Immutable and Read-only Coding

Simon Robinson
Software Developer

@TechieSimon www.SimonRobinson.com

Overview

Making types immutable

- Different for structs and classes

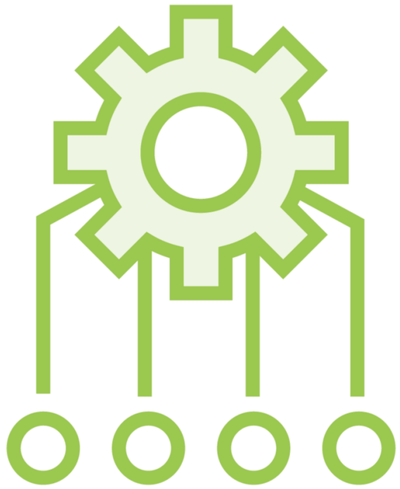
Expose readonly collection properties

- Not the same as non-collection properties

Mutable structs

- Mark readonly methods (for performance)

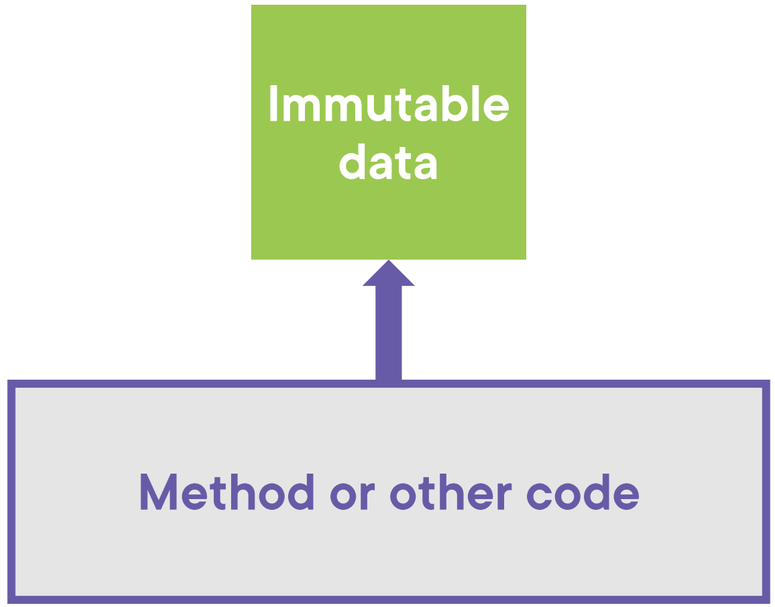
Immutable Types - Benefits



Thread safety

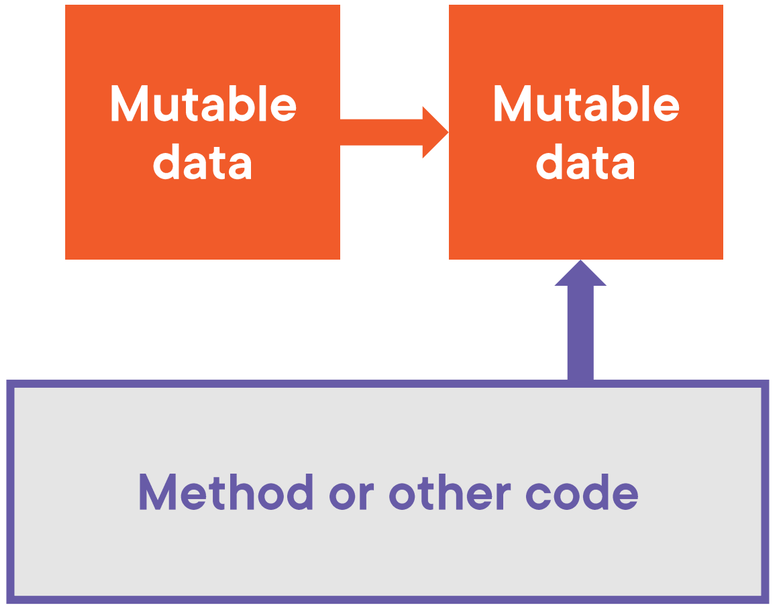
Performance

Immutable Types - Benefits



Pass by reference

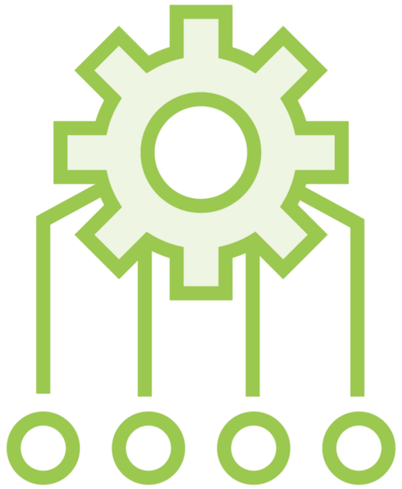
Don’t want method to modify the data



Immutability guarantees method can’t change data

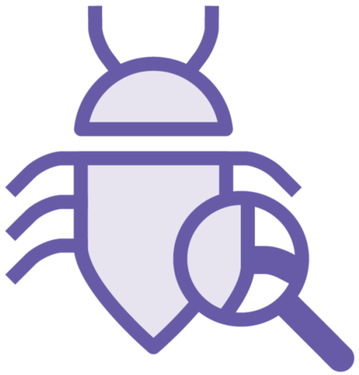
Use defensive copy to protect original data

Immutable Types - Benefits



Thread safety

Performance



Code robustness

Demo

Start with mutable class

- Make it immutable

Making Value Types Immutable

Demo

Value type `Point` added to demo

- We'll make `Point` immutable

Immutability: Structs vs. Classes

Structs	Classes
Extra Language support	
