Collections – Part II

Nico Ludwig (@ersatzteilchen)

TOC

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!

The Collection Story

- A very important topic in programming besides algorithms are collections of data.
  - Virtually, data collections were present before algorithms: computing dealt with mass data from the beginning (business data).
  - The so called "data processing" was applied to predict the results of the US presidential elections in 1952 for the first time basically.
- Terminology alert: Collections are called containers in C++.
- Nowadays we have to deal with huge amounts of data in databases.
- Back to programming with data:
  - Data is held in objects. – "Object" is not a term dedicated to oo-languages, it could be a Pascal RECORD instance as well.
  - Understanding and using collections is key for the art and craft of programming apart from algorithms.
  - In most programming languages (languages) we already have a first class citizen to deal with collections of data: arrays!
- Let's begin our discussion of collections with arrays, or more precisely, collections following the concept of an array.
- Terminology alert for German programmers: Stick to calling an array array, not "Feld", even though German literature does!
  - A Feld is a field of a UDT! This is an example where translation is really inappropriate, leading to ridiculous misunderstandings.

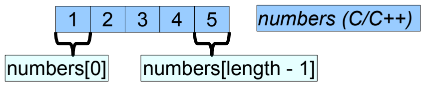
- In 1952 the UNIVAC I (UNIVersal Automatic Computer I) was used for the prediction for the presidential election.
- Mind that compiler and run time usually also always use the term "array". Either they are themselves only printing English messages, or they print German messages, but stick to special terms. – The word "array" is a special term, not a word that needs to be translated!

Arrays – Part I

- Generally, we call a collection an array if following statements hold true:
  - The collection has the semantics of a table.
  - The "rows" of that table, the elements or "slots", can be accessed with a numeric "row number", the so called index.
    - Array indexes usually start at 0 (0-based indexing), the index referencing the last element boils down to length - 1. In Basic, arrays use 1-based indexing.
  - The elements of an array can be accessed randomly with O(1) complexity. – This is very fast!
  - (The elements of an array usually have the same static type, C++ calls this the element type.)
- In many languages arrays have integrated syntactical support, e.g. in Java, C++, C#, VB, JavaScript etc.
  - Creation of arrays with initializer lists and the []-notation:

```
// C++
int numbers[] = {1, 2, 3, 4, 5};
```

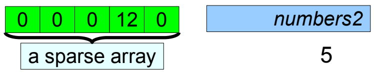
```
// Java
int[] numbers = {1, 2, 3, 4, 5};
int[] numbers2 = new int[5];
```



    - Many languages support an initialization syntax for arrays.
    - In C++ automatic arrays require to be created with a constant length known at compile time.
    - Other languages like Java and C# create arrays only on the heap, and therefor these languages also accept non-constant lengths for arrays.
    - Dynamically created arrays (Java/.NET-heap or C++-freestore) can have the size 0, automatic arrays can't.
  - Accessing and manipulating array elements by accessing them with the index/subscript/[]-operator (C++, Java, C#):

```
// Reading the element at index 3:
int aNumber = numbers[3];
```

```
// Writing the value 12 to the element at index 3:
numbers2[3] = 12;
```



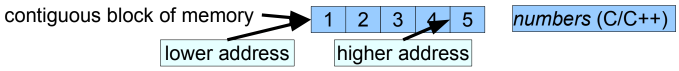
5

    - The array *number2* contains only elements of value 0 with the exception of the element at index 3. Arrays having "gaps" are called sparse arrays.

- The .NET framework allows to create arrays with index-bases (i.e. lower bounds) different from 0 with the method *Array.CreateInstance()*. This is also possible for multidimensional arrays.
- Honorable mention: Fortran's arrays use 1-based indexing by default. However, it also supports to create arrays with other indexing-bases by defining lower and upper bounds.

Arrays – Part II

- Support of arrays in various languages:
  - Arrays are often the only collection that is integrated into a language, i.e. no extra library imports are required to use them.
    - In C/C++ the array elements reside in a contiguous block of memory, which enables the fundamental concept of pointer arithmetics in C/C++



    - Therefor C++ requires array elements to be of the same static type because then all elements have the same size to make pointer arithmetics work!
  - Arrays have a fixed length (i.e. the count of elements) after creation that can't be changed. – We can not add or remove elements.
    - This is not the case for JavaScript arrays (also called "array-like objects")!
  - Arrays can manage objects of dynamic or static types depending on the support of the language.
    - Statically typed languages (e.g. C++) allow only array elements of the same static type:

```
// C++
int numbers[5]; // numbers can only store five ints
Car parkingBay[100]; // parkingBay can only store 100 Car instances
```

    - Dynamically typed languages (e.g. JavaScript) allow individual array elements to be of any type:

```
// JavaScript
var objects = [5, "five", {x: 34, y: 4.4}]; // objects contains an integer, a string and an object
```

  - Arrays are a primary construct of imperative languages. In opposite to immutable lists used in fp languages.

6

- E.g. F# as a functional programming language does have an extra ugly syntax for array-creation, -accessing and -manipulation; F#'s arrays are mutable! So, F# clearly states immutable collections (i.e. F# lists) being more primary than arrays!

Arrays – Part III

- Arrays can be n-dimensional (multidimensional) in many programming languages.
  - N-dimensional arrays can be used to build matrix data structures.

$$A=\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$$

  - 2-dimensional arrays can be seen as an abstraction of tables having more than one column. 3-dimensional arrays are cubes, higher dimensional ones are hypercubes.
  - Terminology alert: sometimes one-dimensional arrays are called vectors!

