哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

Chapter 8: Software Construction for Performance

8.1 Metrics, Principles, and Methods of Construction for Performance

Ming Liu

May 9, 2018

Outline

- **1 Performance Metrics**

- **2 Memory Allocation**
  - Three modes of object management: Static, Stack-based, and Heap-based

- **3 Java Memory Model**

- **4 Garbage Collection**
  - Living without Automatic GC
  - Living with Automatic GC

- **5 Basic Algorithms of Garbage Collection**
  - Reference counting
  - Mark-Sweep
  - Mark-Compact
  - Copying

Outline

- **6 Garbage Collection in JVM**

- **7 Garbage Collection Tuning in JVM**

- **8 Using I/O Efficiently in Java**

- **Summary**

1 Performance Metrics

Performance

- **Performance is the amount of work accomplished by a computer system.**
  - Short response time for a given piece of work
  - Low utilization of computing resource(s) –CPU, memory, disk, etc.

Runtime program performance

- **Time performance**
  - **Execution time** for a single statement, for a complex structure (such as a loop, I/O, network communication, etc), for the whole program;
  - **Distribution of execution time** of different statements/structures (relative time distribution)
  - **Time bottleneck** of a program's execution
- **Space performance**
  - **Memory consumption** for a single variable, for a complex data structure, and for the whole program;
  - **Distribution of memory consumption** of different variables/data structures (relative space distribution)
  - **Space bottleneck** of a program's execution
  - **Evolution of memory consumption** along with time

Factors influencing runtime performance

- **Space performance:**
  - Memory allocation
  - Garbage collection

- **Time performance:**
  - Basic statements
  - Algorithms
  - Data structure
  - I/O
  - Network communication
  - Concurrency / multi-thread / lock

- **In general, a operation is considered as extensive if this operation has a long runtime or a high memory consumption.**

How to get memory consumption?

- **The total used / free memory of an program can be obtained in the program via** `java.lang.Runtime.getRuntime()`, **and the runtime has several method which relates to the memory.**

```java
public class PerformanceTest {
    private static final long MEGABYTE = 1024L * 1024L;

    public static long bytesToMegabytes(long bytes) {
        return bytes / MEGABYTE;
    }

    public static void main(String[] args) {
        // I assume you will know how to create a object Person yourself...
        List<Person> list = new ArrayList<Person>();
        for (int i = 0; i <= 100000; i++) {
            list.add(new Person("Jim", "Knopf"));
        }
        // Get the Java runtime
        Runtime runtime = Runtime.getRuntime();
        // Run the garbage collector
        runtime.gc();
        // Calculate the used memory
        long memory = runtime.totalMemory() - runtime.freeMemory();
        System.out.println("Used memory is bytes: " + memory);
        System.out.println("Used memory is megabytes: "
                + bytesToMegabytes(memory));
    }
}
```

How to get execution time?

- **Use `System.currentTimeMillis()` to get the start time and the end time and calculate the difference.**

```
class TimeTest1 {

    public static void main(String[] args) {

        long startTime = System.currentTimeMillis();

        long total = 0;

        for (int i = 0; i < 10000000; i++) {

            total += i;

        }

        long stopTime = System.currentTimeMillis();

        long elapsedTime = stopTime - startTime;

        System.out.println(elapsedTime);

    }

}
```
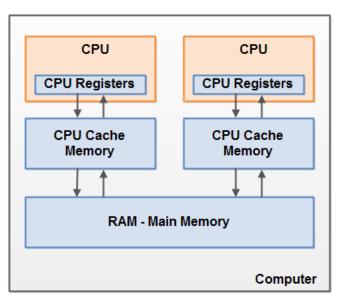
2 Memory Allocation

Recall: Memory handling in OS

- **Native memory** is the memory which is available to a process, e.g. the Java process.

- **Native memory is controlled by OS** and based on physical memory and other physical devices, e.g. disks, flash memory, etc.

  - CPU computes the instructions to execute and stores its computation results into registers. It can access the normal memory over the memory bus.

  - OS normally uses virtual memory to map the physical memory to memory which each process can see. It assigns memory to each process in a virtual memory space for this process and maps access to this virtual memory to the real physical memory.

Memory Management

- **Memory management** is a form of resource management applied to computer memory.

  - The essential requirement of memory management is to provide ways to dynamically allocate portions of memory to programs at their request (memory allocation)

  - Free it for reuse when no longer needed (garbage collection).

- Modern general-purpose computer systems manage memory at two levels:

  - OS level

  - **Application level**

What is the problem?

- **Memory is a scarce and precious resource!**

- **If we had unbounded amounts of memory, we'd never worry.**

- **The PROBLEM is that we don't have unbounded memory.**

What can we do?

- **Dynamically allocate and deallocate memory.**

- **REUSE deallocated memory.**
  - Allocation means finding a free piece of memory in the heap and reserving it for the representation of an object.
  - Deallocation means changing the status of a piece of memory from allocated to free.

- **Dynamical memory allocation is available in many languages, e.g., using languages features:**
  - New() or Constructors   allocates a new object
  - Delete()   deallocated the object X

- **Such features allows programmer to handle allocation themselves.**

Object Model

- **Objects are represented in memory by some number of bits stored contiguously in memory.**

  – They consist of a header (not visible to the user program, called mutator) and zero or more fields.

- **Objects can contain references to other objects.**

  – A reference to an object is usually implemented merely by the memory address of the piece of memory where the object is stored.

- **Basic operation for allocating space to new objects:**

  – Create a new object;

  – Attach it to the reference x;

  – Initialize its fields.

  – E.g., `Student Peter = new Student("18S01558","Computer");`

Object Model

- **Objects are typically allocated in the heap.**

- **An Object Reference is a pointer to an object (typically merely the heap address of the start of the object).**

- **Variables contain object references (ignoring primitive data).**

- **Each object can contain a number of variables and thereby reference other objects.**

- **A reference is attached to at most one object, an object may be attached to two or more references.**

Three modes of object management

Static & dynamic allocation

- **Static allocation — allocation takes place when a program starts – basically memory is laid out by the compiler**

- **Dynamic allocation — allocate new objects while the program is executing.**
  - A simple form of dynamic allocation is stack allocation where objects are allocated on the program stack and deallocated using a stack discipline.
  - Heap allocation is the most general form of allocation: objects are allocated in the heap.

Three modes of object management

- **The form of object management in OO is only one of three commonly found modes:**

  - Static

  - Stack-based

  - Heap-based (free)

- **The choice between these modes determines how an entity can become attached to an object:**

  - An entity is a name in the code representing a run-time value, or a succession of run-time values. Such values are either objects or (possibly void) references to objects. Entities include attributes, formal routine arguments, local entities of routines and Result.

  - The term attached describes associations between entities and objects: at some stage during execution, an entity x is attached to an object O if the value of x is either O (for x of expanded type) or a reference to O (for x of reference type).
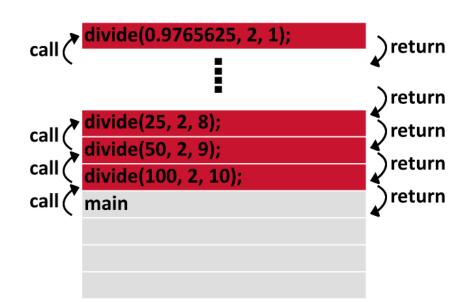
Static mode
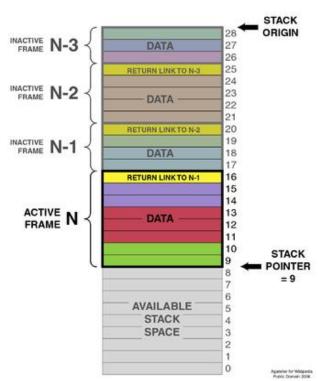
- **In the static mode, an entity may become attached to at most one run-time object during the entire execution of the software.**

- **This is the scheme promoted by languages such as Fortran, designed to allow an implementation technique which will allocate space for all objects (and attach them to the corresponding entities) once and for all, at program loading time or at the beginning of execution.**

  Objects

  Fixed Memory Area

  - It precludes recursion, since a recursive routine must be permitted to have several incarnations active at once, each with its own incarnations of the routine's entities.

