Abstract: - Memory Management is an important area of a programming language for the development of various forms of software. In the safety critical systems, a special emphasis is placed on the error-free memory management operations. An unexpected error in memory management might lead to fatal consequences in safety critical systems. Each language has its own specific storage area management. This paper describes the memory management operations in two programming languages: Java and Ada. A close examination of the storage areas of these two languages also provides a deeper insight into the advantages and disadvantages of the storage area management in Java and Ada. The memory management in the Java language is automatic. It tries to use as little memory as possible, ensuring that enough memory is available. Ada does not have an automatic memory management and, therefore, the developer has to pay attention to the way the storage areas are released.

Key-Words: - Memory Management operation, Java, Ada, storage area, software reliability, safety

1 Introduction
The memory area of a computer is an important part of a framework in which the program is located. The integrity of memory content is a crucial factor that enables the program to work in a correct way. An unreliability of the memory content might cause a damage of the proper operations. Memory management is a central task of programs. There is usually much data in a program which is no longer accessible but still consuming memory. Releasing the memory areas can be done automatically or explicitly by programmers. Automatic memory management is realized in many modern languages by means of the garbage collection mechanism. An advantage of garbage collection is that the programmer does not have to take care of releasing the memory directly. This advantage is a big step in the world of programming languages. Due to the garbage collection mechanism, the following errors cannot occur: releasing an object that is still referenced and unused space is not released. Garbage Collection also has disadvantages such as no control over the time at which objects are released. Garbage Collection takes place at run time [8].

For the programming languages that do not have "garbage collection", the programmers have the task to release unused memory locations.

2 Memory Management in Java
The Java programming language is platform-independent and therefore can be used on all platforms which have the JVM installed. JVM ([Java Virtual Machine]) is, as the name says, a virtual machine located in the operating system, which interprets the Java source code, translates it and passes it as a machine code to the processor. With the JVM, the desired program can be executed. The JVM consists of different areas: method area, heap and threads (see Figure 1).
The method area is a subarea of the JVM, where each loaded class is managed. The method area is created when the JVM starts. Its size can be static or dynamic. The heap manages the class instances and arrays at the run-time and is available for all threads in the JVM. The heap is also created when the JVM starts and can have a static or dynamic size. Heap and Method area are available for all threads within a JVM [1].

For the heap's memory cleaning-up, JVM provides a garbage collection mechanism. Basically, in contrast to many other programming languages, a Java developer does not have to be concerned with cleaning-up the memory area. The heap memory cleanup is the main task of garbage collection. There are various algorithms used for this that ensure, at different points in time, that the space used by objects without any references is released. In the following sections, the various principles and algorithms are described.

2.1 Generational Garbage Collection

This section provides an overview of the General Garbage Collection. For the realization of this idea, the heap will be divided into different areas. The various areas include objects of different ages. Each area of the heap has its own algorithms, adapted to the age of the objects. Some of these objects do not “live” very long, for example, the objects that are defined locally within method. Furthermore, there are also heap objects, for instance, final static attributes of a class, which “live” very long. Figure 2 presents the life-time of an object.

There are lots of objects which are not very old, and relatively few objects with long life-time. In order to handle and process the different objects with different life-times appropriately, objects are assigned to different generations on the heap. Different generations have different garbage collection algorithms.

One of these generations is a “Young generation” which is cleaned frequently. The other one is the “Old generation”, which is rarely cleaned. The Young Generation is the storage area, in which the new objects are created. The Old Generation is the area containing the young objects, which reached a certain age, which means that it has survived a certain number of garbage collections. This division is rational, because young objects from the Young Generation die relatively fast, so this area should have been cleaned-up by the garbage collector more frequently. Consequently, there will be a free memory space available for newly defined objects. The cleanup processes in the Young Generation are called Minor Collections [2].

In the Old Generation, it is not worthwhile doing the frequent cleaning-up. When an object reaches its middle-age, it is highly possible, that it will get older. Therefore, “dead” objects are not frequently found in the old Generation, which means that the memory deallocation process in this area has lower potential. As a consequence, the memory cleanup in the Old Generation takes place considerably less frequently. Figure 3 presents the heap division into two generations (young and old) including a special area (perm).

![Fig. 3: The heap division into two generations](image)

The Perm-Area has nothing in common with the Generational Garbage Collection and the aging of the objects. Normal objects which appear during a program’s run-time, are never placed there. Instead, this area is used for a special object as String Constants, as well as Metadata (Information about classes, fields, methods etc.).

