Immutability

Objektumorientált szoftvertervezés

Object-oriented software design

Dr. Balázs Simon

BME, IIT

Outline

- Immutability
- Problems with mutability
- Advantages of immutability
- Immutable Object-Orientation
- Disadvantages of immutability
- Immutable collections

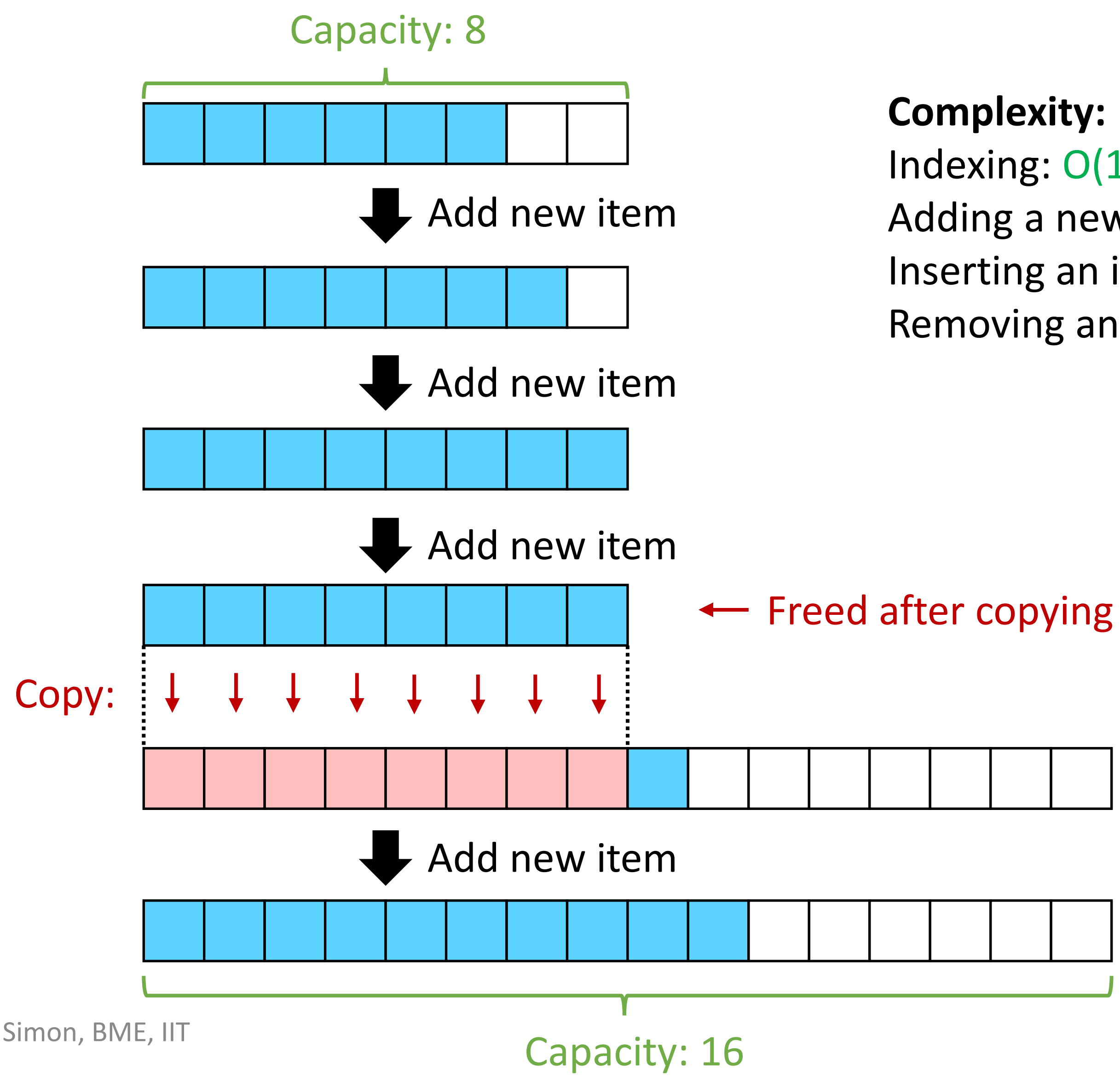
Immutability

Immutability

- An object is immutable if its state cannot change after it is constructed
  - initialized in the constructor
  - no setters
  - no methods that can change state
- Example: String class in Java and in .NET
- Greatest advantage of immutability: immutable classes are inherently thread-safe
  - no need for all those complicated thread-safety patterns
- In imperative languages we are used to mutable objects. So how can we change the state of an immutable object?
  - we can’t
  - we have to create a completely new object
- Question: isn’t this copying slow and wasteful?
  - Not necessarily, the unchanged parts can be reused!