C# – leaving away the element type name on creating Anonymous arrays with array initializers

```
// Assume we have a method with a string-array parameter:
private void AwaitsAStringArray(string[] strings);
// Pass a newly created array with an array initializer:
AwaitsAStringArray(new string[]{ "Monica", "Rebecca", "Lydia" });
// Alternatively leave the type name away on the new keyword:
AwaitsAStringArray(new[]{ "Monica", "Rebecca", "Lydia" });
```

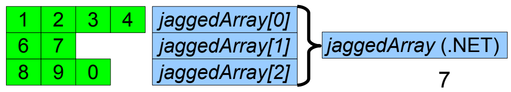
- Languages may support two kinds of n-dimensional arrays:
  - Rectangular n-dimensional arrays. Here a 2-dimensional 3x3 matrix (one big object):

```
// C#
int[,] rectangularArray = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}}; // Represents a 3x3 matrix.
```

$$rectangularArray=\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

  - Jagged n-dimensional arrays are typically build as arrays of arrays (some individual arrays):

```
// C#
int[][] jaggedArray = new int[3][];
// Set the values of the sub-arrays in the jagged array structure:
jaggedArray[0] = new[] {1, 2, 3, 4};
jaggedArray[1] = new[] {6, 7};
jaggedArray[2] = new[] {8, 9, 0};
```



7

- The mathematical numeration of matrix elements is row-major, i.e. the first index is the row's index and second index is the column's index.
- .NET's framework design guidelines suggest using jagged arrays instead of rectangular arrays. The reasoning behind this is the low propability of wasted space, when arrays get sparse and the CLR is able to optimize index access to jagged arrays better than for rectangular arrays.
- Rectangular arrays can consume very much memory if it has many dimensions (even if the array is not filled).
- Jagged arrays can also be defined in Java and Groovy.

Arrays – Part IV

- We should also talk about naming.

- Usually the identifier of array objects reflect the "multiplicity of something". This is called identifier pluralization.
  - E.g. a double array storing several wages could just be called *wages*.

```
// C#
// "wages" is just an array storing multiple wages.
double[] wages = {petersWage, christinasWage, nicosWage};
double allWagesToPay = .0;
for (double wage in wages) {
	allWagesToPay += wage;
}
```

- A variable denoting the count of elements in an array is commonly accepted to be a candidate for a variable prefix: 'n'.
  - When *wages* denotes an array of, well, wages, it seems logical to have another variable denoting count of elements in that array with the count of numbers, or in short ***nWages***.

```
// C#
// "nWages" represents the count of elements in the array wages.
int nWages = wages.Length;
Console.WriteLine($"We have to pay {allWagesToPay} of wages for {nWages} persons");
```

Honorable mention: C's variable length arrays (VLAs)

- Honorable mention: C's variable length arrays (VLAs)
  - Since C99, C supports painless creation of arrays of variable length:

```
// >= C99 VLA example
void create_VLA(size_t length) {
	int myArray[length]; // Yes, myArray resides on the stack!
	myArray[length - 1] = 6;
	printf("%d", myArray[length - 1]); // prints a 6
	const size_t actualLength = sizeof(myArray)/sizeof(int);
	assert(length == actualLength);
}
```

  - Because VLAs are created on the stack in most cases, i.e. w/o *malloc()* and *free()* creation is super fast!
  - Further benefit: having no compile time constants for array lengths closes a source of serious bugs.
  - The downside: VLAs cannot be resized. The standard disallows VLAs of size 0.

- C# kind of supports VLA using stackalloc and pointers, but this an unsafe feature.

- (C99 also supports flexible array members for structs.)

- VLAs are currently not supported by C++ (2019).

Arrays – Argument List of variable Length

- Java/.NET support argument lists of variable length, which are based on dynamically created arrays.

```java
// Java
public static void variableArguments(String... args) {
	assert String[].class == args.getClass();
	for (final String arg : args) {
		System.out.print(" " + arg);
	}
}
```

```java
variableArguments("Hello", "World, ", "see", "my", "collections!");
// >Hello World,  see my collections!
```

```csharp
// C#
public static void VariableArguments(params string[] args) {
	System.Diagnostics.Debug.Assert(typeof(string[]) == args.GetType());
	foreach (string arg in args) {
		Console.Write(" " + arg);
	}
}
```

```csharp
VariableArguments("Hello", "World, ", "see", "my", "collections!");
// >Hello World, see my collections!
```

  - Here, we see Java's ellipsis-operator and .NET's *ParamArrayAttribute* (applied with params keyword in C#) in action.
  - Both idioms create a dynamic one dimensional array on the heap storing the passed arguments.
  - Both idioms support an arbitrary list of parameters in front of the variable parameters parameter.
  - Both idioms disallow any explicitly declared parameter after the variable parameters parameter.
  - In case no arguments are passed, the created array just has a length of 0.

- Also constructed arrays can be passed as a single argument to a method with variable argument list:

```java
VariableArguments(
	new String[]{ "Hello", "World, ", "see", "my", "collections!" });
// >Hello World, see my collections!
```

```csharp
VariableArguments(new []{ "Hello", "World", "see", "my", "collections!" });
// >Hello World, see my collections!
```

10

- Because arrays are handled covariantly in Java and C#, we can have a method with a variable argument list of Object/object and pass anything derived from Object/object.