The Young Generation is divided into three parts: the Eden-Space and two Survivor-Spaces. The Eden-Area contains all the new objects, which are created with the “New” operator. One of the two Survivor-Space areas is basically always empty. The
used and empty areas are called, respectively, From-Space and To-Space [2].

If a minor garbage collection is performed, the objects that have survived from Eden-Space and from From-Space are copied into To-Space. Consequently, the areas in Eden and From-Space are free again. It swaps the From-Space and To-Space roles.

The Old Generation contains objects which became old – have survived several Garbage Collections in the Young Generation. There is a limit value called „Tenuring Threshold“. After the reaching of the age-limit, an object has become too old to be stored in the Young Generation. This moving process from Young to Old Generation is called a Promotion. Objects can also move from Young to Old Generation for other reasons. This takes place, for example, when too many objects that survived in the Young Generation won’t fit into the Survivor Space if they remain. Then, all of them will be moved to the Old Generation. The choice of an appropriate algorithm depends, by the way, on a JVM -implementation. Mostly, the division of heaps and as well as the choice of an algorithm are defined in JVM-implementation. In some cases, the Java developer has an impact on JVM settings, as far as the JVM configuration allows [2]. In the next section, some algorithms are presented, which are widely used in garbage collection. Moreover, it is explained which algorithms are better for which areas.

### 2.2 Mark and Sweep Algorithm

There are some algorithms for implementing the garbage collection. The main tasks of the algorithms are identical – they should destroy the free-standing objects which are no longer referenced and free the memory. There is a Mark and Sweep algorithm in Java which will be explained.

This algorithm is composed of two phases – marking and releasing the memory space. In the first phase, all objects are selected, which are reachable from the root pointer. Afterwards, objects which cannot be reached from the root pointer Garbage Collection are released, see Figure 4.

The advantage here is that cyclic data structures are recognized in a correct way. Another benefit is no overhead by pointer manipulations. There are also disadvantages: the larger the heap, the more complicated the algorithm. The next significant drawback is the memory fragmentation, caused by the Mark and Sweep algorithm. The objects unreachable from the root pointer will be marked as „dead“ and the memory will be released which, after a while, will result in many holes of different sizes in the memory area. Filling these memory gaps after the occurrence of “dead” objects is very expensive and difficult. As an example, a suitable hole for a new object that consists of an appropriate size for storing the object has first to be found. In the Old-Area of the heap, the Mark and Sweep algorithm can be used.

![Fig. 4: Mark and Sweep algorithm](image)

### 2.3 Mark and Copy Algorithm

The next clean-up algorithm, used by the Garbage Collection, is the Mark and Copy algorithm. In order to avoid the fragmentation problem from the Mark and Sweep algorithm, the Mark and Copy algorithm is used. In this algorithm, all living objects are selected. Afterwards, all living objects are copied into the free storage area side by side. Thus, there is no fragmentation of the memory area. For the memory area clean-up by this algorithm, two areas are needed: From-Space and To-Space. In From-Space, the new objects are created, as long as the From-Space is full. All living objects are recognized by the Garbage Collection cycle and copied into the To-Space close to each other. After the copying of all objects, the entire From-Space is released. Now the From-Space is empty and the To-Space full. In this case, both areas are swapped. The advantage of this algorithm is that it does not cause any memory fragmentation. A disadvantage is the memory consumption. There must be an empty space reserved which is not always needed.
2.4 Mark and Compact Algorithm

The Mark and Compact algorithm is an evolved Version of Mark and Sweep algorithm. Since in the Mark and Sweep algorithm the fragmentation has occurred, this drawback should be avoided in Mark and Compact algorithm. The Mark and Compact algorithm accomplishes a compaction of surviving objects and thereby reduces the fragmentation which can be seen as an advantage of this algorithm. The disadvantage of the Mark and Compact approach is that it is a serial algorithm, which means that it stops all threads of the application. This leads to long stop-the-world pauses in which the application is blocked [2]. An alternative for that is a parallel Mark and Compact algorithm. In this algorithm, threads are also stopped, but the entire algorithm's process is performed by multiple threads. Therefore, the resulting breaks are much shorter than in serial. The Mark and Compact algorithm is used by the Old-Area of the Heap.

