WELCOME TO

PARTSTRADER

MARKETS LIMITED

Welcome to PartsTrader

- How do I escape?
- Where are the restrooms?
- What does PartsTrader do?
- Who is talking to us?
- Who else is here?

PARTSTRADER
MARKETS LIMITED

NULLWORRIES

(BE HAPPY)

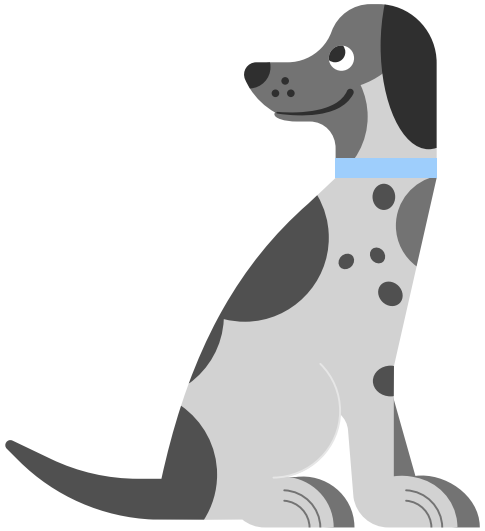
How to Git samples

Following along?

- Clone repo from GitHub
- Navigate to 01-start-here
- Open NullWorries.sln file

Overview

- Why do we use null?
- How does it cause problems?
- What can we do about it?

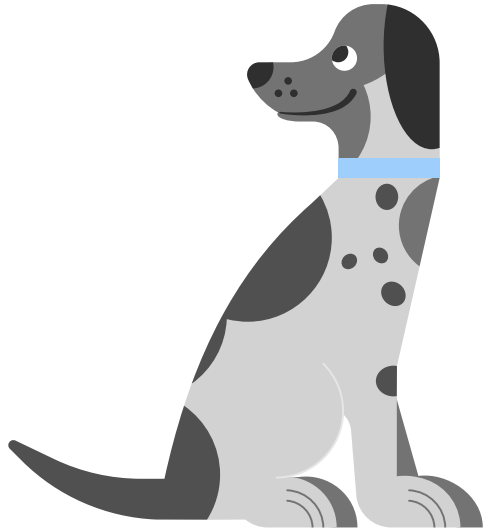


PARTSTRADER
MARKETS LIMITED

Why do we use null?

We use null to represent

- the *absence* of a value
- an *unspecified* value



PARTSTRADER
MARKETS LIMITED

Absence of a value

What is a class? 

The absence of a value can represent

- *unknown* value
- *unnecessary* value

```
public class Person
{
    public Person(string givenName, string familyName)
    {
        GivenName = givenName;
        FamilyName = familyName;
    }

    public string GivenName { get; set; }

    public string FamilyName { get; set; }
}

// Jane Citizen
Person a = new (“Jane”, “Citizen”);

// John ?
Person b = new(“John”, null);

// Lorde
Person c = new(“Lorde”, null);
```

…/null-worries/src/02-absence-of-value

PARTSTRADER MARKETS LIMITED

Unspecified value

What is a nullable value?

An unspecified value can represent

- *invalid* value
- *choice* to omit value

```
public class Person
{
    // adding a date-of-birth property
    public Person(…, DateTime? dateOfBirth)
    {
        if (dateOfBirth != null && dateOfBirth.Value < DateTime.Now)
        {
            DateOfBirth = dateOfBirth;
        }
    }

    public DateTime? DateOfBirth { get; }
}

// born 01 April 3000
Person a = new("Jane", "Citizen", DateTime.Parse("3000-04-01"));

// choice to omit date of birth
Person b = new("Jane", "Citizen", null));
```