Mutable list: List<T>

Mutable list is usually implemented using an array:



Complexity:

Indexing: O(1)

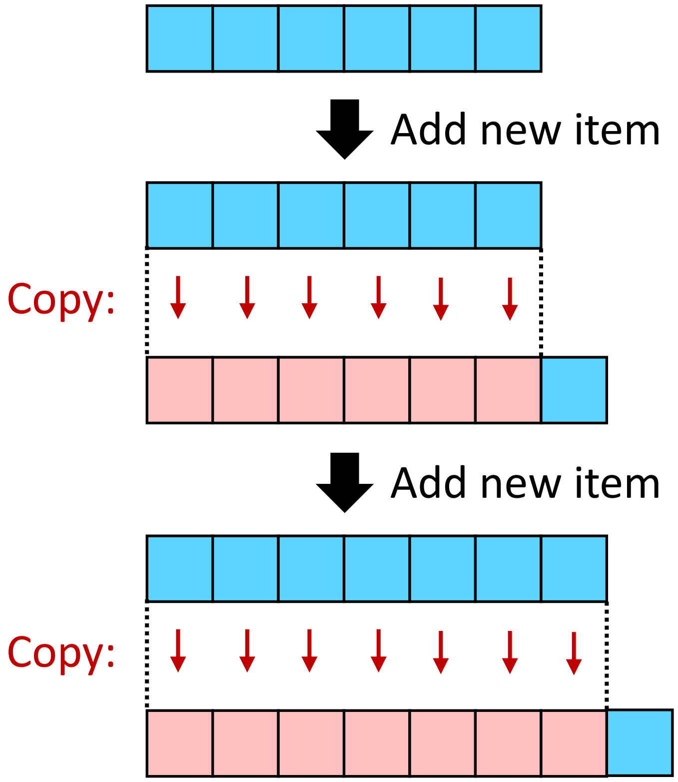
Adding a new item: O(1) *on average*

Inserting an item: O(n)

Removing an item: O(n)

Immutable list with an array

Implementing an immutable list using an array would be inefficient:
capacity is not useful, the array always has to be copied



Complexity:

Indexing: O(1)

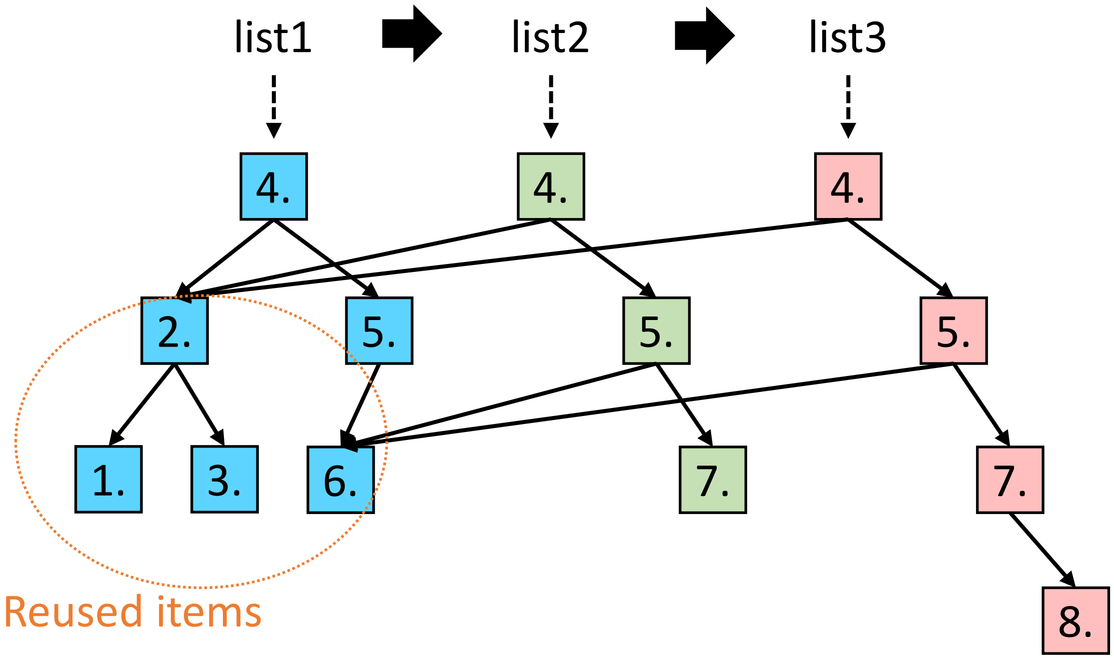
Adding a new item: O(n)

Inserting an item: O(n)

Removing an item: O(n)

Immutable list: ImmutableList<T>

Implementing an immutable list using immutable balanced binary trees:




```
list2 = list1.Add(item)
list3 = list2.Add(item)
```

Complexity:

Indexing: O(log n)

Adding a new item: O(log n)

Inserting an item: O(log n)

Removing an item: O(log n)

If any one of the lists are released the difference is collected by the GC! No memory leaks!

Problems with mutability

Problems with mutability

- readonly/final
- Passing mutable values
- Returning mutable values
- Returning defensive copies
- Multi-threaded access
- Mutable identity
- Corrupted internal state on failure
- Temporal coupling

readonly/final

- A readonly/final field in C#/Java does not mean that the stored object is constant (immutable)
- Only that the reference stored in the field cannot be changed
- The target object of the reference is still mutable:

```
public class Date
{
    public int Year { get; set; }
    public int Month { get; set; }
    public int Day { get; set; }
}

public class Program
{
    private static readonly Date Zero = new Date();

    public static void Main(string[] args)
    {
        Zero.Year = 2000; // Zero is still mutable
    }
}
```

- (In C++ a const field is really immutable, unless you use const_cast...)