2.5 Concurrent Mark and Sweep

The Concurrent Mark and Sweep algorithm was developed for the Old Generation, and can be executed concurrently to the current application. The operation method of Concurrent Mark and Sweep is similar to the Mark and Sweep algorithm in which the problem of heap fragmentation cannot be avoided. On the other hand, there is a big performance advantage, which significantly shortens the stop-the-world breaks [3].

2.6 Garbage First

Garbage First (G1) is a new implementation of the memory management in the JVM. The G1 was designed in order to improve the memory behaviour of a JVM with large heap, and to reduce the length of pauses. The G1 divides the heap into \( n \) subareas of the same size. The size of the areas might vary between 1 and 64 MB. There are four different subareas in the G1, namely the empty area, Eden area, Survivor area and Old area. The “age” of an object is defined on the basis of how long it has been on the heap. Figure 7 illustrates the division of the heap area.

It is important that the range of the heap size is defined by the G1 when a JVM gets started, since it is impossible to define the range during run time. As already mentioned, the new objects are allocated in the Eden-Area. Each of the above mentioned areas has a so-called "bump pointer" which indicates the next free memory location in the region. Each area contains a so-called „Remembered Set“. This "Remembered Set" includes all references from other areas. In order to keep the "Remember Set" up-to-date, a write barrier will be added when allocating new objects. In this write barrier, it is checked whether this object and the newly allocated object are in the same region or not [2].
For the purpose of memory clean-up, G1 selects all areas that have the highest number of unreachable objects. Therefore, it can release more space. Those areas are saved in a „Collection Set“. In the next Garbage Collection passage, all areas in „Collection Set“ will be processed. G1 has got two modes: „Fully Young Mode“ and „Partially Young Mode“, with which it cleans up the memory. „FullyYoung Mode“ cleans up all Young-Areas and „Partially Young Mode“ additionally Old-Areas. When more than a half of the heap has been filled (approx. 80%) a Full Garbage Collection is performed. This means that all regions in the Collection Set are included [3].

2.7 Java in safety critical systems
There are some features in Java which make them difficult to use in a safety critical system, for example the automatic memory management. Since the behaviour of garbage collection is often non-deterministic, its use in safety critical systems is inappropriate. Moreover, the garbage collection requires an unpredictable amount of time for performing its work. In addition, despite the automatic memory management in Java, some Memory leaks or Out-Of-Memory-Errors can occur. The reason is that the objects are no longer used in further program run-time, but they are accessible via a chain of references from the root reference. The Garbage Collector treats the above-mentioned objects as „living“, and never releases. A single object does not cause problems, but if the process is repeated, the number of dead objects that are considered as active increases. Consequently, the JVM can crash after some time with an Out-Of-Memory-Error.

In order to use Java in safety critical systems it should be extended to Real-Time Specification for Java (RTSJ). RTSJ allows soft and hard real-time to develop applications in Java. Soft Real-Time requirements are defined as follows. All program statements are typically processed quickly, but the response time may exceed the given upper time boundary. This is not considered as a failure of the application as long as the defined tolerance value is not exceeded. By hard requirements, the upper time limit must not be exceeded, since this can lead to catastrophic consequences. In this case, the system fails completely [5].

Java cannot meet the mentioned requirements because the essential Java Garbage Collector acts as a single thread and the application can be interrupted at any time due to the memory management. In real-time applications, it is not acceptable because the guarantee of not exceeding the upper time limits is violated. Another problem with Java is that it does not have any threading model. This means that threads have no priorities. The RTSJ was developed in order to solve these problems. Thus, the Garbage Collection is exact and deterministic.

RTSJ extends the JVM to further threading models, which allow real-time Java thread to act with a higher priority than the Garbage Collection thread use. Therefore, a higher-priority thread is given permission to interrupt a thread with a lower priority, even if the system is busy with the memory management at the same time. However, the thread is not allowed to access objects that are in the area of memory management.

Recently, the use of the Java in safety critical systems has been extensively discussed, since the potential of Java was recognized and the safety critical systems have become more complex. The advantage of Java here is that, with the increasing code complexity, it still remains manageable. On the other hand, this language brings much dynamic, which in turn is an argument against using it in critical systems. Aspects like inheritance, polymorphism, type conversion, function overloading, exception handling and dynamic memory management have to be certified in order to pave the way for the development of safety-critical systems with Java.

