: 1, b: 2 };  
  for (let key in obj) {  
   if (Object.prototype.hasOwnProperty.call(obj, key)) {  
   console.log(key);  
   }  
  }

#### **Hard Difficulty**

* **Q1: Incorrect order of async operations in event loop**  
  console.log('Start');  
    
  Promise.resolve().then(() => console.log('Promise'));  
    
  setTimeout(() => console.log('Timeout'), 0);  
    
  console.log('End');
* **Q2: Retained closure in long-lived event handler**  
  function setupButton() {  
   let largeArray = null; // Don't create if not needed in handler  
   const button = document.getElementById('myBtn');  
   const handler = () => {  
   // If largeArray is truly needed, handle its lifecycle or pass relevant data  
   console.log('Button clicked');  
   };  
   button.addEventListener('click', handler);  
   // To prevent leak, remove handler when button is no longer needed:  
   // button.removeEventListener('click', handler);  
   // Or avoid capturing large variables in long-lived closures.  
  }
* **Q3: Incorrect order of async operations**  
  console.log('A');  
  console.log('E');  
  Promise.resolve().then(() => console.log('D'));  
  queueMicrotask(() => console.log('B'));  
  setTimeout(() => console.log('C'), 0);
* **Q4: Detached DOM nodes**  
  let element = document.createElement('div');  
  document.body.appendChild(element);  
  // Ensure all references are nulled out if element is removed and not needed  
  element.remove();  
  element = null; // Explicitly nullify reference if not needed  
  // reference = null; // Also nullify other references
* **Q5: Mixing synchronous and asynchronous array iteration**  
  async function processAllItems() {  
   const items = [1, 2, 3];  
   for (const item of items) {  
   await new Promise(resolve => setTimeout(resolve, 100));  
   console.log(item);  
   }  
   console.log('All items processed');  
  }  
  processAllItems();
* **Q6: Circular references in JavaScript objects**  
  function createCircularRef() {  
   const obj1 = {};  
   const obj2 = {};  
   obj1.ref = obj2;  
   // No circular ref, or carefully manage it if necessary  
   // obj2.ref = obj1;  
   return obj1;  
  }  
    
  let noLeak = createCircularRef();  
  noLeak = null; // Will be collected
* **Q7: Incorrect order of async operations in event loop**  
  console.log('Start');  
  console.log('End');  
  Promise.resolve().then(() => console.log('Promise 1'));  
  Promise.resolve().then(() => console.log('Promise 2'));  
  // In Node.js: process.nextTick > Promises > setImmediate > setTimeout  
  // In browser: Microtasks (Promises, queueMicrotask) > Macrotasks (setTimeout)  
  // Assuming browser context for setTimeout vs Promise:  
  setTimeout(() => console.log('Timeout'), 0);
* **Q8: Detached event listeners**  
  let button = document.createElement('button');  
  document.body.appendChild(button);  
  const handler = function() {  
   console.log('Clicked');  
  };  
  button.addEventListener('click', handler);  
  button.removeEventListener('click', handler); // Explicitly remove listener  
  button.remove();
* **Q9: Incorrect order of async operations in event loop**  
  console.log('sync');  
  const p = Promise.resolve();  
    
  p.then(() => console.log('then'));  
    
  (async () => {  
   await p;  
   console.log('after await');  
  })();
* **Q10: Global variables retaining references**  
  function createLargeObject() {  
   const tempObject = new Array(1000000).fill('data');  
   // Use tempObject for its purpose, then let it go out of scope  
   // Avoid assigning to global scope (window) if not strictly necessary  
  }  
  createLargeObject();  
  // If a global reference is needed, ensure it's explicitly nulled when done  
  // window.largeObject = null;
* **Q11: Incorrect order of async operations in event loop**  
  console.log('First');  
  console.log('Fourth');  
  Promise.resolve().then(() => console.log('Third'));  
  setTimeout(() => console.log('Second'), 0);
* **Q12: Retained closure in long-lived event handler**  
  function setupExpensiveTask() {  
   // If heavyData is not directly used in the handler or only needed temporarily  
   // Consider passing only necessary parts or creating it on demand.  
   const execBtn = document.getElementById('execBtn');  
   const handler = () => {  
   console.log('Executing task');  
   // heavyData = null; if it was local to handler and temporary  
   };  
   execBtn.addEventListener('click', handler);  
   // To prevent leak if element is removed from DOM:  
   // execBtn.removeEventListener('click', handler);  
  }
* **Q13: Incorrect order of async operations**  
  async function main() {  
   console.log(1);  
   await Promise.resolve();  
   console.log(2);  
  }  
  console.log(3);  
  Promise.resolve().then(() => console.log(4));  
  main();
* **Q14: Detached DOM nodes**  
  let container = document.getElementById('container');  
  for(let i=0; i<1000; i++) {  
   let div = document.createElement('div');  
   div.id = 'div-' + i;  
   container.appendChild(div);  
  }  
  // Clear references explicitly if needed, or manage component lifecycle  
  container.innerHTML = '';  
  // If individual 'div' references were stored in an array, clear that too.
* **Q15: Mixing synchronous and asynchronous array iteration**  
  async function processNumbers() {  
   const numbers = [1, 2, 3];  
   const promises = numbers.map(async (num) => {  
   await new Promise(res => setTimeout(res, 50));  
   return num \* 2;  
   });  
   const results = await Promise.all(promises);  
   console.log(results);  
  }  
  processNumbers();
* **Q16: Circular references in JavaScript objects**  
  function createObjectPair() {  
   const objA = {};  
   const objB = {};  
   objA.buddy = objB;  
   // objB.partner = objA; // Remove the direct circular reference  
   return objA;  
  }  
    
  let pair = createObjectPair();  
  pair = null;
* **Q17: Incorrect order of async operations**  
  console.log('C');  
  Promise.resolve().then(() => console.log('B'));  
  setTimeout(() => console.log('A'), 10);
* **Q18: Retained closure in long-lived event handler**  
  let cache = {};  
  const dataButton = document.getElementById('dataButton');  
  const handler = function() {  
   if (!cache.data) {  
   cache.data = Array(100000).fill(Math.random());  
   }  
   console.log('Data loaded');  
  };  
  dataButton.addEventListener('click', handler);  
  // When dataButton is no longer needed:  
  // dataButton.removeEventListener('click', handler);  
  // cache = {}; // If the cache itself should be cleared
* **Q19: Incorrect order of async operations in event loop**  
  console.log('Before async');  
  console.log('Inside async');  
  console.log('After calling async');  
  console.log('After await');
* **Q20: Detached DOM nodes**  
  let list = document.getElementById('myList');  
  const items = [];  
  for(let i=0; i<5; i++) {  
   let item = document.createElement('li');  
   item.textContent = `Item ${i}`;  
   list.appendChild(item);  
   items.push(item); // Store refs if needed  
  }  
    
  list.innerHTML = '';  
  // If references were stored and are no longer needed:  
  // items.length = 0; // Clear array