Passing mutable values

- A function may change a mutable parameter value:

```
public int Min(List<int> values)
{
    values.Sort();
    return values.First();
}
```

- The caller may not expect the change

Returning mutable values

- If an internal representation is returned:

```
public class String
{
    private char[] chars;
    // ...
    public char[] GetChars()
    {
        return this.chars;
    }
}

public class Program
{
    public static void Main(string[] args)
    {
        String str = new String("Hello");
        char[] chars = str.GetChars();
        chars[0] = 'B';
    }
}
```

- The caller may change the returned value

Returning defensive copies

- To prevent callers changing an internal representation, a defensive copy has to be returned:

```
public class String
{
    private char[] chars;

    // ...

    public char[] GetChars()
    {
        return (char[])this.chars.Clone();
    }
}
```

- This is inefficient, since every call on GetChars() makes a copy
- But at least we are protected from callers

Multi-threaded access

- Multiple threads calling the same object may leave it in an inconsistent state
  - two threads entering Push at the same time may end up overwriting each other's items
  - two threads entering Pop at the same time may see the same item popped
- Locking is required to access the Stack class from multiple threads
  - and all the complicated threading patterns are needed...

```
public class Stack<T>
{
    private int top = 0;
    private T[] items = new T[100];

    public void Push(T item)
    {
        items[top++] = item;
    }
    public T Pop()
    {
        return items[top--];
    }
}
```

Mutable identity

- The state, and therefore the identity of an object can change
- The hash-code of a mutable object can also change
- This is a problem especially if the object is used as a key in a Dictionary/HashMap

```
Dictionary<Person, string> map = new Dictionary<Person, string>();

Person p = new Person("Alice");
map.Add(p, "Hello");

p.Name = "Bob";
string value = map[p]; // KeyNotFoundException
```

Corrupted internal state on failure

- A mutable object can be left in an inconsistent state when an error occurs:

```
public class Stack {
  private int size;
  private String[] items;
  // ...
  public void push(String item) {
    size++;
    if (size > items.length) {
      throw new RuntimeException("stack overflow");
    }
    items[size-1] = item;
  }
}
```

- If an exception is thrown, the value of “size” is inconsistent with the “items” array

Temporal coupling

- Example:

```
Request request = new Request("http://example.com");
request.method("POST");
String first = request.fetch();
request.body("text=hello"); // modifies the request object
String second = request.fetch();
```

- Problem: we reuse the configuration of “first” in the configuration of “second”
  - they are in temporal coupling
  - if we reconfigure “first”, we also reconfigure “second”
  - they have to stay together in this order
  - this is a hidden information in code that we have to remember

Advantages of immutability

Advantages of immutable objects

- Simple to construct, test, and use
- Always thread-safe
- Don’t need copy constructor and cloning
- Side-effect free
- No identity mutability
- Failure atomicity
- Easier to cache
- Prevent NULL references, which are bad
- Avoid temporal coupling

Simple to construct, test, and use

- Constructing immutable objects:
  - initialized in the constructor
  - setters do not modify, they return a different immutable object
- Examples:

```
ImmutablePerson p =
    new ImmutablePerson(firstName: "Bob", lastName: "White", age: 45);


ImmutableList<string> l = ImmutableList<string>.Empty;
ImmutableList<string> l2 = l.Add("Bob");
```

Always thread-safe

- Immutable objects can only be read
- No matter how many of them and how often are being called parallel, they are always thread-safe
- Multiple threads can access the same object at the same time, without clashing with another thread

Don’t need copy constructor and cloning

- Never need to make a copy of an immutable object
  - since nobody will be able to change it
- It is enough to keep the reference
  - much more efficient than copying the whole object
- A single immutable instance can be used in many places (e.g. default values like empty lists)
  - uses less memory
  - faster: no new instances needed
- Example:

```
public ImmutableArray<Person> FindPersons(string name)
{
    if (name == null) return ImmutableArray<Person>.Empty;
    // ...
}
```

Side-effect free

- Can be freely passed as parameters:

```
public int Min(ImmutableList<int> values)
{
    ImmutableList<int> sorted = values.Sort(); // "values" is unchanged
    return sorted.First();
}
```

- Internal representations can be freely returned:

```
public class String
{
    private ImmutableArray<char> chars = ImmutableArray<char>.Empty;
    // ...
    public ImmutableArray<char> GetChars()
    {
        return this.chars; // caller won't be able to modify this
    }
}
```