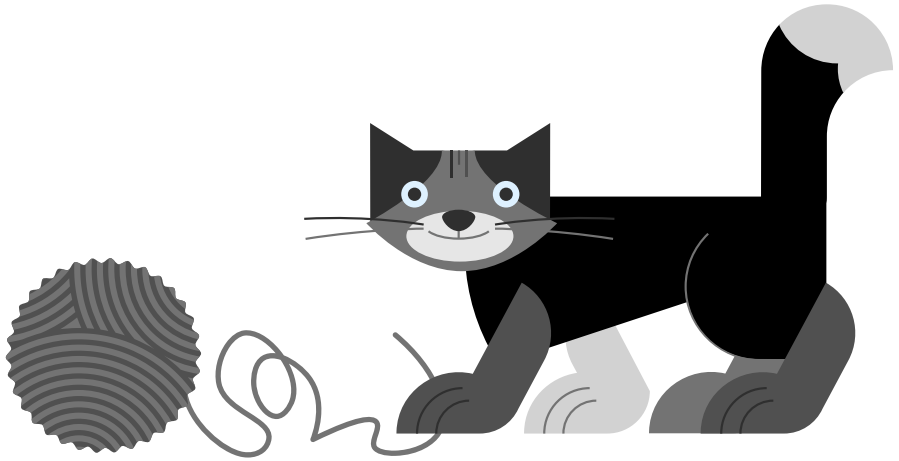
…/null-worries/src/03-unspecified-value

PARTSTRADER MARKETS LIMITED

How does it cause problems?

Null causes problems due to

- uncertainty
- null references



PARTSTRADER
MARKETS LIMITED

Uncertainty

Not clear if *null* is

- *unknown* value
- *unnecessary* value
- *invalid* value
- *choice* to omit value

```
// John ?
Person b = new(“John”, null);

// Lorde
Person c = new(“Lorde”, null);

// born 01 April 3000
Person b = new(“Jane”, “Citizen”, DateTime.Parse(“3000-04-01”));

// choice to omit date of birth
Person c = new(“Jane”, “Citizen”, null));
```

PARTSTRADER
MARKETS LIMITED

What is a null reference?

What is a reference?

A reference points to something in memory

A null reference points at nothing

```
// reference to object
object obj = new();

// reference to string
var str = “Hello world!”;

// null reference
object obj = null;
```

PARTSTRADER MARKETS LIMITED

Is that a problem?

What is the Quicksort algorithm? 

Tony Hoare invented Quicksort in 1959

In 1965 he invented null references

He apologised for them in 2009…

PARTSTRADER
MARKETS LIMITED

Why did he do that?

"I call it my billion dollar mistake"

"I couldn't resist the temptation to put in a null reference, because it was so easy to implement"

"This has led to innumerable errors, vulnerabilities and system crashes"

What is the problem?

~~Developers are lazy~~

Developers are human and make mistakes

It is *very easy* to forget to check for null references

```
var obj = ObjectHelper.GetObject();

// will this work?
var text = obj.ToString();

// if it does, will this work?
var n = text.Length;

// how about this?
Regex pattern = new(@"\\d{4,}");
var isMatch = pattern.IsMatch(text);
```

…/null-worries/src/04-easy-to-forget

PARTSTRADER MARKETS LIMITED

What happens if we forget?

For .NET invoking a member on a null reference

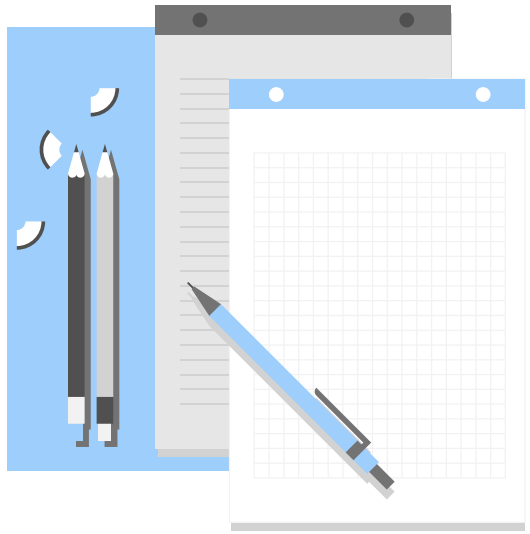
- will cause a *NullReferenceException*
- will abort the current operation
- may cause data/state corruption
- may terminate the application

PARTSTRADER
MARKETS LIMITED

What can we do about it?

