Collections – Part VI

Nico Ludwig (@ersatzteilchen)

TOC

- Collections – Part VI
  - Producing Collection Objects
  - Immutable Collections and defense Copies
  - Empty Collections
  - Ranges and Tuples as pseudo Collections
  - Filling Collections with structured Data
  - Collections abstracting Data Processing: Stack and Queue
  - Kung Fu beyond Collections: Java's Streams and .NET's LINQ
  - Not discussed Topics

Initial Words

Yes, my slides are heavy.

I do so, because I want people to go through the slides at their own pace w/o having to watch an accompanying video.

On each slide you'll find the crucial information. In the notes to each slide you'll find more details and related information, which would be part of the talk I gave.

Have fun!

Producing Collection Objects in Java – Part I

- Java's companion classes *Collections* and *Arrays* provide some simple factories to create collections.

- *Arrays.asList()* produces an index-wise equivalent but unmodifiable *List* from the passed array:

Arrays
+ asList(...a : T) : List<T>

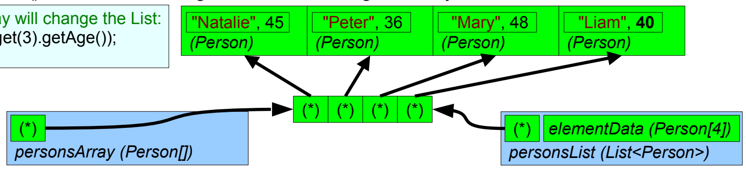
```
Person[] personsArray = {new Person("Natalie", 45), new Person("Peter", 36), new Person("Mary", 48), new Person("Liam", 37)};
List<Person> personsList = Arrays.asList(personsArray); // Creates an unmodifiable List from an array.
```

  - Because Arrays.*asList()* accepts a variable count of arguments, we can directly pass some values, instead of a first class array:

```
// Creates an unmodifiable List from a couple of arguments.
List<Person> personsList2 = Arrays.asList(new Person("Natalie", 45), new Person("Peter", 36), new Person("Mary", 48), new Person("Liam", 37));
```

  - An important point here is, that *Arrays.asList()* only provides a view of the underlying array.
  - It means, that the *List* returned by *Arrays.asList()* reflects the changes done to the original array:

```
personsArray[3].setAge(40); // Changes in the original array will change the List:
System.out.printf("4th Person's age: %d%n", personsList.get(3).getAge());
// >4th Person's age: 40
```



- If a *List* of only one item is needed, the simple factory *Collections.singletonList()* can be more efficient than *Arrays.asList()*:

```
List<Person> oneMary = Collections.singletonList(mary); // Creates an unmodifiable List from exactly one argument.
```

Collections
+ singletonList(o : T) : List<T>

4

  - *Collections.singletonList()* can be more efficient: it doesn't create an real array-backed *List*, but a special object implementing *List*.

Producing Collection Objects in Java – Part II

- The result of *Arrays.asList()* allows to exchange elements, i.e. the result is only immutable in its size, exactly like an array:

```
List<Person> personsList3 = Arrays.asList(new Person("Natalie", 45), new Person("Peter", 36), new Person("Mary", 48), new Person("Liam", 37));
personList3.set(2, new Person("Steve", 32)); // OK! We just set another element at slot 2.
```

```
personList3.add(new Person("Hal", 34)); // Invalid! Throws UnsupportedOperationException! We cannot change the size, the List is backed by an array.
```

- Alternatively we can use the simple factory *List.of()*. It creates a structural immutable *List* from an arbitrary count of arguments.
  - We can not add or remove elements and in opposite to *Arrays.asList()*'s result also not exchange contained elements.
  - *List.of()* follows the idea *Map.of()* and *Set.of()*.
  - The arguments of *List.of()* must not be null!

```
List<Person> personsList4 = List.of(new Person("Natalie", 45), new Person("Peter", 36), new Person("Mary", 48), new Person("Liam", 37));
```

```
personList4.add(new Person("Hal", 34)); // Invalid! Throws UnsupportedOperationException! We cannot change the size, the List is backed by an array.
```

```
personsList4.set(2, new Person("Steve", 32)); // Invalid! Throws UnsupportedOperationException! We cannot change the size, the List is backed by an array.
```

- *List.of()* is available since Java 9, as a call and its result are slightly more efficient than *Arrays.asList()*.
  - It should be said, that *List.of()*'s result cannot be deserialized by pre Java 8-code.
  - Another idiosyncrasy is, that *List.of()*'s result throws NPEs, if *List.contains()* is called with the argument null.

Producing Collection Objects in .NET

- .NET: There exist many means to produce collections through extension methods from the *System.Linq* namespace.
  - E.g. producing a *List* from an array with the extension method *ToList()*:

```
Person[] personsArray = {new Person("Natalie", 45), new Person("Peter", 36), new Person("Mary", 48), new Person("Liam", 37)};
IList<Person> personsList = personsArray.ToList(); // Creates a List from an array.
```

- The class *Enumerable* provides the simple factories *Range()* and *Repeat()* to produce respective sequences.
  - The method-call chain with *ToList()* produces/materializes the lists from the sequences. – The resulting *IList* is modifiable.

```
IList<int> numbers = Enumerable.Range(1, 6).ToList(); // Creates a List of integers from 1 to 6 inclusively.
IList<int> sixCopiesOf42 = Enumerable.Repeat(42, 6).ToList(); // Creates a List containing six copies of the integer 42.
```

Immutable Collections – Part I

- Immutability of objects is a relevant topic meanwhile to write robust applications. How is it supported for collections?

- In C++, container objects can be idiomatically be set as const, then non-const member functions are no longer callable:

```
// Defines a const vector containing some ints:
const std::vector<int> numbers{1, 2, 3, 4, 5, 6};
```

```
// After the const vector was created it cannot be modified:
numbers.push_back(7); // Results in a compile time error!
```

```
// But reading is no problem:
int value = numbers[0];
```

  - The objects managed by a const *std::vector* cannot be modified as well! This is called const-correctness in C++.
  - (This idiomatic usage of const objects in C++ requires no further explanation.)

- In the COCOA framework there exist mutable and immutable collections separately:

```
// Defines an immutable NSArray containing some ints (autoreleased):
NSArray* numbers = @[@1, @2, @3, @4, @5, @6];
// NSArray does not expose any methods that could modify numbers, therefor we
// have to create an NSMutableArray from numbers:
NSMutableArray* mutableNumbers = [NSMutableArray arrayWithArray:numbers];
// NSMutableArray provides methods to modify the collection:
[mutableNumbers addObject:@7];
```

  - *NSMutableArray* inherits from *NSArray*, this idea is used for other COCOA collections as well.

7

- Immutable collections are very relevant to write threadsafe code.

Immutable Collections – Part II

- The .NET framework follows another principle to provide immutability: wrapping.
  - Wrapping means, that a present instance is wrapped into another object in order to front-load another behavior.

```
// Defines a list containing some ints:
IList<int> numbers = new List<int>{1, 2, 3, 4, 5, 6 };
// The mutable IList<int> will be wrapped by an immutable ReadOnlyCollection<int> (from the namespace System.Collections.ObjectModel):
ReadOnlyCollection<int> readOnlyNumbers = new ReadOnlyCollection<int>(numbers);

// ReadOnlyCollection<T> implements IList<T> so that it can partake in APIs requiring objects exposing that interface.
```

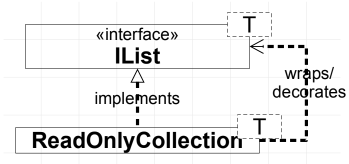
```
// But IList<T> is implemented explicitly, so potentially mutable methods can not be directly called:
readOnlyNumbers.Add(7); // Results in a compile time error.
```

```
// If ReadOnlyCollection<T> is accessed via the IList<T> interface, calling potentially mutable methods will throw a NotSupportedException.
IList<int> readOnlyNumbersAsList = readOnlyNumbers;
readOnlyNumbersAsList.Add(7); // Results in a run time error: NotSupportedException will be thrown.
```

  - Alternatively *List<T>*'s method *AsReadOnly()* can be used, which directly returns readonly wrappers.

```
// Directly create a ReadOnlyCollection<int> from a List<int>
ReadOnlyCollection<int> readOnlyNumbers2 = new List<int>{1, 2, 3, 4, 5, 6 }.AsReadOnly();
```

  - The idea of wrapping is used for *ReadOnlyDictionary* as well.
  - Objects implementing *ICollection<T>* can be checked if these are readonly with the property *ICollection<T>.IsReadOnly*.