- No defensive copies are needed

No identity mutability

- An immutable object's identity is its own state
- The state never changes, therefore, the identity cannot change
  - hence, its HashCode cannot change

```
ImmutablePerson p = new ImmutablePerson("Alice");
map.Add(p, "Hello");

ImmutablePerson p2 = p.SetName("Bob");

string value = map[p]; // OK, "p" cannot be changed
string value2 = map[p2]; // KeyNotFoundException

ImmutablePerson p3 = new ImmutablePerson("Alice");
string value3 = map[p3]; // OK, "p3" has the same identity as "p"
```

Failure atomicity

- An immutable object will never be left in a broken state
- Its state is modified only in its constructor
  - the constructor either succeeds and the object is in a consistent state
  - or the constructor fails and the object is not created

```
public class ImmutableStack {
  private final int size;
  private final ImmutableArray<String> items;
  // ...
  private ImmutableStack(int size, ImmutableArray<String> items) {
    this.size = size;
    this.items = items;
  }
  public ImmutableStack push(String item) {
    if (size+1 > items.length) {
      throw new RuntimeException("stack overflow");
    }
    return new ImmutableStack(size+1, items.set(size, item));
  }
}
```

Easier to cache

- If all the parts of two immutable objects are the same, the two immutable objects are also the same

```
public AddOrSubExpression CreateAddOrSubExpression(Expression left,
                                  Token _operator, Expression right)
{
    int hash;
    var cached =
        SyntaxNodeCache.TryGetNode(SyntaxKind.AddOrSubExpression,
            left, _operator, right, out hash);
    if (cached != null) return (AddOrSubExpression)cached;
    var result =
        new AddOrSubExpressionGreen(SyntaxKind.AddOrSubExpression,
            left, _operator, right);
    if (hash >= 0)
    {
        SyntaxNodeCache.AddNode(result, hash);
    }
    return result;
}
```

Prevent NULL references, which are bad

- Clean-code:
  - don’t pass null
  - don’t return null
- These rules are easy to keep with immutable objects
  - pass or return: default object instead of null
    - e.g. empty collections
  - efficient:
    - no need to create new default objects, new empty lists, etc. on the fly
    - the same immutable default object instance can be used everywhere

```
ImmutableList<int> list = ImmutableList<int>.Empty;
if (!error)
{
    list = list.AddRange(items);
}
ImmutableList<int> sortedList = Sort(list);
```

Avoid temporal coupling

- Example:

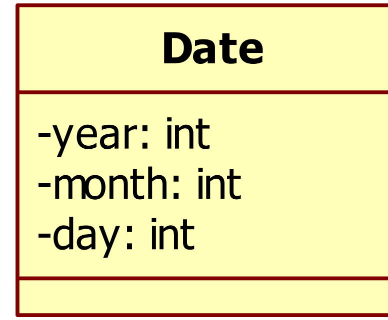
```java
final Request request = new Request("");
final Request post = request.method("POST");
String first = post.fetch();
String second = post.body("text=hello").fetch();
```

- No temporal coupling:
  - “first” and “second” are independent of each other
  - both of them are derived from the same “post” object

Immutable Object-Orientation

Example: mutable Date

```
public class MutableDate
{
    private int year;
    private int month;
    private int day;

    public MutableDate(int year = 0, int month = 0, int day = 0)
    {
        this.year = year;
        this.month = month;
        this.day = day;
    }
    public int Year
    {
        get { return this.year; }
        set { this.year = value; }
    }
    public int Month
    {
        get { return this.month; }
        set { this.month = value; }
    }
    public int Day
    {
        get { return this.day; }
        set { this.day = value; }
    }
}
```

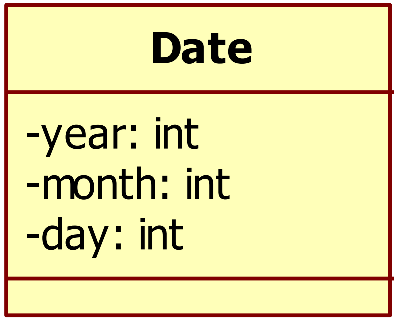


Immutable Object-Orientation

- Design pattern:
  - immutable class
    - initialized in the constructor
    - getters for the fields (Get...)
    - methods for incremental change as a fluent API (With...)
    - a method for multiple changes (Update)
    - other methods for behavior
    - a method for creating a builder from the current object (ToBuilder)
    - a static method for creating a builder (CreateBuilder)
  - builder class
    - mutable: not thread safe!
    - the builder can be more efficient when there are a lot of changes
      - e.g. adding items to a list
    - getters for the fields (Get...)
    - setters for the fields (Set...)
    - setters for the fields as a fluent API (With...)
    - a method for returning the immutable object from the current state (ToImmutable)

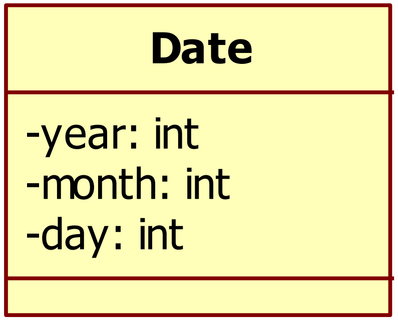
Example: immutable Date (1/5)

```
public class Date
{
    // Immutable default value:
    public static readonly Date Default = new Date();

    // Fields are readonly to prevent accidental change:
    private readonly int year;
    private readonly int month;
    private readonly int day;

    public Date(int year = 0, int month = 0, int day = 0)
    {
        this.year = year;
        this.month = month;
        this.day = day;
    }

    // Getters:
    public int Year { get { return this.year; } }
    public int Month { get { return this.month; } }
    public int Day { get { return this.day; } }

    // ... next slide ...
```