Null worries? Be happy with

- alternatives to null
- ensuring valid object state
- C# language features
- functional programming



PARTSTRADER
MARKETS LIMITED

Default objects

If object not found then return

- null

```
public class Person
{
    public static Person GetPerson(string name)
    {
        // if not found...
        return null;
    }
}

var person = Person.GetPerson("Jane");

// will this work?
var text = person.ToString();
```

…/null-worries/src/05-default-objects-a

PARTSTRADER MARKETS LIMITED

Default objects

If object not found then return

- ~~null~~
- default object

Why string.Empty?

```
public class Person
{
    public static readonly Person Unknown =
        new(string.Empty, string.Empty, null);

    public static Person GetPerson(int id)
    {
        // if not found...
        return Unknown;
    }
}

var person = Person.GetPerson("Jane");

// will this work?
var text = person.ToString();
```

…/null-worries/src/06-default-objects-b

PARTSTRADER MARKETS LIMITED

Are default objects practical?

Advantages?

- null worry eliminated

Disadvantages?

- non-standard
- easy to forget about

PARTSTRADER
MARKETS LIMITED

Enforce valid object state

If object initialised with invalid state then

- carry on

```
public class Book
{
    public Book(string title)
    {
        Title = title;
    }

    public string Title { get; }
}

Book book = new(null);

// will this work?
var searchTerm = “the”;
var isMatch = book.Title.Contains(searchTerm);
```

…/null-worries/src/07-valid-state-a

PARTSTRADER MARKETS LIMITED

Enforce valid object state

If object initialised with invalid state then

- ~~carry on~~
- fail fast

Why nameof(title)? 

```
public class Book
{
    public Book(string title)
    {
        Title = title ?? throw new
            ArgumentNullException(nameof(title));
    }

    public string Title { get; }
}

// will this work?
Book a = new(null);

// how about this?
Book b = new("Screens");
```

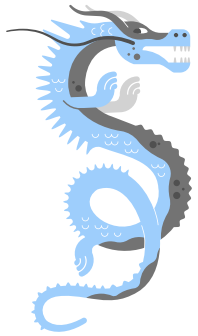
…/null-worries/src/08-valid-state-b

PARTSTRADER MARKETS LIMITED

Beware of value types

What is a struct?

Value types always have a default constructor



```
public struct Book
{
    public Book(string title)
    {
        Title = title ?? throw new
            ArgumentNullException(nameof(title));
    }

    public string Title { get; }
}

// has a default constructor!
Book book = new();

// exception: Title is null
var searchTerm = "the";
var isMatch = book.Title.Contains(searchTerm);
```

PARTSTRADER MARKETS LIMITED

Enforce valid object state

If choosing between default and valid object state then

- choose default state

```
public class Author
{
    public Author()
    {
        // use default initializers
    }

    public Author(IEnumerable<Book> books)
    {
        Books = books ?? throw new
            ArgumentNullException(nameof(books));
    }

    public IEnumerable<Book> Books { get; };
}

Author author = new();

// will this work?
var hasBooks = author.Books.Any();
```

…/null-worries/src/07-valid-state-a

PARTSTRADER MARKETS LIMITED

Enforce valid object state

If choosing between default and valid object state then

- ~~choose default state~~
- choose valid state

Why IEnumerable? 

```
public class Author
{
    public Author()
    {
        Books = Enumerable.Empty<Book>();
    }

    public Author(IEnumerable<Book> books)
    {
        Books = books ?? throw new
            ArgumentNullException(nameof(books));
    }

    public IEnumerable<Book> Books { get; };
}

Author author = new();

// will this work?
var hasBooks = author.Books.Any();
```