3 Memory Management in Ada
The memory management in Ada is dynamic. Ada does not use generic and error-prone access types.

The Ada's semantic allows the activation of a garbage collection, but, because it can lead to unexpected problems in real-time systems, this option is disabled by default. Implementations of Ada usually do not use the automatic garbage collection for memory management. Ada supports runtime tests in order to detect memory overflow, access to non-allocated memory and off-by-one errors in time and to avoid these errors [11].

3.1 Storage pool
The storage pool is the standard tool for memory management in Ada. The storage pools are dynamic
and have fixed block sizes, because of the dynamic memory management, since blocks of any size leads to memory fragmentation. The predefined storage pool is sufficient for the GNAT run-time environment, but the user can also create own memory pools. GNAT is the compiler of Ada.

Each access type defines a storage pool. This does not mean that certain parts of memory are permanently reserved for the data of a specific type. The allocated memory is released when the lifetime of the storage pool has ended [9].

The use of untested release and uncontrolled conversion can lead to incorrect memory references. In order to avoid such problems, the "GNAT.Debug_Pools" storage pool should be used [7].

3.2 Restriction
Restrictions are general mechanisms of programming languages that prohibit access to some features of the program. In the following function

```ada
pragma Restrictions(No_Dependency => Unchecked_Deallocation);
```

the program prohibits the Unchecked_Deallocation function from being invoked, so that as a consequence, no memory can be released. The compiler interrupts the program if this function is not valid. There are some restrictions below which are responsible for the memory management:

```ada
pragma Restrictions(No_Allocators);
pragma Restrictions(No_Implicit_Heap_Allocations);
```

The first restriction prohibits the call of „new”, for instance, „new Cell”, and makes it impossible to use the heap explicitly. The second restriction ensures that temporary objects do not have access to the heap [6].

Problems that appear by anonymous access types can be solved by means of a special "Restriction identifier":

```ada
pragma Restrictions(No_Anonymous_Allocators);
```

This restriction prevents anonymous access types from being used and thus blocks the calling of these functions.

3.3 Ada in safety critical systems
For the safety critical systems, the Ada programming language is the choice of most programmers. Despite this fact, Ada struggles with the following three problems, which cannot always be detected:

- Memory leaks (when the user forgets to release allocated memory)
- Incorrect memory deallocation (when memory is allocated which was not at all allocated)
- Hanging references (when referring to something that was previously released)

Hanging references are created when the explicit memory deallocating of Ada uses “Unchecked_DeAllocation”. As the name says, the “Unchecked_DeAllocation” does not check by releasing the references whether other references refer to this one. For example, this can lead to "hanging-references": Pointer “A” points to “X”, pointer “B” points to “A” and then, the memory is released with the free “A” method. The problem here is that B still refers to A, but A is zero.

All Problems from the memory allocations are assigned to Storage Error. Storage_Error belongs to the Ada Standard Exceptions. The Storage_Error may occur for memory allocation through the "NEW operator" if there is no free space available on the "heap". The Storage_Error can also occur if, during a "subroutine call", the space for local variables is no longer available. Dynamic memory management in Ada is one of the biggest weaknesses for safety critical systems. Whenever possible, a dynamic instantiation should be dispensed after a start-up and an initialization of the program should be carried out [10].

Safety critical systems will be written in Ada in the future, since these languages have stronger real-time system paradigms than Java.

4 Conclusion
This report shows how the memory management of two programming languages, Ada and Java, works. In safety critical systems, Ada has its strengths, such as:

- Tests during compilation
- Tests at runtime
- Static, explicit and strong typing
- Portability
• Interfaces to other programming languages

The Ada programmer must know the memory management mechanisms of this language very well, otherwise memory areas which are already in use can be allocated. This would lead to a fatal error in the application.

In Java, the garbage collection assumes this task and the programmer is only concerned with the logic of the application. As already mentioned, the Garbage Collection is responsible for the memory management in Java, but it does not meet the requirements for being used in safety critical systems, since it is not deterministic and can interrupt the application at any time.

Java Real Time Specification for Java (RTSJ) allows the use of real-time-capable threads, which can block the garbage collection until their tasks are finished. Thus, this specification can be used in safety critical systems.

References:
[3] Patrick Schlott, Christoph Schröder, Garbage Collection, Freie Universität Berlin, 08.05.2012