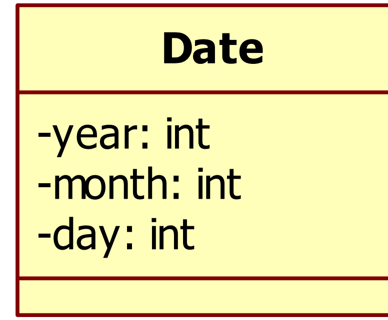
Example: immutable Date (2/5)

```
public class Date
{
    // ... previous slide ...

    // Update multiple fields:
    public Date Update(int year, int month, int day)
    {
        if (year != this.year || month != this.month || day != this.day)
        {
            return new Date(year, month, day);
        }
        return this;
    }

    // Update individual fields as a fluent API:
    public Date WithYear(int year)
    {
        return this.Update(year, this.month, this.day);
    }
    public Date WithMonth(int month)
    {
        return this.Update(this.year, month, this.day);
    }
    public Date WithDay(int day)
    {
        return this.Update(this.year, this.month, day);
    }
    // ... next slide ...
```



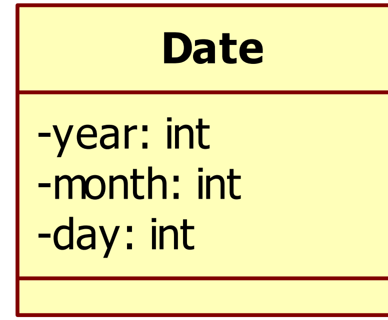
Example: immutable Date (3/5)

```
public class Date {
    // ... previous slide ...

    // Builder from the current object:
    public Date.Builder ToBuilder()
    {
        return new Date.Builder(this);
    }
    // Builder from the default value:
    public static Date.Builder CreateBuilder()
    {
        return new Date.Builder(Date.Default);
    }

    // Inner class of Date:
    public class Builder
    {
        private int year;
        private int month;
        private int day;

        internal Builder(Date date)
        {
            this.year = date.Year;
            this.month = date.Month;
            this.day = date.Day;
        }
        // ... next slide ...
```



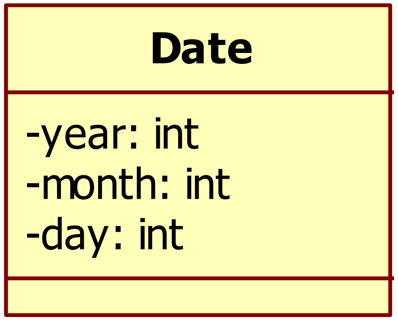
Example: immutable Date (4/5)

```
public class Date
{
    public class Builder
    {
        // ... previous slide ...

        // Getters-setters:
        public int Year
        {
            get { return this.year; }
            set { this.year = value; }
        }

        public int Month
        {
            get { return this.month; }
            set { this.month = value; }
        }

        public int Day
        {
            get { return this.day; }
            set { this.day = value; }
        }

        // ... next slide ...
```



Example: immutable Date (5/5)

```
public class Date {
    public class Builder {
        // ... previous slide ...

        // Setters for the fluent API:
        public Builder WithYear(int year)
        {
            this.Year = year;
            return this;
        }
        public Builder WithMonth(int month)
        {
            this.Month = month;
            return this;
        }
        public Builder WithDay(int day)
        {
            this.Day = day;
            return this;
        }

        // Construct an immutable object from the current state of the builder:
        public Date ToImmutable()
        {
             return new Date(this.year, this.month, this.day);
        }
    }
}
```

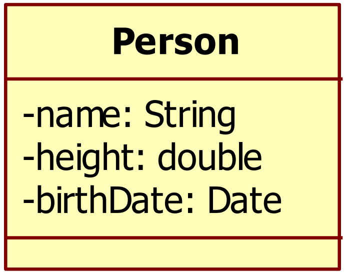


Using the immutable Date

```
Date d1 = Date.Default;                         //    0.0.0.
Date d2 = d1.WithYear(2016);                    // 2016.0.0.
Date d3 = d1.WithYear(2017).WithMonth(9);       // 2017.9.0.
Date d4 = d3.Update(d3.Year, 5, 23);            // 2017.5.23.

Date.Builder b1 = Date.CreateBuilder(); // 0.0.0.
b1.Year = 2016;
b1.Month = 9;
Date d5 = b1.ToImmutable(); // 2016.9.0.
b1.Day = 27;
Date d6 = b1.ToImmutable(); // 2016.9.27.

Date.Builder b2 = d4.ToBuilder(); // 2017.5.23.
b2.Month = 3;
b2.Day = 21;
Date d7 = b2.ToImmutable(); // 2017.3.21.

// 2017.6.18.
Date d8 = d7.ToBuilder().WithMonth(6).WithDay(18).ToImmutable();
```