- The design pattern we've encountered here is called wrapper or decorator.

8

Immutable Collections – Part III

- Java does also apply the wrapper principle to provide immutability.
  - Java hides the wrapper type completely from the caller, but exposes an unmodifiable object implementing a collection interface.
  - The unmodifiable implementation is somehow internally instantiated by the simple factory *Collections.unmodifiable...()*.

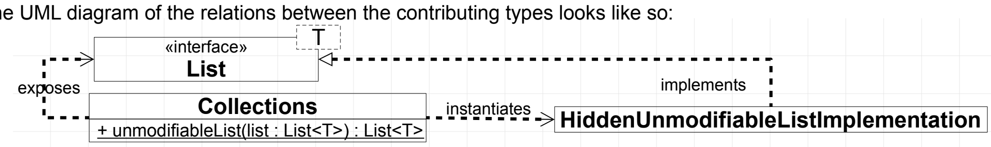
```
// Defines a list containing some ints:
List<Integer> numbers = new ArrayList<>(List.of(1, 2, 3, 4, 5, 6));
// The mutable List<int> will be wrapped by an unmodifiable List<int>:
List<Integer> readOnlyNumbers = Collections.unmodifiableList(numbers); // Calling the simple factory.
// readOnlyNumbers' potentially mutable methods can be called, but these calls will throw UnsupportedOperationExceptions.
```

```
readOnlyNumbers.add(7); // Invalid! UnsupportedOperationException will be thrown.
```

  - *readOnlyNumbers* is an object different from *numbers*, it just wraps *numbers*, but cannot "write through" to the wrapped *List*.
  - But, *Collections.unmodifiable...()* don't create snapshots, so changes on *numbers* "read through" *readOnlyNumbers*:

```
numbers.add(7); // A modification on the wrapped collection ...
int seventhElement = readOnlyNumbers.get(6); // ... "reads through" the wrapping unmodifiable collection.
// seventhElement = 7
```

- Java: More simple factories can be found in the class *Collections* as static methods (*unmodifiableSet()*, *unmodifiableMap()*).
  - The return type of all of these simple factories are just interface types, the unmodifiable implementations are always hidden.
  - "Unmodifiable" means, that the size of the returned *List* cannot be changed afterwards. Always new objects are returned, but they are no snapshots.
  - So the UML diagram of the relations between the contributing types looks like so:



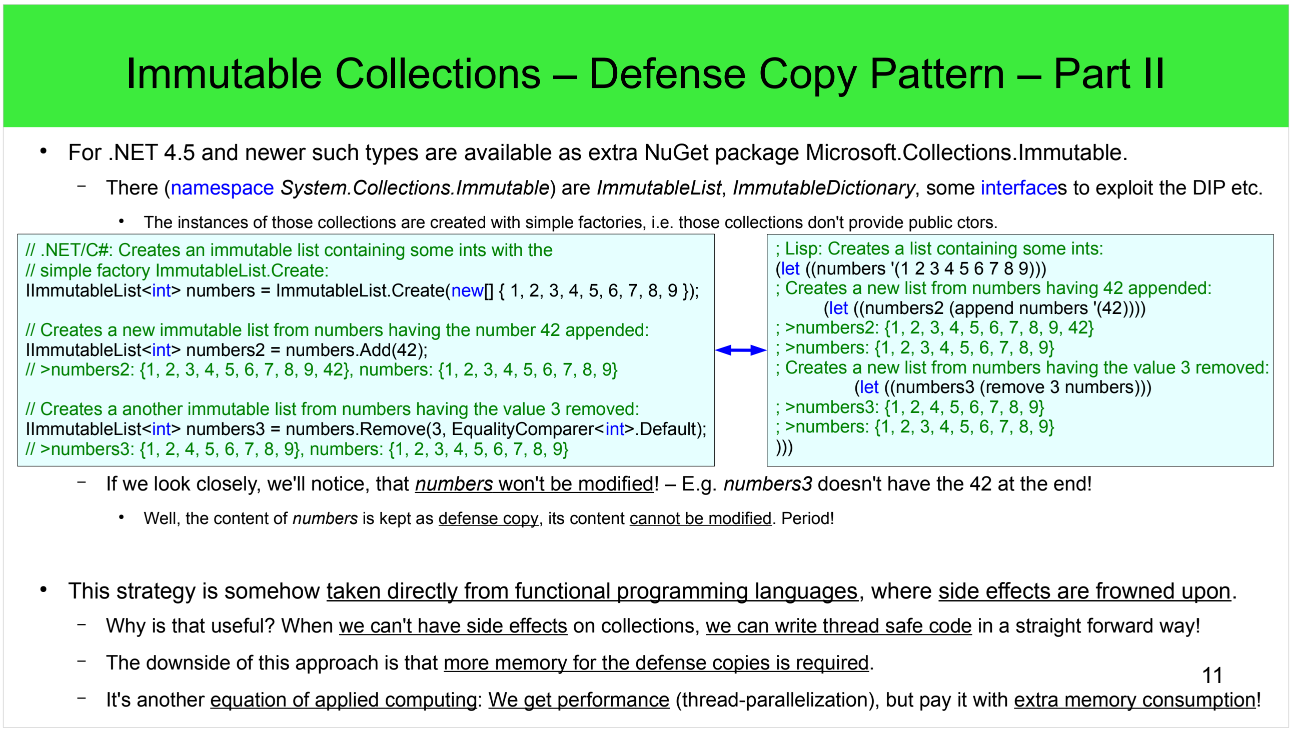
9

Immutable Collections – Defense Copy Pattern – Part I

- The strategy on which the discussed immutable collections are built is "forbidding mutating side effects".

- But there exists a completely different approach: collections that can not be modified, but produce new collections!
  - Sorry?

- Well, when a method is called that has a "mutating-name" (e.g. *add()*), a copy of the mutated collection will be returned!
  - I.e. the original collection won't be changed!
  - We know this strategy from string types in Java and .NET and also from c-strings (const char*):
    - String-operations won't modify the original, but new strings with content different from the original strings are returned.
  - This strategy is sometimes called the defense copy pattern.



- Lists in F# are also immutable, one can only create new lists having different content from the original lists. On the other hand F# does have an extra ugly syntax for array-creation, -accessing and -manipulation (uglier than the list syntax), which are mutable! So, F# clearly states immutable collections (i.e. F# lists) being more primary than arrays! It should be mentioned that memory management of fp languages is very sophisticated. Esp. during processing lists many new lists are generated, because lists are immutable; all the remaining superfluous lists need to be freed from memory. Therefor fp languages/runtimes were the first systems introducing garbage collection. Mind that the general ways of processing lists and also the memory considerations are equivalent to those of processing strings in Java and .NET, where strings are immutable (lists of characters).
- The Java Development Kit (JDK) does not provide collections like *ImmutableXXX*. But such collections can be retrieved from Google (package *com.google.common.collect*).

Empty Collections – Part I

- Often collections are used as return value of an algorithm. The usual result of failure is returning a "invalid" value like null:

```
public static List<Integer> getNumbersFromUser() {
        List<Integer> fromUser = new ArrayList<>();
        try {
             // Get input from user here and fill the list fromUser...
        }  catch (Exception ex) { //... if something went wrong return null.
             return null;
        }
        return fromUser;
}
```

```
List<Integer> result = getNumbersFromUser();
// The returned input must be checked for nullity before accessing it!
if (null != result) {
        for (int item : result) {
             System.out.println(item);
        }
}
```

  - Checking the result for null every time after *getNumbersFromUser()* was called is cumbersome and it could be forgotten!
  - Another idea is to return an empty collection as the result, then no checking for nullity is required:

```
public static List<Integer> getNumbersFromUser() {
        List<Integer> fromUser = new ArrayList<>();
        try {
             // Get input from user here and fill the list fromUser...
        } catch (Exception ex) { //... if something went wrong return an empty list.
             return new ArrayList<>(0);
        }
        return fromUser;
}
```

```
List<Integer> result = getNumbersFromUser();
// The returned input needs no check for nullity before accessing it!
for (int item : result) {
        System.out.println(item);
}
```

- What we've just encountered is called the null-object design pattern.
  - The idea is to return a genuine empty object instead of an invalid object, so that callers need no special treatment of results.

