**Use consistent formatting:**

Consistent formatting makes the code easier to read and understand. This includes consistent indentation, spacing, and braces. For example:

// Bad Example  
public void PrintName(string name)  
{  
Console.WriteLine("Name: " + name);  
}  
  
// Good Example  
public void PrintName(string name)  
{  
 Console.WriteLine($"Name: {name}");  
}

**Use inheritance and polymorphism:**

Inheritance and polymorphism can make the code more flexible and easier to modify. For example:

// Bad Example  
public class Animal  
{  
 public void Walk()  
 {  
 // walk  
 }  
}  
  
public class Dog  
{  
 public void Walk()  
 {  
 // walk  
 }  
  
 public void Bark()  
 {  
 // bark  
 }  
}  
  
// Good Example  
public abstract class Animal  
{  
 public abstract void Walk();  
}  
  
public class Dog : Animal  
{  
 public override void Walk()  
 {  
 // walk  
 }  
  
 public void Bark()  
 {  
 // bark  
 }  
}

**Use descriptive error messages:**

Error messages should be clear and descriptive, to help developers understand the problem and how to fix it. For example:

// Bad Example  
public void CalculateTotal(decimal price, int quantity)  
{  
 if (price < 0 || quantity < 0)  
 {  
 throw new Exception("Invalid input");  
 }  
 // calculate total  
}  
  
// Good Example  
public void CalculateTotal(decimal price, int quantity)  
{  
 if (price < 0)  
 {  
 throw new ArgumentException("Price cannot be negative", nameof(price));  
 }  
 if (quantity < 0)  
 {  
 throw new ArgumentException("Quantity cannot be negative", nameof(quantity));  
 }  
 // calculate total  
}

// Bad Example  
public int Add(int firstNumber, int secondNumber)  
{  
 return firstNumber + secondNumber;  
}  
  
// Good Example  
public int Add(int firstNumber, int secondNumber)  
{  
 if (firstNumber < 0 || secondNumber < 0)  
 {  
 throw new ArgumentException("Both numbers must be positive");  
 }  
 return firstNumber + secondNumber;  
}  
  
[TestFixture]  
public class MathTests  
{  
 [Test]  
 public void Add\_PositiveNumbers\_ReturnsCorrectResult()  
 {  
 Math math = new Math();  
 int result = math.Add(2, 3);  
 Assert.AreEqual(5, result);  
 }  
  
 [Test]  
 public void Add\_NegativeNumber\_ThrowsArgumentException()  
 {  
 Math math = new Math();  
 Assert.Throws<ArgumentException>(() => math.Add(-1, 2));  
 }  
}

**Use comments judiciously:**

While it is important to write code that is self-documenting, sometimes comments are necessary to provide additional context or explain complex logic. However, comments should not be used as a substitute for clean code and should be used judiciously. For example:

// Bad Example  
public void CalculatePrice(decimal price, int quantity)  
{  
 // calculate total  
 decimal total = price \* quantity;  
 // add tax  
 decimal tax = total \* 0.05M;  
 // add shipping  
 decimal shipping = 10;  
 // calculate final price  
 decimal finalPrice = total + tax + shipping;  
 Console.WriteLine(finalPrice);  
}  
  
// Good Example  
public void CalculatePrice(decimal price, int quantity)  
{  
 decimal total = price \* quantity;  
 decimal tax = total \* 0.05M; // tax rate is 5%  
 decimal shipping = 10;  
 decimal finalPrice = total + tax + shipping;  
 Console.WriteLine(finalPrice);  
}

**Avoid duplication:**

Code duplication can make the code harder to understand, maintain, and modify. It is important to identify and remove duplication by using methods, classes, and inheritance. For example:

// Bad Example  
public void GenerateInvoice(string customerName, List<Product> products)  
{  
 Console.WriteLine("Invoice for " + customerName);  
 Console.WriteLine("Product\t\tPrice");  
 decimal total = 0;  
 foreach (Product product in products)  
 {  
 Console.WriteLine(product.Name + "\t\t" + product.Price);  
 total += product.Price;  
 }  
 Console.WriteLine("Total\t\t" + total);  
}  
  
// Good Example  
public class Invoice  
{  
 public string CustomerName { get; set; }  
 public List<Product> Products { get; set; }  
  
 public decimal GetTotal()  
 {  
 decimal total = 0;  
 foreach (Product product in Products)  
 {  
 total += product.Price;  
 }  
 return total;  
 }  
}  
  
public class InvoiceGenerator  
{  
 public void GenerateInvoice(Invoice invoice)  
 {  
 Console.WriteLine("Invoice for " + invoice.CustomerName);  
 Console.WriteLine("Product\t\tPrice");  
 foreach (Product product in invoice.Products)  
 {  
 Console.WriteLine(product.Name + "\t\t" + product.Price);  
 }  
 Console.WriteLine("Total\t\t" + invoice.GetTotal());  
 }  
}

13. **Keep methods small:**

Small methods are easier to understand, test, and modify. It is important to break down larger methods into smaller, focused methods that do one thing and do it well. For example:

// Bad Example  
public void SaveData(string data)  
{  
 if (data == null)  
 {  
 throw new ArgumentNullException(nameof(data));  
 }  
 if (data.Length > 100)  
 {  
 throw new ArgumentException("Data length cannot exceed 100 characters", nameof(data));  
 }  
 if (File.Exists("data.txt"))  
 {  
 File.Delete("data.txt");  
 }  
 File.WriteAllText("data.txt", data);  
}  
  
// Good Example  
public void SaveData(string data)  
{  
 ValidateData(data);  
 DeleteExistingFile("data.txt");  
 WriteFile("data.txt", data);  
}  
  
private void ValidateData(string data)  
{  
 if (data == null)  
 {  
 throw new ArgumentNullException(nameof(data));  
 }  
 if (data.Length > 100)  
 {  
 throw new ArgumentException("Data length cannot exceed 100 characters", nameof(data));  
 }  
}  
  
private void DeleteExistingFile(string fileName)  
{  
 if (File.Exists(fileName))  
 {  
 File.Delete(fileName);  
 }  
}  
  
private void WriteFile(string fileName, string data)  
{  
 File.WriteAllText(fileName, data);  
}