(6) Basics of the Java Programming Language

Nico Ludwig (@ersatzteilchen)

TOC

- (6) Basics of the Java Programming Language
  - Primitive Types/Value Types vs Reference Types
  - Wrapper types
  - Details about Stack and Heap Memory
  - Garbage Collector – General Concepts

- Cited Literature:
  - Just Java, Peter van der Linden
  - Bruce Eckel, Thinking in Java

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 two Worlds of Types in Java

- In Java, we can basically tell primitive types from reference types.

- (1) Primitive types:
  - They are integrated into the Java language (keywords) and have special support in the JVM as well as in byte code.
  - Objects of primitive types are generally created in a dedicated region in the memory: the stack.
  - They have value semantics.
  - There exist only eight primitive types in Java: boolean, byte, char, short, int, long, float and double.

- (2) Reference types:
  - They are types, which are defined in libraries. The names of these types are written in the PascalCase convention.
  - Objects of reference types can only be created in another dedicated region in the memory: the heap.
  - They have reference semantics.
  - We can define our own, new, reference types, so called user defined types (UDTs).
  - The amount of reference types is without limit. We have already used: *String*, *StringBuilder*, *Scanner, ArrayList and int[]*.

- The reference type *String* and all kinds of array types have special support in the JVM and in byte code.

How are Objects created in Java?

- (1) The creation of objects of primitive type means just to define and initialize a variable of that such a type.

```
int aNumber = 0;                          // an int object
double anotherNumber = 73.22;             // a double object
```

  - After creation, we mainly interact with those objects with operators or print their values to any means of output:

```
double yetAnotherResult = aNumber + anotherNumber; // Using operators = and +.
System.out.println(yetAnotherResult);               // Print the result to output.
```

- (2) The creation of objects of reference type has to be done with the new operator. It looks a bit more "involved":

```
StringBuilder text = new StringBuilder();           // a StringBuilder object
```

  - After creation, we mainly interact with those objects with methods:

```
text.append("next number: ");                       // Calling some methods
String effectiveText = text.toString();
```

Good to know
- In Java, the procedure of object creation is emphasized by the phrase "instantiation of an object".
- In C/C++, the procedure of object creation seems to be emphasized by the phrase "allocation of memory", which underscores the technical operation to get memory for a new object.

- But syntactically, some basic things are the same for (1) and (2), when objects are created:

int	aNumber	=	0
type name	variable name		initial value
StringBuilder	text	=	new StringBuilder()

- Often the creation of objects, esp. when created with the new operator, is called instantiation.
  - "The object *text* is an instance of *StringBuilder*."
  - But also: "The object *aNumber* is an instance of int."

Value Types – Primitive Types are Value Types

- What does value type mean?
- The value of a variable of value type is the essence of the object.
- Huh? Well, let's inspect this example of a simple int object:

```
int aNumber = 24;
```

  - Then let's initialize *anotherNumber* with the variable *aNumber*:

```
int anotherNumber = aNumber;
```

  - The point of matter is that *aNumber* and *anotherNumber* are independent objects. If *aNumber* is changed, *anotherNumber* is not!

```
aNumber = 958;
System.out.println("aNumber's value: " + aNumber);
// >aNumber's value: 958

// Changing aNumber won't affect anotherNumber!
System.out.println("anotherNumber's value: " + anotherNumber);
// >anotherNumber's value: 24
```

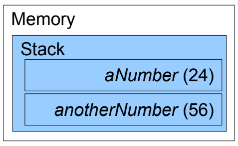
- An object of value type is only defined by its value.

- A variable of value type is often just called "value".

Value Types – Value Types in Memory

- Objects of value type are created in a conceptional region of computer memory, the stack.
- The name "stack" underscores the fact that objects of primitive types are stored in a stacked way in memory.
  - E.g. let's inspect this situation:

```
int aNumber = 24;
int anotherNumber = 56;
```



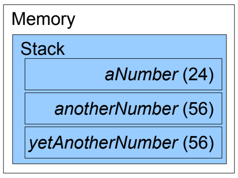
  - The objects 24 and 56 are created on the stack. The memory locations of those are adjacent, or "stacked".
  - The important thing is, that the values themselves are stored on the stack!
- During this course graphics concerning the memory region of the stack will be visualized in blue color.
- The concept of Java's stack memory corresponds to the stack for automatic variables in C/C++.

Value Types – Copying

- Now we'll create *yetAnotherNumber* and assign *anotherNumber*:

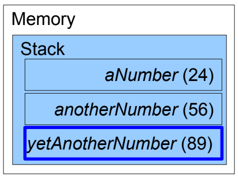
```
int aNumber = 24;
int anotherNumber = 56;
```

```
int yetAnotherNumber = anotherNumber;
```



  - What we see here, is that *yetAnotherNumber* contains just another copy of *anotherNumber*'s value 56.
  - Next, let's change the value of *yetAnotherNumber*:

```
yetAnotherNumber = 89;
```



- The value of a value type is the essence of an object:
  - Only *yetAnotherNumber* contains the new value.
  - The important point is, that the value of the other variable *anotherNumber* was not changed!
  - *anotherNumber* and *yetAnotherNumber* are independent object containing independent values!

Reference Types – User defined Types are Reference Types

- What does reference type mean?

- A variable of reference type is rather a shortcut to the referenced data.
  - A variable of reference type is often just called "reference".
  - Em … huh? Well, let's inspect this example of a *StringBuilder* object:

```
StringBuilder stringBuilder = null;
System.out.println("The stringBuilder: " + stringBuilder);
// >The stringBuilder: null
```

```
StringBuilder stringBuilder; // Invalid! No initialization before usage!
System.out.println("The stringBuilder: " + stringBuilder);
```

    - A reference needs to be initialized before it is used. If a reference does not refer to data, it refers to null.
  - OK, now let's assign a new *StringBuilder* object to *stringBuilder*:

```
stringBuilder = new StringBuilder("content");
System.out.println("The stringBuilder: " + stringBuilder);
// >The stringBuilder: content
```

    - After the assignment of a new *StringBuilder* object, *stringBuilder* refers to this new *StringBuilder* object.

- Looks pretty similar to value types, isn't? So far: yes, at least syntactically, but there is a significant difference!

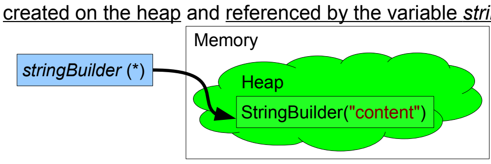
- An important indicator that we use a reference type, is that such objects are created using the new operator.
  - There are two exceptions: *String*s and arrays, which are reference types, but objects thereof can be created w/o the operator new.