Example: immutable Person (1/2)

```
public class Person
{
    public static readonly Person Default = new Person();

    private readonly string name;
    private readonly double height;
    private readonly Date birthDate;

    public Person()
        : this(string.Empty, 0, Date.Default)
    {
    }

    public Person(string name, double height, Date birthDate)
    {
        this.name = name;
        this.height = height;
        this.birthDate = birthDate;
    }

    public string Name { get { return this.name; } }
    public double Height { get { return this.height; } }
    public Date BirthDate { get { return this.birthDate; } }

    // ...
```



Example: immutable Person (2/2)

```
public class Person
{
    // ...
    public class Builder
    {
        private string name;
        private double height;
        private Date.Builder birthDate; // Date builder!

        internal Builder(Person person)
        {
            this.name = person.Name;
            this.height = person.Height;
            // Immutable date to date builder:
            this.birthDate = person.BirthDate.ToBuilder();
        }

        public Person ToImmutable()
        {
            // Date builder to immutable date:
            return new Person(this.name, this.height, this.birthDate.ToImmutable());
        }
        // ...
    }
}
```

Person
-name: String -height: double -birthDate: Date

Converting between immutable objects and their builders can be quite expensive!
Only worth's it if a lot of operations have to be performed
and the builder is more efficient for those operations.
Sometimes it’s just not worth it to have a builder at all.

Disadvantages of immutability

Disadvantages of immutability

- Requires a lot of plumbing
- Inconvenient syntax
- Cheaper to update an existing object than to create a new one
- A small change in a large immutable structure is very inconvenient
- Conversion between immutable objects and builders is inefficient
- No circular reference possible

Requires a lot of plumbing

- Mutable objects: fields + getter-setters
  - mutable date: ~25 lines
- Immutability:
  - Immutable objects: fields + getters + with methods + to builder
    - immutable date: ~50 lines
  - Builder objects: fields + getter-setters + with methods + to immutable
    - immutable date builder: ~50 lines
- ~4x the code!
  - mechanical work
  - but can be generated automatically

Inconvenient syntax

- Imperative languages are designed for mutability
- Immutable objects in imperative languages have an inconvenient syntax

Mutable is more intuitive:

```
Date mutableDate = new Date();
mutableDate.Year = 2017;
mutableDate.Month = 9;
```

Immutable is a bit inconvenient (although still readable):

```
Date immutableDate = Date.Default.WithYear(2017).WithMonth(9);
```

Cheaper to update an existing object than to create a new one

- Imperative languages are designed for mutability
  - because mutability is very efficient
- Modifying a mutable field frequently is more efficient than creating a new object every time
  - especially in large structures
- Examples:
  - a Word document as an immutable object
    - every time you type a character, the whole document must be recreated (although most of it can be reused)
  - a Visual Studio solution with projects and source files
    - every time you type a character, the whole solution must be recreated (although most of it can be reused)
    - Roslyn actually does this!

A small change in a large immutable structure is very inconvenient

- If there is a change in an immutable structure, the whole structure must be recreated
  - no matter how small the change is
- Although most of the structure can be reused
- The recreation requires a lot of (recursive) calls
  - see functional languages where everything is immutable
  - a lot of functions need to be created for a small change deep in the structure
- However, if the structure is traversed as a whole and the recreation functions are there, this is not a problem
  - e.g. immutable binary trees, Roslyn syntax trees, etc.

Conversion between immutable objects and builders is inefficient

- Builders are more efficient if changes are frequent
  - since builders are mutable
  - (but they are usually not thread-safe: cannot be used from multiple threads)
- Converting an immutable object structure to a builder object structure or a builder object structure to an immutable object structure is inefficient
  - requires a lot of copying
  - only worth's it, if there are a lot of changes performed on the builders
  - otherwise, it is more efficient to just omit the builders and use the With... functions of the immutable objects
- Sometimes builders can use the same underlying data structure as the immutable object

No circular reference possible

- Immutable data structures must be DAGs
  - usually they are trees
- No circular references are possible
  - an immutable object cannot be changed after it is created:

```
Husband h = new Husband(???);
Wife w = new Wife(h);
```

- Possible solutions:
  - bi-directional relationships can be stored as separate immutable objects
    - e.g. Marriage
  - lazy initialization
    - (but it makes things a bit more complicated)

```
public class Husband
{
    private Wife wife;

    public Husband(Wife wife)
    {
        this.wife = wife;
    }
}

public class Wife
{
    private Husband husband;

    public Wife(Husband husband)
    {
        this.husband = husband;
    }
}
```