  - It also precludes dynamically created data structures, since the compiler must be able to deduce the exact size of every data structure from the software text.
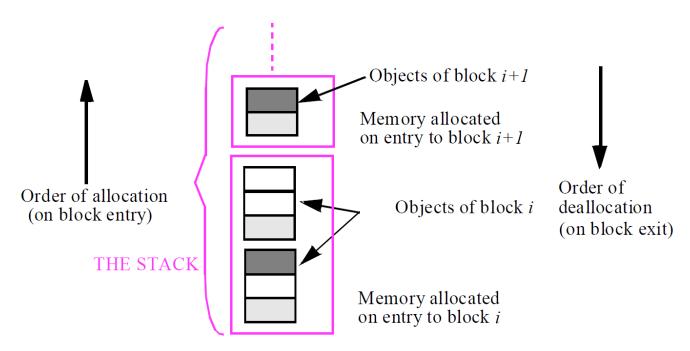
Stack-based mode

- **Stack** is where the method invocations and the local variables are stored.

  - If a method is called, its stack frame is put onto the top of the call stack.

  - The stack frame holds the state of the method including which line of code is executing and the values of all local variables.

  - The method at the top of the stack is always the current running method for that stack.

Stack-based mode

- **An entity may at run time become attached to several objects in succession, and the run-time mechanisms allocate and deallocate these objects in last-in, first-out order in the stack.**

- **When an object is deallocated, the corresponding entity becomes attached again to the object to which it was previously attached, if any.**

Objects of block $i+1$

Memory allocated
on entry to block $i+1$

Order of allocation
(on block entry)

THE STACK

Objects of block i

Order of
deallocation
(on block exit)

Memory allocated
on entry to block i

Stack-based mode
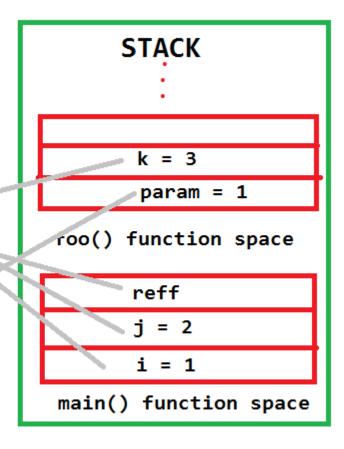
```
public class Stack_Test {

    public static void main(String[] args) {

        int i=1;
        int j=2;

        Stack_Test reff = new Stack_Test();
        reff.foo(i);
    }

    void foo(int param) {
        int k = 3;
        System.out.println(param);
    }

}
```
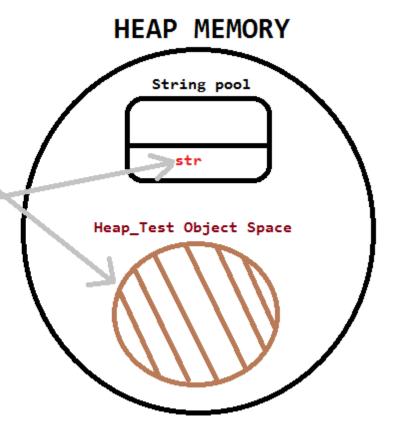
STACK

| k = 3 |
| param = 1 |

foo() function space

| reff |
| j = 2 |
| i = 1 |

main() function space

Heap-based mode

- **A heap which is a part of memory that is split into many pieces each of which either contains the representation of an object or is unused (in which case we say it is free).**

- **Heap-based mode is called free mode. This is the fully dynamic mode in which objects are created dynamically through explicit requests.**

- **An entity may become successively attached to any number of objects; the pattern of object creations is usually not predictable at compile time. Objects may, furthermore, contain references to other objects.**

- **The free mode allows developers to create the sophisticated dynamic data structures.**

Heap-based mode

Three modes of object management

- **Some applications can manage with a bounded amount of memory using static allocation combined with stack allocation.**

- **Others use dynamic allocation of memory because:**
  - Some objects live longer than the method that creates them.
  - Recursive data structures such as lists and trees.
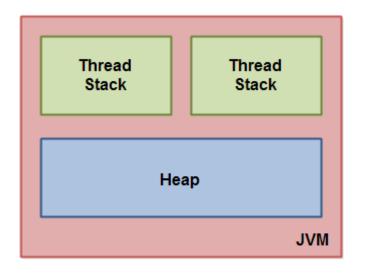 - Avoids fixed hard limits on data structure sizes.
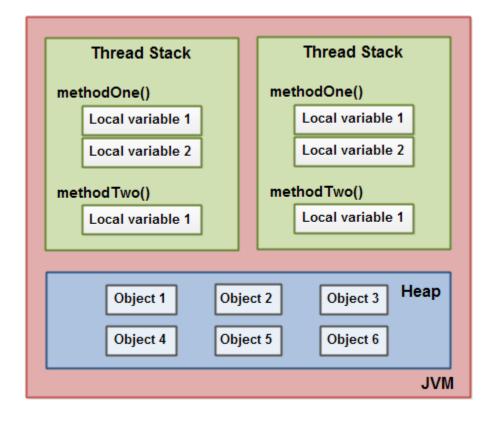
3 Java Memory Model

Memory in Java

- **The Java memory model specifies how the Java virtual machine works with the computer's memory (RAM).**

- **JVM is a model of a whole computer so this model naturally includes a memory model (Java memory model).**

  - New objects created and placed in the heap.

  - Once your application have no reference anymore to an object the Java garbage collector is allowed to delete this object and remove the memory so that your application can use this memory again.

- **The original Java memory model was insufficient, so the Java memory model was revised in Java 1.5.**

- **This version of the Java memory model is still in use in Java 8.**

The Internal Java Memory Model

- **The Java memory model used internally in the JVM divides memory between thread stacks and the heap.**

The Internal Java Memory Model: Stack

- **Each thread running in JVM has its own thread stack.**

  - The thread stack contains information about what methods the thread has called to reach the current point of execution.

  - It is called "call stack". As the thread executes its code, the call stack changes.

- **The thread stack also contains all local variables for each method being executed (all methods on the call stack).**

  - A thread can only access it's own thread stack.

  - Local variables created by a thread are invisible to all other threads than the thread who created it.

  - Even if two threads are executing the exact same code, the two threads will still create the local variables of that code in each their own thread stack.

  - Thus, each thread has its own version of each local variable.

The Internal Java Memory Model: Heap

- **All local variables of primitive types (`boolean`, `byte`, `short`, `char`, `int`, `long`, `float`, `double`) are fully stored on the thread stack and are thus not visible to other threads.**

- **One thread may pass a copy of a pritimive variable to another thread, but it cannot share the primitive local variable itself.**

- **The heap contains all objects created in your Java application, regardless of what thread created the object.**

- **This includes the object versions of the primitive types (e.g. Byte, Integer, Long, etc).**

- **It does not matter if an object was created and assigned to a local variable, or created as a member variable of another object, the object is still stored on the heap.**
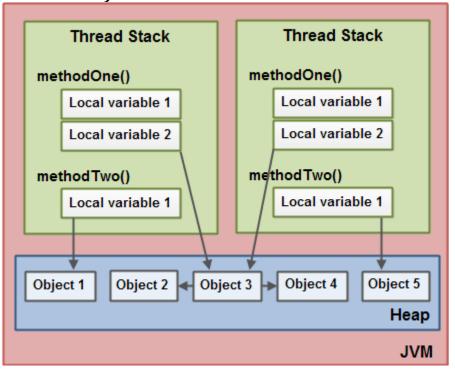
Some key points about Java Memory Model

- **A local variable may be of a primitive type, in which case it is totally kept on the thread stack.**

- **A local variable may also be a reference to an object. In that case the reference (the local variable) is stored on the thread stack, but the object itself if stored on the heap.**

- **An object may contain methods and these methods may contain local variables. These local variables are also stored on the thread stack, even if the object the method belongs to is stored on the heap.**

- **An object's member variables are stored on the heap along with the object itself. That is true both when the member variable is of a primitive type, and if it is a reference to an object.**

- **Static class variables are also stored on the heap along with the class definition.**

Some key points about Java Memory Model

- **Objects on the heap can be accessed by all threads that have a reference to the object.**

- **When a thread has access to an object, it can also get access to that object's member variables.**
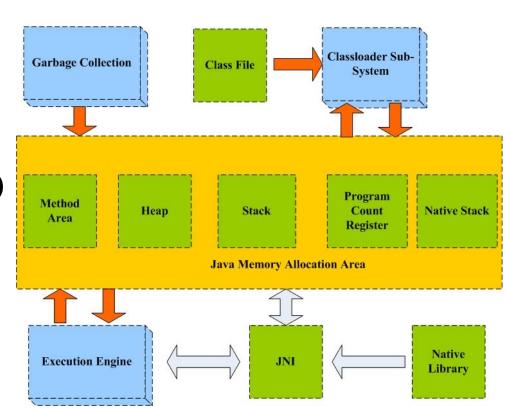
- **If two threads call a method on the same object at the same time, they will both have access to the object's member variables, but each thread will have its own copy of the local variables.**