Reference Types – Reference Types in Memory – Part I

- Objects of reference type are created in another conceptional region of computer memory, the heap.
- The name "heap" also underscores following fact: referenced objects are stored in memory as if it was a heap.
  - E.g. let's inspect this situation:

```
StringBuilder stringBuilder = new StringBuilder("content");
```

  - The object *StringBuilder("content")* is created on the heap and referenced by the variable *stringBuilder*:



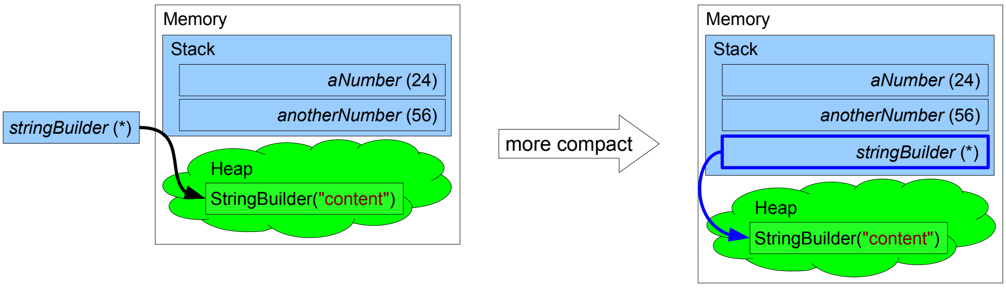
- In comparison to the stack, the heap is very unordered (not "stacked"), it will be represented by a "fuzzy" cloud.
  - During this course graphics concerning the memory region of the heap will be visualized in green color.
- A variable of reference type is a reference to the object of reference type in the heap.
  - A variable of reference type, lives on the stack, but the referenced object resides on the heap.
  - The value of a variable of reference type will be represented with the '*' symbol in memory graphics in this course:

Reference Types – Reference Types in Memory – Part II

- It's time to bring the memory types stack and heap together to have a more complete picture of memory.

```
// Some objects, which reside in memory:
int aNumber = 24;
int anotherNumber = 56;
StringBuilder stringBuilder = new StringBuilder("content");
```

- What's about the variable *stringBuilder*? In which type of memory does it reside?



  - The blue color of *stringBuilder*'s box indicates, that the variable resides on the stack!
  - *StringBuilder("content")* is created on the heap, but referenced by the variable *stringBuilder* on the stack.
- In short: the *stringBuilder*'s data is stored on the heap and is is only referenced from the stack!
  - Mind, that the reference *stringBuilder* is stored on the stack! The reference *stringBuilder* acts like "shortcut" to "its" data in the heap.

- This slide should make clear, how Java's references resemble pointers in C/C++. But as far as memory is concerned, Java's references can not refer to data on the stack.

Reference Types – Aliasing

- Let's assign the just created *StringBuilder* object to yet another reference *anotherStringBuilder*:

```
StringBuilder anotherStringBuilder = stringBuilder;
```

  - Then let's call a method of *StringBuilder* on *anotherStringBuilder*, which modifies the object:

```
anotherStringBuilder.insert(3, " in between ");
System.out.println("The anotherStringBuilder: " + anotherStringBuilder);
// >The anotherStringBuilder: con in between tent
```

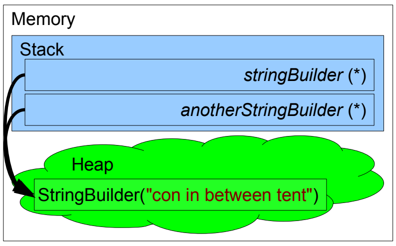
  - It meets our expectations that *anotherStringBuilder* shows the performed modifications. Now we'll inspect *stringBuilder*'s content:

```
System.out.println("The stringBuilder: " + stringBuilder);
// >The stringBuilder: con in between tent
```

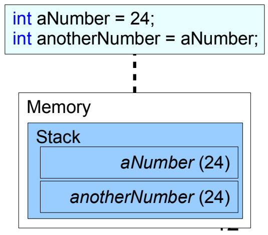
  - The "content" of *stringBuilder* has been modified as well! -> There's a connection between *stringBuilder* and *anotherStringBuilder*!

- References can refer to the same object, this is what we call aliasing.

Reference type *StringBuilder*: Aliasing



Value type int: Copies

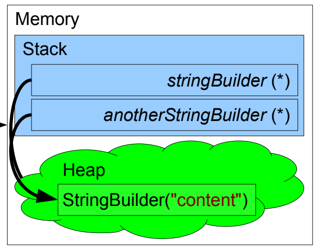


  - This is very different from the last example, where we've used the value type int: Value type keep copies of their values in stack.

Reference Types – the null-Reference

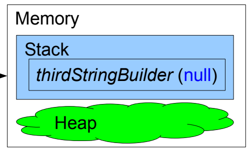
- We already discussed the alias effect, where two or more references refer to the same data in the heap:

```
StringBuilder stringBuilder = new StringBuilder("content");
StringBuilder anotherStringBuilder = stringBuilder;
```



- However, there also exists another extreme: a reference, which refers to nothing, i.e. it has no connection to the heap:

```
// A reference referring to no data in the heap:
StringBuilder thirdStringBuilder = null;
```



Good to know
It is good practice to distinguish "0" from "null" by calling the "0" "zero".

  - In Java, a reference does either refer to an object in heap or is a null-reference.

- Trying to call methods on a null-reference will raise a *NullPointerException* (NPE):

```
// Will throw a NullPointerException:
thirdStringBuilder.append("some text");
```

Good to know
The NPE is the by far most frequently thrown exception in Java, not only beginners encounter them on a regular basis!

13

- *NullPointerException* is of course a bad name: it is a null-reference not a null-pointer.
- There are at least two other important differences between C/C++' pointers and Java's references:
  - The state of references is clearly defined in Java: it either refers to an object or is a null-reference. – A reference cannot be in an undefined state.
  - Calling methods on a null-reference in Java (i.e. dereferencing a pointer in C/C++) will throw an NPE. In C/C++ dereferencing an uninitialized pointer or null-pointer leads to undefined behavior.

Reference Types – Being null-aware

- Before we access a reference, which could be a null-reference, we should check for "nullity", before the access:

```
if (null != thirdStringBuilder) { // Excellent!
	thirdStringBuilder.append("some text");
}
```

  - Code, or methods, which consequently check references for nullity before the access takes place are often said to be "null-aware".

- In opposite to C/C++ does with nullptr/NULL, Java does not interpret null-references as booleans!

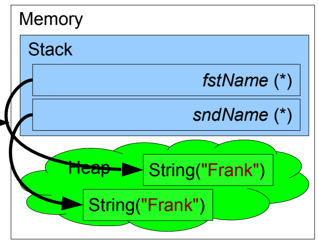
```
if (thirdStringBuilder) { // Invalid! Error: java: incompatible types: java.lang.StringBuilder cannot be converted to boolean
	thirdStringBuilder.append("some text");
}
```

- In Java, null is the absence of a value of a reference. null is not 0 (zero) and not an empty *String*.

Reference Types – Comparison of Objects

- In an earlier lecture we discussed, how *String* objects have to be compared correctly with the method *equals()*:
  - To understand why == comparison is wrong, let's analyze the situation in memory:

```
// Two different objects having the same content:
String fstName = "Frank";
String sndName = String.format("%s", "Frank");
```



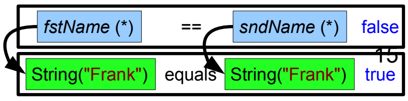
  - In the memory visualization, we can see, that *fstName* and *sndName* refer to different data in heap.
  - I.e. the arrows refer from different references on the stack to different objects in the heap.