Empty Collections – Part II

- Java supports the idea of empty collections and the null-object design pattern by *Collections*' simple factories:

```
List<Integer> fromUser = new ArrayList<>();
try {
        // Get input from user here and fill the list fromUser...
} catch (Exception ex) { //... if something went wrong return an empty list.
        return Collections.emptyList();
}
return fromUser;
```

- Besides *Collections.emptyList()* there exist simple factories producing other empty collections, e.g. *Map*s, *Set*s and *Iterator*s:

```
Map<String, Integer> emptyMap = Collections.emptyMap();
Set<String> emptySet = Collections.emptySet();
Iterator<String> emptyIterator = Collections.emptyIterator();
```

Collections
+ emptyList() : List<T>
+ emptyMap() : Map<K, V>
+ emptySet() : Set<T>
+ emptyIterator() : Iterator<T>

  - These simple factories produce null-objects that are unmodifiable!
  - The produced null-object of a specific simple factory is always identical.

- Besides the companion class *Collections* we can also use the "*of()*"-style simple factories in respective collection types:

```
List<Integer> emptyList = List.of();
Map<String, Integer> emptyMap = Map.of();
Set<String> emptySet = Set.of();
```

  - Mind that resulting collections throw NPEs, if *Collection.contains()*, *Map.containsKey()* and *Map.containsValue()* are called with null.
  - Further mind, that those collections cannot be deserialized by pre-Java 9 code.

13

- Java: it's recommended to use methods like *Collections.emptyList()* instead of fields like *Collections.EMPTY_LIST*! *Collections.EMPTY_LIST* is just a raw type, whereas with *Collections.emptyList()* the compiler can infer the correct type and the code will be type safe.

Empty Collections – Part III

- The .NET framework doesn't directly support null-objects for empty collections. Instead *Enumerable*'s static methods can be used:

```
IList<int> fromUser = new List<int>();
try {
        // Get input from user here and fill the list fromUser...
}  catch (Exception ex) { //... if something went wrong return an empty list.
        return Enumerable.Empty<int>().ToList(); // uses System.Linq.Enumerable
}
return fromUser;
```

  - The method-call chain *Empty().ToList()* produces/materializes an empty list. – The resulting *List* is modifiable.
  - The methods *ToDictionary()* and *ToArray()* can be used to create empty collections the same way.

- Starting with .NET 4.6, we can use the simple factory *Array.Empty<T>()* to create empty arrays:

```
int[] emptyArray = new int[0];
```

```
int[] emptyArray = Array.Empty<int>();
```

  - *Array.Empty<T>()* is the recommended way to express empty arrays, because it features interning.
  - *Array.Empty<T>()* can only be used where no compile-time constant is required for arrays.

14

- Empty collections in .NET:
  - When variadic arrays in methods are no filled with values, *Array.Empty<T>()* will be inserted by the compiler.
  - There are no pre-canned empty *IDictionary*s or *IList*s in .NET. To create an empty dictionary, just go with *new Dictionary<T>(0)*/*new List<T>(0)*.
  - The .NET framework guidelines mandate the usage of empty collections instead of null.
    - However, this guideline doesn't work for optional arguments: the default value must be a compile time constant, so we cannot use anything different than null, esp. no empty collection. Esp. if optional arguments are own UDTs, we are better off just using overloads and pass an explicitly created null-object (like *Array.Empty<T>()*) of that UDT. Null-objects are generally a more stable pattern than optional arguments!

Singleton Objects

- We have already presented *Collections.singletonList()*, it can be used if an unmodifiable *List* of only one item is needed.

- Besides *Collections.singletonList()* there exist more simple factories producing singleton *Map*s and *Set*s:

```
List<String> singletonList = Collections.singletonList("Fran");
Set<String> singletonSet = Collections.singleton("Julian");
Map<String, Integer> singletonMap = Collections.singletonMap("Joe", 71);
```

```
Collections
+ singletonList(o : T) : List<T>
+ singleton(o : T) : Set<T>
+ singletonMap(key : K, value : V) : Map<K, V>
```

  - These simple factories produce respective collections that are unmodifiable!
  - The results are structural unmodifiable: we can neither add new items nor remove items nor replace the single contained item.

- Since Java 9 we can also use *List.of()*, *Map.of()* or *Set.of()* to produce singletons:

```
List<String> singletonList2 = List.of("Fran");
Set<String> singletonSet2 = Set.of("Julian");
Map<String, Integer> singletonMap2 = Map.of("Joe", 71);
```

  - But there are some caveats:
    - "*of*"-style simple factories can't handle nulls, if null ist passed, they'll raise NPEs.
    - The resulting collections cannot be serialized by Java code of version < 9.

15

Ranges as pseudo Collections

- There are some constructs in programming, which act or look like collections, but aren't.
  - Those are esp. idioms or special types, which support iteration (such as ranges) or data processing (such as tuples).
  - What they have in common with collections is a certain size / count of elements and the containment of "some" items.
  - Such constructs originate in scripting languages but gradually make their way into dominant programming languages.

- Ranges act as compact representation of a set of adjacent items by only specifying a start-value and an end-value.
  - The definition of adjacent items is only possible, if the type is of "countable set" or the step, difference between items gets specified.

- In Groovy ranges are represented idiomatically in the language.
  - The .. and ..< syntax allows defining closed and half-open ranges:

```
// Represents the closed range [1, 5]:
def numbers = 1..5;
for (def i in numbers) {
        println(i);
}
```

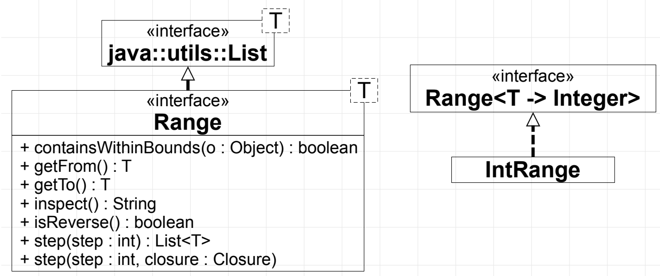
```
Terminal
1
2
3
4
5
```

```
// Represents the half-open range [1, 5[:
def numbers = 1..<5;
for (def i in numbers) {
        println(i);
}
```

```
Terminal
1
2
3
4
```

Ranges in Groovy – Part I

- Groovy's ranges are represented as instances of the generic UDT *groovy.lang.Range<T>*.
  - For primitive types like ints, Groovy provides primitive specializations, e.g. *IntRange*.



- Interestingly, *Range* also implements *List*, which provides functionality to index-access items and get the size of a range:

```
def numbers = 1..10;
println("Ranges size: ${numbers.size()}");
// >Ranges size: 10
println("Item at index 3: ${numbers[3]}");
// >Item at index 3: 4
```

  - However, ranges are unmodifiable in Groovy, so we can't modify an *IntRange* even though it implements *List*:

```
numbers[3] = 17; // Invalid! Throws UnsupportedOperationException
```

17

  - Obviously, *IntRange* applies the optional feature pattern to "disable" unsupported features.