…/null-worries/src/08-valid-state-b

PARTSTRADER MARKETS LIMITED

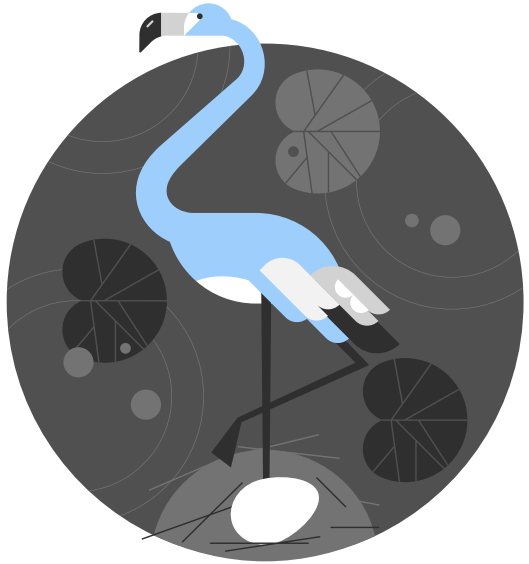
Is enforcing state practical?

Disadvantages?

- failures will happen

Advantages?

- known failure points
- principal of least surprise



PARTSTRADER
MARKETS LIMITED

Use setters to enforce state

If property set to invalid state then

- carry on

```
public class Book
{
    public Book(string title)
    {
        Title = title;
    }

    public string Title { get; set; }
}

// no exception
Book book = new("Screens");
book.Title = null;

// exception: Title is null
var searchTerm = "the";
var isMatch = book.Title.Contains(searchTerm);
```

Use setters to enforce state

If property set to invalid state then

- ~~carry on~~
- fail fast

```
public class Book
{
    public Book(string title)
    {
        Title = title;
    }

    private string _title;

    public string Title
    {
        get => _title;
        set => _title = value ?? throw new
            ArgumentNullException(nameof(title));
    }
}

Book book = new("Screens");

// exception: fail fast
Book.Title = null;
```

PARTSTRADER MARKETS LIMITED

Is enforcing state with setters practical?

Advantages? 

- known failures points

Disadvantages? 

- additional failure points
- greater chance of surprise

PARTSTRADER
MARKETS LIMITED

Mutable and immutable types

Mutable type
value *can* be changed

```
public class MutableAuthor
{
    public MutableAuthor (string name)
    {
        Name = name;
    }

    public string Name { get; set; }
}

MutableAuthor author = new("Christopher Laine");

// name can be changed
author.Name = "Peter F. Hamilton";
```

Mutable and immutable types

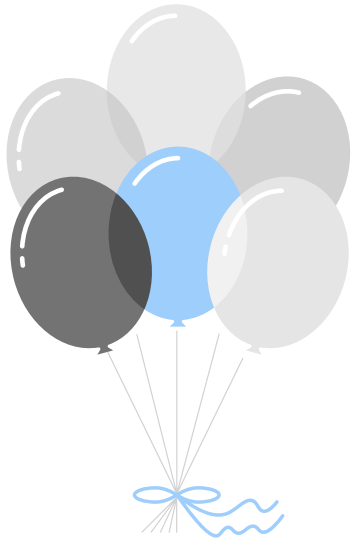
Mutable type
value *can* be changed

Immutable type
value *cannot* be changed

```
public class ImmutableAuthor
{
    public ImmutableAuthor(string name)
    {
        Name = name;
    }

    public string Name { get; }
}

ImmutableAuthor author = new("Christopher Laine");

// compiler error
author.Name = "Peter F. Hamilton";
```

Advantages of immutable types

- Constructors enforce valid object state
- Safer/easier multi-threading
  - ✓ Cannot be changed by other threads
  - ✓ Does not require synchronisation
- WYSIWYG