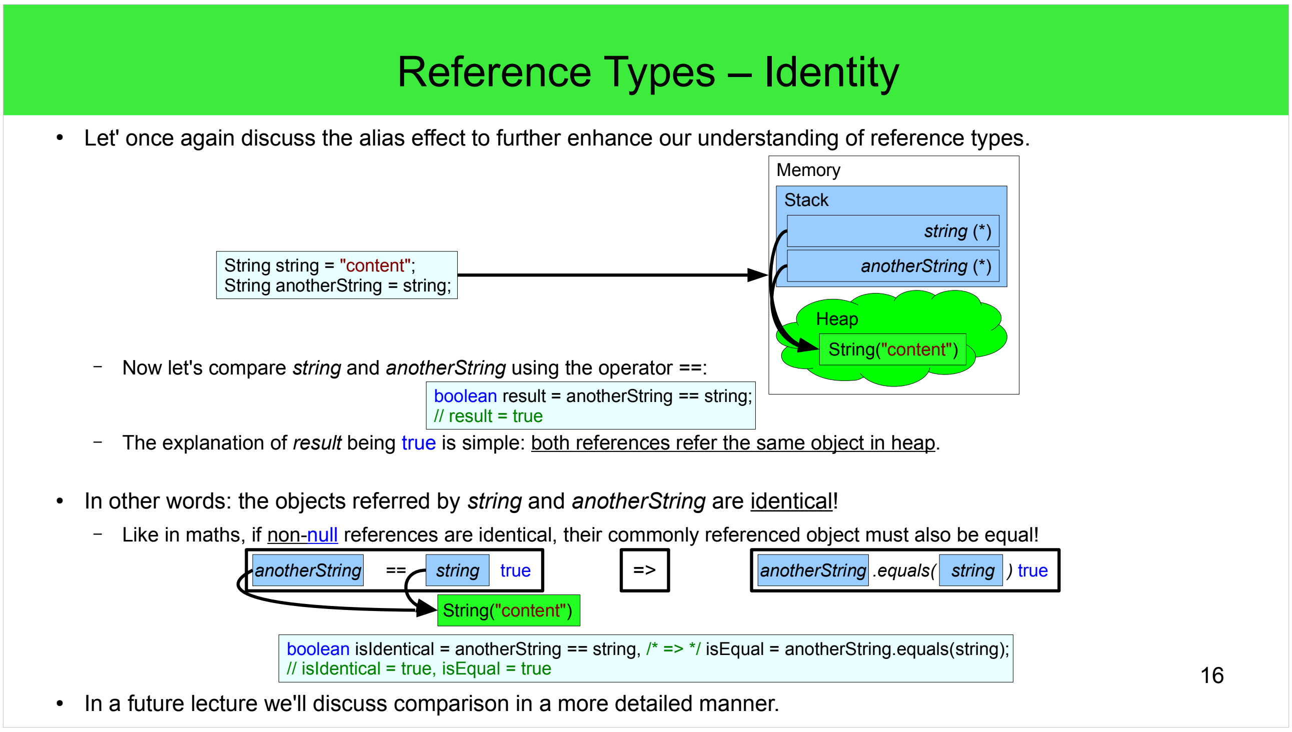
- Java's operator == does only compare data on the stack, i.e. the content of objects of value type or references!

```
// Wrong:
boolean result = fstName == sndName;
// result = false
```

```
// Correct:
boolean result = fstName.equals(sndName);
// result = true
```

  - This operator == doesn't compare the data, which is referenced by a reference.
  - Instead *equals()* performs a deep comparison of objects.





- We can only identity-compare references of the same or related type:

```
String string = "content";
int[] numbers = {1,2,3,4,5};
boolean result = string == numbers;   // Invalid! Incomparable types:
                                      // java.lang.String and int[]
```

  - We will clarify the meaning of comparable and incomparable types in a future lecture.

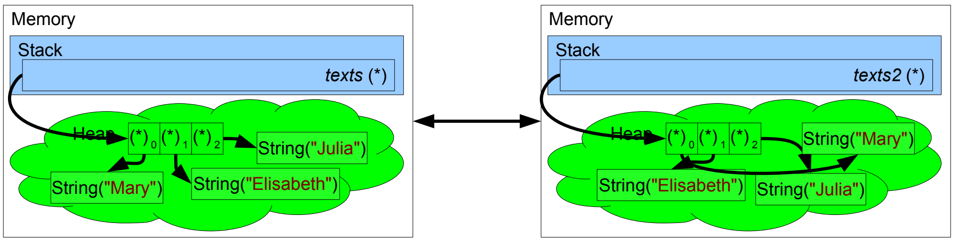
Reference Types – Arrays

- In Java, all array types are reference types.
  - Similar to *String*s, the array creation syntax might not unleash this fact, because in certain cases the operator new can be left away:

```
// Arrays can be created with the operator new:
String[] texts = new String[] {"Mary", "Elisabeth", "Julia"};
```

```
// Or applying the short array initializer syntax (without new):
String[] texts2 = {"Mary", "Elisabeth", "Julia"};
```

- Independent from syntax, *texts* and *texts2* are both created on the heap! Their memory can be visualized like so:



- The arrays *texts* and *texts2* represent multiple slots, which in whole reside on the heap.
  - The visualization of the arrays should be no surprise, the slots of the arrays are arranged on the heap like a list (with indexes 0-2).
- Each of the slots refers to an individual element on the heap, which is of type *String*.
  - If we create arrays with elements of reference type, those arrays are arrays of references.
  - Here *texts* and *texts2* are arrays created on the heap with references referring to elements also residing on the heap.

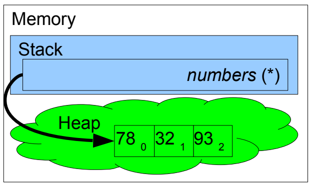
- The Java language specification doesn't specify, whether data is stored adjacently in an array.

Reference Types – Arrays – Arrays of Value Type Objects

- We have discussed arrays of objects of reference type, now we have to discuss arrays of objects of value type, e.g. int:

```
// An array of int objects:
int[] numbers = new int[] {78, 32, 93};
```

- In Java, all arrays are created on the heap, so is *numbers*! The memory of *numbers* can be visualized like so:



- The symbol *numbers* refers to the array in heap. Each "slot" of the array directly contains the int value respectively.
  - I.e. here, the slots do not contain references, which refer to other objects each.

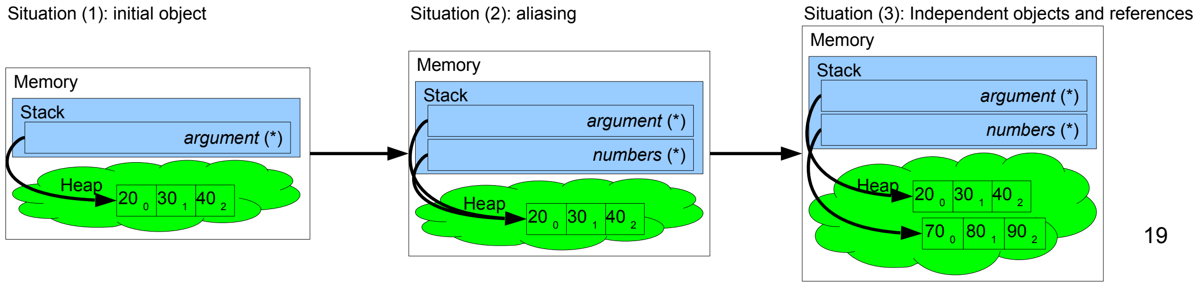
- Important conclusion: in arrays, also objects of primitive types (e.g. int) are stored on the heap.
  - Value type objects "living" on the heap are sometimes called wrapped values (an older Java term) or boxed values.