### **Java**

#### **Easy Difficulty**

* **Q1: Missing semicolon at the end of a statement**  
  public class MyClass {  
   public static void main(String[] args) {  
   System.out.println("Hello World");  
   }  
  }
* **Q2: Incorrect loop condition**  
  for (int i = 0; i < 5; i++) {  
   if (i == 4) { // Or change loop condition  
   System.out.println("Found 4");  
   }  
  }
* **Q3: Assigning incompatible types**  
  int number = 5;
* **Q4: Dereferencing a null object**  
  String str = "hello";  
  System.out.println(str.length());
* **Q5: Accessing array out of bounds**  
  int[] arr = {1, 2, 3};  
  System.out.println(arr[2]); // Access last element correctly
* **Q6: Missing curly brace**  
  public class Test {  
   public static void main(String[] args) {  
   System.out.println("Hello");  
   }  
  }
* **Q7: Incorrect boolean comparison**  
  boolean isValid = false;  
  if (isValid == true) { // Correct comparison  
   System.out.println("Valid");  
  }
* **Q8: Implicit conversion from float to int**  
  int x = (int) 5.5f; // Explicit cast  
  // Or: float x = 5.5f;
* **Q9: Dereferencing a null object**  
  List<String> names = new ArrayList<>();  
  System.out.println(names.size());
* **Q10: Accessing array out of bounds**  
  String[] colors = {"red", "green"};  
  System.out.println(colors[1]);
* **Q11: Missing import statement**  
  import java.util.ArrayList;  
    
  public class Main {  
   public static void main(String[] args) {  
   ArrayList<String> list = new ArrayList<>();  
   }  
  }
* **Q12: Integer division losing precision**  
  int a = 5;  
  int b = 2;  
  double result = (double) a / b;  
  System.out.println(result);
* **Q13: Using wrong wrapper class for primitive**  
  Integer i = 10;  
  Long l = Long.valueOf(i.longValue()); // Explicit conversion  
  // Or: long lPrim = i.longValue();
* **Q14: Dereferencing a null object**  
  Map<String, String> map = new HashMap<>();  
  map.put("key", "value");
* **Q15: Off-by-one error in loop**  
  int[] numbers = {10, 20, 30};  
  for (int i = 0; i < numbers.length; i++) {  
   System.out.println(numbers[i]);  
  }
* **Q16: Missing 'static' keyword for main method**  
  public class MyProgram {  
   public static void main(String[] args) {  
   System.out.println("Hello");  
   }  
  }
* **Q17: Incorrect string comparison**  
  String s1 = new String("hello");  
  String s2 = new String("hello");  
  if (s1.equals(s2)) {  
   System.out.println("Strings are equal");  
  }
* **Q18: Trying to cast incompatible types**  
  Object o = "string";  
  String s = (String) o;
* **Q19: Dereferencing a null object**  
  class Student {  
   String name = "John";  
   String getName() { return name; }  
  }  
  Student student = new Student();  
  System.out.println(student.getName());
* **Q20: Accessing array out of bounds**  
  int[] data = {5, 10};  
  System.out.println(data[1]);

#### **Medium Difficulty**

* **Q1: Race condition with non-atomic increment**  
  import java.util.concurrent.atomic.AtomicInteger;  
    
  class Counter {  
   public AtomicInteger count = new AtomicInteger(0);  
   public void increment() {  
   count.incrementAndGet();  
   }  
  }  
    
  // Or using synchronized:  
  // class Counter {  
  // public int count = 0;  
  // public synchronized void increment() {  
  // count++;  
  // }  
  // }
* **Q2: Static collection holding object references**  
  import java.util.ArrayList;  
  import java.util.List;  
    
  public class LeakExample {  
   // Ensure objects are removed from static lists when no longer needed.  
   // Or use WeakHashMap/WeakReference if appropriate for caching scenarios.  
   private static final List<Object> leakList = new ArrayList<>();  
    
   public void addAndRemoveObject(Object obj) {  
   leakList.add(obj);  
   // ... later when obj is no longer needed ...  
   leakList.remove(obj);  
   }  
  }
* **Q3: Deadlock with two locks**  
  public class DeadlockExample {  
   private Object lock1 = new Object();  
   private Object lock2 = new Object();  
    
   // Consistent locking order to prevent deadlock  
   public void method1() {  
   synchronized (lock1) {  
   System.out.println("Method 1 holding lock1");  
   try { Thread.sleep(10); } catch (InterruptedException e) {}  
   synchronized (lock2) {  
   System.out.println("Method 1 holding lock1 and lock2");  
   }  
   }  
   }  
    
   public void method2() {  
   synchronized (lock1) { // Same order as method1  
   System.out.println("Method 2 holding lock1");  
   try { Thread.sleep(10); } catch (InterruptedException e) {}  
   synchronized (lock2) {  
   System.out.println("Method 2 holding lock1 and lock2");  
   }  
   }  
   }  
  }
* **Q4: Not closing I/O stream**  
  import java.io.BufferedReader;  
  import java.io.FileReader;  
  import java.io.IOException;  
    
  public class ReadFile {  
   public void readFileContent(String fileName) throws IOException {  
   // Use try-with-resources to ensure resource closure  
   try (BufferedReader reader = new BufferedReader(new FileReader(fileName))) {  
   String line;  
   while ((line = reader.readLine()) != null) {  
   System.out.println(line);  
   }  
   }  
   }  
  }
* **Q5: Non-volatile field used in multithreading**  
  class Flag {  
   volatile boolean running = true; // Use volatile  
   public void stop() {  
   running = false;  
   }  
   public void runLoop() {  
   while(running) {  
   // do work  
   }  
   System.out.println("Loop stopped.");  
   }  
  }
* **Q6: Static collection holding object references**  
  import java.util.Map;  
  import java.util.WeakHashMap;  
    
  public class Cache {  
   // Use WeakHashMap if values should be GC'd when only referenced by the cache  
   private static final Map<String, Object> dataCache = new WeakHashMap<>();  
    
   public static void putData(String key, Object value) {  
   dataCache.put(key, value);  
   }  
    
   public static void removeData(String key) {  
   dataCache.remove(key);  
   }  
  }
* **Q7: Deadlock with two locks**  
  public class PhilosopherProblem {  
   Object fork1 = new Object();  
   Object fork2 = new Object();  
    
   // Standardize the order of acquiring locks  
   public void eatFirstThenSecond() {  
   synchronized (fork1) {  
   synchronized (fork2) {  
   // Eating  
   }  
   }  
   }  
    
   public void eatSecondThenFirst() {  
   synchronized (fork1) { // Always try for fork1 first  
   synchronized (fork2) { // Then fork2  
   // Eating  
   }  
   }  
   }  
  }
* **Q8: Not closing database connection**  
  import java.sql.\*;  
    
  public class DBUtils {  
   public ResultSet executeQuery(String query) throws SQLException {  
   Connection conn = null;  
   Statement stmt = null;  
   ResultSet rs = null;  
   try {  
   conn = DriverManager.getConnection("jdbc:url");  
   stmt = conn.createStatement();  
   rs = stmt.executeQuery(query);  
   return rs;  
   } finally {  
   // Close resources in reverse order of creation  
   if (rs != null) try { rs.close(); } catch (SQLException e) { /\* log error \*/ }  
   if (stmt != null) try { stmt.close(); } catch (SQLException e) { /\* log error \*/ }  
   if (conn != null) try { conn.close(); } catch (SQLException e) { /\* log error \*/ }  
   }  
   }  
  }
* **Q9: Non-volatile field used in multithreading**  
  class Stopper {  
   volatile boolean shouldStop = false;  
    
   public void stopNow() {  
   shouldStop = true;  
   }  
    
   public boolean isStopped() {  
   return shouldStop;  
   }  
    
   // Thread 1 calls stopNow(), Thread 2 calls isStopped() in a loop  
  }
* **Q10: Incorrect use of ThreadLocal leading to leaks**  
  import java.lang.ThreadLocal;  
    
  public class ThreadLocalLeak {  
   private static final ThreadLocal<LargeObject> threadLocalData = new ThreadLocal<>();  
    
   public void setAndUseData(LargeObject data) {  
   threadLocalData.set(data);  
   try {  
   // ... use data ...  
   } finally {  
   threadLocalData.remove(); // Always remove data when done  
   }  
   }  
  }
* **Q11: Race condition with non-atomic increment**  
  import java.util.concurrent.atomic.AtomicInteger;  
    
  class Counter {  
   public AtomicInteger count = new AtomicInteger(0);  
   public void increment() {  
   count.incrementAndGet();  
   }  
  }
* **Q12: Static collection holding object references**  
  import java.util.ArrayList;  
  import java.util.List;  
    
  public class LeakExample {  
   private static final List<Object> leakList = new ArrayList<>();  
    
   public void addAndRemoveObject(Object obj) {  
   leakList.add(obj);  
   // If not using WeakHashMap, ensure explicit removal:  
   // leakList.remove(obj);  
   }  
  }
* **Q13: Deadlock with two locks**  
  public class DeadlockExample {  
   private Object lockA = new Object();  
   private Object lockB = new Object();  
    