Memory Structure of Java Virtual Machine (JVM)
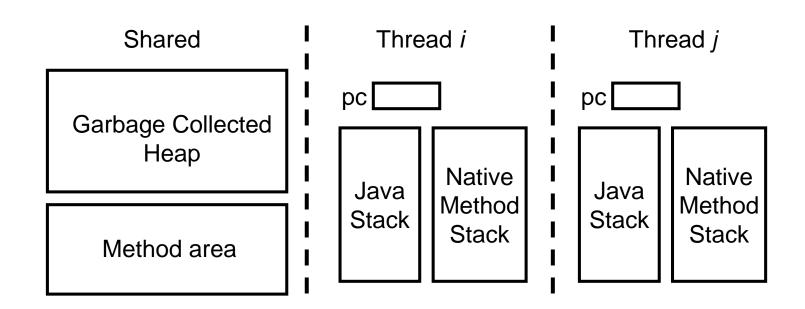
- **Stacks**

- **Native Stacks**

- **Heap**

- **Program Counter Register (PC)**

- **Method Area**

Memory Structure of Java Virtual Machine (JVM)

- **Besides OO concepts, JVM also supports multi-threading. Threads are directly supported by the JVM.**

- **Two kinds of runtime data areas:**
  - Shared between all threads
  - Private to a single thread

  To be discussed in Chapter 10

Shared	Thread *i*	Thread *j*
Garbage Collected Heap	pc	pc
Method area	Java Stack / Native Method Stack	Java Stack / Native Method Stack

Memory Structure of Java Virtual Machine (JVM)
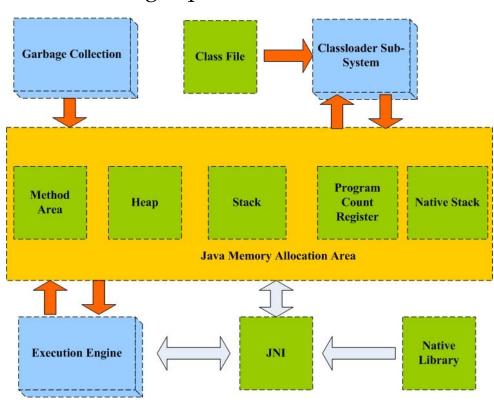
- **The stack is used to:**
  - Pass arguments to methods
  - Return a result from a method
  - Store intermediate results while evaluating expressions
  - Store local variables

- **The heap is used to:**
  - Objects
  - Arrays
  - By using "new" operators

Garbage Collection	Class File	→	Classloader Sub-System

Java Memory Allocation Area

Method Area	Heap	Stack	Program Count Register	Native Stack

Execution Engine	JNI	Native Library

4 Garbage Collection

Space reclamation in static modes

- **The ability to create objects dynamically, as in the stack-based and heap-based modes, raises the question:**

  – What to do when an object becomes unused: is it possible to reclaim its memory space, so as to use it again for one or more new objects in later creation instructions?

- **In the static mode, the problem does not exist:**

  – For every object, there is exactly one attached entity;

  – Execution needs to retain the object's space as long as the entity is active.

  – So there is no possibility for reclamation in the proper sense.

Space reclamation in stack-based modes

- **With the stack-based mode, the objects attached to an entity may be allocated on a stack.**

  - Block-structured language make things particularly simple: object allocation occurs at the same time for all entities declared in a given block, allowing the use of a single stack for a whole program.

Dynamic Property (event at execution time)	Static Property (location in the software text)	Implementation Technique
Object allocation	Block entry.	Push objects (one for each of the entities local to the block) onto stack.
Object deallocation	Block exit.	Pop stack.

Space reclamation in heap-based (free) modes

- **With the free mode, things cease to be so simple.**

- **The problem comes from the very power of the mechanism: since the pattern of object creation is unknown at compile time, it is not possible to predict when a given object may become useless.**

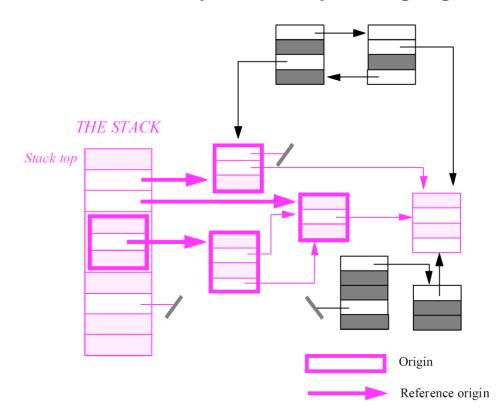# Reachable Objects vs. Unreachable Objects

- **At any point during the execution of a system, the set of origins / roots is made of the following objects:**

  - The system's root object.

  - Any object attached to a local entity or formal argument of a routine currently being executed (including the local entity Result for a function).

- **Any dependent, direct or indirect, of these origins/roots is reachable, and any other object is unreachable:**

  - For an unreachable object, it is possible to reclaim the memory it occupies (for example to recycle it for other objects) without affecting the correct semantics of the system's execution

- **The first step towards addressing the problem of memory management under the free mode is to separate the reachable objects from the unreachable ones.**

What are the roots?

- **What are the roots of a computation?**
  - Determining roots is, in general, language-dependent
  - Depends on the run-time structure defined by the underlying language.

- **In common language implementations roots include**
  - Words in the static area
  - Registers
  - Words on the execution stack that point into the heap.

Live objects vs. dead objects

- **The objects and references can be considered a directed graph:**
  The live objects of the graph are those reachable from a root.
  The process executing a computation is called the mutator because it is viewed as dynamically changing the object graph.

Live objects (in color) and dead objects (in black) in a combined stack-based and free model

Tracing reachable/unreachable objects

- **We can formalise our definition of reachability:**

$$\mathbf{live = \{ N \in Objects \mid (\exists\, r \in Roots\,.\,r \rightarrow N) \ \lor\ (\exists\, M \in live\,.\,M \rightarrow N) \}}$$

- **We can encode this definition simply:**
  - Step1: Start at the roots; the live set is empty
  - Step2: Add any object that a root points at to the live set
  - Step3: Repeat
    Add any object a live object points at to the live set
    Until no new live objects are found
  - Step4: Any objects not in the live set are garbage

What is garbage?

- **Almost all garbage collectors assume the following definition of live objects called liveness by reachability: if you can get to an object, then it is live.**

- **More formally, an object is live if and only if:**
  - It is referenced in a predefined variable called a root,
  - Or, it is referenced in a variable contained in a live object (i.e. it is transitively referenced from a root).

- **Non-live objects are called dead objects, i.e. garbage.**

Garbage Collection

- **Identifying garbage and deallocating the memory it occupies is called Garbage Collection(GC).**

- **Why is garbage collection needed?**
  - Language requirement: many OO languages assume GC, e.g. allocated objects may survive much longer than the method that created them
  - Problem requirement: the nature of the problem may make it very hard/impossible to determine when something is garbage

Garbage Collection

- **We can try to handle the garbage collection housekeeping chores related to object allocation and deallocation ourselves.**

- **Such housekeeping can be simple but for many applications the chores become complex – and error prone.**

- **Can we do it ourselves?**

- **Should it be automatic??**

Living without Automatic GC

Living without GC: Doing it yourself

- **Defensive programming**

- **Pairing Principle**

- **The Ownership concept**

- **Monitoring technique**

- **Administrator technique**

- **Tools**

- **Living without GC in C++**

Defensive programming

- **Defensive strategies for doing without GC**

- **Sharing:**
  - Copying objects instead of sharing.
  - Transform a global deallocation decision to a local one

- **Can be wasteful of space, but can be useful.**

- **Example:**
  - Everyone gets their own set of lights: turn your lights off when leaving. (Simple, but obviously wasteful.)

Pairing Principle

- **For each `new()` pair it with a `delete()`, make sure there is a one-to-one correspondence.**

- **For every `new()` check that the corresponding `delete()` is there.**

- **One way is to have the allocating object also be the deallocating object.**

- **Example**

  - If you turned the light on, YOU turn it off.

  - Allocate in the constructor; deallocate in the destructor.

```
Class A {
   Xclass X;
   void A()  { X = new Xclass;}
   void ~A() { delete X;}
   . . .
}
```

Ownership Concept

- **Observation: Objects are often passed around**
  - Thus it is often an object other than the creator who must do the deallocation.

- **Ownership Concept**
  - Initially, the allocating object is the owner of the newly allocated object. When passing a reference to the allocated object, the ownership can also be passed. (The previous owner should throw away the reference – it may become dangling very soon!)
  - Only the owner is allowed to deallocate the object; the last owner does the deallocation.
  - Each owner either passes on the ownership rights – or deallocates.