Reference Types – Passing References to Methods

- We've noticed, that arguments are passed to methods by value: a parameter's value is a copy of the argument's value.
- This is also true for reference types: if we pass a reference to a method, it will be copied:

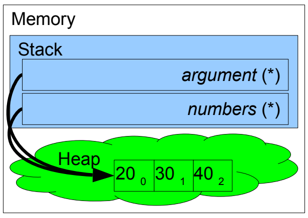
```
// This is a silly method, trying to modify parameters:
static void silly(int[] numbers) {
	// numbers = {20, 30, 40} (2) aliasing situation, argument and numbers refer to the some data in heap
	numbers = new int[] {70, 80, 90}; // The assignment to numbers only has an effect local to silly()!
	// numbers = {70, 80, 90} (3) two independent arrays in heap
}

int[] argument = {20, 30, 40}; // (1) initial situation
silly(argument); // The reference argument will be copied into the parameter numbers!
// argument = {20, 30, 40} // The passed argument is still {20, 30, 40}!
```

  - Practically, it means that *argument* and *numbers* are different references, which finally refer to different objects:



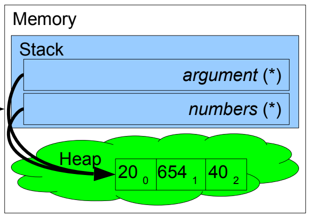
Reference Types – Modifying Objects via References

- OK, references are getting copied when passed to a method. But as we saw both copies refer to same data in heap:



- The interesting point concerning references is, that we can modify the referred object within the called method:

```
// This method modifies the referred object:
static void modifyReferredObject(int[] numbers) {
    // numbers = {20, 30, 40} (2) aliasing situation, argument and numbers refer to the some data in heap
    numbers[1] = 654; // The assignment to a "slot" in numbers modifies the same referred object!
    // numbers = {20, 654, 40} (3) the array referred by numbers and argument has been modified
}

int[] argument = {20, 30, 40}; // (1) initial situation
modifyReferredObject(argument); // The reference argument will be copied into the parameter numbers!
// argument = {20, 654, 40} // The referred object was modified {20, 654, 40}!
```

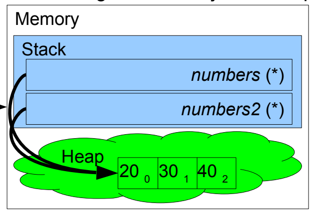


- Important conclusion: the elements of arrays, can be modified via references.
  - This is true for all objects of reference type: via references we can potentially modify referred objects.

Reference Types – Copying Arrays

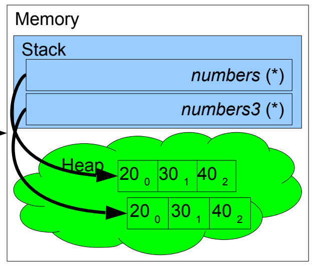
- Now, we'll discuss a topic, which shows a practical example of the aliasing effect: copying arrays.
  - Obviously arrays can not be copied this way, because it'll only produce two references referring to same object in heap:

```
// Once again an aliasing situation:
int[] numbers = new int[] {70, 80, 90};
int[] numbers2 = numbers;
```



- Java provides a special method to copy arrays: *java.util.Arrays.copyOf()*.
  - This method accepts the original array as well as the length of the new array as arguments:

```
// The correct way creating a copy of an array:
int[] numbers = new int[] {70, 80, 90};
int[] numbers3 = java.util.Arrays.copyOf(numbers, numbers.length);
```



  - Sometimes programmers call copying an object in "heap creating a deep copy".
  - The word "deep" and how "deep" a "deep-copy" is, is not clearly defined.

Wrapper Types – Part I

- In an earlier lecture we discussed the parameterized type *ArrayList*, which works like an array, that can grow and shrink.

```
ArrayList<Integer> numbers = new ArrayList<Integer>();
```

"formal" syntax

ArrayList<TypeArgument>	*identifier*

- A crucial difference to arrays is, that *ArrayList* does not work with primitive types, but only with wrapper types.
  - In the example above we have to create an *ArrayList<Integer>* and not an *ArrayList<int>*!
  - Now its time to understand what's behind wrapper types.
- The type *ArrayList* is a so called parametrized type, in which a type parameter is a placeholder for a *TypeArgument*.
  - *ArrayList*s "collect" objects of another type, given as *TypeArgument*. This is pretty similar to arrays.
  - But parametrized types can only work with reference types as *TypeArgument*!
- But we also want to use values, e.g. objects of primitive type (e.g. int or double) with parameterized types like *ArrayList*!
- Sure! – That is the idea of wrapper types. A wrapper type wraps a value (of value type) into a reference type.
  - In other words: A wrapper type allows to put a value on the stack into an object living on the heap.

Wrapper Types – Part II

- Java provides wrapper types for all primitive types, so that we can use values with parametrized types like *ArrayList*.
  - Here a selection of commonly used primitive types associated to their wrapper types:

Primitive Type	Wrapper (reference) Type	Example
int	*Integer*	*ArrayList<Integer>*
double	*Double*	*ArrayList<Double>*
boolean	*Boolean*	*ArrayList<Boolean>*
char	*Character*	*ArrayList<Character>*

- On the following slides, we'll discuss what's going on in memory, when we use wrapper types.

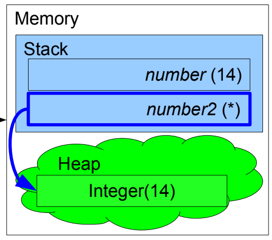
- We'll concentrate on the wrapper type *Integer*, which wraps an int value into a reference type.
  - We can use the type *Integer* independently from *ArrayList<Integer>* like so:

```
int number = 14
Integer number2 = number;
```

Wrapper Types – Part III

- Consider what is going on here:

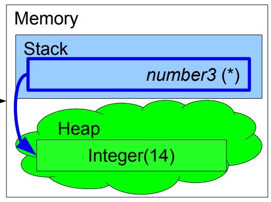
```
int number = 14
Integer number2 = number;
```



  - The original int value 14 is stored in *number*. *number* is of value type int, therefor it is stored on the stack.
  - When *number* is assigned to *number2*, which is of wrapper type *Integer*, a copy of 14 is put in the heap wrapped in an *Integer* object.

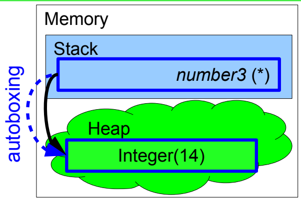
- We can also directly "wrap an int literal into the heap":
  - Mind, that *number3* is still residing on the stack!

```
Integer number3 = 14;
```



Wrapper Types – Autoboxing

```
Integer number3 = 14;
```



Good to know
Before Java 5, autoboxing was not available. In older versions, *Integer*s needed to be created from ints using the cast syntax:

```
// Java 1.4
Integer number3 = (Integer) 14;
```

Before Java 5, the term "boxing" was not even used for this operation.

