Methods and Properties

Simon Robinson
Software Developer

@TechieSimon www.SimonRobinson.com

Overview

Protect against bugs:

- Guard clauses

Simpler code:

- Writing fluent methods
- Choosing property types (Auto-implemented, expression-bodied, etc.)

Returning multiple values

Performance:

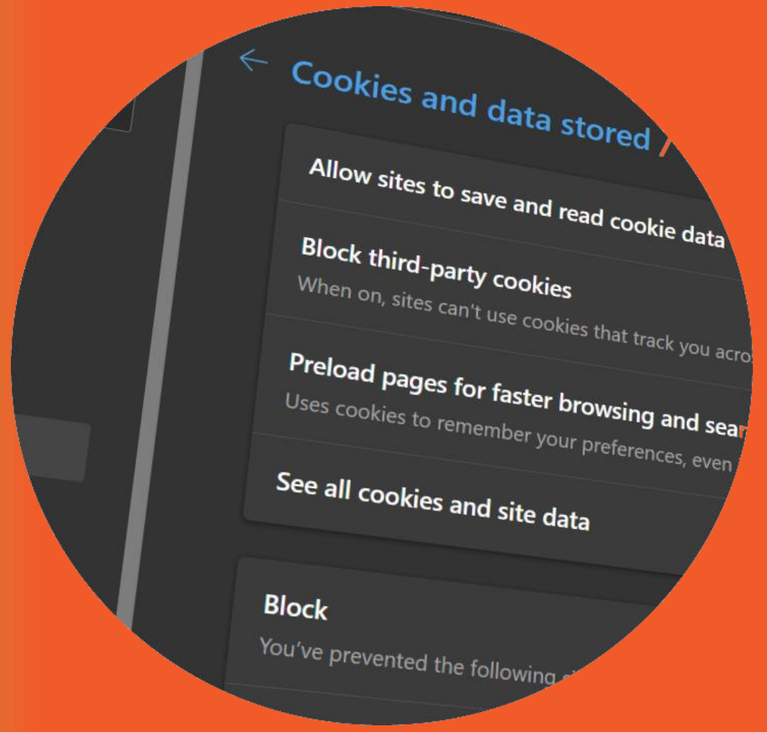
- Passing value types by reference

The Demo: Storing Cookie Sales

These:

⚠️

Not these:



Method Guard Clauses

Demo

Customer class

- Guard clauses to ensure data is valid
- Helps debugging

Choosing Types of Property

Demo

Allow customer name changes

- Make `CookieCustomer.Name` settable
- Will show the need for different property styles

Choosing a Property Type

Use simplest possible syntax:

Backing field but no logic: Use auto-property

```
public string? Notes { get; set; }
```

No backing field: Prefer expression-bodied property

```
public char NameFirstChar => Name[0];
```

Backing field and logic: Use full property syntax

```
private string _name;
public string Name
{
    get { /* getter code */ }
    set { /* setter code */ }
}
```

Designing for Fluent Code

Fluent Coding

A style of coding involving chaining method calls

Can simplify code when performing multiple operations

Demo

Add cookie sales to list of sales

- Redesign this method in a fluent style
- Simplifies calling code

Return Multiple Values from a Method

Demo

Identify customer who has spent the most

- Method must supply
  - Customer
  - How much spent
  - How many sales
- Method must return three pieces of information

Returning Multiple Values

Clear

Values are grouped together

Using a value tuple

```
public (string CustomerName, decimal TotalSpent, int NSales)
     GetHighestValueCustomer()
{
```

May be slightly faster

Using out parameters

```
public string GetHighestValueCustomer(
   out decimal totalSpent, out int nTransactions)
{
```

Passing Value Types by Reference

Demo

Add a business rule to the cookie sales app

- Will pass by reference
  - This may help performance

Passing by Reference

May improve performance

Allows methods to modify data in their callers

Passing by Value

```
MyValueType data = new();
DoSomething(MyValueType data);
```

data
20

```
void DoSomething(MyValueType data) { }
```

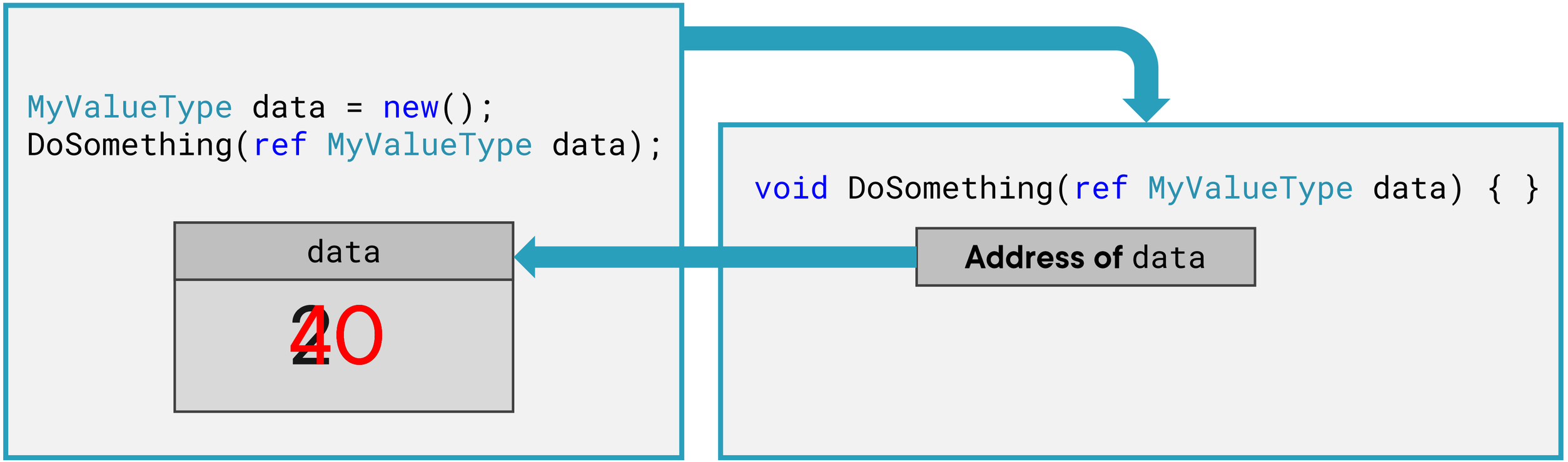
copy of data
~~2~~40

Method gets own copy of value types

Changes to copy don’t affect original

✔ This helps protect data

Passing by Reference



Method gets the address of the data in the caller

⚠️ Can mutate the same instance seen by the caller

Passing by Reference

Declare as `in`

```
void DoSomething(in MyValueType data) { }
```

Do this if by reference is just for performance

Declare as `ref`

```
void DoSomething(ref MyValueType data) { }
```

Do this if you want to modify the value in the caller

Declare as `out`

```
void DoSomething(out MyValueType data) { }
```

Do this to use the value as a return value

Demo

Brand new app

- Shows how to find value type sizes
- Will reveal if passing by reference might help performance

Summary

Techniques for robust / performant code

Guard code

- Keep this simple

Properties

- Auto-property if no logic
- Expression-bodied property if no backing field
- Full syntax if logic and backing field are required

Summary

Fluent coding

- `return this`
- Allows method chaining

Return multiple values

- Using value tuples
- Using `out` parameters

Pass value types by reference

- Using `in` for performance
- Using `out` or `ref` to pass data to the caller