   // Standardize lock acquisition order for all tasks  
   public void performTask1() {  
   synchronized (lockA) {  
   synchronized (lockB) {  
   // ...  
   }  
   }  
   }  
    
   public void performTask2() {  
   synchronized (lockA) { // Acquire lockA first  
   synchronized (lockB) { // Then lockB  
   // ...  
   }  
   }  
   }  
  }
* **Q14: Not closing I/O stream**  
  import java.io.FileInputStream;  
  import java.io.IOException;  
    
  public class FileProcessor {  
   public void processFile(String path) throws IOException {  
   try (FileInputStream fis = new FileInputStream(path)) {  
   int data;  
   while ((data = fis.read()) != -1) {  
   // process data  
   }  
   }  
   }  
  }
* **Q15: Non-volatile field used in multithreading**  
  class StateHolder {  
   private volatile boolean state = false;  
    
   public void setState(boolean newState) {  
   this.state = newState;  
   }  
    
   public boolean getState() {  
   return this.state;  
   }  
  }
* **Q16: Static collection holding object references**  
  import java.util.LinkedList;  
  import java.util.Queue;  
    
  public class EventQueue {  
   private static final Queue<Runnable> eventQueue = new LinkedList<>();  
    
   public static void addEvent(Runnable event) {  
   eventQueue.add(event);  
   }  
    
   public static Runnable getNextEvent() {  
   return eventQueue.poll(); // Polls and removes  
   }  
  }
* **Q17: Deadlock with two locks**  
  public class ResourceUser {  
   private Object resA = new Object();  
   private Object resB = new Object();  
    
   public void useAthenB() {  
   synchronized (resA) {  
   System.out.println("Holding A");  
   synchronized (resB) {  
   System.out.println("Holding A and B");  
   }  
   }  
   }  
    
   public void useBthenA() {  
   synchronized (resA) { // Consistent order: always A then B  
   System.out.println("Holding A");  
   synchronized (resB) {  
   System.out.println("Holding A and B");  
   }  
   }  
   }  
  }
* **Q18: Not closing database connection**  
  import java.sql.\*;  
    
  public class ConnectionManager {  
   public void fetchData() throws SQLException {  
   try (Connection conn = DriverManager.getConnection("jdbc:example");  
   Statement stmt = conn.createStatement();  
   ResultSet rs = stmt.executeQuery("SELECT \* FROM users")) {  
   // Use rs  
   }  
   }  
  }
* **Q19: Non-volatile field used in multithreading**  
  class DataCache {  
   private volatile int cachedValue = 0;  
    
   public void setValue(int val) {  
   this.cachedValue = val;  
   }  
    
   public int getValue() {  
   return this.cachedValue;  
   }  
  }
* **Q20: Incorrect use of ThreadLocal leading to leaks**  
  import java.lang.ThreadLocal;  
    
  public class ThreadLocalUser {  
   private static final ThreadLocal<StringBuilder> threadLocalBuffer = new ThreadLocal<StringBuilder>();  
    
   public StringBuilder getBuffer() {  
   if (threadLocalBuffer.get() == null) {  
   threadLocalBuffer.set(new StringBuilder());  
   }  
   return threadLocalBuffer.get();  
   }  
    
   public void clearBuffer() {  
   threadLocalBuffer.remove(); // Essential to prevent leaks in thread pools  
   }  
  }

#### **Hard Difficulty**

* **Q1: Improper synchronization leading to data inconsistency**  
  class BankAccount {  
   private double balance = 0;  
    
   public synchronized void deposit(double amount) {  
   balance += amount;  
   }  
    
   public synchronized void withdraw(double amount) {  
   balance -= amount;  
   }  
    
   public double getBalance() {  
   return balance;  
   }  
  }
* **Q2: Unclosed resources in try-catch without finally**  
  import java.io.FileOutputStream;  
  import java.io.IOException;  
    
  public class FileWriterExample {  
   public void writeToFile(String filename, String data) {  
   try (FileOutputStream fos = new FileOutputStream(filename)) {  
   fos.write(data.getBytes());  
   } catch (IOException e) {  
   e.printStackTrace();  
   }  
   }  
  }
* **Q3: Incorrect use of wait() and notify() without loops**  
  class SharedResource {  
   private boolean produced = false;  
   public synchronized void consume() {  
   while (!produced) { // Use loop for wait condition  
   try { wait(); } catch (InterruptedException e) {}  
   }  
   System.out.println("Consumed");  
   produced = false;  
   }  
   public synchronized void produce() {  
   System.out.println("Produced");  
   produced = true;  
   notifyAll(); // Use notifyAll to wake up all waiting threads  
   }  
  }
* **Q4: Incorrect implementation of equals() and hashCode()**  
  import java.util.HashSet;  
  import java.util.Objects;  
  import java.util.Set;  
    
  class CustomKey {  
   int id;  
   String name;  
   public CustomKey(int id, String name) {  
   this.id = id;  
   this.name = name;  
   }  
    
   @Override  
   public boolean equals(Object o) {  
   if (this == o) return true;  
   if (o == null || getClass() != o.getClass()) return false;  
   CustomKey customKey = (CustomKey) o;  
   return id == customKey.id && Objects.equals(name, customKey.name);  
   }  
    
   @Override  
   public int hashCode() {  
   return Objects.hash(id, name);  
   }  
  }  
    
  public class MapLeak {  
   public static void main(String[] args) {  
   Set<CustomKey> set = new HashSet<>();  
   CustomKey k1 = new CustomKey(1, "test");  
   CustomKey k2 = new CustomKey(1, "test");  
   set.add(k1);  
   set.add(k2);  
   System.out.println(set.size());  
   }  
  }
* **Q5: Improper synchronization leading to data inconsistency**  
  class SharedData {  
   private volatile int data = 0; // Use volatile for visibility  
    
   // Or synchronize methods for atomic operations on complex objects:  
   public synchronized void write(int value) {  
   data = value;  
   }  
    
   public synchronized int read() {  
   return data;  
   }  
  }
* **Q6: Unclosed resources in try-catch without finally**  
  import java.io.PrintWriter;  
  import java.io.FileNotFoundException;  
    
  public class LogWriter {  
   public void logMessage(String msg) {  
   try (PrintWriter writer = new PrintWriter("log.txt")) {  
   writer.println(msg);  
   } catch (FileNotFoundException e) {  
   e.printStackTrace();  
   }  
   }  
  }
* **Q7: Incorrect use of wait() and notify() without loops**  
  class ProducerConsumer {  
   private int item = 0;  
   private boolean hasItem = false;  
    
   public synchronized void produce(int value) {  
   while (hasItem) { // Loop for robust waiting  
   try { wait(); } catch (InterruptedException e) {}  
   }  
   item = value;  
   hasItem = true;  
   notifyAll(); // Notify all potentially waiting consumers/producers  
   }  
    
   public synchronized int consume() {  
   while (!hasItem) { // Loop for robust waiting  
   try { wait(); } catch (InterruptedException e) {}  
   }  
   hasItem = false;  
   notifyAll();  
   return item;  
   }  
  }
* **Q8: Incorrect implementation of equals() and hashCode()**  
  import java.util.HashMap;  
  import java.util.Map;  
  import java.util.Objects;  
    
  class Coordinate {  
   int x, y;  
   public Coordinate(int x, int y) { this.x = x; this.y = y; }  
    