- The procedure of putting a value into a reference object is called autoboxing.
  - Read: wrapping a primitive types' object into a wrapper types' object is called autoboxing.
  - 14 is now a boxed value, i.e. a boxed int.

- Restrictions:
  - A boxed value cannot be changed! Boxed objects are immutable!
    - It just means, that we cannot change the boxed 14. Instead we have to create a new *Integer* object:

```
number3 = 245; // This creates a new Integer(245) object in heap, overwriting the content of number3
```

  - There are no standard conversions between "related" wrapper types, e.g. *Integer* and *Double*: the types are just not related at all!

```
Integer ii = 34;
Double dd = ii; // Invalid! Incompatible types.
```

```
Integer ii = 34;
Double dd = ii.doubleValue(); // Excellent!
```

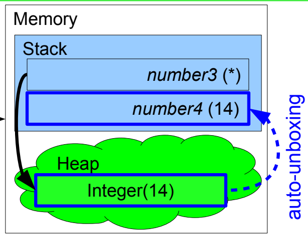
  - During boxing, standard conversions across "related" wrapper and primitive types are not allowed, e.g. from double to *Integer*:

```
// Surprisingly, this won't work:
Double dd = 34; // Invalid! Incompatible types.
```

```
Double dd = new Double(34); // Excellent!
```

Wrapper Types – Auto-Unboxing

```
Integer number3 = 14;
int number4 = number3;
```



Good to know
Before Java 5, auto-unboxing was not available. In older versions, ints needed to be unboxed from *Integer*s by using *Integer*'s method *intValue()*:

```
// Java 1.4
int number4 = number3.intValue();
```

Other wrapper types provide similar "*xxxValue()*" methods for unboxing (e.g. *Double* provides *doubleValue()*).

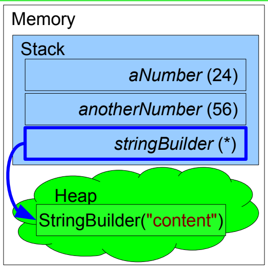
- The procedure of putting a boxed value into a value is called auto-unboxing.
  - Read: unwrapping a wrapper types' object into a primitive types' object is called auto-unboxing.
- If a reference to a boxed value is a null-reference, auto-unboxing will fail with an *NPE*:

```
Integer number5 = null; // number5 is a null reference
int number6 = number5; // Will throw an NPE!
```

- During unboxing, standard conversions "crossing the box" from "related" wrapper and primitive types are allowed!
  - E.g. from *Integer* to double:

```
Integer ii = 34;
double dd = ii; // OK!
```

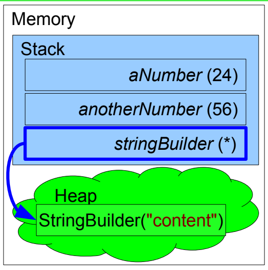
Reference and Value Types Reality Double Check – Part I



- A value type stores its essential value in the stack. The value is the essence of a value type object.
  - Think: "all happens in the stack"/"all happens in the blue part of the memory".
  - Think: "value semantics means copying"
- A reference type's object resides on the heap. A reference type's variable resides on the stack and refers to "its" object on the heap.
  - Think: "the object lives in the heap, a variable lives on the stack", but "objects in the green memory, variables in the blue memory".
  - Think: "reference semantics means shared ownership"

- What we see here is only a simplified model of memory of one Java program, at run time.
  - The heap is really very much bigger than the stack.
  - The contents of the heap are in principle completely unordered.
  - Other sorts of memory apart from stack and heap are not in the model.
  - However, stack and heap are really separated, i.e. residing in different address spaces, but in the same memory.

Reference and Value Types Reality Double Check – Part II



- Identity:
  - Objects of value type have no identity.
  - Objects of reference type have an identity, which is represented by their "reference" (think: "address").
- Passing around:
  - When objects of value type are passed to methods, copies are made.
  - When objects of reference type are passed to methods, shortcuts are passed around.
- Both types can be connected: Objects of reference type can only be used by references on the stack.

Java Programs and Memory

- Up to now we did never talk about memory consumption explicitly!
- As we said, memory is consumed in Java, of course it is! – Every computer program consumes resources!
  - Computer memory is not really "consumed", however, memory is occupied, i.e. it is no longer available for other allocations.
- First let's see what happens, if we run out of memory in a Java program on the stack (during run time):

```
// Let's initiate an infinite recursion:
public static void f() {
	f(); // f() calls itself.
}
// Will throw a java.lang.StackOverflowError
```

Good to know
A computer program consumes two kinds of resources during run time: CPU processing time and memory.

- And what happens, if we run out of memory on the heap:

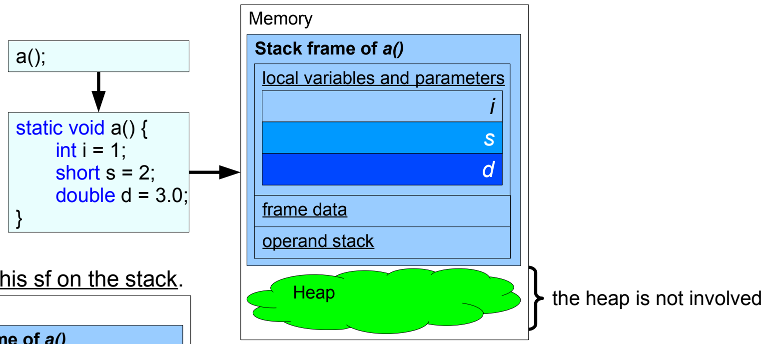
```
// Let's try to create an int array of ~2_000_000_000⁴ elements:
int[][][][] numbers = new int[Integer.MAX_VALUE][Integer.MAX_VALUE][Integer.MAX_VALUE][Integer.MAX_VALUE];
// Will throw a java.lang.OutOfMemoryError: Requested array size exceeds VM limit
```

- Meanwhile we know that Java signals run time errors by throwing *Exception*s, or throwing *Error*s.
  - Java does also use *Exception*s to indicate run time errors regarding memory.
  - A *java.lang.StackOverflowError* is thrown, when memory allocation exceeds all the (current thread's) stack.
  - A *java.lang.OutOfMemoryError* is thrown, when memory allocation exceeds all the computer's memory.

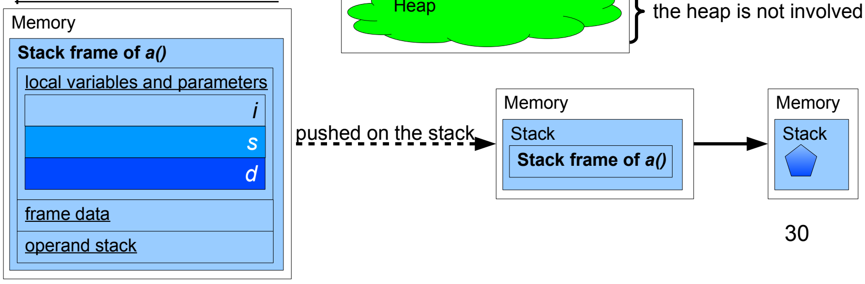
On the Stack – The Lifetime of local Values

- When a method is called, the JVM creates a so called stack frame (sf), which holds the data making up a method's memory.