Monitoring Technique

- **A simple mechanism to help find bugs is to maintain a table of allocated objects.**

- `Malloc()` **is replaced by a version that store the address of newly allocated objects in a table.**

- `Free()` **is replaced by a version that checks the table before freeing.**

- **Such monitoring can help find bugs:**
  - memory leaks (the table will fill with a particular type of object),
  - dangling references (the table can be checked to see, if a reference is valid before using it)
  - double deallocations (free will protest if a non-allocated object is free'd).

Multiple Owners: Shared Objects

- **Handling Shared Objects Using Reference Counting**
  - We could attempt to handle shared objects by trying to keep track of multiple owners, e.g., by expanding the monitoring table by a count field.
  - For every new owner, we must increment the count.
  - And decrement it every time an owner is done with the object.
  - When the last owner is done (the count goes to zero), the object is deallocated.

  - Requires extra code for deallocation, allocation, copying of references, etc.

Dangers of Explicit Deallocation

- **With explicit deallocation, the programmer ends up:**
  - Doing too little
    - Garbage objects are not deallocated and slowly but surely clutters memory and so the program runs out of memory (such a failure to delete garbage objects is called a memory leak).
  - Doing too much
    - Throwing away a non-garbage object. Subsequent use of a live reference to the object will cause the program to fail in inexplicable ways. Such a reference is a dangling reference.
    - Throwing away a garbage object twice! Likely to break the memory manager.

The REAL bad thing about explicit deallocation

- **The problems are real and omnipresent in explicit deallocation systems and they cause the real problem**
  - Memory leaks
  - Dangling references
  - Double deallocation

- **Wasting huge amounts of debugging time!**

- **Despite this, programs may still fail in mysterious ways long after being put into production.**

- **Finding and fixing MM bugs can account for 40% of debug time.**

Living with Automatic GC

Why AUTOMATIC Garbage Collection?

- **Because human programmers just can't get it right:**
  - Too little is collected leading to memory leaks
  - Too much is collected leading to broken programs.

Memory leaks?

- **Human programmers can:**
  - Forget to delete an object when it is no longer needed.
  - Return a newly allocated object – but when will it be deallocated?
  - Not figure out when a shared objects should be deleted.

- **Sharing is a significant problem**
  - Can be handled by using the principle last one to leave the room turns off the light.

- **However, this is easily forgotten, and, worse, in a large building, it can be close to impossible to detect that you are the last!**

Dangling Pointers

- **Eager human programmers can delete objects too early leading to dangling pointers**

- **Consider an object that is shared between two different parts of a program each having its own pointer to the object. If one of the pointers is deleted then the other pointer is left pointing to a non-existent object – we say that it is a dangling pointer.**

  - Memory leaks and dangling pointers are two sides of the same coin:
  - The difficulty is managing objects in the presences of sharing.

5 Basic Algorithms of Garbage Collection

The basic algorithms of GC

- **The main task of GC is to separate the reachable objects from the unreachable ones(live objects from dead objects).**

- **Reference counting:**
  - Keep a note on each object in your garage, indicating the number of live references to the object. If an object's reference count goes to zero, throw the object out (it's dead).

- **Mark-Sweep:**
  - Put a note on objects you need (roots).
  - Then recursively put a note on anything needed by a live object.
  - Afterwards, check all objects and throw out objects without notes.

The basic algorithms of GC

- **Mark-Compact:**
  - Put notes on objects you need.
  - Move anything with a note on it to the back of the garage.
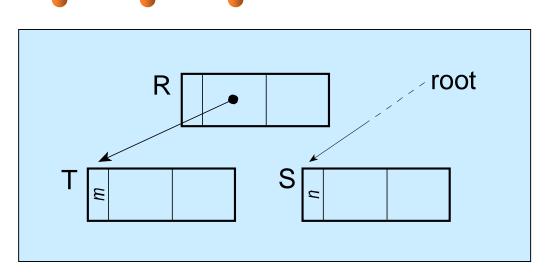 - Burn everything at the front of the garage (it's all dead).
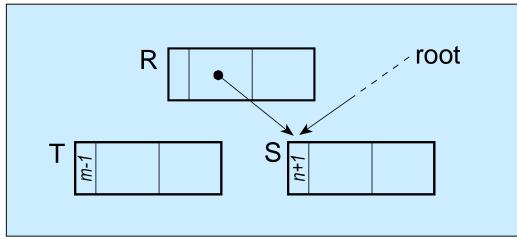
- **Copying:**
  - Move objects you need to a new garage.
  - Then recursively move anything needed by an object in the new garage.
  - Afterwards, burn down the old garage (any objects in it are dead)!

Reference counting

Reference counting

- **Reference counting: a mechanism to share ownership**

- **Goal: to identify when you are the only owner, and thus you can make the disposal decision.**

- **Basic idea: count the number of references from live objects.**
  - Each object has a reference count (RC)
  - When a reference is copied, the referent's RC is incremented
  - When a reference is deleted, the referent's RC is decremented
  - An object can be reclaimed when its RC = 0

Reference counting

Update(left(R), S)

Reference counting

- **Recursive freeing**
  - Once an object's RC=0, it can be freed.
  - But object may contain references to further objects.
  - Before this object is freed, the RCs of its constituents should also be freed.

```
New() {
  if free_list == nil
    abort "Memory exhausted"
  newcell = allocate()
  RC(newcell) = 1
  return newcell
}

Update(R, S) {
  RC(S) = RC(S) + 1
  delete(*R)
  *R = S
}
```

```
delete(T) {
  RC(T) = RC(T) - 1
  if RC(T) == 0 {
    for U in Children(T)
      delete(*U)
    free(T)
  }
}

free(N) {
  next(N) = free_list;
  free_list = N;
}
```

Advantages of reference counting

- **Simple to implement**

- **Costs distributed throughout program**

- **Good locality of reference: only touch old and new targets' RCs**

- **Works well because few objects are shared and many are short-lived**

- **Zombie time minimized**

  - the zombie time is the time from when an object becomes garbage until it is collected

- **Immediate finalisation is possible (due to near zero zombie time)**

Disadvantages of reference counting

- **Not comprehensive (does not collect all garbage): cannot reclaim cyclic data structures**

- **High cost of manipulating RCs: cost is ever-present even if no garbage is collected**

- **Bad for concurrency; RC manipulations must be atomic — need Compare&Swap operation**

- **Tightly coupled interface to mutator**

- **High space overheads**

- **Recursive freeing cascade is only bounded by heap size**

Mark-Sweep

Mark-Sweep

- **Mark-sweep is such a tracing algorithm — it works by following (tracing) references from live objects to find other live objects.**
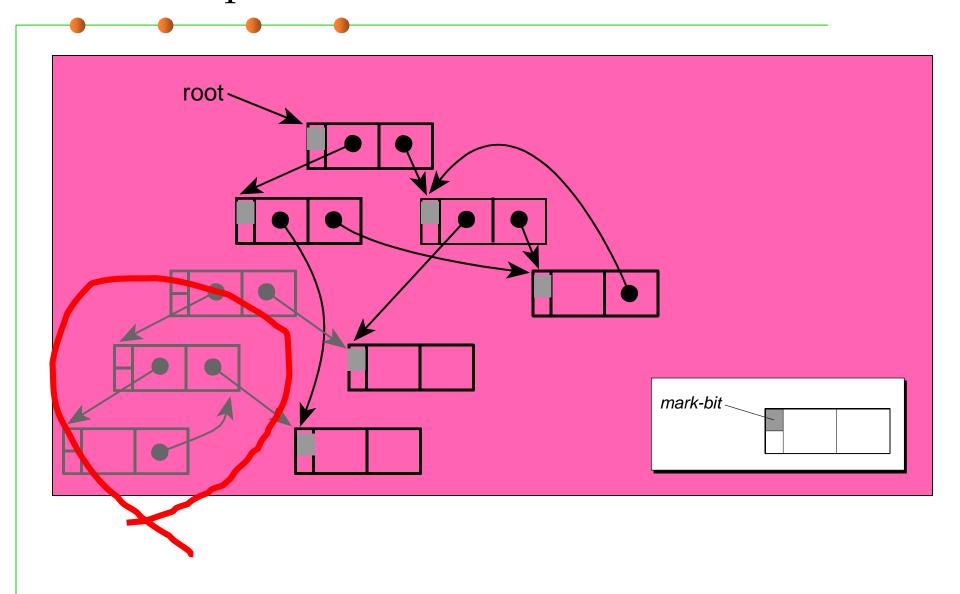
- **Implementation of the live set:**
  - Each object has a mark-bit associated with it, indicating whether it is a member of the live set.