Excursus: Variable Length Argument Lists in C

- Esp. C is well known for its feature of functions, which can cope with variable argument lists (vargs):

```
#include <stdarg.h>
// C variadic function example
int sum(int nNumbers, ...) {
	int sum = 0;
	va_list args;
	va_start(args, nNumbers);
	for (int i = 0; i < nNumbers; ++i) {
		nSum += va_arg(args, int);
	}
	va_end(args);
	return sum;
}
```

```
const int full_sum = sum(3, 1, 2, 3);
// >6
```

Good to know
All standard C/C++ functions have the calling convention __cdecl. Only __cdecl allows variable argument lists, because only the caller knows the argument list and only the caller can then pop the arguments. __stdcall functions execute a little bit faster than __cdecl functions, because the stack needs not to be cleaned on the callee's side (i.e. within a __stdcall function).

  - Featured by the ubiquitous function *printf()*, applied via the ...-operator (ellipsis-operator).
  - The mandatory vargs' first argument must be interpreted, to guess how many vargs follow.
  - Vargs are harmful: It's a way to introduce security leaks through stack overruns. (Just call *sum(**4**, 1, 2, 3)* and see what happens.)
- How does it work? The vargs-features works very near the metal:
  - The compiler calculates the required stack depending on the arguments and decrements the stack pointer by the required offset.
  - As arguments are laid down on the stack from right to left, *nNumbers* is on offset 0.
  - Then *nNumbers* is analyzed and the awaited offsets are read from the stack. Here an offset of, e.g., 4B for each int passed to *sum()*.

11

- The calling convention __cdecl is a C/C++ compiler's default, __stdcall is the calling convention of the Win32 API, because it works better with non-C/C++ languages. __cdecl requires to prefix a function's name with an underscore when calling it (this is the exported name, on which the linker operates). A function compiled with __stdcall carries the size of its parameters in its name (this is also the exported name). – Need to encode the size of bytes or the parameters: If a __cdecl function calls a __stdcall function, the __stdcall function would clean the stack and after the __stdcall function returns the __cdecl function would clean the stack again. - The naming of the exported symbol of __stdcall functions allow the caller to know how many bytes to "hop", because they've already been removed by the __stdcall function. Carrying the size in a function name is not required with __cdecl, because the caller needs to clean the stack. - This feature allowed C to handle variadic functions with __cdecl (nowadays the platform independent variadic macros can be used in C and C++).
- Other calling conventions:
  - pascal: This calling convention copies the arguments to the stack from left to right, the callee needs to clean the stack.
  - fastcall: This calling convention combines __cdecl with the usage of registers to pass parameters to get better performance. It is often used for inline functions. The callee needs to clean the stack. The register calling convention is often the default for 64b CPUs.
  - thiscall: This calling convention is used for member functions. It combines __cdecl with passing a pointer to the member's instance as if it was the leftmost parameter.
- In this example the RV (EAX on x86) register can only store values of 4B. In reality the operation can be more difficult.
  - For floaty results the FPU's stack (ST0) is used.
  - User defined types (e.g. structs) are stored to an address that is passed to the function silently.
  - It is usually completely different on micro controllers.

Multidimensional Arrays in C/C++

- C/C++ allow the definition of arrays that act like "n-dimensional" arrays.
  - "N-dimensional" arrays are equivalent to "normal" arrays in C/C++.
    - I. e. the memory layout is equivalent for both. N-dimensional arrays have no genuine concept in C/C++!
  - C/C++ provide alternative syntaxes for defining and accessing "mimicked" n-dimensional arrays.
  - The definition/initialization syntax differs, however.
    - Creation and memory layout:

```
// Creates a "normal" 6-int-array in C++:
int array[6] = {1, 2, 3, 4, 5, 6};
```

```
// Creates an "n-dimensional" 2x3-int-array in C++:
int mArray[2][3] = {{1, 2, 3}, {4, 5, 6}};
```

1	2	3	4	5	6	*array*
1	2	3	4	5	6	*mArray*

    - The memory layout of C/C++ arrays is done by row-major order, in which the rows are laid down in linear memory after another.
    - The elements of multidimensional arrays in C/C++ are accessed with multiple applications of the []-operator:

```
for (int i = 0; i < 6; ++i) {
	array[i] = 0; // Uses i as index.
}
```

```
for (int i = 0; i < 2; ++i) { // 1. "dimension" (columns)
	for (int j = 0; j < 3; ++j) { // 2. "dimension" (rows)
		mArray[i][j] = 0; // Uses i and j as "coordinates".
	}
}
```

- The way C/C++ arrange n-dimensional arrays is critical for optimizations in the CPU's cache and vectorization.
  - (But to gain maximum performance, developers have to access elements in a special order.)
  - Closely related is the performance gain when using the GPU to process large amounts of data to relief the CPU.
  - You should notice that n-dimensional arrays are no topic for application programming, but for high performance computing.

12

- Data vectorization means that blocks of data, such as arrays, are not manipulated element-wise in loops, but manipulated as a whole. CPUs provide special instructions to apply vectorization.

Value and Reference Semantics of Elements

- An important point we've to clarify, is, whether collections hold <u>copies or references to "contained" objects</u>.
  - <u>We have to understand, how the language we use handles this.</u>
  - Do the collection and the contained elements <u>share the same lifetime or not</u>?