Ranges in Groovy – Part II

- Because *Range* implements *List*, it transitively implements *Collection* and *Iterable*, and can thus be used in for-in-loops.
  - But there is a little more support, esp. we can define the step of the range's items:

```
def numbers = (1..10).step(2);
for (def i in numbers) {
        println(i);
}
```

```
Terminal
1
3
5
7
9
```

- More features:
  - *IntRange* implements equality "as expected":

```
def A = 1..10;
def B = 1..10;
def C = (1..10).step(2);
println("A == B: ${A == B}");
// >A == B: true
println("A == C: ${A == C}");
// >A == C: false
```

  - *IntRange.toString()* is overridden to yield a useful representation of the items, *Range.inspect()* returns the literal *IntRange*-expression:

```
println("A's items: ${A}");
// >A's items: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
println("A: ${A.inspect()}");
// >A: 1..10
```

Tuples as Pseudo Collections

- Tuples are fixed-sized, ordered lists of heterogeneous elements.
  - … whereas (typical) arrays for example are fixed-sized ordered lists of homogeneous elements.
  - Tuples are types, which are data containers, a kind of immutable lists, but not yet collections.
  - Tuples close the gap between UDTs (like records: limited count of fields of different types) and collections (size, element-access).

- Originally tuples are taken from maths, where they represent an ordered and finite sequence of items.
  - E.g. the components of the spatial vector *a* can be decomposed into a tuple:

$$\vec{a}=\begin{pmatrix}2\\7\\5\end{pmatrix} \text{ can be represented as a tuple like so: } a_{comp}=(2,7,5)$$

  - *a*'s components are represented as the 3-tuple a_{comp}. A 3-tuple is sometimes also called triple
    - 0-tuples can be called null-tuples,1-tuples can be called singles or monads, 2-tuple can be called pair or twin or couple or double, others simply n-tuples.
    - The "items" of the tuple are called components: 2, 7, and 5 are the components of a.
  - A complex number can be represented as 2-tuple of a real number for the real part and another real number for the imaginary part:

$$z=2+3i\,;\,z\in C \text{ can be represented like so: } z_{comp}=(\Re(z),\Im(z))=(2,3)$$

  - In opposite to sets, a tuple's components need not to be distinct.

- Tuples are a bit different in cs, where we can typically have tuple-components of different types.

Tuples in .NET – Part I

- As an example how to ally tuples in programming, we can represent a's and z's components as .NET *Tuples*:

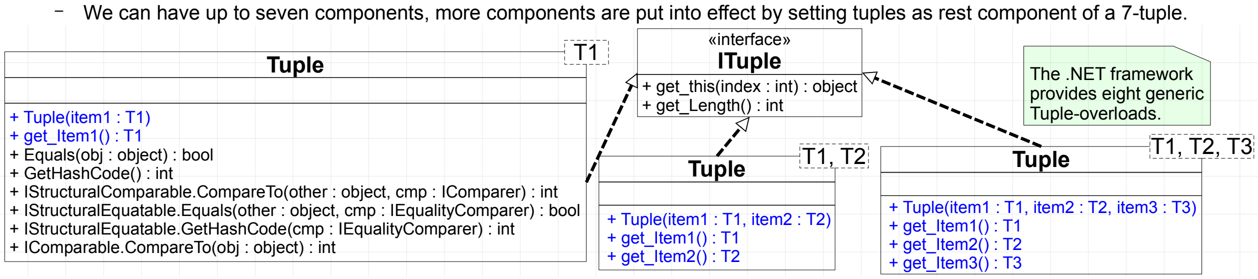
$\vec{a}=\begin{pmatrix}2\\7\\5\end{pmatrix} \quad a_{comp}=(2,7,5)$

```
Tuple<int, int, int> aComp = new Tuple<int, int, int>(2, 7, 5);
```

$z=2+3i \quad z_{comp}=(2,3)$

```
Tuple<int, int> zComp = new Tuple<int, int>(2, 3);
```

- The count and the static types of *Tuple*-components is not restricted. Esp. components need not to be numbers!
  - We can have up to seven components, more components are put into effect by setting tuples as rest component of a 7-tuple.



- Because *Tuples* are generic classes, we have the freedom to create *Tuples* with components of basically any type:

```
// Creating some tuples:
Tuple<int, string> twoTuple = new Tuple<int, string>(2, "two"); // A couple using the simple factory Tuple.Create and type inference.
Tuple<string, int, double> triple = new Tuple<string, int, double>("three", 3, .7); // A triple.
// Accessing a tuple's components:
Console.WriteLine($"couple's item1: {twoTuple.Item1} couple's item2: {twoTuple.Item2}"); // Mind the fix property names Item1 and Item2!
// >couple's item1: 2, couple's item2: two
```

20

Tuples in .NET – Part II

- The .NET framework provides another non-generic static class *Tuple*, that only provides simple factories to create *Tuple*s:

Tuple
+ Create<T1>(item1 : T1) : Tuple<T1> + Create<T1, T2>(item1: T1, item2 : T2) : Tuple<T1, T2> + Create<T1, T2, T3>(item1 : T1, item2 : T2, item3 : T3) : Tuple<T1, T2, T3> ...

- These simple factories are generic methods, which allows the compiler to infer the types of the parameters:

```
Tuple<int, string> twoTuple = new Tuple<int, string>(2, "two");
Tuple<string, int, double> triple = new Tuple<string, int, double>("three", 3, .7);
```

```
Tuple<int, string> twoTuple = Tuple.Create(2, "two");
Tuple<string, int, double> triple = Tuple.Create("three", 3, .7);
```

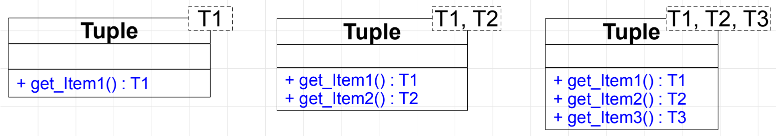
- The simple factories are not only for convenience to avoid writing types explicitly, but to handle *Tuple*s of anonymous types:

```
var records =
        File .ReadLines("/Users/nico/journal.txt")
              .Select((arg, lineNo) =>
                     Tuple.Create(lineNo, new { Value = arg, IsLarge = args.Length > 200 })
              );

foreach (var record in records) {
        Console.WriteLine($"#line: {record.Item1}, record: {record.Item2}");
}
```

Tuples in .NET – Part III

- The class diagram showed, that *Tuple*s only have getter-properties for their components:



- Fair enough, it means we can access/read the components of a *Tuple*N by properties named *Item1* to *Item*N:

```
Tuple<int, string> twoTuple = Tuple.Create(2, "two");
```

```
Console.WriteLine($"Item1: {twoTuple.Item1}, Item2: {twoTuple.Item2}");
// >Item1: 2, Item2: two
```

- *Tuples*' implementation of *ToString()* is also quite useful:

```
Console.WriteLine($"The Tuple: {twoTuple}");
// >The Tuple: (2, two)
```

- However, after a *Tuple* was created, it cannot be structurally changed, we cannot overwrite the values of the components:

```
twoTuple.Item1 = 3; // Invalid! Tuple<int, string>.Item1 cannot be assigned!
twoTuple.Item2 = "three"; // Invalid! Tuple<int, string>.Item2 cannot be assigned!
```

  – => The components of a .NET *Tuple* are immutable!

22

Tuples in .NET – Part IV

- Tuples were added to .NET because the (then) new .NET language F# introduced tuples as genuine concept.
  - F# provides integrated syntactical support for tuples, similar to the support of arrays in other languages:

```
// Creating some tuples with F#' integrated syntax:
let a = (2, 7, 5) // The representation of the spatial vector a. The type is inferred to (int * int * int).
let twoTuple = (2, "two") // A couple. The inferred tuple type is (int * string).
// Accessing a couple's components:
printfn "couple's item1: %A couple's item2: %A" (fst twoTuple) (snd twoTuple)
// >couple's item1: 2, couple's item2: "two"
```

- Back to .NET: Collections of *Tuples* are common sense: Why creating a UDT "car", if we can use a *Tuple* as a "container"?

```
Tuple<int, string>[] cars = new[] { // A car represented by tuples (int Horsepower, string Name)
        Tuple.Create(200, "500 Abarth Assetto Corse"),
        Tuple.Create(130, "Ford Focus 2.0"),
        Tuple.Create(55, "VW Kaffeemühle"),
        Tuple.Create(354, "Lotus Esprit V8"),
        Tuple.Create(112, "Alfa Romeo Giulia ti Super")
};
```

  - It should be underscored that the semantics of "car" is no longer present. Its abstraction was flattened into a (int, string) tuple.
    - A (int, string) tuple could also represent (int phonenumber, string name) or (int postalCode, string street).

