Improving the readability of C#

Some new (and old) features that can help.

Damien Guard (BSc)

https://damieng.com

10 July 2021

Why is readability important?

- Code is read **over and over** after being written once
- As codebases expand the ability to **quickly navigate** is critical
- Readable code is **more easily maintained**, less chance of regressions
- Well-structured code can keep future changes from sprawling
- Less code is better... right up until it becomes obscure

What can we do to aid readability?

- Clear variable and class names that indicate their **purpose**
- Standards and consistency... but not when it obscures a particular case
- Small methods that do what they say... and **nothing more**
- Methods that **exit early** if they can't do what they say
- Writing blocks of code with an eye to speed-reading

Speed-reading code - writing

Reduce the cognitive load of future readers.

- Reduce the quantity of code that needs to be read
- Allow skipping a line or block within a couple of tokens
- Cyclomatic complexity determines "number of possible paths"
- Nested structures are the enemy of speed reading
- Optimize proximity of structures, data, and operations

Speed-reading code - reading

Use tools to navigate what you have.

- Step-debugging can be fastest (especially if you have tests)
  - Learn how to set intra-line breakpoints `F9`
  - Remember breakpoints can be conditional
  - Moving the instruction pointer back is useful
- Go-To Definition & Implementation `Ctrl-Click` `F12` `Ctrl+F12`
- Use the mouse back and forward buttons
- Find All References & View Call Hierarchy

Major C# features by version

- **2.0** Generics
- **3.0** LINQ (Lambdas, expressions, anonymous types, initializers)
- **4.0** Dynamic + generic co/contra-variance
- **5.0** Async + caller info
- **6.0** String interpolation, null-conditionals, auto properties
- **7.0** Tuples, deconstruction, discards, out vars, pattern matching

Major C# features by version (continued)

- **7.1** Async main, generic type matching, inferred tuple element names
- **7.2** Conditional ref, in parameter modifiers, non-trailing arguments
- **7.3** Attributes on auto-properties, out for params
- **8.0** Async streams, null-coalescing assignment, ranges, switch++
- **9.0** Records, target-typed new, top-level statements, lambda discards
- **10.0** TBD but maybe global usings, required & backing field properties

Why new C# features?

They either enable you to

- do something you couldn't do before
  or
- write ***shorter, clearer*** code to do the same thing

The first few times you encounter a new feature or pattern you'll slow down... but that passes quickly.

Switch expressions [C# 8.0]

A `switch` for returning values.

Before

```
TimeSpan interval;
switch (job.Repeats)
{
  case Repeat.Daily:
    interval = TimeSpan.FromDays(1);
    break;
  case Repeat.Weekly:
    interval = TimeSpan.FromDays(7);
    break;
  case Repeat.Hourly:
    interval = TimeSpan.FromHours(1);
    break;
  default:
    interval = TimeSpan.FromMinutes(10);
    break;
}
```

After

```
TimeSpan interval = job.Repeats switch
{
  Repeat.Daily => TimeSpan.FromDays(1),
  Repeat.Weekly => TimeSpan.FromDays(7),
  Repeat.Hourly => TimeSpan.FromHours(1),
  _ => TimeSpan.FromMinutes(10)
};
```

Enum extension methods [C# 3.0]

Enums can have extension methods too.

Before

```
var interval = job.Repeats switch
{
  Repeat.Daily => TimeSpan.FromDays(1),
  Repeat.Weekly => TimeSpan.FromDays(7),
  Repeat.Hourly => TimeSpan.FromHours(1),
  _ => TimeSpan.FromMinutes(10)

};
```

Pushed into extension method

```
var interval = job.Repeats.GetInterval();

static TimeSpan GetInterval(this Repeat repeat)
{
  return job.Repeats switch
  {
    Repeat.Daily => TimeSpan.FromDays(1),
    Repeat.Weekly => TimeSpan.FromDays(7),
    Repeat.Hourly => TimeSpan.FromHours(1),
    _ => TimeSpan.FromMinutes(10);
  };
}
```

Conditional operator [C# 1.0]

Converting a `bool` to two possible values.

Before

```
decimal tradeValue;
if (confidence > 0.7) {
  tradeValue = TradeValue.High;
} else {
  TradeValue = TradeValue.Low;
}
```

After

```
decimal tradeValue = confidence > 0.7
  ? TradeValue.High
  : TradeValue.Low;
```

Null-coalescing (a default value for nulls)

Default a value when encountering a `null`.