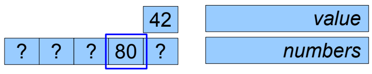
<u>C# – automatically implemented properties</u>

```
public class Car {
    // Power's backing field will be created automatically!
    public int Power {get; set;}
}
```

- If a language defines <u>value semantics for contents of variables</u>, arrays will hold <u>copies</u>.
  - E.g. an array of int in C/C++:

```
// C++
int value = 42;
int numbers[5];
numbers[3] = value; // Copies the content of value.
```

```
numbers[3] = 80; // Doesn't modify value.
```



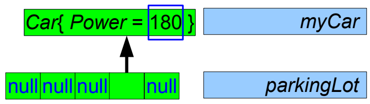
<u>C# – object initializers</u>

```
// Create a Car object and set the property
// Power:
Car car = new Car();
car.Power = 180;
// Alternatively create a Car object and set
// the property Power within one expression
// as object initializer:
Car car2 = new Car{ Power = 180 };
```

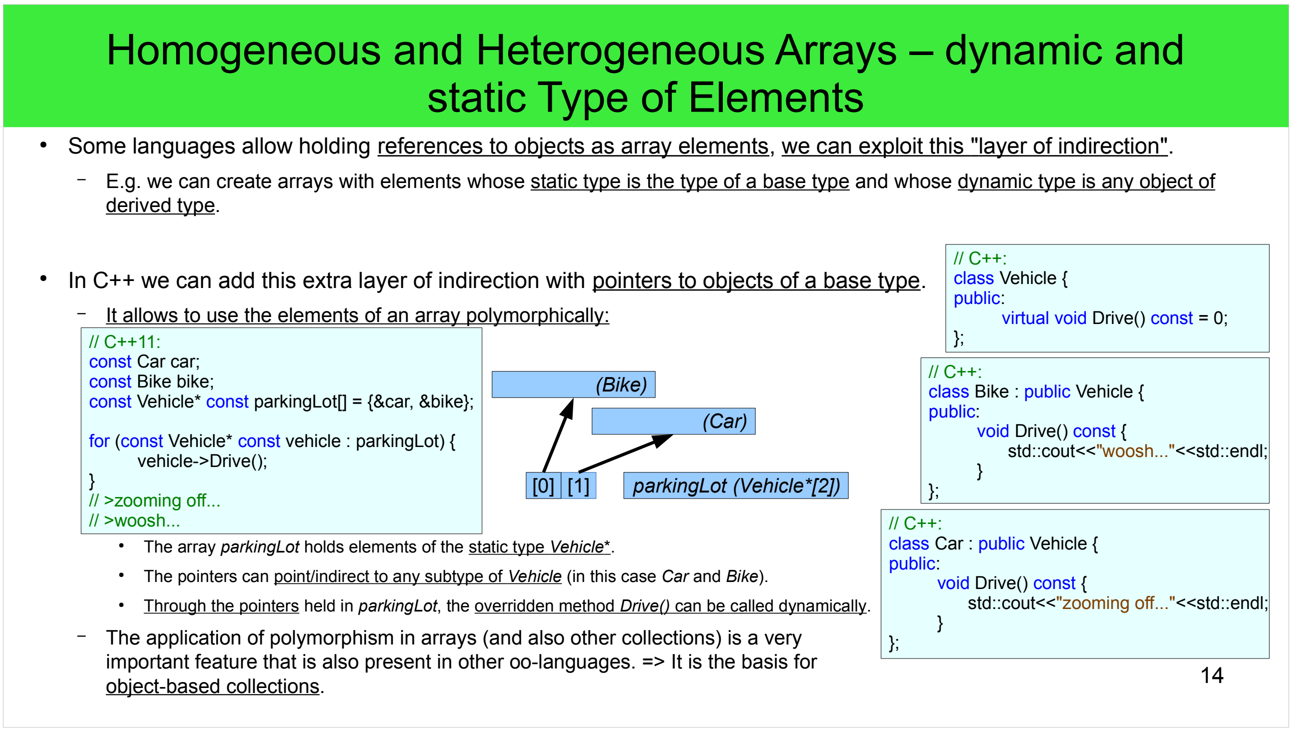
- If a language defines <u>reference semantics for contents of variables</u>, arrays will hold <u>references</u>.
  - E.g. an array of reference type (the class *Car* in this example) in C#:

```
// C#
Car myCar = new Car{ Power = 120 };
Car[] parkingLot = new Car[5];
parkingLot[3] = myCar; // Copy a reference to myCar.
```

```
parkingLot[3].Power = 180; // Will also modify myCar!
```



<u>Good to know</u>
Think:
value semantics: copy,
reference semantics: shared ownership



- In most languages the array elements need to be of the same static type. JavaScript arrays can be really heterogeneous, but JavaScript does only have dynamic types altogether...

Shortcomings of Arrays

- Most languages have arrays that don't expose their length. So programmers have to pass the array's length separately.
  - More modern languages/platforms do have arrays that expose their length, e.g. Java, .NET and JavaScript, but not C/C++!.
  - (C++11 provides the STL wrapper type *std::array* for working with arrays with compile time known size.)
- Esp. in C/C++ arrays are directly associated to physical memory. This is a further source of potential bugs and problems.
  - Therefor higher level C++ STL container types should be used.
- The length (i.e. the count of elements) of an array is fixed.
  - Not so in JavaScript.
- .NET's framework design guidelines suggest using "genuine" collections instead of arrays in public interfaces.

Platform-agnostic Categorization of Collections

- Object-based vs. generic
- Indexed, sequential vs. associative
  - Terminology alert: The C++ STL tells sequential (those are indexed and sequential containers) from associative containers.
  - Terminology alert: Java tells *Collection*s (sequential collections (and "somehow" indexed collections also)) from *Map*s.
- Ordered or unordered
- Mutable or readonly
- Synchronized or unsynchronized

Indexed Collections – Part I

- We start with discussing indexed, sequential and associative collections, beginning with indexed collections.
- Indexed collections have following basic features:
  - Elements can be accessed and modified via an index number (0-based or not).
  - Elements have a defined order.
  - Elements can be randomly accessed (usuallay with O(1) complexity – this is very fast).
- Ok, these basic features do just describe arrays as collections, but indexed collections have more features:
  - The collection exposes the count of elements it holds, (i.e. length; with 0-based indexes the last index would be length - 1).
  - Elements can be added or removed after creation of an indexed collection.
    - The length of an indexed collection can grow or shrink during run time!
- Esp. in C/C++, arrays are a major source of problems, indexed collections help, because they expose their length.