- Java provides no dedicated UDTs to represent tuples, but those can be created easily.

23

ValueTuples – Part I

- .NET's *Tuple*s are useful, but bear some downsides:
  - Components can only be accessed by fixed-named properties, such as *Item*N.
  - The naming of the components' properties is relevant to remember, because they reflect the order of the components in the *Tuple*.
  - The names of these properties have no problem-domain-connection to the data they hold: *Item2* is a pretty meaningless name.
  - We can only create 7-tuples in a convenient way, else we have to apply nesting.
  - Although they are "simple" types, they are reference types, which need heap-allocations and other somewhat costly CPU-operations.

- <u>To remedy *Tuple*'s downsides, .NET 7 introduced the value type *ValueTuple*.</u>

- On a first look they don't differ that much from .NET 4's *Tuple*, e.g. as far as instantiation of *ValueTuple*s is concerned:

```
ValueTuple<int, int, int> a = new ValueTuple<int, int, int>(2, 7, 5);
ValueTuple<int, string> twoTuple = ValueTuple.Create(2, "two");
ValueTuple<string, int, double> triple = ValueTuple.Create("three", 3, .7);
```

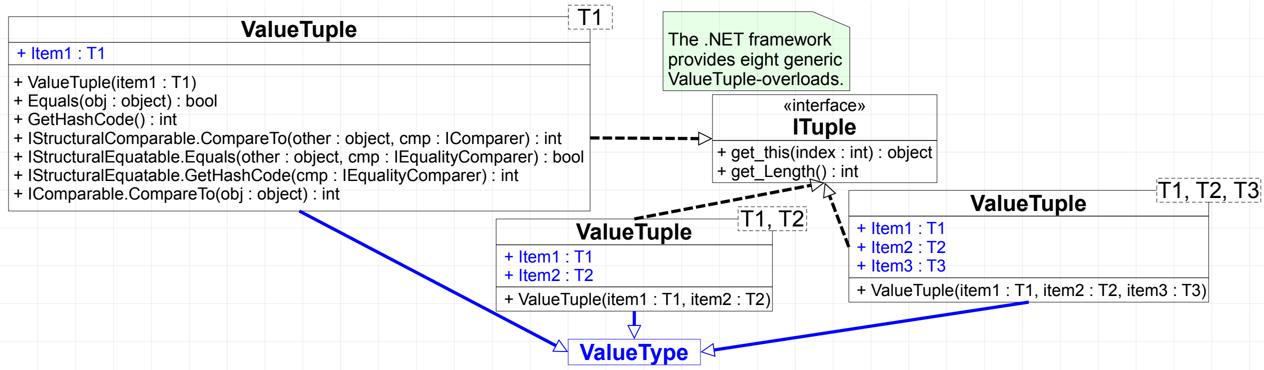
ValueTuples – Part II

- Seen under this perspective, *ValueTuples* are really not so spectacular and similar to *Tuples*:

```
Tuple<int, int, int> a = new Tuple<int, int, int>(2, 7, 5);
```

```
ValueTuple<int, int, int> a = new ValueTuple<int, int, int>(2, 7, 5);
```

- Also *ValueTuple*'s class diagram looks similar to *Tuple*'s:



- But the class diagram unleashes two important differences to *Tuple*:
  - All *ValueTuple*-classes inherit *ValueType*, therefor they are value types, which typically reside on the stack like primitive type objects.
  - A *ValueTuple*'s components are accessible and manipulatable as public fields instead of public getter-properties.

25

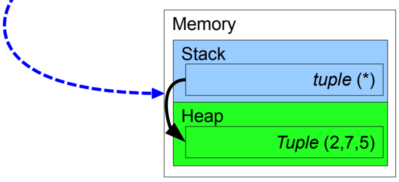
ValueTuples – Part III

- For completeness: there also exists a non-generic static class *ValueTuple*, providing simple factories for type inference:

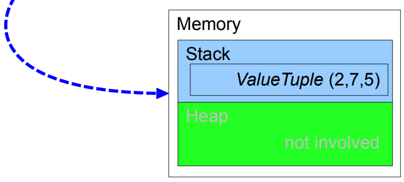
ValueTuple
+ Create() : ValueTuple + Create<T1>(item1 : T1) : ValueTuple<T1> + Create<T1, T2>(item1: T1, item2 : T2) : ValueTuple<T1, T2> + Create<T1, T2, T3>(item1 : T1, item2 : T2, item3 : T3) : ValueTuple<T1, T2, T3> ...

- *ValueTuple*s are value types. If all components of a *ValueTuple* are of value type, the *ValueTuple* fully resides on the stack:

```
Tuple<int, int, int> tuple = Tuple.Create(2, 7, 5);
```




```
ValueTuple<int, int, int> valueTuple = ValueTuple.Create(2, 7, 5);
```



- In opposite to *Tuple*s, *ValueTuple*s can be structurally changed post creation, we can change the values of the components:

```
valueTuple.Item1 = 3; // OK! Just assigning the ValueTuple<int, int, int>.Item1.
valueTuple.Item2 = 4; // OK! Just assigning the ValueTuple<int, int, int>.Item2.
```

  - => The components of a .NET *ValueTuple* are <u>mutable</u> fields!

ValueTuples – Part IV

- C# 7 added special syntactical support for *ValueTuples*:
  - Literals for *ValueTuple*-objects.
  - Literals for constructed *ValueTuple*.
  - Support for structural operations for deconstruction, discarding and pattern matching.

- Let's begin by significantly simplifying the syntax needed for *ValueTuple*-creation.
  - Instead of explicitly calling *ValueTuple*'s ctor or *ValueTuple.Create()*, we can just write a comma-separated list of values:

```
ValueTuple<int, int, int> valueTuple = ValueTuple.Create(2, 7, 5);
```
→
```
ValueTuple<int, int, int> valueTuple = (2, 7, 5);
```

  - The new syntax is so simple, it should not require an extra explanation.

- The next simplification concerns the syntax for the constructed type.
  - Instead of explicitly writing the name of the constructed type with all type arguments we can write a comma-separated list of types:

```
ValueTuple<int, int, int> valueTuple = (2, 7, 5);
```
→
```
(int, int, int) valueTuple = (2, 7, 5);
```

- In upcoming examples we'll stick to the simplest notation we have in avail for *ValueTuples*:

```
ValueTuple<int, int, int> valueTuple = new ValueTuple<int, int, int>(2, 7, 5);
```
→
```
(int, int, int) valueTuple = (2, 7, 5);
```

27

ValueTuples – Part V

- Esp. the dense type-literal can also be used to declare constructed *ValueTuple* as field, return and parameter-types:

```
public class PersonalData {
        private (int, string, string) address;

        public void SetAddress((int, string, string) address) {
            this.address = address;
        }
        public (int, string, string) GetAddress() {
            return address;
        }
}
```

```
PersonalData personalData = new PersonalData();
personalData.SetAddress((34, "Millerstreet", "55-77"));
(int, string, string) address = personalData.GetAddress();
```

- We can use a tuple-like syntax for assigned-to variables to decompose a tuple into its components.
  - E.g. *PersonalData* returns an address, and we want its components be directly represented as distinct variables:

```
// address is a ValueTuple<int, string, string>:
(int, string, string) address = personalData.GetAddress();
Console.WriteLine($"Street name: {address.Item2}");
// > Street name: Millerstreet
```

```
// The ValueTuple<int, string, string> is decomposed into three variables:
(int no, string street, string postalCode) = personalData.GetAddress();
Console.WriteLine($"Street name: {street}");
// > Street name: Millerstreet
```

  - We can also use type inference to let the compiler resolve the components' types by using var:

```
(int no, var street, string postalCode) = personalData.GetAddress();
```

  - As further simplification we let the compiler infer all the components' types by prefixing the component-list with the var keyword.

```
var (no, street, postalCode) = personalData.GetAddress();
```

ValueTuples – Part VI

- If we are only interested in some of the components, we can use a special variable '_' to discard irrelevant components:

```
// Do the discard of two components:
var (_, street, _) = personalData.GetAddress();
Console.WriteLine($"Street name: {street}");
// > Street name: Millerstreet
```

  - The "discard" is no ordinary identifier, it can be used multiply in the same scope. It's type is inferred.