PARTSTRADER
MARKETS LIMITED

Disadvantages of immutable types

- ~~Greater memory usage~~
- Can be painful to initialise

```
public class RealisticBook
{
    public RealisticBook(
        string title,
        string isbn,
        string subject,
        decimal retailPrice,
        Author author,
        IEnumerable<Review> reviews,
        /* and so on… */)
    {
        // TODO: validate every argument...
    }
}
```

Playing together: the builder pattern

Builder pattern

- allows bit-by-bit initialisation
- guides us through it
- can build an immutable type

…/null-worries/src/09-builders-a

PARTSTRADER
MARKETS LIMITED

Start with a part

```
public class Part
{
    internal Part(
        string partNumber,
        string description,
        decimal listPrice)
    {
        PartNumber = partNumber;
        Description = description;
        ListPrice = listPrice;
    }

    public string PartNumber { get; }

    public string Description { get; }

    public decimal ListPrice { get; }
}
```

- Immutable type
- Not enforcing valid state
- Internal constructor
- Not value type

…/null-worries/src/10-builders-b

PARTSTRADER MARKETS LIMITED

Scaffold the part builder

```
public class PartBuilder
{
    private string _partNumber;
    private string _description;
    private decimal _listPrice;

    public PartBuilder()
    {
        _description = string.Empty;
    }
}
```

Require

- part number
- list price

Optional

- description

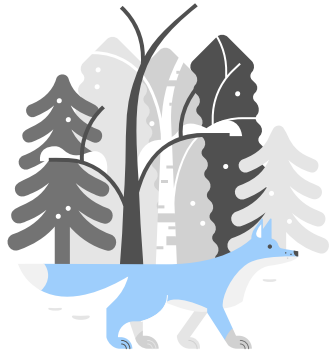
…/null-worries/src/10-builders-b

PARTSTRADER MARKETS LIMITED

Begin at the end

What is an interface?

What is an explicit interface implementation?



```
public interface IListPriceInitialized
{
    Part Build();
}

public class PartBuilder : IListPriceInitialized
{
    Part IListPriceInitialized.Build() =>
        new(_partNumber, _description, _listPrice);
}
```

…/null-worries/src/10-builders-b

PARTSTRADER MARKETS LIMITED

A required list price

```
public interface IDescriptionInitialized
{
    IListPriceInitialized ListPrice(decimal listPrice);
}

public class PartBuilder : IDescriptionInitialized, …
{
    IListPriceInitialized
        IDescriptionInitialized.ListPrice(decimal listPrice)
    {
        if (listPrice < 0m)
        {
            throw new ArgumentOutOfRangeException(…);
        }

        _listPrice = listPrice;

        return this;
    }
}
```

- Omitting code for clarity
- Enforce valid object state
- Return next actions

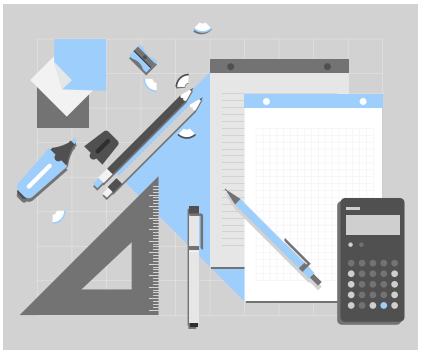
…/null-worries/src/10-builders-b

PARTSTRADER MARKETS LIMITED

An optional description

```
public interface IPartNumberInitialized : IDescriptionInitialized
{
    IDescriptionInitialized Description(string description);
}

public class PartBuilder : IPartNumberInitialized, …
{
    IDescriptionInitialized
        IPartNumberInitialized.Description(string description)
    {
        if (string.IsNullOrWhiteSpace(description))
        {
            throw new ArgumentException(…);
        }

        _description = description;

        return this;
    }
}
```