   @Override  
   public boolean equals(Object o) {  
   if (this == o) return true;  
   if (o == null || getClass() != o.getClass()) return false;  
   Coordinate that = (Coordinate) o;  
   return x == that.x && y == that.y;  
   }  
    
   @Override  
   public int hashCode() {  
   return Objects.hash(x, y);  
   }  
  }  
    
  public class HashBug {  
   public static void main(String[] args) {  
   Map<Coordinate, String> map = new HashMap<>();  
   map.put(new Coordinate(1, 1), "A");  
   System.out.println(map.get(new Coordinate(1, 1)));  
   }  
  }
* **Q9: Improper synchronization leading to data inconsistency**  
  class DataProcessor {  
   private StringBuilder log = new StringBuilder();  
    
   public synchronized void appendLog(String message) {  
   log.append(message);  
   }  
    
   public synchronized String getLog() {  
   return log.toString();  
   }  
  }
* **Q10: Unclosed resources in try-catch without finally**  
  import java.sql.\*;  
    
  public class DBConnector {  
   public void executeUpdate(String sql) {  
   try (Connection conn = DriverManager.getConnection("jdbc:url");  
   Statement stmt = conn.createStatement()) {  
   stmt.executeUpdate(sql);  
   } catch (SQLException e) {  
   e.printStackTrace();  
   }  
   }  
  }
* **Q11: Improper synchronization leading to data inconsistency**  
  class BankAccount {  
   private double balance = 0;  
    
   public synchronized void deposit(double amount) {  
   balance += amount;  
   }  
    
   public synchronized void withdraw(double amount) {  
   balance -= amount;  
   }  
    
   public double getBalance() {  
   return balance;  
   }  
  }
* **Q12: Unclosed resources in try-catch without finally**  
  import java.io.FileOutputStream;  
  import java.io.IOException;  
    
  public class FileWriterExample {  
   public void writeToFile(String filename, String data) {  
   try (FileOutputStream fos = new FileOutputStream(filename)) {  
   fos.write(data.getBytes());  
   } catch (IOException e) {  
   e.printStackTrace();  
   }  
   }  
  }
* **Q13: Incorrect use of wait() and notify() without loops**  
  class SharedResource {  
   private boolean produced = false;  
   public synchronized void consume() {  
   while (!produced) {  
   try { wait(); } catch (InterruptedException e) {}  
   }  
   System.out.println("Consumed");  
   produced = false;  
   }  
   public synchronized void produce() {  
   System.out.println("Produced");  
   produced = true;  
   notifyAll();  
   }  
  }
* **Q14: Incorrect implementation of equals() and hashCode()**  
  import java.util.HashSet;  
  import java.util.Objects;  
  import java.util.Set;  
    
  class CustomId {  
   int id;  
   public CustomId(int id) {  
   this.id = id;  
   }  
    
   @Override  
   public boolean equals(Object o) {  
   if (this == o) return true;  
   if (o == null || getClass() != o.getClass()) return false;  
   CustomId customId = (CustomId) o;  
   return id == customId.id;  
   }  
    
   @Override  
   public int hashCode() {  
   return Objects.hash(id);  
   }  
  }  
    
  public class SetBug {  
   public static void main(String[] args) {  
   Set<CustomId> set = new HashSet<>();  
   CustomId i1 = new CustomId(1);  
   CustomId i2 = new CustomId(1);  
   set.add(i1);  
   set.add(i2);  
   System.out.println(set.size());  
   }  
  }
* **Q15: Improper synchronization leading to data inconsistency**  
  import java.util.concurrent.atomic.AtomicIntegerArray;  
    
  class DataStore {  
   private AtomicIntegerArray data = new AtomicIntegerArray(new int[]{0, 0, 0});  
    
   public void updateData(int index, int value) {  
   data.set(index, value);  
   }  
    
   public int getData(int index) {  
   return data.get(index);  
   }  
  }
* **Q16: Unclosed resources in try-catch without finally**  
  import java.io.BufferedInputStream;  
  import java.io.FileInputStream;  
  import java.io.IOException;  
    
  public class DataReader {  
   public void readBytes(String filePath) throws IOException {  
   try (FileInputStream fis = new FileInputStream(filePath);  
   BufferedInputStream bis = new BufferedInputStream(fis)) {  
   int b;  
   while ((b = bis.read()) != -1) {  
   // read byte  
   }  
   }  
   }  
  }
* **Q17: Improper synchronization leading to data inconsistency**  
  class ValueHolder {  
   private volatile int value = 0;  
    
   public synchronized void increment() {  
   value++;  
   }  
    
   public int getValue() {  
   return value;  
   }  
  }
* **Q18: Incorrect implementation of equals() and hashCode()**  
  import java.util.HashMap;  
  import java.util.Map;  
  import java.util.Objects;  
    
  class Person {  
   String name;  
   int age;  
   public Person(String name, int age) { this.name = name; this.age = age; }  
    
   @Override  
   public boolean equals(Object o) {  
   if (this == o) return true;  
   if (o == null || getClass() != o.getClass()) return false;  
   Person person = (Person) o;  
   return age == person.age && Objects.equals(name, person.name);  
   }  
    
   @Override  
   public int hashCode() {  
   return Objects.hash(name, age);  
   }  
  }  
    
  public class PersonMap {  
   public static void main(String[] args) {  
   Map<Person, String> users = new HashMap<>();  
   users.put(new Person("Alice", 30), "user1");  
   System.out.println(users.get(new Person("Alice", 30)));  
   }  
  }
* **Q19: Incorrect use of wait() and notify() without loops**  
  class MessageQueue {  
   private String message = null;  
    
   public synchronized void put(String msg) {  
   while (message != null) {  
   try { wait(); } catch (InterruptedException e) {}  
   }  
   message = msg;  
   notifyAll();  
   }  
    
   public synchronized String take() {  
   while (message == null) {  
   try { wait(); } catch (InterruptedException e) {}  
   }  
   String temp = message;  
   message = null;  
   notifyAll();  
   return temp;  
   }  
  }
* **Q20: Unclosed resources in try-catch without finally**  
  import java.io.InputStream;  
  import java.io.IOException;  
  import java.net.URL;  
    
  public class URLReader {  
   public void readURL(String urlString) {  
   try (InputStream is = new URL(urlString).openStream()) {  
   int data;  
   while ((data = is.read()) != -1) {  
   // process data  
   }  
   } catch (IOException e) {  
   e.printStackTrace();  
   }  
   }  
  }