Before

```
var msg = GetMessage();

if (msg == null)
{
  msg = defaultMessage;
}
```

After using assignment [C# 8.0]

```
decimal tradeValue = confidence > 0.7
  ? TradeValue.High
  : TradeValue.Low;
```

After using operator [C# 2.0]

```
var msg = GetMessage() ?? defaultMessage;
```

Out variables [C# 7.0]

Declare `out` *parameters inline.*

Before

```
int quantity;
if (int.TryParse(qty, out quantity))
{
  Allocate(quantity);
}
```

After

```
if (int.TryParse(qty, out var quantity)) {
  Allocate(quantity);
}
```

Interpolated strings [C# 6.0]

Build formatted strings with in-place evaluation.

Before

```
var msg = "Hello ";
if (name == null) {
 msg += name;
} else {
 msg += "user";
}

msg += "!";
```

After

```
var msg = $"Hello {name ?? "user"}!";
```

CallerMemberName vs `nameof` [C# 5.0]

Capture the name of the calling method silently.

nameof

```
Post GetPost(Guid id)
{
  var post = db.Get<Post>(id);
  if (post == null)
    Log($"Post {id} missing", nameof(GetPost));
  return post;
}

void Log(string message, string method)
{
  logger.Log(message, method, ...);
}
```

CallerMemberName

```
Post GetPost(Guid id)
{
  var post = db.Get<Post>(id);
  if (post == null)
    Log($"Post {id} missing");
  return post;
}

void Log(string message,
  [CallerMemberName] string method = null)
{
  logger.Log(message, method, ...);
}
```

LINQ "query syntax" [C# 3.0]

Query syntax is clearer when projecting or grouping.

Lambda/method syntax

```
var results = db.Customers
  .Where(c => c.Region == "UK")
  .Select(c =>
      new { Customer = c,
            Address = GetAddress(c) })
  .Where(x => x.Address.Postcode.StartsWith("GY"))
  .Select(x => x.Customer);
```

Query/comprehension syntax

```
var results = from c in db.Customers
  let address = GetAddress(c)
  where address.PostCode.StartsWith("GY")
  select c;
```

LINQ vs `foreach` [C# 3.0]

`foreach` can often be eliminated.

Lambda/method syntax

```
var customers = db.Customers
  .Where(c => c.Region == "UK");

var addresses = new List<Address>();
foreach(var customer in customers) {
  addresses.Add(GetAddress(customer));
}
```

Query/comprehension syntax

```
var addresses = db.Customers
  .Where(c => c.Region == "UK")
  .Select(c => GetAddress(c))
  .ToList();
```

Null-conditional operator [C# 6.0]

Avoid the need to handle `null` *explicitly.*

Before

```
var validator = GetValidator();
// …
if (validator != null)
  validator.Validate(customer);
// …
if (validator != null)
  validator.Validate(address);
// ..
if (validator != null)
  validator.Validate(transaction);
```

After

```
var validator = GetValidator();
// …
validator?.Validate(customer);
// …
validator?.Validate(address);
// ..
validator?.Validate(transaction);
```

Object initializers [C# 6.0]

Move as much object initialization together as possible.

Before

```
var customer = new Customer(id);
if (!validate(form, out var errors))
  return ValidationError(errors);

Customer.Name = form["Name"];
Customer.Country = CountryMap[form["Country"]];
```

After

```
if (!validate(form, out var errors))
  return ValidationError(errors);

var customer = new Customer(id) {
  Name = form["Name"],
  Country = CountryMap[form["Country"]];
}
```

Guard conditions [C# 1.0]

Validate conditions up-front and exit to avoid nesting.

Before

```
decimal GetBalance(string id, DateTime asAt)
{
  if (id!= null) {
   throw new ArgumentNullException(acc);
  }

  if (asAt <= DateTime.UtcNow) {
    var balance = db.Transactions
      .Where(t => t.TransactionDate <= asAt)
      .OrderByDesc(t => t.TransactionDate)
      .Select(t => t.Balance)
      .FirstOrDefault();
    }
  else {
    throw new ArgumentOutOfRangeException(nameof(asAt));
  }
}
```

After

```
decimal GetBalance(string id, DateTime asAt)
{
  if (id != null)
   throw new ArgumentNullException(nameof(id));

  if (asAt > DateTime.UtcNow)
    throw new ArgumentOutOfRangeException(nameof(asAt));

  var balance = db.Transactions
    .Where(t => t.TransactionDate <= asAt)
    .OrderByDesc(t => t.TransactionDate)
    .Select(t => t.Balance)
    .FirstOrDefault();
}
```

Throw expressions [C# 7.0]

Throw exceptions inline.