What if description is *null?*



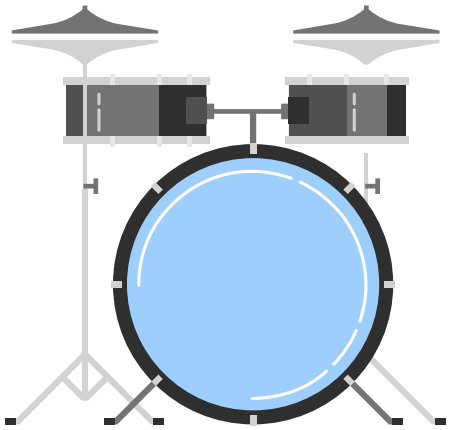
…/null-worries/src/10-builders-b

PARTSTRADER MARKETS LIMITED

A required part number

```
public interface IUninitialized
{
    IPartNumberInitialized PartNumber(string partNumber);
}

public class PartBuilder : IUninitialized, …
{
    public IPartNumberInitialized PartNumber(string partNumber)
    {
        if (string.IsNullOrWhiteSpace(partNumber))
        {
            throw new ArgumentException(…);
        }

        _partNumber = partNumber;

        return this;
    }
}
```

- Implicit implementation



…/null-worries/src/10-builders-b

PARTSTRADER MARKETS LIMITED

What have we got?

```
var b = new PartBuilder()
    .PartNumber("1234-abcd")
    .Description("Sample Part")
    .ListPrice(12.95)
    .Build();

var c = new PartBuilder()
    .PartNumber("5678-efgh")
    .ListPrice(3.14)
    .Build();

// compiler error
var d = new PartBuilder()
    .PartNumber("1234-abcd")
    .Build();

// run-time error
var e = new PartBuilder()
    .PartNumber("5678-efgh")
    .ListPrice(3.14)
    .Build();
```

- ✓ Bit-by-bit initialisation
- ✓ Guides us through it
- ✓ Creates immutable type
- ✓ Enforces valid object state

…/null-worries/src/10-builders-b

PARTSTRADER MARKETS LIMITED

C# language features

- Nullable reference types (C# 8)
- Records (C# 9)
- Init-only setters (C# 9)
- Null parameter checking (C# 10)*
- Required properties (C# 10) *

PARTSTRADER
MARKETS LIMITED

Nullable reference types

- C# 8 (.NET Core 3.x)
- Value type syntax
- Indicates whether reference *can* be null

```
// value type
int a = 0;

// nullable value type
int? b = null;

// “non-nullable” reference type
object c = new object();

// nullable reference type
object? d = null;
```

…/null-worries/src/11-nullable-references

PARTSTRADER MARKETS LIMITED

Disabled by default

- Edit project file
- Add <Nullable> element
- Set to *enable*

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <Nullable>enable</Nullable>
  </PropertyGroup>
</Project>
```

…/null-worries/src/11-nullable-references

PARTSTRADER MARKETS LIMITED

How does it help?

- Shows warning if reference *can* be null
- Reveals unexpected edge cases in old code
- Suggestion: treat warnings as errors

```
// valid: can be null
object? a = null;

// CS8600: Converting null literal or
// possible null value to non-nullable type
object b = null;

object obj = new();
var text = obj.ToString();

// CS8602 Dereference of a possibly null reference
var n = text.Length;

if (text != null)
{
    // no warning: text is non-null
    var m = text.Length;
}
```

…/null-worries/src/11-nullable-references

PARTSTRADER MARKETS LIMITED

Records

- C# 9 (.NET 5.x)
- Automatic properties
- Does *not* automatically enforce valid object state
- Does give deconstructor Equals, ToString, etc…

```
public class Person
{
    public Person(string givenName, string familyName)
    {
        GivenName = givenName;
        FamilyName = familyName;
    }

    public string GivenName { get; }

    public string FamilyName { get; }
}

public record Person
{
    public Person(string GivenName, string FamilyName);
}
```

Init-only setters

- C# 9 (.NET 5.x)
- Immutable properties
- Optional initializers
- Use backing field to enforce property state

```
public class Part
{
    public string PartNumber { get; init; }

    public string Description { get; init; }

    public decimal ListPrice { get; init; }
}

Part part = new
{
    PartNumber = “1234-abcd”,
    ListPrice = 12.95
};