- We can have named components to build "labeled *ValueTuple*s":
  - Here we hold a *ValueTuple* as a single variable, but give the components names, when the *ValueTuple* is assigned:

```
(int no, string street, string postalCode) address = personalData.GetAddress();
Console.WriteLine($"Street name: {address.street}");
// > Street name: Millerstreet
```

- Also when we create an instance of a *ValueTuple*, we can name each component:

```
var address = (no: 34, street: "Millerstreet", postalCode: "55-77");
personalData.SetAddress(address);
```

  - However, the name have no semantics, we can name the components freely to our liking:

```
var address = (a: 34, b: "Millerstreet", c: "55-77");
personalData.SetAddress(address);
```

  - Only the "type-wise" order of components matters, so this won't work:

```
var address = (street: "Millerstreet", no: 34, postalCode: "55-77");
personalData.SetAddress(address);  // Invalid! Cannot convert from '(string
                                   // street, int no, string postalCode)' to '(int,
                                   // string, string)'
```

ValueTuples – Part VII

- We can name the components of a *ValueTuple* everywhere, also in field-, return- and parameter-types:

```
public class PersonalData {
        private (int no, string street, string postalCode) address;

        public void SetAddress((int no, string street, string postalCode) address) {
             this.address = address;
        }
        public (int no, string street, string postalCode) GetAddress() {
             return address;
        }
}
```

  - The idea is to more clearly show the semantics of the components with the names.

- But only the "type-wise" order of components is relevant to the when checking compatibility of *ValueTuple*-instances:
  - We can name the components freely to our liking:

```
var address = (a: 34, b: "Millerstreet", c: "55-77");
personalData.SetAddress(address);
```

  - Only the "type-wise" order of components matters, so this won't work:

```
var address = (street: "Millerstreet", no: 34, postalCode: "55-77");
personalData.SetAddress(address);  // Invalid! Cannot convert from '(string
                                   // street, int no, string postalCode)' to '(int,
                                   // string, string)'
```

30

ValueTuples – Part VIII

- Like we have done it with *Tuple*, we can use collections of *ValueTuples* are common sense: Why creating a UDT "car", if we can use a *ValueTuple* as a "container"?

```
Tuple<int, string>[] cars = new[] { // Each car represented by Tuple (int, string)
        Tuple.Create(200, "500 Abarth Assetto Corse"),
        Tuple.Create(130, "Ford Focus 2.0"),
        Tuple.Create(55, "VW Kaffeemühle"),
        Tuple.Create(354, "Lotus Esprit V8"),
        Tuple.Create(112, "Alfa Romeo Giulia ti Super")
};
```

```
(int, string)[] cars = new[] { // Each car represented by ValueTuple (int, string)
        (horsePower: 200, name: "500 Abarth Assetto Corse"),
        (horsePower: 130, name: "Ford Focus 2.0"),
        (horsePower: 55, name: "VW Kaffeemühle"),
        (horsePower: 354, name: "Lotus Esprit V8"),
        (horsePower: 112, name: "Alfa Romeo Giulia ti Super")
};
```

- Let's again stress that the semantics of "car" is no longer present. Its abstraction was flattened into a (int, string) tuple.
  - A (int, string) tuple could also represent (int phonenumber, string name) or (int postalCode, string street).
  - Yes, we can name the components of a *ValueTuple*, but still only the "type-wise" order of components is relevant.

- The equality of *Tuple*/*ValueTuple* bases on structural equality, objects of equal "type-wise" order and value are equal:

```
// The basic equality-comparison:
bool isEqual = (130, "Ford Focus 2.0").Equals((130, "Ford Focus 2.0"));
// isEqual = true
```

```
// The same values and types, but different order:
bool isEqual = ("Ford Focus 2.0", 130).Equals((130, "Ford Focus 2.0"));
// isEqual = false
```

  - The rules of equality are basically the same as those for anonymous types.
  - But *Tuple*s and *ValueTuple*s cannot be positively compared, even though they are "kind of" structural equal:

```
bool isEqual = Tuple.Create(130, "Ford Focus 2.0").Equals((130, "Ford Focus 2.0"));
// isEqual = false
```

    - For structural equality to work the run-time-type of the objects must be same.

Pattern Matching – Part I

- The syntax defining *ValueTuple*s can be used for pattern matching to match a combination of multiple values:

```
var selectedCars = cars.Where(car =>
            car switch {
                (< 100, _) => true, // Tuple pattern and relational pattern
                (_, "Lotus Esprit V8") => true, // Tuple pattern
                _ => false
            });
foreach (var car in selectedCars) {
        Console.WriteLine(car);
}
// (55, VW Kaffeemühle)
// (354, Lotus Esprit V8)
```

- This way of pattern matching is far more powerful: we can put any values into components of a *ValueTuple*, consider:

```
public class Person {
        public int Age { get; private set; }
        public string Name { get; private set; }

        public Person(string name, int age) {
             Age = age;
             Name = name;
        }
}
```

```
var persons = new[] { new Person("James", 37), new Person("Miranda", 55), new Person("Helen", 22) };
var selectedPersons = persons.Where((Person person) =>
        (person.Age, person.Name) switch { // Deconstruct Person's properties into components and do the match!
             (_, < 30) => true,
             ("Miranda", _) => true,
             _ => false
            });
foreach (var person in selectedPersons) {
        Console.WriteLine($"Name: {person.Name}, Age: {person.Age}");
}
// >Name: Miranda, Age: 55
// >Name: Helen, Age: 22
```

Pattern Matching – Part II

- In a .NET UDT we can also predefine, how this UDT should be deconstructed into a *ValueTuple* whilst pattern matching.
  - Therefor we implement a void-method with the special name *Deconstruct()*:

```
// UDT Person with Deconstruction-overload:
public class Person { // (details hidden)

        public void Deconstruct(out string name, out int age) {
              name = Name;
              age = Age;
        }
}
```

  - *Deconstruct()*-overloads act like "opponents" to ctor-overloads: we specify components of the target-*ValueTuple* as out-parameters in order.

- This way of pattern matching is far more powerful, we can put any values into components of a *ValueTuple*, consider:

```
var selectedPersons = persons.Where((Person person) =>
                person switch { // Automatically deconstructs Person's properties into components and do the match!
                    (_, < 30) => true,
                    ("Miranda", _) => true,
                    _ => false
                });
foreach (var person in selectedPersons) {
        Console.WriteLine($"Name: {person.Name}, Age: {person.Age}");
}
// >Name: Miranda, Age: 55
// >Name: Helen, Age: 22
```

Filling Collections with Data – .NET/C#

- Many languages provide ways to fill collections with new objects in a concise and compact manner. Let's review those.
  - We mainly discuss compact ways to instantiate objects in order to fill collections directly or even literally.

- Anonymous types: Besides filling collections with tuples, C# allows filling collections with instances of anonymous types:

```
Tuple<int, string>[] cars = new[] { // A car represented by tuples
                                    // (int, string)
        Tuple.Create(200, "500 Abarth Assetto Corse"),
        Tuple.Create(130, "Ford Focus 2.0"),
        Tuple.Create(55, "VW Kaffeemühle"),
        Tuple.Create(354, "Lotus Esprit V8"),
        Tuple.Create(112, "Alfa Romeo Giulia ti Super")
};
```

```
var cars = new[] { // A car represented by instances of anonymous types
                   // {int Horsepower, string Name}
        new { Horsepower = 200, Name = "500 Abarth Assetto Corse" },
        new { Horsepower = 130, Name = "Ford Focus 2.0" },
        new { Horsepower = 55,   Name = "VW Kaffeemühle" },
        new { Horsepower = 354, Name = "Lotus Esprit V8" },
        new { Horsepower = 112, Name = "Alfa Romeo Giulia ti Super" }
};
```

- Anonymous types as alternative to .NET *Tuple*s/*ValueTuple*s:
  - Pro: anonymous types have richer semantics, because named properties are part of the type and can be used ad hoc.
  - Contra: It's needed to use type inference (var), which makes code more arcane. Instances can't be returned from or passed to methods.
    - However, we could use dynamic typing.