### **DotNet**

#### **Easy Difficulty**

* **Q1: Missing semicolon at end of statement**  
  using System;  
  public class Program  
  {  
   public static void Main(string[] args)  
   {  
   Console.WriteLine("Hello DotNet");  
   }  
  }
* **Q2: Incorrect comparison operator**  
  int age = 17;  
  if (age >= 18) {  
   Console.WriteLine("Eligible to vote");  
  }
* **Q3: Implicit conversion failure**  
  int number = 123; // Or: int number = int.Parse("123");
* **Q4: Dereferencing a null object**  
  string name = "John Doe";  
  Console.WriteLine(name.Length);
* **Q5: Accessing array out of bounds**  
  int[] numbers = {1, 2, 3};  
  Console.WriteLine(numbers[2]);
* **Q6: Missing curly brace**  
  public class MyClass  
  {  
   public void MyMethod()  
   {  
   Console.WriteLine("Hello");  
   }  
  }
* **Q7: Incorrect boolean assignment**  
  bool isActive = false;  
  if (isActive == true) {  
   Console.WriteLine("Active");  
  }
* **Q8: Attempting to assign float to int without cast**  
  int value = (int)10.5f;
* **Q9: Dereferencing a null object**  
  List<string> items = new List<string>();  
  Console.WriteLine(items.Count);
* **Q10: Accessing list out of bounds**  
  List<int> data = new List<int> {10, 20};  
  Console.WriteLine(data[1]);
* **Q11: Missing namespace reference**  
  using System.Collections.Generic;  
    
  public class Example {  
   public void Method() {  
   var list = new List<int>();  
   }  
  }
* **Q12: Integer division losing precision**  
  int numerator = 10;  
  int denominator = 3;  
  double result = (double)numerator / denominator;  
  Console.WriteLine(result);
* **Q13: Attempting to cast unrelated types**  
  object obj = 123;  
  int val = (int)obj;  
  // Or: string s = (string)obj;
* **Q14: Dereferencing a null object**  
  Dictionary<string, int> dict = new Dictionary<string, int>();  
  dict.Add("one", 1);
* **Q15: Off-by-one error in loop**  
  int[] values = {5, 6, 7};  
  for (int i = 0; i < values.Length; i++) {  
   Console.WriteLine(values[i]);  
  }
* **Q16: Missing return type in method**  
  public class Calc {  
   public int Sum(int a, int b) {  
   return a + b;  
   }  
  }
* **Q17: Incorrect string comparison**  
  string s1 = "hello";  
  string s2 = "Hello";  
  if (s1.Equals(s2, StringComparison.OrdinalIgnoreCase)) {  
   Console.WriteLine("Strings are equal");  
  }
* **Q18: Boxing/unboxing issue with value types**  
  object obj = 100;  
  short s = (short)(int)obj;
* **Q19: Dereferencing a null object**  
  public class MyClass { public void DoSomething() { Console.WriteLine("Done"); } }  
  MyClass instance = new MyClass();  
  instance.DoSomething();
* **Q20: Accessing list out of bounds**  
  List<string> names = new List<string> { "Alice" };  
  Console.WriteLine(names[0]);

#### **Medium Difficulty**

* **Q1: Race condition with non-atomic increment**  
  using System;  
  using System.Threading;  
    
  public class Counter  
  {  
   private int \_count = 0;  
   public void Increment()  
   {  
   Interlocked.Increment(ref \_count); // Use Interlocked for atomic operations  
   }  
   public int GetCount() { return \_count; }  
  }  
    
  // Or using lock:  
  // public class Counter {  
  // private int \_count = 0;  
  // private readonly object \_lock = new object();  
  // public void Increment() {  
  // lock (\_lock) {  
  // \_count++;  
  // }  
  // }  
  // public int GetCount() { lock(\_lock) { return \_count; } }  
  // }
* **Q2: Static event handler preventing object garbage collection**  
  using System;  
  using System.Collections.Generic;  
    
  public class EventSource  
  {  
   public static event Action GlobalEvent;  
   public static void FireEvent() { GlobalEvent?.Invoke(); }  
  }  
    
  public class Listener : IDisposable  
  {  
   public string Name { get; set; }  
   public Listener(string name)  
   {  
   Name = name;  
   EventSource.GlobalEvent += HandleEvent;  
   }  
    
   private void HandleEvent()  
   {  
   Console.WriteLine($"{Name} handled event");  
   }  
    
   public void Dispose()  
   {  
   EventSource.GlobalEvent -= HandleEvent; // Unsubscribe to prevent leak  
   GC.SuppressFinalize(this);  
   }  
  }  
  // Usage:  
  // using (var listener = new Listener("MyListener")) {  
  // // ...  
  // }  
  // Or manually call listener.Dispose();
* **Q3: Deadlock with multiple locks**  
  using System;  
  using System.Threading;  
    
  public class Account  
  {  
   public int Id { get; set; }  
   public double Balance { get; set; }  
   private object \_lock = new object();  
    
   public void Transfer(Account destination, double amount)  
   {  
   // Ensure consistent lock ordering (e.g., by ID to prevent deadlocks)  
   object lock1 = this.Id < destination.Id ? this.\_lock : destination.\_lock;  
   object lock2 = this.Id < destination.Id ? destination.\_lock : this.\_lock;  
    
   lock (lock1)  
   {  
   Thread.Sleep(10);  
   lock (lock2)  
   {  
   if (Balance >= amount) {  
   Balance -= amount;  
   destination.Balance += amount;  
   }  
   }  
   }  
   }  
  }  
  // Simulating two accounts trying to transfer to each other simultaneously
* **Q4: Not disposing IDisposable objects**  
  using System;  
  using System.IO;  
    
  public class FileProcessor  
  {  
   public void ReadAndProcess(string path)  
   {  
   using (StreamReader reader = new StreamReader(path)) // Use 'using' statement  
   {  
   string line = reader.ReadLine();  
   Console.WriteLine(line);  
   }  
   }  
  }
* **Q5: Non-volatile field leading to stale data**  
  using System;  
  using System.Threading;  
    
  public class StatusFlag  
  {  
   private volatile bool \_stop = false; // Use volatile keyword  
    
   public void SetStop()  
   {  
   \_stop = true;  
   }  
    
   public bool IsStopped()  
   {  
   return \_stop;  
   }  
    
   public void RunLoop()  
   {  
   while (!\_stop)  
   {  
   // Perform work  
   }  
   Console.WriteLine("Loop exited.");  
   }  
  }
* **Q6: Static event handler preventing object garbage collection**  
  using System;  
    
  public class GlobalPublisher  
  {  
   public static event Action OnDataReady;  
   public static void PublishData() { OnDataReady?.Invoke(); }  
  }  
    
  public class DataConsumer : IDisposable  
  {  
   public DataConsumer()  
   {  
   GlobalPublisher.OnDataReady += HandleData; // Leak: no unsubscribe  
   }  
   private void HandleData() { Console.WriteLine("Data received"); }  
   public void Dispose()  
   {  
   GlobalPublisher.OnDataReady -= HandleData; // Unsubscribe  
   GC.SuppressFinalize(this);  
   }  
  }
* **Q7: Deadlock with multiple locks**  
  using System.Threading;  
    
  public class DiningPhilosophers  
  {  
   private object \_fork1 = new object();  
   private object \_fork2 = new object();  
    
   public void EatWith1Then2()  
   {  
   lock (\_fork1)  
   {  
   Thread.Sleep(10);  
   lock (\_fork2) { /\* Eat \*/ }  
   }  
   }  
    
   public void EatWith2Then1()  
   {  
   lock (\_fork1) // Always acquire in the same order  
   {  
   Thread.Sleep(10);  
   lock (\_fork2) { /\* Eat \*/ }  
   }  
   }  
  }
* **Q8: Not disposing IDisposable objects**  
  using System.Data.SqlClient;  
    