Before

```
ExRate(Currency currency, Exchange exchange, float rate)
{
  if (currency == null)
   throw new InvalidArgumentException(nameof(currency));

  if (exchange == null)
   throw new ArgumentNullException(nameof(exchange));

  this.currency = currency;
  this.exchange = exchange;
  this.amount = amount;
}
```

After

```
ExRate(Currency currency, Exchange exchange, float rate) {

  this.currency = currency
    ?? throw new InvalidArgumentException(nameof(currency));
  this.exchange = exchange
    ?? throw new ArgumentNullException(nameof(exchange));
  this.amount = amount;
}
```

`var` vs type [C# 3.0]

Don't repeat obvious types.

Before

```
// Has to be a "var"
var output = new { name = "ABC", status = "ready" };

// Good candidate to "var"
Dictionary<string, int> counts =
  new Dictionary<string, int>();

// Less clear candidate to "var"
IEnumerable<ContractBase> contracts =
  GetNextContracts();
```

After

```
// Has to be a "var"
var output = new { name = "ABC", status = "ready" };

// Good candidate to "var"
var counts = new Dictionary<string, int>();

// Less clear candidate to "var"
var contracts = GetNextContracts();
```

Target-typed `new` [C# 9.0]

Eliminate the object type from the `new` statement.

Before

```
public class MyClass {
  private Dictionary<string, int> counts =
    new Dictionary<string, int>();
}
```

After

```
public class MyClass {
  private Dictionary<string, int> counts = new();
}
```

Elimination of defaults [C# 1.0]

Consider learning defaults and eliminating them.

Before

```
internal class MyClass {
  private decimal total = 0.0m;
}
```

After

```
class MyClass {
  decimal total;
}
```

Exception filtering [C# 6.0]

Catch exceptions you can handle rather than rethrowing those you can't.

Before

```
var response = await client.GetAsync(uri);
try
{
   response.EnsureSuccessStatusCode();
   var text = await response.Content.ReadAsStringAsync();
   File.WriteAllText(path, text);
}

catch (HttpRequestException ex)
{
   if (ex.StatusCode == HttpStatusCode.Redirect)
      await SaveUrl(path, response.Headers.Location);
   else
      throw;
}
```

After

```
var response = await client.GetAsync(uri);
try
{
    response.EnsureSuccessStatusCode();

    var text = await response.Content.ReadAsStringAsync();
    File.WriteAllText(path, text);
}
catch (HttpRequestException ex) when
      (ex.StatusCode == HttpStatusCode.Redirect)
{
    await SaveUrl(path, response.Headers.Location);
}
```

Records [C# 9.0]

A simpler way to declare data transfer objects (DTOs)

Before

```
class Person
{
    public string FirstName { get; private set; }
    public string LastName { get; private set; }

    public Person(string firstName, string lastName)
    {
        FirstName = firstName;
        LastName = lastName;
    }

    public override int GetHashCode()
    {
        int hash = 17;
        if (FirstName != null)
          hash = hash * 23 + FirstName.GetHashCode();
        if (LastName != null)
          hash = hash * 23 + LastName.GetHashCode();
        return hash;
    }

    public bool Equals(Person other)
    {
        if (Object.ReferenceEquals(this, other))
           return true;
        if (other == null
            || other.GetType() != typeof(Person))
           return false;
        return other.FirstName == FirstName
           && other.LastName == LastName;
    }
```

After

```
record Person
{
    public string LastName { get; }
    public string FirstName { get; }

    public Person(string firstName, string lastName)
    {
        FirstName = firstName;
        LastName = lastName;
    }
}
```

or even...

```
record Person(string FirstName, string LastName);
```

Replace comment with method [C# 1.0]

Before

```
decimal total = 0m;
...
// Total up the tax at current 20% VAT rate
const float taxRate = 0.20;
decimal tax = 0;
foreach(var line in order.Lines) {
  if (!VatExempt(line.Product))
     total += item.Total * taxRate;
```

After

```
decimal total = 0m;
...
total += CalculateTax(order.Lines, 0.20);
```

```
decimal CalculateTax(IEnumerable<Line> lines, float rate)
{
  decimal tax = 0;
  foreach(var line in lines) {
    if (!VatExempt(line.Product))
       tax += line.Total * rate;
  }
  return tax;
}
```

Comments

- You should not need comments to explain ***what*** code is doing
- You might need comments to explain ***why*** it is doing it
- Eliminate smells caused by the code
- Comment smells outside your control (business rules, external systems)
- **Do or Do Not** – there is no `// TODO`
  (Unless you go create a bug or story card for it)

More information

- C# Evolution with before & after code
  https://damieng.github.io/csharp-evolution
- Martin Fowler's Refactoring
  https://refactoring.com
- Clean Code
  https://www.indiebound.org/book/9780132350884
- The Pragmatic Programmer
  https://www.indiebound.org/book/9780201616224