17

- Java provides the marker interface *RandomAccess*, which indicates collections that provide random access.

Indexed Collections – Part II

- In modern languages we can find following indexed collections beyond arrays:
  - "Genuine" lists or vectors
  - Deques (singular deque, pronounced [dɛk], for double ended queues)

- Strings are a special kind of indexed collection in many languages. In C/C++ c-strings are just arrays.
  - On some platforms (Java, .NET) strings act as readonly indexed collections.
    - I.e. the elements of strings (the characters) can't be modified and elements can't be added or removed to strings.
    - String operations will not modify the original string, but a new string with a content different from the original string will be created. This is called the "defense copy pattern".
  - The above mentioned platforms do also define mutable string types.
    - Those encapsulate an original string object and allow accessing it with an interface providing mutating operations.
    - Examples: *StringBuilder* (.NET/Java), *StringBuffer* (Java), *NSMutableString* (Cocoa), *std::string* (C++).
    - (To make working with string-like types simpler in Java, *String*, *StringBuffer* and *StringBuilder* implement the interface *CharSequence*.)

Indexed Collections and "Bounds checked Arrays"

- An important shortcoming of arrays in C/C++ is their undefined behavior, if indexes exceed array bounds.

```
// C++
int nNumbers = 5;
int numbers[nNumbers];
numbers[10] = 10; // Undefined behavior! Above number's bounds.
int value = numbers[-3]; // Undefined behavior! Below number's bounds.
```

- Modern platforms (Java/.NET) introduce bounds checking of array-access via indexes during run time with exceptions:

```
// C#:
int nNumbers = 5;
int numbers = new int[nNumbers];

numbers[10] = 10; // Well defined! Will throw ArgumentOutOfRangeException.

// We could also handle ArgumentOutOfRangeException (not recommended):
try
{
	int value = numbers[-3]; // Well defined! Will throw ArgumentOutOfRangeException.
}
catch (ArgumentOutOfRangeException)
{
	Console.WriteLine("ArgumentOutOfRangeException thrown.");
}
```

Good to know
A variable denoting the count of elements in an array (or any kind of "collection"), is commonly accepted to be a candidate for a variable prefix: 'n'. The idea is, that when *numbers* denotes an array of, well, numbers, it seems logical to have another variable denoting count of elements in that array with the count of numbers, or in short *nNumbers*.

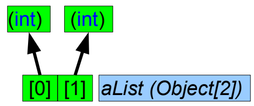
- Indexed collections beyond arrays (e.g. lists) usually support bounds checking with exceptions as well.

Indexed Object-based Collections

- Now its time to introduce our first collection "beyond" arrays: the <u>list</u>.
  - Java/.NET provide a implementation of list: *ArrayList*.(.NET: *System.Collections.ArrayList*, Java: *java.util.ArrayList*) *ArrayList* is an <u>object-based collection</u>, let's understand what that means for us.
- Terminology alert: <u>in Lisp and C++ lists</u> are <u>linked lists</u> and <u>no indexed collections</u>!
- <u>*ArrayList*s can only store objects of the static .NET type *Object*</u>! Therefor *ArrayList* is said to be an <u>object-based collection</u>.
  - <u>That's no problem!</u> – <u>The dynamic type stored in *ArrayList* can be any type derived from *Object*.</u>
  - <u>Each .NET type is derived from *Object*, therefor any .NET type can be stored in *ArrayList* as dynamic type.</u>

```
// .NET System.Collections.ArrayList (C#)
public class ArrayList { // (Details hidden)
        // The "operator[]":
        public object this[int index] {
            get { /* pass */ }
            set { /* pass */ }
        }
}
```

```
// C#:
// Create an ArrayList of two elements (passing an int[] to the ctor):
ArrayList aList = new ArrayList(new[]{42, 18});
// 42 and 18 are then stored on the indexes 0 and 1 (ArrayList's index is 0-based).
// When we fetch the values back, we just get objects of the static type Object back:
object the42 = aList[0];
object the18 = aList[1];
// We have to use cast contact lenses to get the dynamic types out of the objects:
int the42AsInt = (int)the42;
// We can also cast directly from the indexer notation:
int the18AsInt = (int)aList[1];
```



(logical representation)

- There is just one <u>quirk</u> with object-based collections: <u>we have to use downcasts to get the formerly stored values back</u>!
  - This means: we as programmers have to <u>remember</u> the types of the stored values, i.e. <u>their dynamic types</u>!

20

- Mind how C#'s indexer (i.e. the "operator[]") makes using *ArrayList*s look like arrays.