// compiler error
part.Description = “Sample Part”;
```

PARTSTRADER
MARKETS LIMITED

Null parameter checking

- C# 10 (.NET 6.x)*
- Prototype feature/syntax
- Only checks for *null*

```
public Book(string title)
{
    if (title == null)
    {
        throw new ArgumentNullException(…);
    }
    …
}

public Book(string title!!)
{
    …
}
```

PARTSTRADER
MARKETS LIMITED

Required properties

- C# 10 (.NET 6.x)*
- Prototype feature/syntax
- Use backing fields to enforce valid object state

```
public class Part
{
    public required string PartNumber { get; init; }

    public string Description { get; init; }

    public required decimal ListPrice { get; init; }
}

// compiler error: ListPrice not initialized
Part part = new
{
    PartNumber = “1234-abcd”,
    Description = “Sample Part”
};
```

PARTSTRADER MARKETS LIMITED

Option data type (functional programming)

Also known as *Maybe* (Haskell)

Union of two types:

- *None* – value is absent
- *Some* – value is present

Enforces checking of None/Some

PARTSTRADER
MARKETS LIMITED

Building the option type

```
public struct Option<T>
{
    private readonly bool _hasValue;
    private readonly T _value;

    public TResult Match<TResult>(
        Func<TResult> none,
        Func<T, TResult> some)
    {
        return _hasValue ? some(_value) : none();
    }
}
```

- Why value type? 
- No external access to underlying value

…/null-worries/src/12-options

PARTSTRADER MARKETS LIMITED

Building the none and some types

```
public struct None
{
    internal static readonly None Value = new();
}

public struct Some<T>
{
    internal Some(T value)
    {
        Value = value ?? throw new
            ArgumentNullException(nameof(value));
    }

    internal T Value { get; }
}
```

- What is internal? 
- Using value types
- No external access to underlying value

…/null-worries/src/12-options

PARTSTRADER MARKETS LIMITED

Making it easy to use

```
public static class F
{
    public static None None =>
        None.Value;

    public static Some<T> Some<T>(T value) =>
        new(value);
}
```

```
public struct Option<T>
{
    public static implicit operator
        Option<T>(Infrastructure.None _) =>
            new();

    public static implicit operator
        Option<T>(Infrastructure.Some<T> some) =>
            new(some.Value);

    public static implicit operator
        Option<T>(T value) =>
            value == null ? None : Some(value);
}
```

…/null-worries/src/12-options

PARTSTRADER MARKETS LIMITED

A quick detour

```
public static class DictionaryExtensions
{
    public static Option<TValue> GetValue<TKey, TValue>(
        this IDictionary<TKey, TValue> source, TKey key) =>
            source.TryGetValue(key, out var value)
                ? Some(value) : None;
}
```

…/null-worries/src/12-options

PARTSTRADER MARKETS LIMITED

What have we got?

```
var key = Guid.NewGuid().ToString();

Dictionary<string, string> dictionary =
    new() { { key, "Hello world!" } };

Console.WriteLine(
    dictionary.GetValue(key).Match(
        none: () => "Key not found",
        some: value => $"Value is \"{value}\"."));
```

- ✓ No null worries
- ✓ Explicit failure handling
- ✓ Success case handled
- ✓ Be happy!

…/null-worries/src/12-options

PARTSTRADER MARKETS LIMITED

Summary

- We use null when we do not have a better value to use
- It causes problems because others do not know what we mean by it
- We can do something about it by making our intentions clearer

PARTSTRADER
MARKETS LIMITED

How do we make our intentions clearer?

- Default objects
- Ensuring valid object state
- Immutable objects and builders
- C# language features
- Functional programming

PARTSTRADER
MARKETS LIMITED

THANKYOU

QUESTIONS?

alistair-grant/summer-of-tech/null-worries

PARTSTRADER
MARKETS LIMITED

alistair-grant/summer-of-tech/null-worries