- A sf of a method has has three sections:
  - local variables and parameters
  - frame data
  - and the operand stack.

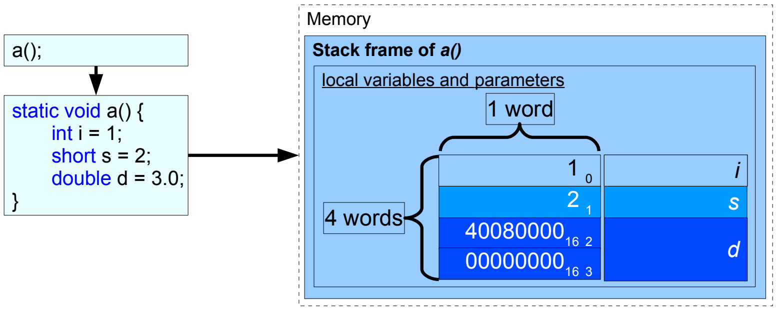


- When the JVM creates a new sf, it pushes this sf on the stack.
  - On the next slides we'll use the symbol to denote *a()*'s sf.



On the Stack – Section Local Variables and Parameters

- The first section of the stack is a structure, which contains all local variables and parameters. For *a()* it looks like this:

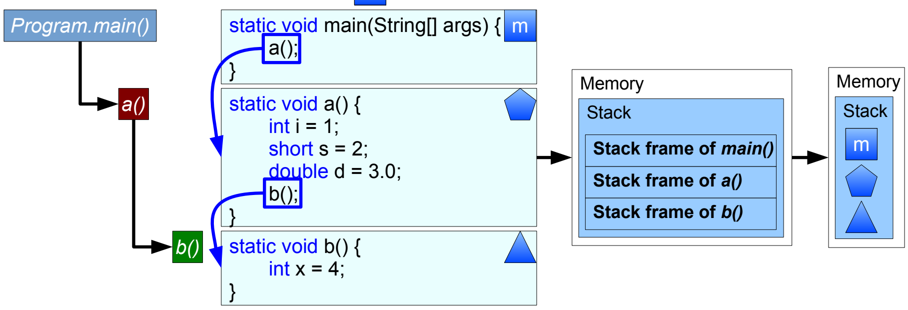


  - The structure is organized like a 0-based array, which stores local variables in the order they are defined in the method.
  - Each element of that array has a fixed size, the so called word size.
  - The word size is an implementation detail, each JVM implementation can do it differently.
  - A word must at least hold one object of type byte, short, int, char, float, returnAddress, or reference, so its minimum size is 4B.
    - We know most of these types already, returnAddress just stores the address, to which execution jumps, when a method completes normally.
  - double and long are allowed to fit into two words.

- This section is very important, because it allows us to discuss the call stacks of methods and the lifetime of locals.

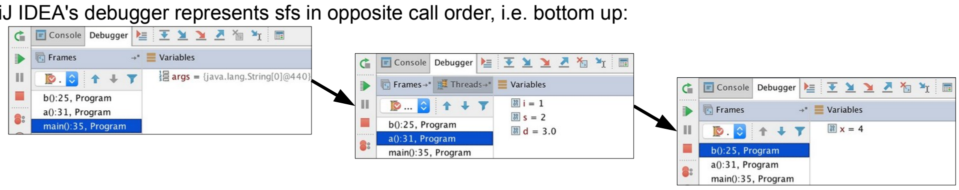
On the Stack – The Call Stack – Part I

- When a method *b()* is called from *a()*, the JVM creates another sf for *b()* (▲), which is pushed on the stack as well:
  - Usually sfs start at the method *main()* (m)!



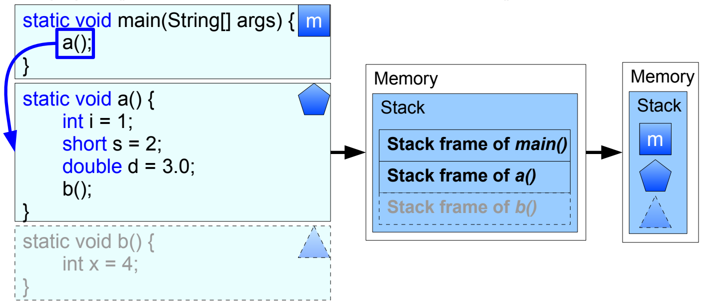
Good to know
Imagine, how a method, that calls itself recursively, could quickly floods the stack with multiple sf's! It can lead to a *java.lang.StackOverflowError* being thrown.

  - The way, how *a()* and *b()*'s sfs are pushed on the stack "stackwise" on top of each other, underscore why the stack is called stack.
  - IntelliJ IDEA's debugger represents sfs in opposite call order, i.e. bottom up:



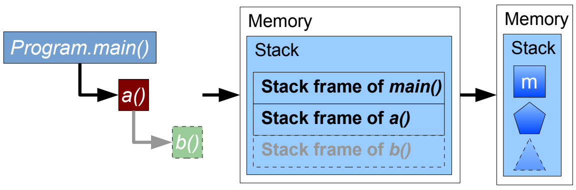
On the Stack – The Call Stack – Part II

- After *b()* returns, the JVM pops *b()*'s sf from the stack, this will remove *b()*'s sf from the memory.



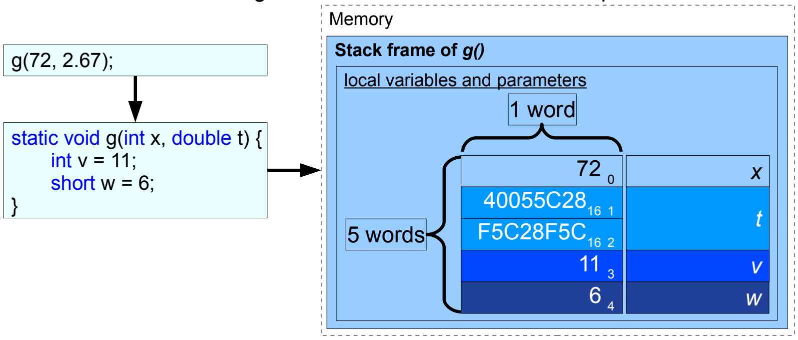
  - When *b()*' sf is removed from the memory, all its value objects are removed from memory as well.
  - Removing stack objects is a very fast operation Java, it is directly performed by the JVM.

- We've just discussed sfs from a "method call hierarchy perspective", the so called call stack perspective.



On the Stack – For Completeness: Parameters, Return Values and Addresses

- Parameters on the stack are also living in the section local variables and parameters

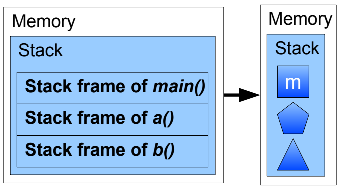


- As can be seen, parameters are just put before the other local variables in that section.
  - Read: they get the first indexes of the local variables and parameters array.
  - This means: parameters are treated like local variables by the JVM.
  - They also contribute to the call stack and have the lifetime of local variables.