List – Basic Operations

- Creating the empty list *names*:

```
// C#:
ArrayList names = new ArrayList();
```

```
// Java:
ArrayList names = new ArrayList();
```

- Then we can add two elements like so:

```
names.Add("James");
names.Add("Miranda");
```

```
names.add("James");
names.add("Miranda");
```

C# – collection initializers

```
// Alternatively create and initialize with a collection
// initializer:
ArrayList names2 = new ArrayList{ "James", "Miranda" };
```

- We can access the elements of a list like so (bounds checked):

```
// Get the values back (object-based collection):
object name1 = names[0]; // "James" (static type Object)
object name2 = names[1]; // "Miranda"
// Cast the strings out of the objects:
string name1AsString = (string)name1;
// We can also cast directly from the index notation:
string name2As = (string)names[1];
```

```
// Get the values back (object-based collection):
Object name1 = names.get(0); // "James" (static type Object)
Object name2 = names.get(1); // "Miranda"
// Cast the strings out of the objects:
String name1AsString = (String)name1;
// We can also cast directly from the getter-call:
String name2As = (String)names.get(1);
```

- We can set the elements of a list to new values like so (bounds checked):

```
names[1] = "Meredith";
Console.WriteLine(names[1]);
// >Meredith
```

```
names.set(1, "Meredith");
System.out.println(names.get(1));
// >Meredith
```

- We can get the current count of elements stored in the list:

```
names.Add("Christie"); // Adding one more element.
Console.WriteLine(names.Count);
// >3
```

```
names.add("Christie"); // Adding one more element.
System.out.println(names.size());
// >3
```

21

- As can be seen, in Java expressions like *array*[i] just need to be replaced by *list.get*(i)/*list.set*(i).
  - Groovy's syntax (Groovy is based on the Java platform) for creating arrays does directly create *ArrayList*s instead of bare arrays. (But Groovy permits the []-operator to access *ArrayList*s!)
- Here we use *ArrayList* in Java, an alternative would be *Vector*, but *Vector* is a (object-) synchronized collection, which is more inefficient than the unsynchronized *ArrayList*. – *ArrayList* should be our default, until synchronization is needed. But if synchronization is needed, it is better to use Java's simple factory *Collections.synchronizedList()* instead of the old-fashioned *Vector*.
  - *Vector*, which was introduced with Java 1, was synchronized from the start, because people wanted to force multithreaded programming from the start and *Vector* should then be a functional default collection for multithreaded programming.

List – Other important Operations

- Removing elements (bounds checked):

```
// Removes the first occurrence of "James" from names.
// - If the specified value is not in the list, nothing will be removed.
names.Remove("James");
// - If the specified index is out of bounds of the list, an
// ArgumentOutOfRangeException will be thrown.
names.RemoveAt(0);
```

```
// Removes the first occurrence of "James" from names.
// - If the specified value is not in the list, nothing will be removed.
names.remove("James");
// - If the specified index is out of bounds of the list, an
// IndexOutOfBoundsException will be thrown.
names.removeAt(0);
```

- Inserting elements (bounds checked):

```
// Inserts "Karl" into names at the specified index.
// - If the specified index is out of bounds of the list, an
// ArgumentOutOfRangeException will be thrown.
names.Insert(0, "Karl");
```

```
// Inserts "Karl" into names at the specified index.
// - If the specified index is out of bounds of the list, an
// IndexOutOfBoundsException will be thrown.
names.add(0, "Karl");
```

- Find an element:

```
// Returns the index where the first occurrence of "Karl" resides
// in names or -1.
int foundIndex = names.IndexOf(value);
```

```
// Returns the index where the first occurrence of "Karl" resides in
// names or -1.
int foundIndex = names.indexOf(value);
```

- Other operations (selection):

```
names.Clear();
names.AddRange(new[]{"Mary", "Jane"});
names.Reverse();
```

```
names.clear();
names.addAll(Arrays.asList("Mary", "Jane"));
boolean listIsEmpty = names.isEmpty();
```

List – Practical Example: Reading a File Line by Line into a List

- (Yes, it can be done simpler. This is only an example.)
- We don't know the count of lines in advance: a list seems to be a good collection, because it can grow! Solve it with list!

```
// C#:
// Begin with an empty list:
ArrayList allLines = new ArrayList();

string line = null;
using (StreamReader file = new StreamReader("/Library/Logs/Software Update.log"))
{
	// Read the file line by line:
	while (null != (line = file.ReadLine()))
	{
		// Add each line to the list as String:
		allLines.Add(line);
	}

	// Output all the read lines to the console:
	for (int i = 0; i < allLines.Count; ++i)
	{
		// As ArrayList is object-based, we've to cast the element back to String:
		string storedLine = (string)allLines[i];
		Console.WriteLine(storedLine);
	}
}
```

List – Practical Example: Does the List contain a certain element?

- We could iterate over the list to search a specific entry ("2008-11-24 23:02:24 +0100: Installed \"Safari\" (3.2.1)"):

```
bool containsSafari = false;
// Iterate over all entries of the list:
for (int i = 0; i < allLines.Count; ++i)
{
	// Find a certain element in the list:
	string entry = (string)allLines[i];
	if (entry.Equals("2008-11-24 23:02:24 +0100: Installed \"Safari\" (3.2.1)"))
	{
		containsSafari = true;
		break;
	}
}
Console.WriteLine($"Contains 'Safari': {containsSafari}");
// >Contains 'Safari': True
```

- Often collections provide more clever operations to check containment of objects directly, e.g. with .NET's method *Contains()*:
  - This code does effectively the same as the snippet above, but this time we delegate the search procedure to the list itself!

```
// Let's ask the list to find the contained element:
bool containsSafari = allLines.Contains("2008-11-24 23:02:24 +0100: Installed \"Safari\" (3.2.1)");
Console.WriteLine($"Contains 'Safari': {containsSafari}", );
// >Contains 'Safari': True
```

  - As can be seen no loop or comparison is required, list's method *Contains()* does everything for us!