- **There are two phases:**
  - Mark phase:
    - starting from the roots, the graph is traced and the mark-bit is set in each unmarked object encountered.
    - At the end of the mark phase, unmarked objects are garbage.
  - Sweep phase: starting from the bottom, the heap is swept
    - mark-bit not set: the object is reclaimed
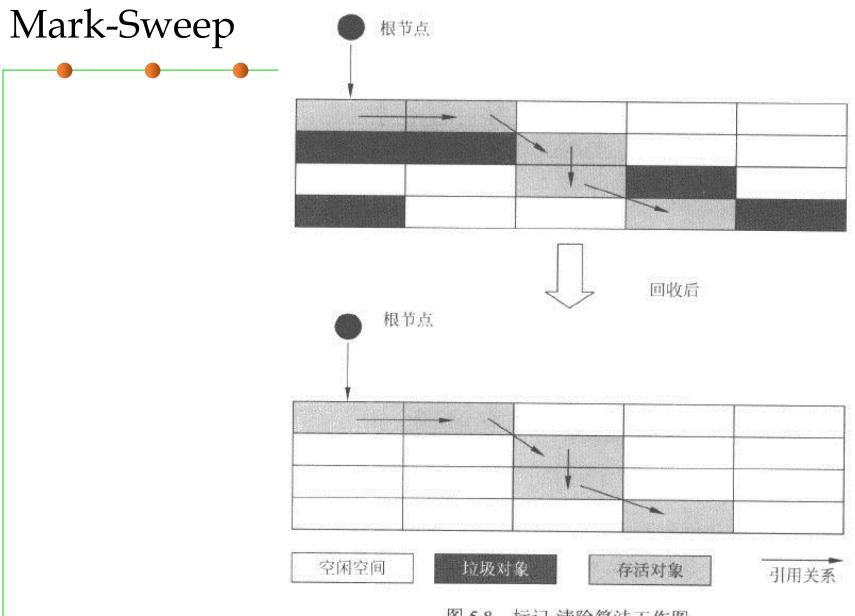   - mark-bit set: the mark-bit is cleared

Mark-Sweep

root

mark-bit

Mark-Sweep

根节点

回收后

根节点

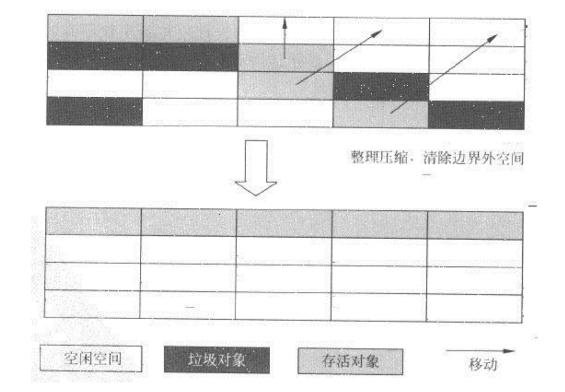空闲空间　　　垃圾对象　　　存活对象　　　引用关系

图 5.8　标记-清除算法工作图

Mark-Compact

Mark-Compact

- **Mark-Compact:**
  - Put notes on objects you need (as Mark-Sweep).
  - Move anything with a note on it to the back of the garage.
  - Burn everything at the front of the garage (it's all dead).

整理压缩，清除边界外空间

空闲空间　　　垃圾对象　　　存活对象　　　移动

The mark-stack

- **The simplest solution is to implement marking recursively: walk a minimum spanning tree of the object graph**

```
mark(N) {
    if markBit(N) == UNMARKED {
        markBit(N) = MARKED
        for M in Children(N)
            mark(*M)
    }
}
```

- **A more efficient method is to use a marking stack: repeat until the mark stack is empty.**

  – Pop the top item

  – If it is unmarked, mark it.

  – If it is a branch point in the graph, push any unmarked children onto the stack

The mark-stack

```
New() {
  if free_pool.empty
    markHeap()
    sweep()
  newobj = allocate()
  return newobj
}

markHeap() {
  markStack = empty
  for R in Roots
    markBit(R) = MARKED
    markStack.push(R)
    mark()
}
```

```
mark() {
  while markStack not empty {
    N = pop(markStack)
    for M in Children(N)
      if markBit(M)== UNMARKED {
        markBit(M) = MARKED
        if not atom(*M)
          push(markStack,*M)
      }
  }
}

sweep() {
  N = Heap_bottom
  while N < Heap_top
    if markBit(N) == UNMARKED
      free (N)
    else markBit(N) = UNMARKED
  N += N.size
}
```

Advantages of mark-sweep

- **Comprehensive: cyclic garbage collected naturally**

- **No run-time overhead on pointer manipulations**

- **Loosely coupled to mutator**

- **Does not move objects**
  - does not break any mutator invariants
  - optimiser-friendly
  - requires only one reference to each live object to be discovered (rather than having to find every reference)

Disadvantages of mark-sweep

- **Stop/start nature leads to disruptive pauses and long zombie times.**

- **Complexity is O(heap) rather than O(live)**
  - every live object is visited in mark phase
  - every object, alive or dead, is visited in sweep phase

- **Degrades with residency (heap occupancy)**
  - the collector needs headroom in the heap to avoid thrashing
  - Example: lots of marking to do if heap is full and we do this often

- **Fragmentation and mark-stack overflow are issues**

- **Tracing collectors must be able to find roots (unlike reference counting)**
  - This needs some understanding of the run-time system or cooperation from the compiler.

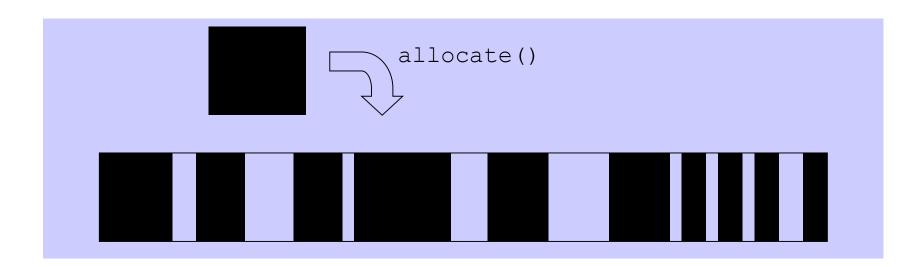# Fragmentation and Copying

Fragmentation

- **Fragmentation: inability to use available memory**

  - External: allocated memory scattered into blocks; free blocks cannot be coalesced

  - Internal: memory manager allocated more space than actually required — common causes are headers, rounding sizes up

- **Fragmentation is a problem for explicit memory managers as well; free() is often not free.**

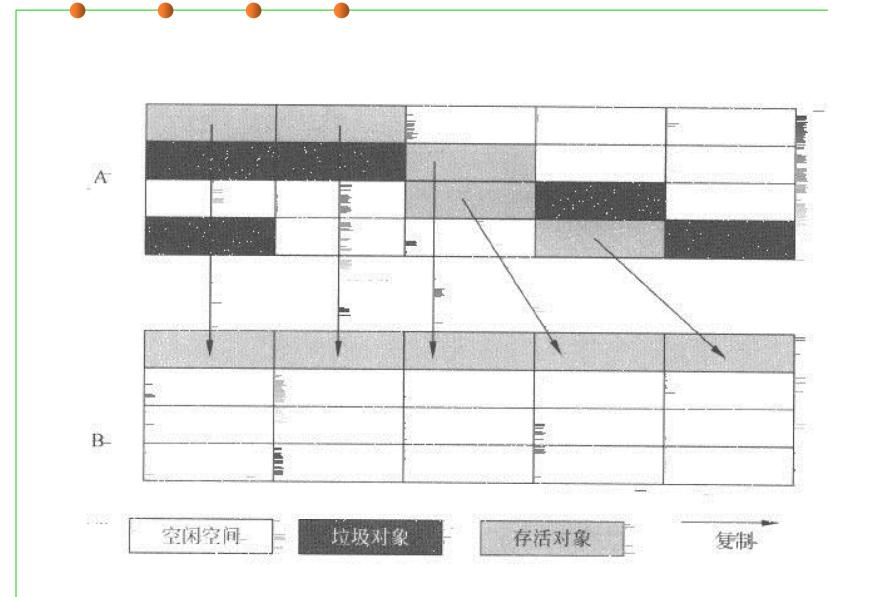# Copying

- **GC is a set-partitioning problem**

  - A mark-bit is one way of defining two sets.