Immutability is best practice	Immutability is a design choice
Immutability avoids possible subtle bugs involving boxing	Classes don’t get boxed!

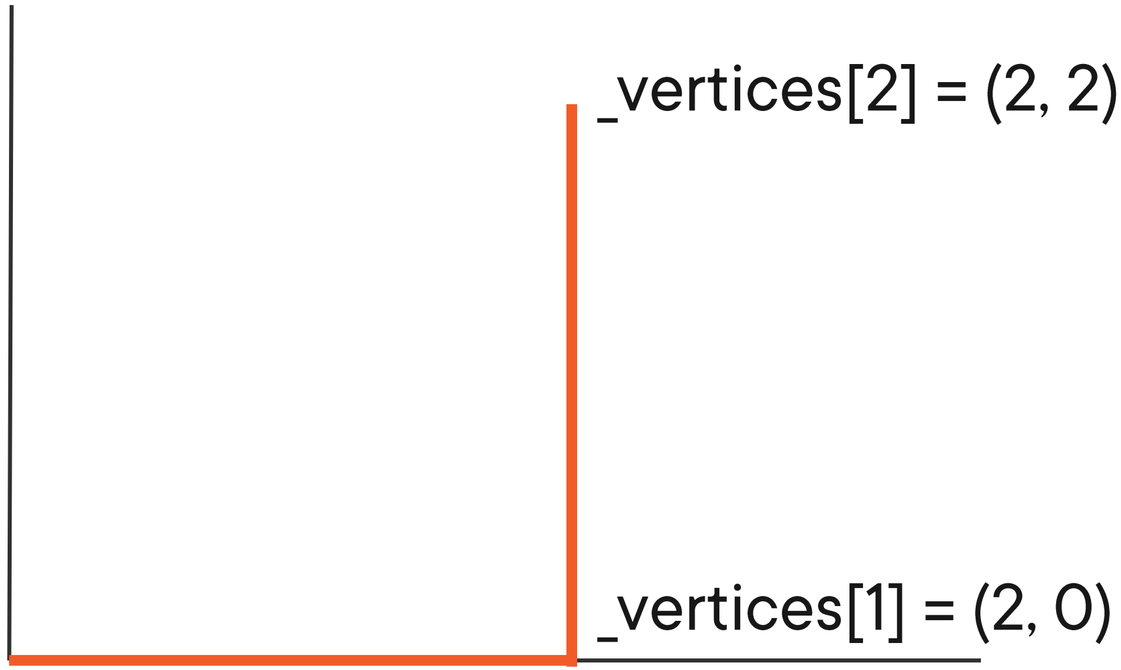
Struct and Class Immutability

To make immutable:

- Make fields `readonly`
- No property setters
- For structs:
  - Declare entire type as `readonly`
  - Immutable is good practice

Exposing Read-only Collection Properties

```
public class PolyLine
{
    private List<Point> _vertices = new();
```



Represents an open shape consisting of line segments

Demo

Polyline:

- Expose `_vertices` in read-only way

Passing Mutable Structs by Reference

Demo

Mutable struct:

- Get immutability performance benefits:
  - Mark ‘readonly’ methods that don’t mutate the struct

Summary

Protecting data from changes

- Immutability
  - Readonly fields, no property setters
  - For structs, declare type as readonly
- Collection members
  - Cache readonly copies as properties
- Mutable structs
  - Declare non-mutating methods as readonly
  - Same for property getters