- Java's interfaces *List*/*Collection* provide the method *contains()*.

List – List Comprehensions

- Esp. in functional programming (fp) languages there exist very sophisticated ways to create filled lists without loops.
  - This is required, because lists can usually not be modified after creation in fp languages. For example in Lisp and Haskell:

$$numbers = \{n \mid n \in \mathbf{N},\ n < 10\}$$

$$numbersSquared = \{n^2 \mid n \in \mathbf{N},\ n < 10\}$$

Mathematical set builder notation

```
; Common Lisp (loop macro)
(loop for n from 1 to 9 collect n)
; =(1 2 3 4 5 6 7 8 9)
(loop for n from 1 to 9 collect (* n n))
; =(1 4 9 16 25 36 49 64 81)
```

```
-- Haskell
numbers = [1..9]
-- numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
numbersSquared = [n * n | n <- [0..9]]
-- numbersSquared = [1, 4, 9, 16, 25, 36, 49, 64, 81]
```

  - What we see here in action is called list comprehension, it means to create collections with complex content in one expression.
  - Comprehensions allow to mimc closed forms (comprehensions are often implemented as mathematical procedures).

- Non-fp languages have been updated to provide list comprehensions in order to get compact expressiveness:

```
// Java (version >= 8 provides streams)
List<Integer> numbers
	= IntStream.range(1, 10).boxed().collect(Collectors.toList());
// numbers = 1, 2, 3, 4, 5, 6, 7, 8, 9
List<Integer> numbersSquared
	= IntStream.range(1, 10).map(n -> n * n).boxed().collect(Collectors.toList());
// numbersSquared = 1, 4, 9, 16, 25, 36, 49, 64, 81
```

```
// F#
let numbers = [1..9]
// numbers = [0; 1; 2; 3; 4; 5; 6; 7; 8; 9]
let numbersSquaredList = [ for n in 1 .. 9 -> n * n ]
// numbersSquaredList = [1; 4; 9; 16; 25; 36; 49; 64; 81]
let numbersSquaredArray = [| for n in 1 .. 9 -> n * n |]
// numbersSquaredArray = [|1; 4; 9; 16; 25; 36; 49; 64; 81|]
```

```
// C# (version >= 3 provides query expressions)
var numbers = Enumerable.Range(1, 9).ToList();
// numbers = 1, 2, 3, 4, 5, 6, 7, 8, 9
var numbersSquared = (from n in Enumerable.Range(1, 9) select n * n).ToList();
// numbersSquared = 1, 4, 9, 16, 25, 36, 49, 64, 81
```

```
// Groovy
def numbers = (1 .. 9).toList()
// numbers = [0; 1; 2; 3; 4; 5; 6; 7; 8; 9]
def numbersSquared = (1 .. 9).collect{it * it}
// numbersSquared = [1, 4, 9, 16, 25, 36, 49, 64, 81]
```

25

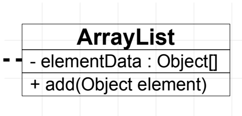
- The *Stream*-based Java code snippet needs some explanation: The *range()* method to get into int-based list comprehensions in Java, is only available on *IntStream*. And *IntStream* provides a stream of int, which is not an object derived from *Stream* at all, *IntStream* is a primitive stream. Java's specific collector, that collects a generic *Stream* into a generic *List* does not accept primitive streams. Therefor we have to use *boxed()* to convert the primitive *IntStream* into a generic *Stream<Integer>* (i.e. boxing all ints), that can be consumed by the *Collectors.toList() Collector*.

Indexed Collections – Size and Capacity – Part I

- When we think about the implementation of an indexed collection we can assume that it is backed by an array.
  - This means that indexed collections encapsulate or simulate arrays of varying size. – That's straight forward!
  - Now we'll clarify how managing of indexed collections' internal space for contained elements works.

- Most interesting are obviously methods like *ArrayList*'s method *add()* in Java. It could be implemented like so:

```
// Java: (fictive implementation of ArrayList.add())
public boolean add(Object element) {
        // Enlarge and append:
        elementData = Arrays.copyOf(elementData, elementData.length + 1);
        elementData[elementData.length - 1] = element;
}
```



- But the assumed implementation of *add()* has a serious problem: it is very inefficient!

```
ArrayList names = new ArrayList(); // Creates an ArrayList of zero elements.
names.add("James");    // Create a new internal array of size one.
names.add("Miranda");  // Create a new internal array of size two.
names.add("Helen");    // Create a new internal array of size three.
```

  - Every time a new element is added two operations are going on basically:
    - (1) enlarge: the encapsulated array *elementData* is copied to a new array having the old length + 1
    - (2) append: and then the new element is set as last element in the new array.

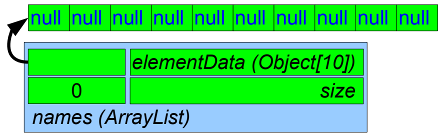
26

Indexed Collections – Size and Capacity – Part II

- Indeed the real implementation of *ArrayList* is more clever! *ArrayList* uses the idea of capacity for more efficiency.

- Let's create another empty *ArrayList* in Java:

```
// Java:
ArrayList names = new ArrayList();
```



  - The real representation of names looks not very empty!

- The actual memory representation:
  - As can be seen *names* is not really empty.
  - *names*' field *elementData* (this is the encapsulated *Object*-array) has an initial length of ten.
  - All elements of *elementData* have the initial value null.
  - But *names* exposes a *size* of zero! – What the heck is going on?