  public class DataAccess  
  {  
   public void GetData(string connString, string query)  
   {  
   using (SqlConnection conn = new SqlConnection(connString))  
   {  
   conn.Open();  
   using (SqlCommand cmd = new SqlCommand(query, conn))  
   {  
   using (SqlDataReader reader = cmd.ExecuteReader())  
   {  
   // Use reader  
   }  
   }  
   }  
   }  
  }
* **Q9: Non-volatile field leading to stale data**  
  using System;  
  using System.Threading;  
    
  public class SharedCounter  
  {  
   private volatile int \_value = 0; // Ensure visibility with volatile  
    
   public void SetValue(int val)  
   {  
   \_value = val;  
   }  
    
   public int GetValue()  
   {  
   return \_value;  
   }  
  }
* **Q10: Improperly used WeakReference for caching**  
  using System;  
  using System.Collections.Generic;  
    
  public class ObjectCache  
  {  
   private static Dictionary<string, WeakReference<object>> cache = new Dictionary<string, WeakReference<object>>();  
    
   public static void Add(string key, object obj)  
   {  
   cache[key] = new WeakReference<object>(obj);  
   }  
    
   public static object Get(string key)  
   {  
   if (cache.TryGetValue(key, out WeakReference<object> weakRef))  
   {  
   return weakRef.TryGetTarget(out object target) ? target : null;  
   }  
   return null;  
   }  
    
   public static void CleanCache()  
   {  
   // Periodically clean up collected entries  
   var keysToRemove = new List<string>();  
   foreach (var entry in cache)  
   {  
   if (!entry.Value.TryGetTarget(out \_))  
   {  
   keysToRemove.Add(entry.Key);  
   }  
   }  
   foreach (var key in keysToRemove)  
   {  
   cache.Remove(key);  
   }  
   }  
  }
* **Q11: Race condition with non-atomic increment**  
  using System;  
  using System.Threading;  
    
  public class Counter  
  {  
   private int \_value = 0;  
   public void Add()  
   {  
   Interlocked.Increment(ref \_value);  
   }  
   public int GetValue() { return \_value; }  
  }
* **Q12: Static event handler preventing object garbage collection**  
  using System;  
    
  public class GlobalNotifier  
  {  
   public static event Action NotifyAll;  
   public static void Trigger() { NotifyAll?.Invoke(); }  
  }  
    
  public class Subscriber : IDisposable  
  {  
   public Subscriber() { GlobalNotifier.NotifyAll += OnNotify; }  
   private void OnNotify() { Console.WriteLine("Notified"); }  
   public void Dispose()  
   {  
   GlobalNotifier.NotifyAll -= OnNotify;  
   GC.SuppressFinalize(this);  
   }  
  }
* **Q13: Deadlock with multiple locks**  
  using System.Threading;  
    
  public class BankTransfer  
  {  
   public void TransferFunds(Account source, Account dest, decimal amount)  
   {  
   object lock1 = source.Id < dest.Id ? source.SyncRoot : dest.SyncRoot; // Assuming SyncRoot property on Account  
   object lock2 = source.Id < dest.Id ? dest.SyncRoot : source.SyncRoot;  
    
   lock (lock1)  
   {  
   lock (lock2)  
   {  
   // Perform transfer logic  
   }  
   }  
   }  
  }  
  // Requires Account class to have a SyncRoot or unique lock object per instance
* **Q14: Not disposing IDisposable objects**  
  using System.Net.Http;  
    
  public class WebClient {  
   public async Task<string> FetchData(string url) {  
   using (HttpClient client = new HttpClient()) {  
   string data = await client.GetStringAsync(url);  
   return data;  
   }  
   }  
  }
* **Q15: Non-volatile field leading to stale data**  
  using System;  
  using System.Threading;  
    
  public class SharedFlag  
  {  
   private volatile bool \_flag = false;  
    
   public void SetFlag()  
   {  
   \_flag = true;  
   }  
    
   public bool GetFlag()  
   {  
   return \_flag;  
   }  
  }
* **Q16: Static event handler preventing object garbage collection**  
  using System;  
    
  public class GlobalMonitor  
  {  
   public static event Action OnUpdate;  
   public static void TriggerUpdate() { OnUpdate?.Invoke(); }  
  }  
    
  public class ReportingService : IDisposable  
  {  
   public ReportingService() { GlobalMonitor.OnUpdate += GenerateReport; }  
   private void GenerateReport() { Console.WriteLine("Report generated"); }  
   public void Dispose()  
   {  
   GlobalMonitor.OnUpdate -= GenerateReport;  
   GC.SuppressFinalize(this);  
   }  
  }
* **Q17: Deadlock with multiple locks**  
  using System.Threading;  
    
  public class ResourceLocker  
  {  
   private readonly object \_resource1 = new object();  
   private readonly object \_resource2 = new object();  
    
   public void Acquire1then2()  
   {  
   lock (\_resource1) { lock (\_resource2) { /\* Use resources \*/ } }  
   }  
    
   public void Acquire2then1()  
   {  
   lock (\_resource1) { lock (\_resource2) { /\* Use resources \*/ } }  
   }  
  }
* **Q18: Not disposing IDisposable objects**  
  using System.Net.Sockets;  
    
  public class SocketClient {  
   public void ConnectAndSend(string host, int port, string message) {  
   using (Socket socket = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp)) {  
   socket.Connect(host, port);  
   byte[] data = System.Text.Encoding.ASCII.GetBytes(message);  
   socket.Send(data);  
   }  
   }  
  }
* **Q19: Non-volatile field leading to stale data**  
  using System;  
  using System.Threading;  
    
  public class CacheUpdater  
  {  
   private volatile bool \_dataReady = false;  
    
   public void MarkDataReady()  
   {  
   \_dataReady = true;  
   }  
    
   public bool IsDataReady()  
   {  
   return \_dataReady;  
   }  
  }
* **Q20: Improperly used WeakReference for caching**  
  using System;  
  using System.Collections.Concurrent;  
  using System.Collections.Generic;  
    
  public class SharedCache  
  {  
   private static ConcurrentDictionary<string, WeakReference<object>> \_cache = new ConcurrentDictionary<string, WeakReference<object>>();  
    
   public static void AddItem(string key, object item)  
   {  
   \_cache[key] = new WeakReference<object>(item);  
   }  
    
   public static object GetItem(string key)  
   {  
   if (\_cache.TryGetValue(key, out var weakRef))  
   {  
   if (weakRef.TryGetTarget(out var target))  
   {  
   return target;  
   }  
   else  
   {  
   // Remove the entry if the target has been collected  
   \_cache.TryRemove(key, out \_);  
   }  
   }  
   return null;  
   }  
    
   public static void CleanUpCollectedEntries()  
   {  
   foreach (var entry in \_cache)  
   {  
   if (!entry.Value.TryGetTarget(out \_))  
   {  
   \_cache.TryRemove(entry.Key, out \_);  
   }  
   }  
   }  
  }