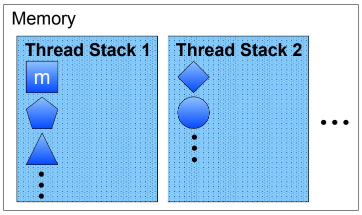
- Return values and the return address, to which a method returns, when its sf is removed are also stored in the sf.

On the Stack – Threads

- All right, so when methods are called, the JVM creates sfs for each method call on the stack:



- But, this is not the full truth: More exactly, the JVM creates a new stack frame for the current thread's stack.
  - A Java program is usually not a single big process, but it can consist of many smaller threads of execution, or short threads.

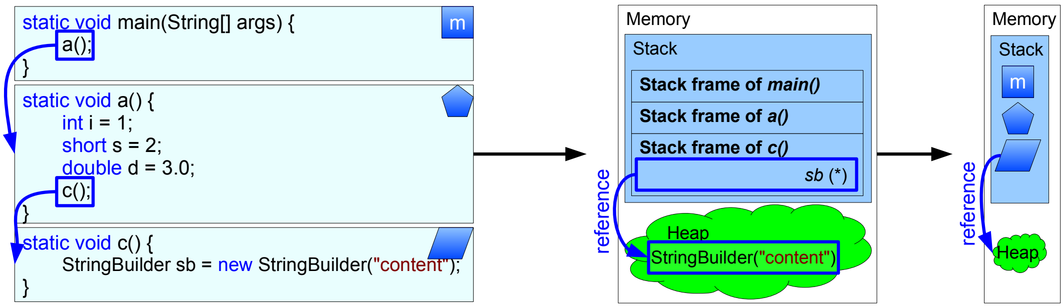


  - Interestingly, there can only be one thread, in which the *main()* method "lives".

- Actually, the idea of threads is very important in Java, but it is not important at this stage of the Java course.

Call Stack and Heap

- Let's discuss the method *c()*, which is called by method *a()*, which is called by *main()*:

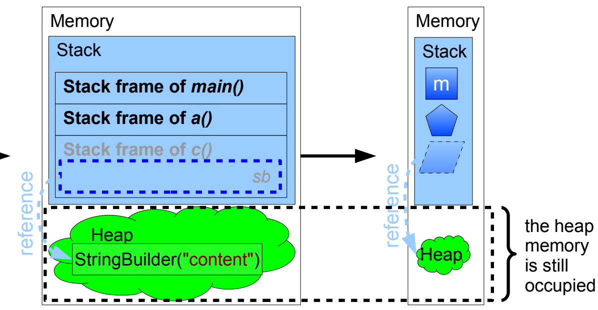


  - As we see, the usual call stack of *main()*, *a()* and *c()* is built.
  - The specialty of *c()*'s sf: the reference variable *sb* of type *StringBuilder* resides on its sf, but *sb* refers an object on the heap!
  - Objects of reference type, are referenced by variables on the stack! So there is a connection between stack and heap!
  - In our graphics the arrow connecting *sb* on the stack with the *StringBuilder("content")* object represents the reference!

Call Stack and Heap – Where does unused Memory go?

- After *c()* returns, we have this situation in memory:

```
static void main(String[] args) {
    a();
}
static void a() {
    int i = 1;
    short s = 2;
    double d = 3.0;
    c();
}
static void c() {
    StringBuilder sb = new StringBuilder("content");
}
```



  - The JVM popped *c()*'s sf from the stack, this will remove *c()*'s sf from the memory.
  - But <u>only the stack's memory occupied by *sb* is removed</u>! -> Only the <u>"shortcut"</u> to the heap was removed!
  - => I.e., after *c()* returned, the object <u>*StringBuilder("content")* is still occupying heap memory</u>!

- <u>Question: How can we remove *StringBuilder("content")* from the heap?</u>
  - <u>If we cannot remove objects like *StringBuilder("content")* from the heap, we would run short on memory very soon!</u>

Memory Consumption and Memory Release – Part I

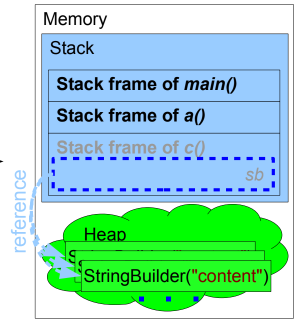
- To be frank, memory running short is a rare situation in Java… But why?
  - Remember: we will get *Exception*s, if the JVM runs out of memory, and we'll almost never have seen those *Exception*s...
- Up to now we only discussed, how to create objects, but what happens, when they're no longer needed?
  - Obviously, we can provoke errors by deliberate consumption of very large portions of memory during run time:

```
int[][][][] numbers = new int[Integer.MAX_VALUE][Integer.MAX_VALUE][Integer.MAX_VALUE][Integer.MAX_VALUE];
// Will throw a java.lang.OutOfMemoryError: Requested array size exceeds VM limit
```

- However, the same effect should gradually develop with heap memory not being cleaned up after we've finished with it.
  - This code creates an infinite amount of *StringBuilder("content")*s in the heap:

```
static void a() {
    for ( ; ; ) {
        c();
    }
}
```

```
static void c() {
    StringBuilder sb = new StringBuilder("content");
}
```



  - But nothing happens! The code can run for good, but no *Exception* is thrown!
  - How can this work? Is the heap somehow "infinite"?

Memory Consumption and Memory Release – Part II

- Of course heap memory is not infinite!
- After memory was used, it will be recycled and can be used as fresh memory for other purposes in the program.
  - (1) How do we know, when memory can be recycled?
  - (2) When do we have to do the recycling?

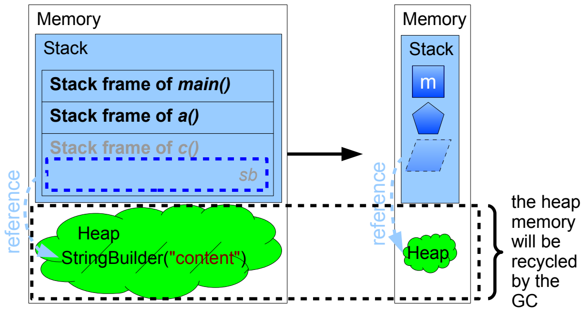
Good to know
An acceptable definition of the term "efficiency": used resources should be lead back into their cycles as fast as possible. This is what we call recycling.

- The answer to both question is: The JVM will do it for us!
- The JVM employs its own garbage collector (GC), which recycles unused memory.
  - How does it know, that memory is unused?
- The GC assumes, that an object in heap is no longer occupied, if no stack variable references it from somewhere.
  - The GC runs in its own background-thread during the "main" Java program is running.
  - The GC periodically scans the run time structure of the main program and counts references to objects in the heap.
  - When the GC reaches a count of 0 references, the heap memory of the now "orphaned" heap object, will be marked as "unused".
    - => This completes the recycling of the heap memory.
  - The memory manager is allowed to allocate unused heap memory, when we create new objects on the heap.