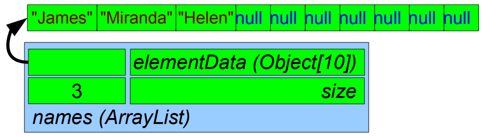
- *ArrayList* has an initial size of zero and an initial capacity of ten!
  - In modern collection APIs, collections maintain capacity and size separately, let's understand why this is a good idea ...

27

Indexed Collections – Capacity Control – Part I

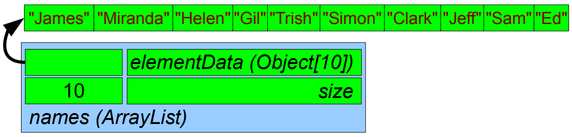
- With *ArrayList*'s initial capacity of ten we can add ten elements, before a new *elementData*-array needs to be created:
  – Let's begin with adding three *String*s to *names*:

```
names.add("James");
names.add("Miranda");
names.add("Helen");
```



  – Add another seven Strings to *names*:

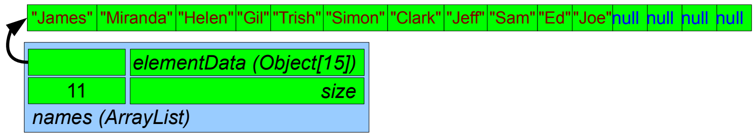
```
names.addAll(Arrays.asList("Gil", "Trish", "Simon"));
names.addAll(Arrays.asList("Clark", "Jeff", "Sam", "Ed"));
```



  – Then we add another, the eleventh *String* to *names*...

```
names.add("Joe");
```

- ...when the capacity of the *ArrayList* is exhausted, the encapsulated array (*elementData*) needs to be enlarged.
  – (1) A new bigger array is created and
  – (2) the content of the old array is copied to the new array.
    - (In this implementation, *elementData* is enlarged for five elements, so the new capacity is 15.)
  – (3) The last element (the eleventh element) will be set to the added element.



28

- The values of the initial capacity as well the amount of growth/increment of the capacity should be taken from the spec. of the platform in question. Tip: In practice programmers should not rely on these specs, but rather control the capacity themselves.

Indexed Collections – Capacity Control – Part II

- We just learned that when the required space of an *ArrayList* exceeds its capacity, the internal memory is "re-capacitized".
- Of course it is! The idea of the encapsulated array's capacity apart from the collection's size is to defer the need to "re-capacitize" the encapsulated array to the latest thinkable occasion: When size gets greater than capacity.
  - But there is a problem: if we constantly exceed the capacity of a collection, it will often be re-capacitized.
- Yes, we can solve this problem with capacity control!
  - One idea of capacity control is to set the required capacity of a collection, before elements are added, e.g. like so:

```
int initialCapacity = 11; // Ok, we know in advance that we're going to add eleven elements to names. So we can set the required capacity early!
ArrayList names = new ArrayList(initialCapacity); // E.g. we can specify the capacity right in the ctor.
```

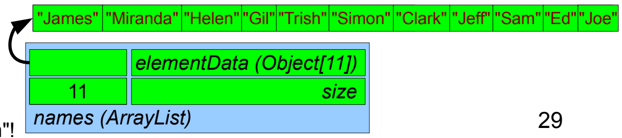
  - Add ten elements; the size will be below the capacity:

```
names.addAll(Arrays.asList("James", "Miranda", "Helen", "Gil", "Trish", "Simon", "Clark", "Jeff", "Sam", "Ed"));
```

  - Then we add the eleventh element...

```
names.add("Joe");
```

  - ...this just meets the capacity of *names*
  - => With capacity control there was no need for "re-capacitation"!



29

Indexed Collections – Capacity Control – Part III

- Collections (not only indexed collections) often provide a set of methods/properties to control capacity.
  - The capacity is the number of elements a collection can store; the size is the actual number of elements a collection contains.
  - => This must always be true: capacity >= size.

- Collections may have ctors to set the initial capacity overriding the default capacity, e.g. initializing it w/ a capacity of eleven:

```
// Java:
ArrayList names = new ArrayList(11);
names.addAll(Arrays.asList("James", "Miranda", "Helen", "Gil",
        "Trish", "Simon", "Clark", "Jeff", "Sam", "Ed", "Joe"));
```

```
// C#:
ArrayList names = new ArrayList(11); // Set initial capacity to 10.
names.AddRange(new []{"James", "Miranda", "Helen", "Gil",
        "Trish", "Simon", "Clark", "Jeff", "Sam", "Ed", "Joe"});
```

- Some collections allow setting a certain minimum capacity, or to set the capacity directly:

```
names.ensureCapacity(20);
```

```
names.Capacity = 20;
```

- But *names*' size is still eleven!

```
boolean hasEleven = names.size() == 11;
// hasEleven = true
```

```
bool hasEleven = names.Count == 11;
// hasEleven = true
```

Performance guideline
- If the effective size of a collection to be created is known ahead, set the capacity as soon as possible, e.g. in the ctor.
- If memory is low, it can be helpful to set a small capacity for a new collection or to trim the capacity of existing collections to size.

30

- Some IDEs allow enabling a warning during code inspection, if a collection w/o explicitly specified capacity is filled.

Indexed Collections – Capacity Control – Part IV

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

```java
// 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);
}
```

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

```csharp
// 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);
}
```

```csharp
// … 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 filled with items in bulk after creation:

```java
// 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 (it is usually faster):
Collections.addAll(numbers, 1, 2, 3, 4);
```

```csharp
// 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});
```

31

- Java's *Collections.addAll()* is faster than *Collection.addAll()* because the latter creates a redundant array.

Thank you!