#### **Hard Difficulty**

* **Q1: Incorrect use of Monitor.Wait and Monitor.Pulse**  
  using System;  
  using System.Threading;  
  using System.Collections.Generic;  
    
  public class ProducerConsumer  
  {  
   private Queue<int> \_queue = new Queue<int>();  
   private readonly int \_capacity = 5;  
   private readonly object \_lock = new object();  
    
   public void Produce(int item)  
   {  
   lock (\_lock)  
   {  
   while (\_queue.Count == \_capacity) // Use while loop  
   {  
   Monitor.Wait(\_lock);  
   }  
   \_queue.Enqueue(item);  
   Console.WriteLine($"Produced: {item}");  
   Monitor.PulseAll(\_lock); // Use PulseAll  
   }  
   }  
    
   public int Consume()  
   {  
   lock (\_lock)  
   {  
   while (\_queue.Count == 0) // Use while loop  
   {  
   Monitor.Wait(\_lock);  
   }  
   int item = \_queue.Dequeue();  
   Console.WriteLine($"Consumed: {item}");  
   Monitor.PulseAll(\_lock); // Use PulseAll  
   return item;  
   }  
   }  
  }
* **Q2: Unmanaged resources not properly released (COM objects, handles)**  
  using System;  
  using System.Runtime.InteropServices;  
    
  public class MyComWrapper : IDisposable  
  {  
   private IntPtr \_comObjectPointer;  
   private bool disposed = false;  
    
   public MyComWrapper(IntPtr comPtr)  
   {  
   \_comObjectPointer = comPtr;  
   }  
    
   public void Dispose()  
   {  
   Dispose(true);  
   GC.SuppressFinalize(this);  
   }  
    
   protected virtual void Dispose(bool disposing)  
   {  
   if (disposed) return;  
    
   if (disposing)  
   {  
   // Dispose managed resources here  
   }  
    
   // Release unmanaged resources  
   if (\_comObjectPointer != IntPtr.Zero)  
   {  
   // Marshal.Release(\_comObjectPointer); // Example for COM object  
   \_comObjectPointer = IntPtr.Zero;  
   }  
   disposed = true;  
   }  
    
   ~MyComWrapper() // Finalizer for unmanaged resources  
   {  
   Dispose(false);  
   }  
  }
* **Q3: Incorrect use of ReaderWriterLockSlim**  
  using System.Threading;  
    
  public class DataStore  
  {  
   private ReaderWriterLockSlim \_lock = new ReaderWriterLockSlim();  
   private int \_data = 0;  
    
   public int ReadData()  
   {  
   \_lock.EnterReadLock();  
   try  
   {  
   return \_data;  
   }  
   finally  
   {  
   \_lock.ExitReadLock();  
   }  
   }  
    
   public void WriteData(int value)  
   {  
   \_lock.EnterWriteLock();  
   try  
   {  
   \_data = value;  
   }  
   finally  
   {  
   \_lock.ExitWriteLock(); // Ensure release even on exception  
   }  
   }  
  }
* **Q4: Capturing this in async lambdas without ConfigureAwait(false)**  
  using System;  
  using System.Threading.Tasks;  
    
  public class MyService  
  {  
   private object \_data;  
    
   public MyService() {  
   \_data = new object();  
   }  
    
   public async Task DoWorkAsync()  
   {  
   // Use ConfigureAwait(false) to avoid capturing the current SynchronizationContext  
   // This helps prevent deadlocks and can sometimes help with GC if context isn't needed.  
   await Task.Run(() => {  
   // Do some work  
   }).ConfigureAwait(false);  
    
   // If \_data is not needed after the await point, consider structuring differently  
   Console.WriteLine("Work done for " + \_data.GetHashCode());  
   }  
  }
* **Q5: Incorrect use of CancellationTokenSource and CancellationToken**  
  using System;  
  using System.Threading;  
  using System.Threading.Tasks;  
    
  public class Worker  
  {  
   public async Task DoWork(CancellationToken token)  
   {  
   try  
   {  
   for (int i = 0; i < 100; i++)  
   {  
   token.ThrowIfCancellationRequested(); // Check for cancellation regularly  
   Thread.Sleep(100); // Simulate work  
   }  
   }  
   catch (OperationCanceledException)  
   {  
   Console.WriteLine("Work cancelled!");  
   }  
   }  
  }
* **Q6: Unmanaged resources not properly released (COM objects, handles)**  
  using System;  
  using System.Runtime.InteropServices;  
    
  public class NativeResourceWrapper : IDisposable  
  {  
   private IntPtr \_nativeHandle; // Represents a handle to an unmanaged resource  
   private bool \_disposed = false;  
    
   public NativeResourceWrapper(IntPtr handle)  
   {  
   \_nativeHandle = handle;  
   }  
    
   public void Dispose()  
   {  
   Dispose(true);  
   GC.SuppressFinalize(this);  
   }  
    
   protected virtual void Dispose(bool disposing)  
   {  
   if (\_disposed) return;  
    
   if (disposing)  
   {  
   // Dispose managed resources here  
   }  
    
   if (\_nativeHandle != IntPtr.Zero)  
   {  
   // Release the unmanaged handle (e.g., CloseHandle, FreeLibrary, etc.)  
   // Example: CloseHandle(\_nativeHandle);  
   \_nativeHandle = IntPtr.Zero;  
   }  
   \_disposed = true;  
   }  
    
   ~NativeResourceWrapper()  
   {  
   Dispose(false);  
   }  
  }
* **Q7: Incorrect use of Monitor.Wait and Monitor.Pulse**  
  using System.Threading;  
  using System.Collections.Generic;  
    
  public class BoundedBuffer  
  {  
   private Queue<string> \_buffer = new Queue<string>();  
   private readonly int \_maxSize = 10;  
   private readonly object \_lock = new object();  
    
   public void AddItem(string item)  
   {  
   lock (\_lock)  
   {  
   while (\_buffer.Count == \_maxSize) Monitor.Wait(\_lock);  
   \_buffer.Enqueue(item);  
   Monitor.PulseAll(\_lock); // Wake up all waiting consumers  
   }  
   }  
    
   public string GetItem()  
   {  
   lock (\_lock)  
   {  
   while (\_buffer.Count == 0) Monitor.Wait(\_lock);  
   string item = \_buffer.Dequeue();  
   Monitor.PulseAll(\_lock); // Wake up all waiting producers  
   return item;  
   }  
   }  
  }
* **Q8: Capturing this in async lambdas without ConfigureAwait(false)**  
  using System;  
  using System.Threading.Tasks;  
  using System.Collections.Generic;  
    
  public class DataProcessor  
  {  
   private List<string> \_dataList = new List<string>();  
    
   public async Task ProcessLongRunningAsync()  
   {  
   // Use ConfigureAwait(false) if the continuation does not need to resume on the original context.  
   await Task.Delay(100).ConfigureAwait(false);  
   // If \_dataList is needed after the await, `this` is still implicitly captured.  
   // If not, consider passing data explicitly or breaking object lifecycle.  
   await Task.Run(() => {  
   lock(\_dataList) { \_dataList.Add("processed"); } // Also consider thread-safety for \_dataList  
   }).ConfigureAwait(false);  
   }  
  }
* **Q9: Incorrect use of ReaderWriterLockSlim**  
  using System.Threading;  
  using System.Collections.Generic;  
    
  public class ConcurrentCache  
  {  
   private ReaderWriterLockSlim \_cacheLock = new ReaderWriterLockSlim();  
   private Dictionary<string, object> \_data = new Dictionary<string, object>();  
    
   public object Get(string key)  
   {  
   \_cacheLock.EnterReadLock();  
   try  
   {  
   return \_data[key];  
   }  
   finally  
   {  
   \_cacheLock.ExitReadLock();  
   }  
   }  
    
   public void Set(string key, object value)  
   {  
   \_cacheLock.EnterWriteLock();  
   try  
   {  
   \_data[key] = value;  
   }  
   finally  
   {  
   \_cacheLock.ExitWriteLock();  
   }  
   }  
  }
* **Q10: Unmanaged resources not properly released (COM objects, handles)**  
  using System;  
  using System.Runtime.InteropServices;  
    
  public class ExternalDependencyWrapper : IDisposable  
  {  
   private IntPtr \_handle; // Represents a handle to a native DLL resource  
   private bool \_disposed = false;  
    
   public ExternalDependencyWrapper(IntPtr h)  
   {  
   \_handle = h;  
   }  
    
   public void Dispose()  
   {  
   Dispose(true);  
   GC.SuppressFinalize(this);  
   }  
    
   protected virtual void Dispose(bool disposing)  
   {  
   if (\_disposed) return;  
    
   if (disposing)  
   {  
   // Dispose managed state (managed objects).  
   }  
    
   // Free unmanaged resources (unmanaged objects).  
   if (\_handle != IntPtr.Zero)  
   {  
   // Call native function to release handle  
   // e.g., NativeMethods.CloseHandle(\_handle);  
   \_handle = IntPtr.Zero;  
   }  
   \_disposed = true;  
   }  
    