Immutable collections

Advantages of immutable collections

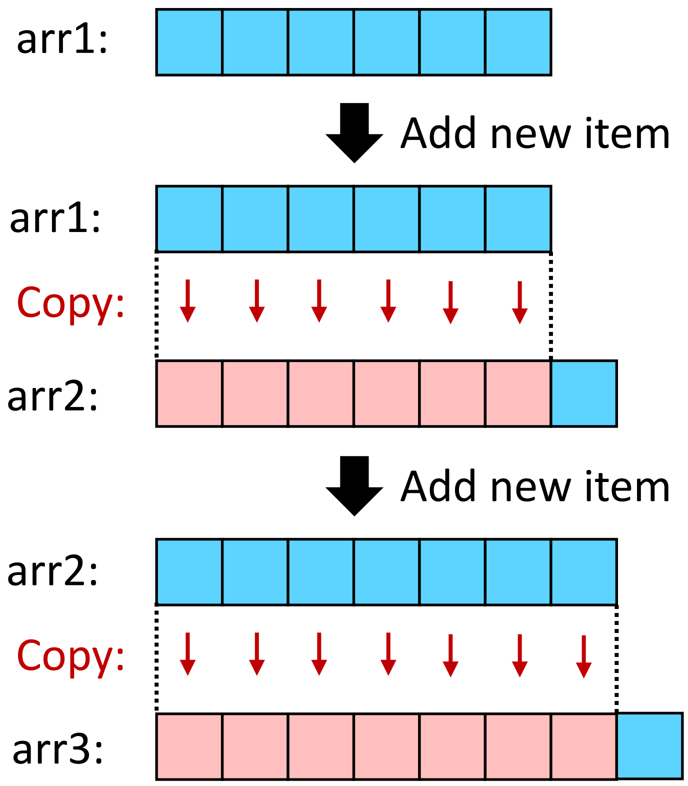
- Snapshot semantics: allowing you to share your collections in a way that the receiver can count on never changing
- Implicit thread-safety in multi-threaded applications: no locks required to access collections
- Any time you have a class member that accepts or returns a collection type and you want to include read-only semantics in the contract
- Functional programming friendly
- Allow modification of a collection during enumeration, while ensuring original collection does not change
- They implement the same IReadOnly* interfaces that your code already deals with so migration is easy

System.Collections.Immutable

- Structures:
  - ImmutableArray<T>
- Classes:
  - ImmutableStack<T>
  - ImmutableQueue<T>
  - ImmutableList<T>
  - ImmutableHashSet<T>
  - ImmutableSortedSet<T>
  - ImmutableDictionary<K, V>
  - ImmutableSortedDictionary<K, V>

ImmutableArray<T>

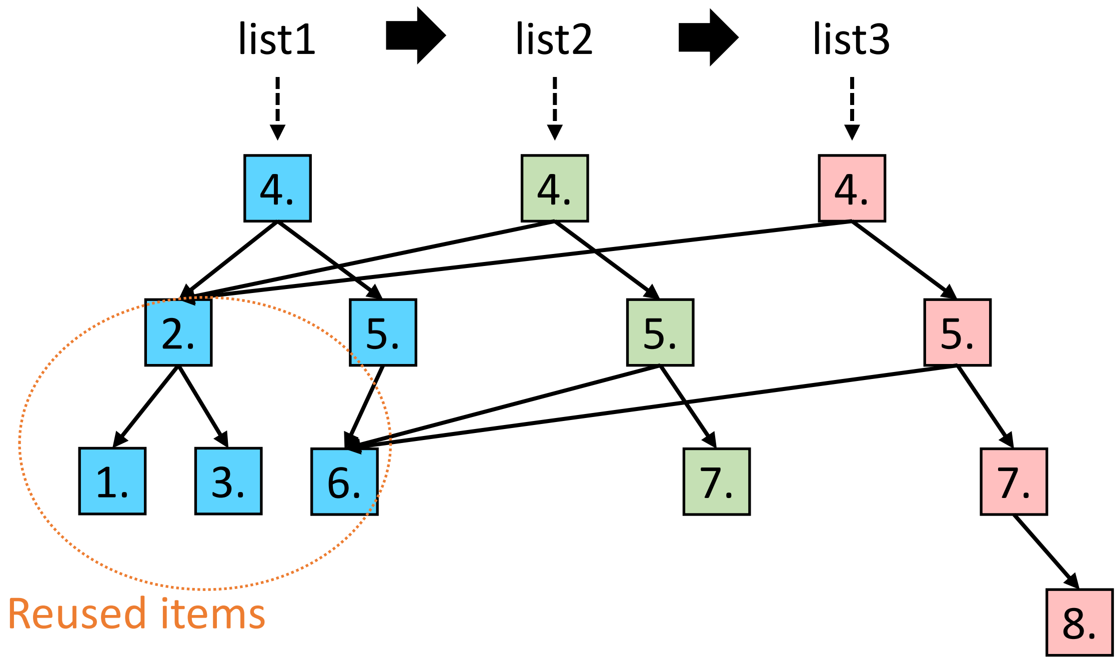
```
arr2 = arr1.Add(item)
arr3 = arr2.Add(item)
```



Complexity:
Indexing: O(1)
Adding a new item: O(n)
Inserting an item: O(n)
Removing an item: O(n)

ImmutableList<T>

```
list2 = list1.Add(item)
list3 = list2.Add(item)
```



Complexity:
Indexing: O(log n)
Adding a new item: O(log n)
Inserting an item: O(log n)
Removing an item: O(log n)