- **Mark-compact physically moves members of the live set to a different part of the heap**

  - the free pointer marks the dividing line between live data and memory that can be overwritten

- **Copying collection is a simpler solution: it picks out live objects and copies them to a 'fresh' heap**
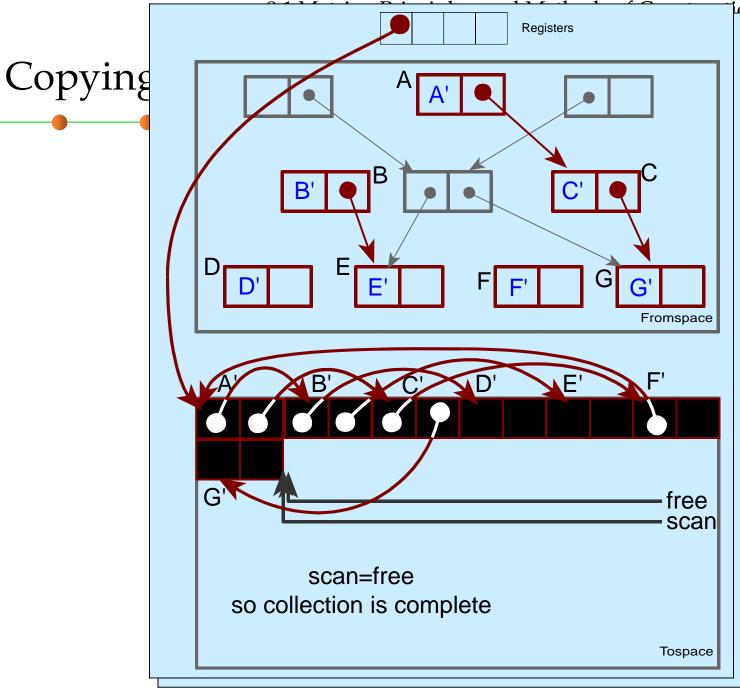
Copying garbage collection

- **Divide heap into 2 halves called semi-spaces and named Fromspace and Tospace**

- **Allocate objects in Tospace**

- **When Tospace is full**
  - Flip the roles of the semi-spaces
  - Pick out all live data in Fromspace and copy them to Tospace
  - Preserve sharing by leaving a forwarding address in the Fromspace replica
  - Use Tospace objects as a work queue

Copying garbage collection

| 空闲空间 | 垃圾对象 | 存活对象 | 复制 |

Copying

Copying garbage collection

```
flip(){
    Fromspace, Tospace = Tospace, Fromspace
    top_of_space = Tospace + space_size
    scan = free = Tospace
    for R in Roots {R = copy(R)}
    while scan < free {
        for P in Children(scan) {*P = copy(*P)}
        scan = scan + size (scan)
    }
}


copy(P) {
    if forwarded(P){return forwarding_address(P)}
    else {
        addr = free
        move(P,free)
        free = free + size(P)
        forwarding_address(P) = addr
        return addr
    }
}
```

Advantages of copying GC

- **Compaction for free**

- **Allocation is very cheap for all object sizes**
  - out-of-space check is pointer comparison
  - simply increment free pointer to allocate

- **Only live data is processed (commonly a small fraction of the heap)**

- **Fixed space overheads**
  - free and scan pointers
  - forwarding addresses can be written over user data

- **Comprehensive: cyclic garbage collected naturally**

- **Simple to implement a reasonably efficient copying GC**

Disadvantages of copying GC

- **Stop-and-copy may be disruptive**

- **Degrades with residency**

- **Requires twice the address space of other simple collectors**
  - touch twice as many pages
  - trade-off against fragmentation

- **Cost of copying large objects**

- **Long-lived data may be repeatedly copied**

- **All references must be updated**

- **Moving objects may break mutator invariants**

- **Breadth-first copying may disturb locality patterns**

Complexity: caveat emptor

- **Claim: "Copying is always better than mark-sweep GC"**

- **The collectors we've seen so far are very simple minded
  Let us compare their basic performance…**

- **Copying is more expensive than setting a bit**

- **Efficient implementations of mark-sweep are dominated by cost of mark phase**
  - linear scanning less expensive than tracing, and
  - cost of sweep can be reduced further

Simple asymptotic complexity analyses are misleading
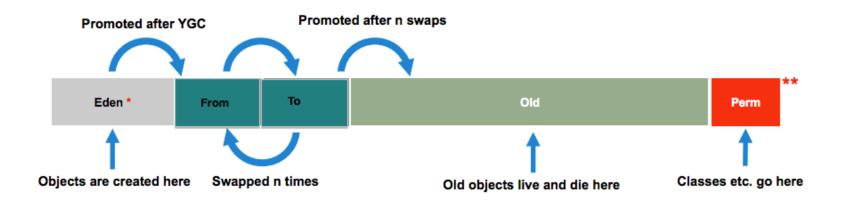
6 Garbage Collection in JVM

Java Garbage Collector

- **The Java garbage collector can logically separate the heap into different areas, so that the GC can faster identify objects which can get removed.**

- **The JVM automatically re-collects the memory which is not used any more.**

  - The memory for objects which are not referred any more will be automatically released by the garbage collector.

  - To see that the garbage collector starts working add the command line argument "`-verbose:gc`" to your virtual machine.
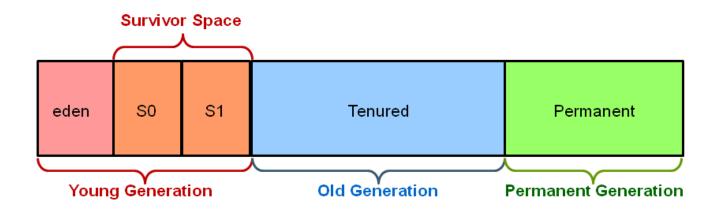
Garbage Collection in JVM

- **The HotSpot VM (Sun JVM) has three major spaces: young generation, old generation, and permanent generation.**

  – When a Java application allocates Java objects, those objects are allocated in the young generation space.

  – Objects that survive, that is, those that remain live, after some number of minor garbage collections are promoted into the old generation space.

  – The permanent generation space holds VM and Java class metadata as well as interned Strings and class static variables

Promoted after YGC   Promoted after n swaps

| Eden * | From | To | Old | Perm ** |

Objects are created here   Swapped n times   Old objects live and die here   Classes etc. go here

Generational Collection GC algorithm

- **Generational Collection GC algorithm**

    – The various GC algorithms discussed above have their own advantages and disadvantages. It's wise to use appropriate algorithm according to the characteristics of GC objects.

- **Generational Collection divides memory space into several pieces (young generation, old (tenured) generation, and permanent generation) and use different GC algorithms according to the characteristics of each memory pieces, in order to improve the efficiency of garbage collection.**
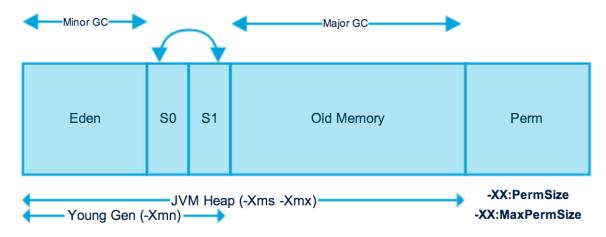
The Yong and Old Generations

- **In the young generation**

  - Each time garbage collection will find a large number of objects die, and only a small amount survive.

  - So the copying algorithm is appropriate which only needs to pay the cost of copying a small amount of living objects.

- **In the old generation**

  - Because objects have a high survival rate and no additional memory space to be allocated, the Mark-Sweep or Mark-Compact algorithm must be used for collection.

When Garbage Collection occurs

- **It is important to understand that a garbage collection occurs when any one of the three spaces, young generation, old generation, or permanent generation, is in a state where it can no longer satisfy an allocation event.**

- **In other words, a garbage collection occurs when any one of those three spaces is considered full and there is some request for additional space that is not available.**

- **When the young generation space does not have enough room available to satisfy a Java object allocation, the HotSpot VM performs a minor garbage collection to free up space. Minor garbage collections tend to be short in duration relative to full garbage collections.**

Minor and Full Garbage Collection

- **Objects that remain live for some number of minor garbage collections eventually get promoted (copied) to the old generation space.**

- **When the old generation space no longer has available space for promoted objects, the HotSpot VM performs a full garbage collection.**

- **It actually performs a full garbage collection when it determines there is not enough available space for object promotions from the next minor garbage collection. This is a less costly approach rather than being in the middle of a minor garbage collection and discovering that the promotion of an object will fail.**

- **Recovering from an object promotion failure is an expensive operation. A full garbage collection also occurs when the permanent generation space does not have enough available space to store additional VM or class metadata.**

7 Garbage Collection Tuning in JVM

Garbage Collection Tuning in JVM

- **A best practice is to tune the time spent doing garbage collection to within 5% of execution time.**