Filling Collections with Data – Groovy and JavaScript

- Groovy allows defining lists of maps (associative collections). The maps act like bags of properties as fully blown objects:

```
def cars = [ // A car represented by instances of maps
             // [Horsepower : Integer, Name : String]
    [ HorsePower : 200, Name : "500 Abarth Assetto Corse" ],
    [ HorsePower : 130, Name : "Ford Focus 2.0" ],
    [ HorsePower : 55,  Name : "VW Kaffeemühle" ],
    [ HorsePower : 354, Name : "Lotus Esprit V8" ],
    [ HorsePower : 112, Name : "Alfa Romeo Giulia ti Super" ]
]
```

- In JavaScript, we have true dynamic typing, in which we don't need specific types. We can just write object literals directly.
  - Here we have an array holding five object literals representing cars:

```
var cars = [ // A car represented by object literals
             // {Horsepower, Name}
       { HorsePower : 200, Name : "500 Abarth Assetto Corse" },
       { HorsePower : 130, Name : "Ford Focus 2.0" },
       { HorsePower : 55,  Name : "VW Kaffeemühle" },
       { HorsePower : 354, Name : "Lotus Esprit V8" },
       { HorsePower : 112, Name : "Alfa Romeo Giulia ti Super" }
]
```

- Java does not yet provide simple means to freely define instances (to be used for filling collections).
  - Following the Java philosophy a suitable JVM language (like Scala or Groovy) should be used.

Filling Collections – do it in Bulk

- Collections can also be created along with a <u>bulk of data</u>, which can yield <u>performance benefits</u> due to efficient capacitation:
  - (Not all platforms apply capacity control on bulk operations!)

```
// Java; instead of adding items to a list with a loop...
List<Integer> numbers = new ArrayList<>();
for (int item : new int[]{1, 2, 3, 4}) {
	numbers.add(item);
}
```

```
// … the list could be initialized with another "ad hoc" collection like an array:
List<Integer> numbers2 = new ArrayList<>(Arrays.asList(1, 2, 3, 4));
```

```
// C#; instead of adding items to a list with a collection initializer or loop...
IList<int> numbers = new List<int>{1, 2, 3, 4};

IList<int> numbers2 = new List<int>();
foreach (int item in new []{1, 2, 3, 4}) {
	numbers2.Add(item);
}
```

```
// … the list could be initialized with another "ad hoc" collection like an array:
IList<int> numbers3 = new List<int>(new []{1, 2, 3, 4});
```

- Often collections can also be <u>filled with items in bulk after creation</u>:

```
// Java; adding a couple of items to a list in bulk:
List<Integer> numbers = new ArrayList<>();
numbers.addAll(Arrays.asList(1, 2, 3, 4))
// Alternatively Collections.addAll() can be used (its usually faster, but the
// second argument needs to be an array (!)):
Collections.addAll(numbers, 1, 2, 3, 4);
```

```
// C#; adding a couple of items to a list in bulk:
List<int> numbers = new List<int>();
// (Mind that AddRange() isn't available via the interface IList!)
numbers.AddRange(new []{1,2,3,4});
```

36

- Java's *Collections.addAll()* is faster than *Collection.addAll()* because the latter creates an extra array. But *Collections.addAll()* only works for adding the contents of an array to another collection!

Excursus: Observable Collections

- Java provides collections, which can report structural changes on items. This is esp. interesting in GUI-applications.

- Those collections can be found in package *javafx.collections*, which is part of module *javafx.base*.
  - JavaFX is a modern GUI-framework to create platform-independent GUI-application in Java.
  - Such collections carry the prefix *Observable*: *ObservableArray<T>*, *ObservableList<E>*, *ObservableMap<K, V>*, *ObservableSet<E>*
  - These are only interfaces: implementations are hidden and instances must be created by simple factories from class *FXCollections*:

```
ObservableList<Integer> observableList = FXCollections.observableArrayList();
observableList.add(10);
observableList.add(20);
observableList.add(30);
```

- The more interesting part is, that we can register a *ListChangeListener* to get notifications, when *observableList* changes:

```
observableList.addListener((ListChangeListener<Integer>) change -> {
        while (change.next()) {
            System.out.printf("Change observed: %s%n", change);
        }
});

observableList.add(23);
// >{ Change observed: [23] added at 3 }
```

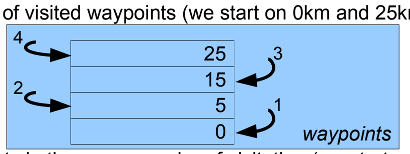
  - As can be seen, the *ListChangeListener* is also called back, when bulk-changes were performed (notice the while-loop).

Collections abstracting Data Processing – Stack

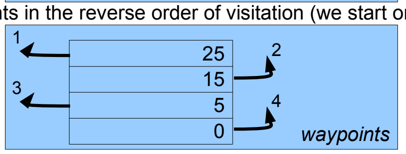
- There exist collections that directly represent ways how data is processed in computing whilst holding the data.
  - We'll discuss the collections stack and queue.
  - (The heap is also a collection of this category, but we'll ignore it in this course.)

- The idea of a stack is to take the put-in items out in the reverse order. This principle is called Last-In-First-Out (LIFO).

- E.g. assume that we want to put waypoints on a hike and we want to reach the waypoints again when we hike back.
  - After we hiked there, we get following stack of visited waypoints (we start on 0km and 25km was the last visited waypoint):



  - When we hike back, we pick up the waypoints in the reverse order of visitation (we start on the last waypoint visited and drill down):



- Now we'll put this into code.

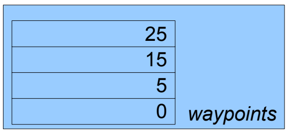
38

- The stack as it is used in a CPU is of course an implementation of "stack". But the CPU's stack is if you will an array of fix length and push/pop operations just move the stackpointer around. – So instead of explicitly freeing memory, the stack pointer moves and leaves the stack's slots alone, which are no longer of relevance.

Collections abstracting Data Processing – Stack – fundamental Operations

- Let's discuss the .NET collection *Stack*:
  - The method *Push()* pushes items on the stack, it corresponds to an "add-item" operation:

```
// Create a Stack and push four waypoint km-marks on it:
Stack<int> waypoints = new Stack<int>(4);
waypoints.Push(0);
waypoints.Push(5);
waypoints.Push(15);
waypoints.Push(25);
```



  - The method *Pop()* pops items from the stack (LIFO), it corresponds to a "remove-item" operation:

```
int lastWaypoint = waypoints.Pop();
// >lastWaypoint evaluates to 25, because it was put in last
```

  - The method *Peek()* returns the last pushed item, but doesn't pop it from the stack.

```
int penultimateWaypoint = waypoints.Peek();
// >penultimateWaypoint evaluates to 15, because it was put in penultimately
```

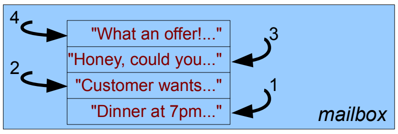
  - If *Pop()* or *Peek()* are applied on an empty *Stack* an *InvalidOperationException* will be thrown.

- Stack collection types on other platforms: C++ STL provides *std::stack* (<stack>) and Java provides *Stack*.
  - Java's *Stack* is derived from *Vector*, which a synchronized list, if sync is not needed use *ArrayQueue<T>* as unsynchronized stack.

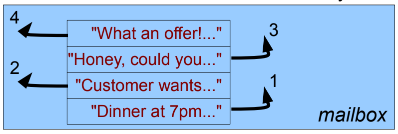
Collections abstracting Data Processing – Queue

- The idea of a queue is to take the put-in items out in the same order. This principle is called First-In-First-Out (FIFO).