ImmutableArray<T> vs. ImmutableList<T>

- Reasons to use immutable array:
  - Updating the data is rare or the number of elements is quite small (less than 16 items)
  - You need to be able to iterate over the data in performance critical sections
  - You have many instances of immutable collections and you can't afford keeping the data in trees
- Reasons to use immutable list:
  - Updating the data is common or the number of elements isn't expected to be small
  - Updating the collection is more performance critical than iterating the contents

Using the immutable collections: Empty

```
var emptyFruitBasket = ImmutableList<string>.Empty;
```

No constructor: no memory allocation is needed

Empty is a static singleton: can be shared throughout the application

Using the immutable collections

```
var emptyFruitBasket = ImmutableList<string>.Empty;

var fruitBasketWithApple = emptyFruitBasket.Add("Apple");
```

Each operation returns a new collection

The original emptyFruitBasket is still empty

Using the immutable collections

```
var fruitBasket = ImmutableList<string>.Empty;
fruitBasket = fruitBasket.Add("Apple");
fruitBasket = fruitBasket.Add("Banana");
fruitBasket = fruitBasket.Add("Celery");
```

We can reassign the same local variable

Intermediate lists are still not modified, and internal parts of them may be reused

Intermediate lists and their non-reused internal parts will be garbage collected

Using the immutable collections

```
var list = ImmutableList<int>.Empty;
list.Add(3); // forgot to reassign result to a local variable
             // contents of the "list" local variable is still empty!
```

Caution: make sure to save the result of an operation!

Using the immutable collections

```
fruitBasket = fruitBasket.AddRange(new[] { "Kiwi", "Lemons", "Grapes" });
```

Add several elements at once to prevent many intermediate objects

Using the immutable collections

```
var fruitBasket = ImmutableList<string>.Empty
    .Add("Apple")
    .Add("Banana")
    .Add("Celery")
    .AddRange(new[] { "Kiwi", "Lemons", "Grapes" });
```

Instead of saving the intermediate results, we can use them as a fluent API.

Builders

- Each top-level operation results in allocating a few new nodes in that binary tree
  - increases GC pressure
  - just like concatenating strings
- Immutable collections have builders
  - just like StringBuilder for strings
- Immutable collection builder:
  - it is a *mutable* collection that uses exactly the same data structure as the immutable collection, but without the immutability requirement
  - implements the same mutable interface that we are used to
  - to restore immutability call **ToImmutable()**
    - creates a snapshot: the builder freezes its internal structure and returns it as an immutable collection

Builders

```
/// <summary>Maintains the set of customers.</summary>
private ImmutableHashSet<Customer> customers;

public ImmutableHashSet<Customer> GetCustomersInDebt()
{
    // Since most of our customers are debtors, start with
    // the customers collection itself.
    var debtorsBuilder = this.customers.ToBuilder();

    // Now remove those customers that actually have positive balances.
    foreach (var customer in this.customers)
    {
        if (customer.Balance >= 0.0)
        {
            // We don’t have to reassign the result because
            // the Builder modifies the collection in-place.
            debtorsBuilder.Remove(customer);
        }
    }

    return debtorsBuilder.ToImmutable();
}
```

In this example:
- no immutable collection is modified
- no collection is copied entirely

Bulk modification without Builders

```
private ImmutableHashSet<Customer> customers;

public ImmutableHashSet<Customer> GetCustomersInDebt()
{
    return this.customers.Except(c => c.Balance >= 0.0);
}
```

As efficient as the Builder approach

However, not all bulk modifications are expressible as a single method call with a lambda: use the Builder in those cases

Immutable vs. mutable collections

Immutable collections can be faster than mutable collections:

```
private List<int> collection;

public IReadOnlyList<int> SomeProperty
{
    get
    {
        lock (this)
        {
            return this.collection.ToList();
        }
    }
}
```

```
private ImmutableList<int> collection;

public IReadOnlyList<int> SomeProperty
{
    get { return this.collection; }
}
```

Immutable vs. mutable collections

Algorithmic complexity:

	Mutable (amortized)	Mutable (worst case)	Immutable
Stack.Push	O(1)	O(n)	O(1)
Queue.Enqueue	O(1)	O(n)	O(1)
List.Add	O(1)	O(n)	O(log n)
ImmutableArray.Add			O(n)
HashSet.Add	O(1)	O(n)	O(log n)
SortedSet.Add	O(log n)	O(n)	O(log n)
Dictionary.Add	O(1)	O(n)	O(log n)
SortedDictionary.Add	O(log n)	O(n log n)	O(log n)

Immutability vs. mutability

Immutability vs. mutability

- Use immutability whenever possible: immutability is good
- Immutability protects you from a lot of bugs and threading issues
- Immutability is great for simple objects and tree hierarchies
- There are Garbage Collectors optimized for immutability:
  - an immutable object always references older objects than itself
- Mutability is usually more efficient than immutability when there are frequent changes
- However, it is harder to write thread-safe mutable code