- **The JVM runs with fixed available memory. Once this memory is exceeded you will receive "java.lang.OutOfMemoryError".**
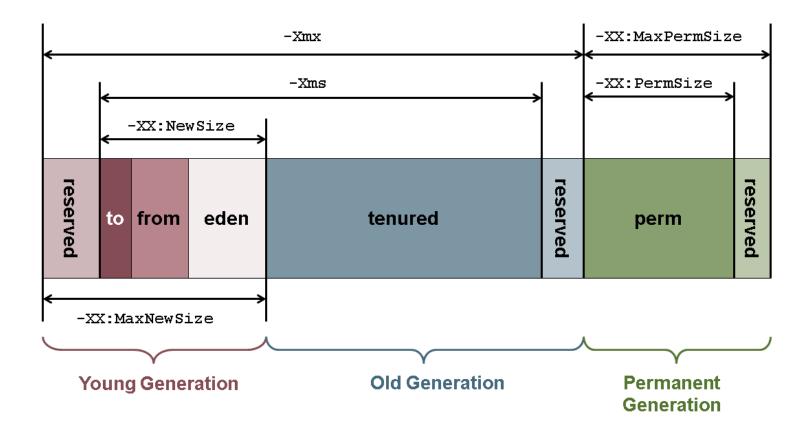
    ```
    "Exception in thread java.lang.OutOfMemoryError:
                     Java heap space".
    ```

- **The JVM tries to make an intelligent choice about the available memory at startup but you can overwrite the default with the following settings.**

Tuning JVM's garbage collection

- **Specifying VM heap size**

- **Choosing a garbage collection scheme**

- **Using verbose garbage collection to determine heap size**

- **Automatically logging low memory conditions**

- **Manually requesting garbage collection**

- **Requesting thread stacks**

(1) Tuning VM Heap Size

(1) Tuning VM Heap Size

- **The Java heap is where the objects of a Java program live. It is a repository for live objects, dead objects, and free memory.**

  - When an object can no longer be reached from any pointer in the running program, it is considered "garbage" and ready for collection.

- **The JVM heap size determines how often and how long the VM spends collecting garbage.**

  - An acceptable rate for garbage collection is application-specific and should be adjusted after analyzing the actual time and frequency of garbage collections.

  - If you set a large heap size, full garbage collection is slower, but it occurs less frequently.

  - If you set your heap size in accordance with your memory needs, full garbage collection is faster, but occurs more frequently.

(1) Tuning VM Heap Size

- **The `-Xmx` and `-Xms` command line options specify the initial and maximum total size of the young generation and old generation spaces. This initial and maximum size is also referred to as the Java heap size.**
  - `Java –Xms 1024M`
  - `Java –Xmx 2048M`
- **When `-Xms` is smaller than `-Xmx`, the amount of space consumed by young and old generation spaces is allowed to grow or contract depending on the needs of the application.**
  - The growth of the Java heap will never be larger than `–Xmx`, and the Java heap will never contract smaller than `–Xms`.
  - Growing or contracting the size of either the young generation space or old generation space requires a full garbage collection.
  - Full garbage collections can reduce throughput and induce larger than desired latencies.

(1) Tuning VM Heap Size

- **The young generation space is specified using any one of the following command line options:**
  - `-XX: NewSize=<n>[g|m|k]` The initial and minimum size of the young generation space. <n> is the size. [g|m|k] indicates whether the size should be interpreted as gigabytes, megabytes, or kilobytes.
  - `-XX: MaxNewSize=<n>[g|m|k]` The maximum size of the young generation space.
  - `-Xmn<n>[g|m|k]` Sets the initial, minimum, and maximum size of the young generation space.

(1) Tuning VM Heap Size

- **The size of the old generation space is implicitly set based on the size of the young generation space.**
  - The initial old generation space size is the value of `-Xms minus -XX:NewSize`.
  - The maximum old generation space size is the value of `-Xmx minus -XX:MaxNewSize`.
  - If `-Xms` and `-Xmx` are set to the same value and `-Xmn` is used, or `-XX:NewSize` is the same value `as -XX:MaxNewSize`, then the old generation size is `-Xmx (or -Xms) minus -Xmn`.

If the garbage collector has become a bottleneck, you may wish to customize the generation sizes.
Check the verbose garbage collector output, and then explore the sensitivity of your individual performance metric to the garbage collector parameters.

(2) Choosing a Garbage Collection Scheme

- Depending on which JVM you are using, you can choose from several garbage collection schemes to manage your system memory.

- Some garbage collection schemes are more appropriate for a given type of application.

- Once you have an understanding of the workload of the application and the different garbage collection algorithms utilized by the JVM, you can optimize the configuration of the garbage collection.

(2) Choosing a Garbage Collection Scheme

- **The serial collector**: uses a single thread to perform all garbage collection work   -XX:+UseSerialGC

- **The throughput (parallel) collector**: performs minor collections (on the young generation) in parallel, which can significantly reduce garbage collection overhead. (but major collections are performed using a single thread)   -XX:+UseParallelGC

- **The concurrent low pause collector**: collects tenured generation and does most of the collection concurrently with the execution of the application. It is paused for short periods during the collection.

                                                    -XX:+UseConcMarkSweepGC

- **The incremental low pause collector**: collects a portion of the tenured generation at each minor collection and tries to minimize large pause of major collections   -XX:+UseTrainGC

(3) Using verbose garbage collection

- **The verbose garbage collection option (verbosegc) enables you to measure exactly how much time and resources are put into garbage collection.**

```
-verbose:gc
```

- **To determine the most effective heap size, turn on verbose garbage collection and redirect the output to a log file for diagnostic purposes.**

- **From log file:**
  - How often is garbage collection taking place?
  - How long is garbage collection taking? Full garbage collection should not take longer than 3 to 5 seconds.
  - What is your average memory footprint? In other words, what does the heap settle back down to after each full garbage collection? If the heap always settles to 85 percent free, you might set the heap size smaller.

(4) Manually request garbage collection

- **You can manually request that the JVM perform garbage collection.**

- **When you perform garbage collection, the JVM often examines every living object in the heap.**

- **Garbage Collect calls the JVM's `System.gc()` method to perform garbage collection. The JVM implementation then decides whether or not the request actually triggers garbage collection.**

8 Using I/O Efficiently in Java
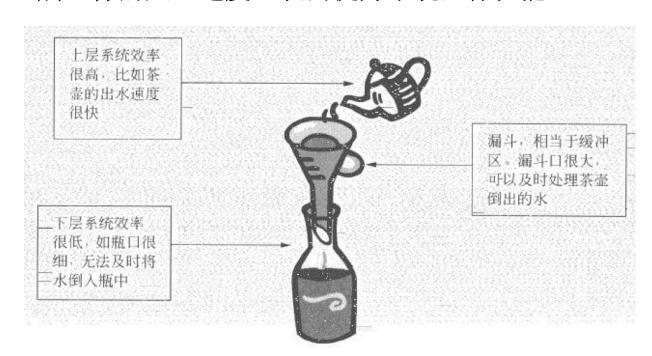
Speeding up I/O

- **In software systems, the reading-writing speed of I/O is slower than that of memory.**

- **Therefore, I/O reading and writing on many occasions will become bottleneck of a system.**

- **Speeding up I/O has a great advantage to enhance the overall performance of the system.**

- **A main technique to speed up I/O is buffer.**

Buffer

- **Data buffer (or just buffer) is a region of a physical memory storage used to temporarily store data while it is being moved from one place to another.**

- **Typically, the data is stored in a buffer as it is retrieved from an input device (such as a microphone) or just before it is sent to an output device (such as speakers).**

- **Buffers can be implemented in a fixed memory location in hardware—or by using a virtual data buffer in software, pointing at a location in the physical memory. In all cases, the data stored in a data buffer are stored on a physical storage medium.**

- **A majority of buffers are implemented in software, which typically use the faster RAM to store temporary data, due to the much faster access time compared with hard disk drives.**

Buffer

- **Buffers are typically used when there is a difference between the rate at which data is received and the rate at which it can be processed, or in the case that these rates are variable, for example in a printer spooler or in online video streaming.**

- 上层应用组件不需要等待下层组件真实地接受全部数据，即可返回操作，加快了上层组件的处理速度，从而提升系统整体性能。

Using Buffer to speed up I/O

- **E.g., FileWriter in Java**

```
Writer writer = new FileWriter(new File("file.txt"));

long begin = System.currentTimeMillis();

for (int i=0; i< CIRCLE; i++){

    writer.write(i);

}

Writer.close();

System.out.println("testFileWriter spend:" +

                    (System.currentTimeNillis() - begin));
```

Set CIRCLE to 100,000
Running time of the original program is 63ms;
Running time of the program with buffer is 32ms;

```
Writer writer = new BufferedWriter(new FileWriter(new File("file.txt")));
```

java.nio: High Performance I/O for Java

- **Everyday use – Buffered{Reader, Writer}**

- **Casual use - Scanner**
  - Easy but not general and swallows exceptions

- **Stream integration – Files.lines**
  - No parallelism support yet

- **Async – java.nio.AsynchronousFileChannel**

- **Many niche APIs, e.g. mem mapping, line numbering**
  - Search them out as needed

- **Consider Okio if third party API allowed**

NIO: New I/O

- **Prior to the J2SE 1.4 release of Java, I/O had become a bottleneck.**

- **The old java.io stream classes had too many software layers to be fast:**

  – The specification implied much copying of small chunks of data;

  – There was no way to multiplex data from multiple sources without incurring thread context switches;

  – There was no way to exploit modern OS tricks for high performance I/O, like memory mapped files.