Memory Consumption and Memory Release – Part III

- In our example:

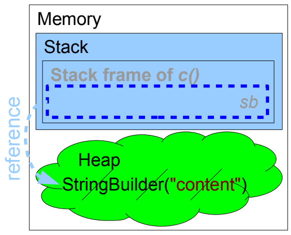
```
static void main(String[] args) {
    a();
}
static void a() {
    int i = 1;
    short s = 2;
    double d = 3.0;
    c();
}
static void c() {
    StringBuilder sb = new StringBuilder("content");
}
```



- All right! So, the GC counts references to *StringBuilder("content")* in the heap:
  - (1) *c()*'s sf is removed from the stack.
  - (2) The reference *sb*, which references *StringBuilder("content")* is removed from the stack together with *s()*'s sf.
  - (3) Somewhen, the GC notices, that there are no more references to *StringBuilder("content")*.
  - (4) The GC marks the heap memory occupied by *StringBuilder("content")* as unused.
    - Now, *StringBuilder("content")* loses its identity, it is no longer a "box" having a meaning as object in the heap.
  - (5) => Recycling of *StringBuilder("content")* completed!

GC Concepts

- GC concepts:
  - Generally, orphaned memory is memory, which is no longer addressed via reference variables.
  - In assumes, that objects created early during run time, live longer, that just recently created objects.
    - I.e. global objects live longer than local objects.
    - Therefor, the GC scans references to newer objects more often than older objects. It put objects into generations to allow exact scanning.
  - It does all this activities in the background! It means we don't know, when memory is really freed!

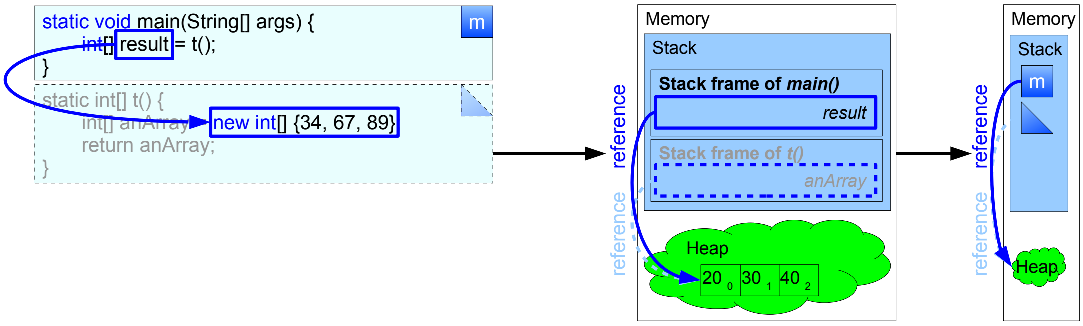


    - => The lifetime of *StringBuilder("content")* is completely independent from the reference *sb*, the GC will manage it.
- GC consequences:
  - The GC is a piece of software, doing its work in a background thread.
  - Therefor, removing objects from heap is much slower, than removing objects from the stack. The same is true for object creation!
  - Remember, that removing objects from stack comes to happen by popping sfs from the stack, which is super fast.

- *String* literals (or more accurately, the *String* objects that represent them) are historically stored in a section called the "permgen" heap. (Permgen is short for permanent generation.)
- Under normal circumstances, *String* literals and much of the other stuff in the permgen heap are "permanently" reachable, and are not garbage collected.

Reference Types – returning Arrays from Methods

- The next topic shows a <u>very relevant application</u> of the aliasing effect: <u>returning arrays from a method</u>.



- *result* in *main()* refers to the same int[] created in *t()*: int[] is created on the heap and survives sfs.
  - This is completely different from C/C++, where arrays can be created on the stack.

- <u>Notice, that the array created in *f()* is not gc'd!</u>

The GC was important for Java's Success

- When Java was introduced in 1995 it freed programmers from freeing memory generally with the GC.
  - In contrary to most other languages or IDEs that time, in which the programmer needed to care for memory explicitly.
  - Can we conclude from these statements, that memory (consumption) isn't a big topic in Java? – The answer is "yes", it isn't "big"!
  - However, there are situations, in which we have to care about object destruction in Java, but we'll come to this later.

```java
// Java. Creates an int[], which will be freed by the GC:
static void createsIntArray() {
        int numbers = new int[] {70, 80, 90};
} // numbers will be popped from the stack, and the
  // referenced memory in the heap will be "GC'd"
  // somewhen.
```

```cpp
// C++. Creates an int[] in the heap, which must be freed explicitly and
// manually:
void createsIntArray() {
        int* numbers = static_cast<int*>(std::malloc(sizeof(int) * 3));
        if (numbers) {
                numbers[0] = 70; numbers[1] = 80; numbers[2] = 90;
                std::free(numbers);   // Removes the memory referenced by
                                      // numbers from the heap.
        }
}
```

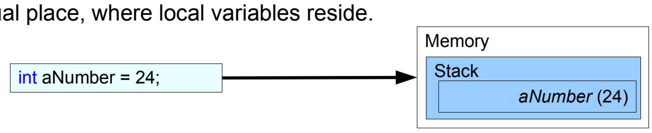
Good to know
To be frank, it must be pointed out, that memory-related terminology and the overall memory management of objects of value type and reference type works differently in Java compared to C/C++. Java's and C++' memory concepts cannot be directly compared and this example is really "contrived".

- Pretty much all modern programming architectures apply garbage collection:
  - It makes programming approachable to hobbyists.
  - The power of modern CPUs makes the parallel garbage collection process very unobtrusive.
  - But the idea of garbage collection is not new: e.g. Lisp, Smalltalk and esp. JavaScript.

- Lisp applies a garbage collector to manage consumed memory automatically. This is required for fp languages, because they deal with a lot of temporary data!

Stack vs. Heap: It's not a Mystery, just two Concepts – Part I

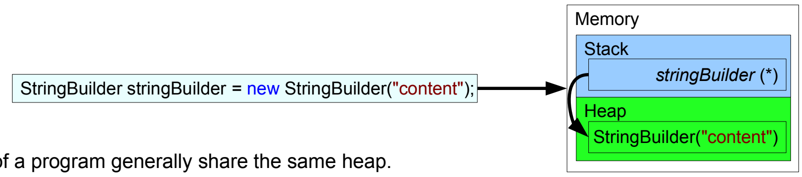
- When we create an object of value or reference type, the created object will consume memory!
  - Technically, objects are just regions in memory with a special meaning for an application.
- The stack is a conceptual place, where local variables reside.



  - Simply spoken each method has its own share of the stack called stackframe (sf).
  - Each thread of a running Java application has its own stack, and methods are executed "in threads" (read: in a thread's context).
- The lifetime of an object on the stack is bound to its scope. – The lifetime of an object on the stack is exactly defined.
  - The stack is conceptually controlled by the program's algorithm, esp. by how the scopes of local variables are organized.
  - In reality the stack is controlled by the JVM by pushing and popping sfs of called methods to and from the stack

Stack vs. Heap: It's not a Mystery, just two Concepts – Part II

- The heap is a conceptual place, where all the contents of references (read: referenced contents) reside.



  - All methods of a program generally share the same heap.
  - Usually, creation of objects on the heap is an implicit procedure, when objects of reference type/UDT are created.

- The lifetime of objects on the heap is not bound to the scopes of variables.
  - Their lifetime is also not controlled by the programmer.
  - In Java, objects allocated on the heap, will be deallocated automatically by the GC, when resources are no longer referenced.
  - A Java programmer does not even know, when the lifetime of an object on the heap ends.

Thank you!