- E.g. assume that we want to read emails in the order they receive the mailbox.
  - The emails are received in this order:



  - When we're going to read the emails we read them in the same order, in which they have been received:

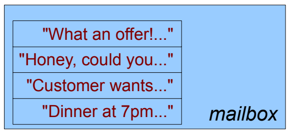


- Now we'll put this into code.

Collections abstracting Data Processing – Queue – fundamental Operations

- Let's discuss the .NET collection *Queue*:
  - The method *Enqueue()* enqueues items into the queue, it corresponds to an "add-item" operation:

```
// Create a Queue and enqueue four emails in receipt order:
Queue<string> mailbox = new Queue<string>(4);
mailbox.Enqueue("Dinner at 7pm...");
mailbox.Enqueue("Customer wants...");
mailbox.Enqueue("Honey, could you...");
mailbox.Enqueue("What an offer!...");
```



  - The method *Dequeue()* dequeues items from the queue (FIFO), it corresponds to a "remove-item" operation:

```
string firstEmail = mailbox.Dequeue();
// >firstEmail evaluates to "Dinner at 7pm...", because it was put in at first
```

  - The method *Peek()* returns the last enqueued item, but doesn't dequeue it from the queue.

```
string secondEmail = mailbox.Peek();
// >secondEmail evaluates to "Customer wants...", because it was put in at second
```

  - If *Dequeue()* or *Peek()* are applied on an empty *Queue* an *InvalidOperationException* will be thrown.

- Queue collection types on other platforms: C++ STL provides *std::queue*/*std::priority_queue* (both in <queue>) and *std::deque* (<deque>). Among others, Java provides *LinkedList* as implementation of the interface *Queue*.

Stack vs Queue

- The general difference between stack and queue is basically only the LIFO (stack) vs the FIFO (queue) concept.

- A table showing the complexity of essential operations unleashes this similarity.

	Accessing	**Searching**	**Insert**	**Delete**
Stack	O(n)	O(n)	O(1)	O(1)
Queue	O(n)	O(n)	O(1)	O(1)

  – Just the operations insert/delete are either effective on only the first or last element respectively as enqueue/dequeue or push/pop.

Common behavior of Collections and Maps in Java

- <u>*Collection*s and *Map*s offer @*Override*s of *toString()*</u>, which <u>call *toString()* of each item</u> with a nice *String*-representation:

```
Map<String, Integer> map = Map.of("A", 1, "B", 2, "C", 3);
String mapAsString = map.toString();
// mapAsString = {C=3, B=2, A=1}
```

```
List<String> list = List.of("A", "B", "C");
String listAsString = list.toString();
// listAsString = "[A, B, C]"
```

  - These @*Override*s are provided by the classes *AbstractCollection* and *AbstractMap*.
  - These @*Override*s are recursion-proof: if a raw-type *List* contains itself as element, *AbstractCollection.toString()* won't recurse:

```
List list = new ArrayList();
list.addAll(List.of("A", "B", "C", list)); // <- adds the List as element of itself
String recursiveListAsString = list.toString();
// recursiveListAsString = "[A, B, C, (this Collection)]"
```

- <u>*Collection*s and *Map*s offer @*Override*s of *equals()*</u>, which allow equality-comparing *Collection*s and *Map*s of related types.
  - It <u>sounds</u> not spectacular, but it means, that <u>we can equality-compare any two objects, which only implement *List*, *Set* or *Map*</u>:

```
List<String> listA = new ArrayList<>(List.of("A", "B", "C"));
List<String> listB = new LinkedList<>(List.of("A", "B", "C"));
boolean areEqual = listA.equals(listB);
// areEqual = true
```

  - <u>Usually</u>, the JDK implements the equals contract so that the types of the objects to be compared <u>must have the same dynamic type</u>.
    - Also own UDTs should usually work this way <u>to avoid symmetry-problems with the equals contract</u>.
  - However, this is not the case for *List*, *Set* or *Map*: <u>The objects to be compared must only both implement *List* or *Set* or *Map*.</u>
  - <u>=> This is a unique implementation of the equals contract in the JDK.</u>

Kung Fu beyond Collections – Part I

- Up to now we've mentioned functional programming (fp) languages a lot in the context of collections.
  - The interesting point is that those languages deal with data in a special way: data is generally immutable after creation.
  - The lack of mutability, which is fundamental for, e.g. imperative languages, leads to fps having mighty features for data creation:
    - Support of immutability, list comprehensions, generic types, type inference.
  - The "main stream" languages are going to adopt or have already adopted these features as new language idioms.

- Why are these adopting fp idioms?
  - The most important point is more productivity whilst programming.
  - A secondary aspect is that immutable types makes writing threadsafe a piece of cake.
    - Modern code should exploit the availability of parallel computing power today.

- You should know or learn about these features! They are real game changers for programmers!

- Let's rewrite an imperative code snippet that produces a new collection from a present collection.

Kung Fu beyond Collections – Part II

- We have a list of numbers and we want to filter the odd numbers out to get only the even numbers, the classical way:

```
// C#
IList<int> numbers = new List<int>{ 1, 2, 3, 4, 5, 6, 7, 8, 9 };
IList<int> evenNumbers = new List<int>();

foreach (int number in numbers) {
        if (0 == number % 2) {
            evenNumbers.Add(number);
        }
}
```

```
// Java
List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9);
List<Integer> evenNumbers = new ArrayList<>();

for (int number : numbers) {
        if (0 == number % 2) {
            evenNumbers.add(number);
        }
}
```

  - In both snippets we have a couple of statements, esp. a loop, a branch and adding elements to an initially empty collection, which will be filled. We already discussed this idea, when list comprehensions were presented: here we have sophisticated list compr.

- Now we'll rewrite the snippets using fp features of C# and Java (and Groovy and Scala):

```
// C#
IList<int> numbers = new List<int>{ 1, 2, 3, 4, 5, 6, 7, 8, 9 };
IList<int> evenNumbers = (    from number in numbers
                              where 0 == number % 2
                              select number).ToList();
// >{2, 4, 6, 8}
```

```
// Java
List<Integer> evenNumbers =
        Stream  .of(1, 2, 3, 4, 5, 6, 7, 8, 9)
                .filter(number -> 0 == number%2)
                .collect(Collectors.toList());
// >{2, 4, 6, 8}
```

```
// Groovy
def numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]
def evenNumbers = numbers.grep{0 == it % 2}
// >[2, 4, 6, 8]
```

```
// Scala
def numbers = List(1, 2, 3, 4, 5, 6, 7, 8, 9)
def evenNumbers = numbers.filter(number => 0 == number % 2)
// >List(2, 4, 6, 8)
```

Kung Fu beyond Collections – Part III

- What we saw in action were .NET's LINQ and Java's *Streams* API.

```
// C#/.NET's LINQ:
IList<int> numbers = new List<int>{ 1, 2, 3, 4, 5, 6, 7, 8, 9 };
IList<int> evenNumbers = (  from number in numbers
                            where 0 == number % 2
                            select number).ToList();
// >{2, 4, 6, 8}
```

```
// Java's Streams:
List<Integer> evenNumbers =
        Stream  .of(1, 2, 3, 4, 5, 6, 7, 8, 9)
                .filter(number -> 0 == number%2)
                .collect(Collectors.toList());
// >{2, 4, 6, 8}
```

  - In both cases we don't have a couple of statements, but rather single expressions describing the results of the operations.
  - => .NET's LINQ and Java's streams support a declarative programming style instead of an imperative programming style.
  - The declarative style, hiding the internals (internal iteration) makes declarative parallelization doable.

- We'll not discuss LINQ and *Stream*s in this lecture, because there are some advanced idioms to understand:
  - Generic types in depth
  - Type inference
  - Lambda expressions
  - Fluid interfaces and the "pipes and filters" architectural style

46

- Java's *Stream*s API makes an interesting distinction between (old) outer loops and the internal loops of *Stream*s.
- LINQ and *Stream*s put it to extremes: there needs not even to be a solid collection to do operations on it!

Not Discussed Topics

- Concurrent collections
- Collections and memory consumption
- Collections and amortized costs for different operations

Thank you!