   ~ExternalDependencyWrapper()  
   {  
   Dispose(false);  
   }  
  }
* **Q11: Incorrect use of Monitor.Wait and Monitor.Pulse**  
  using System;  
  using System.Threading;  
  using System.Collections.Generic;  
    
  public class ProducerConsumer  
  {  
   private Queue<int> \_queue = new Queue<int>();  
   private readonly int \_capacity = 5;  
   private readonly object \_lock = new object();  
    
   public void Produce(int item)  
   {  
   lock (\_lock)  
   {  
   while (\_queue.Count == \_capacity)  
   {  
   Monitor.Wait(\_lock);  
   }  
   \_queue.Enqueue(item);  
   Console.WriteLine($"Produced: {item}");  
   Monitor.PulseAll(\_lock);  
   }  
   }  
    
   public int Consume()  
   {  
   lock (\_lock)  
   {  
   while (\_queue.Count == 0)  
   {  
   Monitor.Wait(\_lock);  
   }  
   int item = \_queue.Dequeue();  
   Console.WriteLine($"Consumed: {item}");  
   Monitor.PulseAll(\_lock);  
   return item;  
   }  
   }  
  }
* **Q12: Unmanaged resources not properly released (COM objects, handles)**  
  using System;  
  using System.Runtime.InteropServices;  
    
  public class MyComWrapper : IDisposable  
  {  
   private IntPtr \_comObjectPointer;  
   private bool disposed = false;  
    
   public MyComWrapper(IntPtr comPtr)  
   {  
   \_comObjectPointer = comPtr;  
   }  
    
   public void Dispose()  
   {  
   Dispose(true);  
   GC.SuppressFinalize(this);  
   }  
    
   protected virtual void Dispose(bool disposing)  
   {  
   if (disposed) return;  
    
   if (disposing)  
   {  
   }  
    
   if (\_comObjectPointer != IntPtr.Zero)  
   {  
   \_comObjectPointer = IntPtr.Zero;  
   }  
   disposed = true;  
   }  
    
   ~MyComWrapper()  
   {  
   Dispose(false);  
   }  
  }
* **Q13: Incorrect use of ReaderWriterLockSlim**  
  using System.Threading;  
    
  public class DataStore  
  {  
   private ReaderWriterLockSlim \_lock = new ReaderWriterLockSlim();  
   private int \_data = 0;  
    
   public int ReadData()  
   {  
   \_lock.EnterReadLock();  
   try  
   {  
   return \_data;  
   }  
   finally  
   {  
   \_lock.ExitReadLock();  
   }  
   }  
    
   public void WriteData(int value)  
   {  
   \_lock.EnterWriteLock();  
   try  
   {  
   \_data = value;  
   }  
   finally  
   {  
   \_lock.ExitWriteLock();  
   }  
   }  
  }
* **Q14: Capturing this in async lambdas without ConfigureAwait(false)**  
  using System;  
  using System.Threading.Tasks;  
    
  public class MyService  
  {  
   private object \_data;  
    
   public MyService() { \_data = new object(); }  
    
   public async Task DoWorkAsync()  
   {  
   await Task.Run(() => { }).ConfigureAwait(false);  
   Console.WriteLine("Work done for " + \_data.GetHashCode());  
   }  
  }
* **Q15: Incorrect use of CancellationTokenSource and CancellationToken**  
  using System;  
  using System.Threading;  
  using System.Threading.Tasks;  
    
  public class Worker  
  {  
   public async Task DoWork(CancellationToken token)  
   {  
   try  
   {  
   for (int i = 0; i < 100; i++)  
   {  
   token.ThrowIfCancellationRequested();  
   Thread.Sleep(100);  
   }  
   }  
   catch (OperationCanceledException)  
   {  
   Console.WriteLine("Work cancelled!");  
   }  
   }  
  }
* **Q16: Unmanaged resources not properly released (COM objects, handles)**  
  using System;  
  using System.Runtime.InteropServices;  
    
  public class NativeResourceWrapper : IDisposable  
  {  
   private IntPtr \_nativeHandle;  
   private bool \_disposed = false;  
    
   public NativeResourceWrapper(IntPtr handle)  
   {  
   \_nativeHandle = handle;  
   }  
    
   public void Dispose()  
   {  
   Dispose(true);  
   GC.SuppressFinalize(this);  
   }  
    
   protected virtual void Dispose(bool disposing)  
   {  
   if (\_disposed) return;  
    
   if (disposing)  
   {  
   }  
    
   if (\_nativeHandle != IntPtr.Zero)  
   {  
   \_nativeHandle = IntPtr.Zero;  
   }  
   \_disposed = true;  
   }  
    
   ~NativeResourceWrapper()  
   {  
   Dispose(false);  
   }  
  }
* **Q17: Incorrect use of Monitor.Wait and Monitor.Pulse**  
  using System.Threading;  
  using System.Collections.Generic;  
    
  public class BoundedBuffer  
  {  
   private Queue<string> \_buffer = new Queue<string>();  
   private readonly int \_maxSize = 10;  
   private readonly object \_lock = new object();  
    
   public void AddItem(string item)  
   {  
   lock (\_lock)  
   {  
   while (\_buffer.Count == \_maxSize) Monitor.Wait(\_lock);  
   \_buffer.Enqueue(item);  
   Monitor.PulseAll(\_lock);  
   }  
   }  
    
   public string GetItem()  
   {  
   lock (\_lock)  
   {  
   while (\_buffer.Count == 0) Monitor.Wait(\_lock);  
   string item = \_buffer.Dequeue();  
   Monitor.PulseAll(\_lock);  
   return item;  
   }  
   }  
  }
* **Q18: Capturing this in async lambdas without ConfigureAwait(false)**  
  using System;  
  using System.Threading.Tasks;  
  using System.Collections.Generic;  
    
  public class DataProcessor  
  {  
   private List<string> \_dataList = new List<string>();  
    
   public async Task ProcessLongRunningAsync()  
   {  
   await Task.Delay(100).ConfigureAwait(false);  
   await Task.Run(() => {  
   lock(\_dataList) { \_dataList.Add("processed"); }  
   }).ConfigureAwait(false);  
   }  
  }
* **Q19: Incorrect use of ReaderWriterLockSlim**  
  using System.Threading;  
  using System.Collections.Generic;  
    
  public class ConcurrentCache  
  {  
   private ReaderWriterLockSlim \_cacheLock = new ReaderWriterLockSlim();  
   private Dictionary<string, object> \_data = new Dictionary<string, object>();  
    
   public object Get(string key)  
   {  
   \_cacheLock.EnterReadLock();  
   try  
   {  
   return \_data[key];  
   }  
   finally  
   {  
   \_cacheLock.ExitReadLock();  
   }  
   }  
    
   public void Set(string key, object value)  
   {  
   \_cacheLock.EnterWriteLock();  
   try  
   {  
   \_data[key] = value;  
   }  
   finally  
   {  
   \_cacheLock.ExitWriteLock();  
   }  
   }  
  }