NIO: New I/O
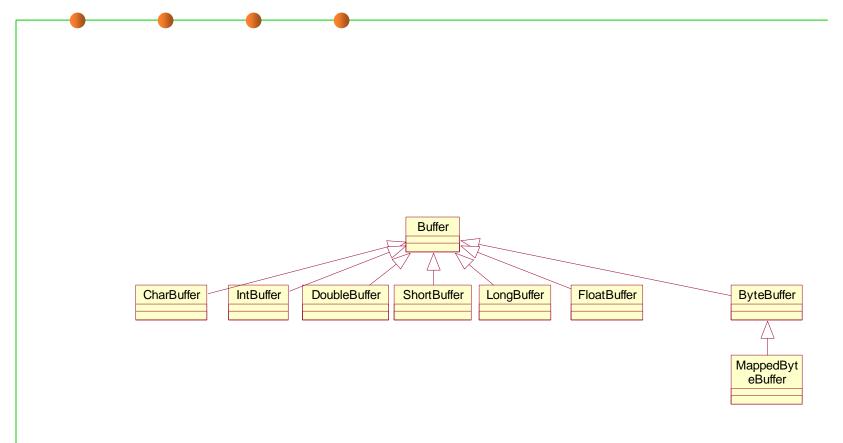
- **New I/O changes that by providing:**
  - A hierarchy of dedicated buffer classes that allow data to be moved from the JVM to the OS with minimal memory-to-memory copying, and without expensive overheads like switching byte order; effectively buffer classes give Java a "window" on system memory.

  - A unified family of channel classes that allow data to be fed directly from buffers to files and sockets, without going through the intermediaries of the old stream classes.

  - A family of classes to directly implement selection (AKA readiness testing, AKA multiplexing) over a set of channels.

  - NIO also provides file locking for the first time in Java.

Buffers in NIO

- **A Buffer object is a container for a fixed amount of data.**

- **It behaves something like a byte [] array, but is encapsulated in such a way that the internal storage can be a block of system memory.**

  – Thus adding data to, or extracting it from, a buffer can be a very direct way of getting information between a Java program and the underlying operating system.

  – All modern OS's provide virtual memory systems that allow memory space to be mapped to files, so this also enables a very direct and high-performance route to the file system.

  – The data in a buffer can also be efficiently read from, or written to, a socket or pipe, enabling high performance communication.

- **The buffer APIs allow you to read or write from a specific location in the buffer directly; they also allow relative reads and writes, similar to sequential file access.**

The java.nio.Buffer Hierarchy
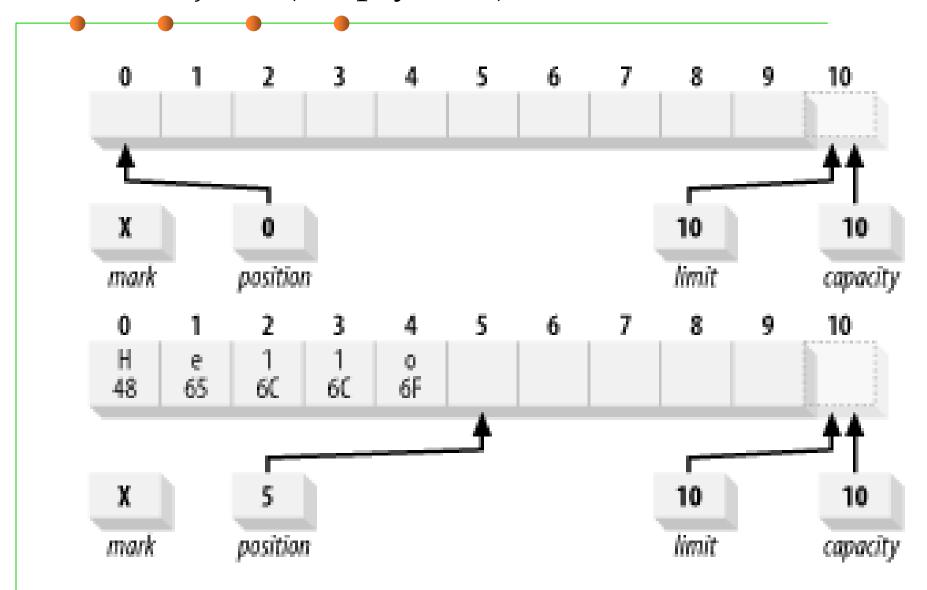
The ByteBuffer Class

- **The most important buffer class in practice is probably the ByteBuffer class. This represents a fixed-size vector of primitive bytes.**

- **Important methods on this class include:**
  - byte get()
  - byte get(int index)
  - ByteBuffer get(byte [] dst)
  - ByteBuffer get(byte [] dst, int offset, int length)
  - ByteBuffer put(byte b)
  - ByteBuffer put(int index, byte b)
  - ByteBuffer put(byte [] src)
  - ByteBuffer put(byte [] src, int offset, int length)
  - ByteBuffer put(ByteBuffer src)

File Position and Limit
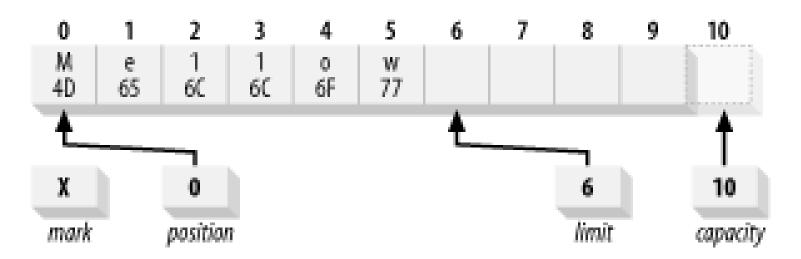
- **Apart from forms with an index parameter, these are all relative operations: they get data from, or insert data into, the buffer starting at the current position in the buffer; they also update the position to point to the position after the read or written data. The position property is like the file pointer in sequential file access.**

- **The superclass Buffer has methods for explicitly manipulating the position and related properties of buffers, e.g:**
  - int position()
  - Buffer position(int newPosition)
  - int limit()
  - Buffer limit(int newLimit)
  - The ByteBuffer or Buffer references returned by these various methods are simply references to this buffer object, not new buffers. They are provided to support cryptic invocation chaining. Feel free to ignore them.

Buffer Objects (Empty/Fill)

File Position and Limit

- **The limit property defines either the last space available for writing, or how much data has been written to the file.**

  - After finishing writing a flip() method can be called to set limit to the current value of position, and reset position to zero, ready for reading.

- **Various operations implicitly work on the data between position and limit.**

Creating Buffers

- **Four factory methods can be used to create a new ByteBuffer:**
  - ByteBuffer allocate(int capacity)
  - ByteBuffer allocateDirect(int capacity)
  - ByteBuffer wrap(byte [] array)
  - ByteBuffer wrap(byte [] array, int offset, length)

- **These are all static methods of the ByteBuffer class.**
  - allocate() creates a ByteBuffer with an ordinary Java backing array of size capacity.
  - allocateDirect()—perhaps the most interesting case—creates a direct ByteBuffer, backed by capacity bytes of system memory.
  - The wrap() methods create ByteBuffer's backed by all or part of an array allocated by the user.

- **The other typed buffer classes (CharBuffer, etc) have similar factory methods, except allocateDirect() method.**

Other Primitive Types in ByteBuffer

- **It is possible to write other primitive types (char, int, double, etc) to a ByteBuffer by methods like:**

  – ByteBuffer putChar(char value)

  – ByteBuffer putChar(int index, char value)

  – ByteBuffer putInt(int value)

  – ByteBuffer putInt(int index, int value)

  – …

- **The putChar() methods do absolute or relative writes of the two bytes in a Java char, the putInt() methods write 4 bytes, and so on.**

  – Of course there are corresponding getChar(), getInt(), … methods.

Other Primitive Types in ByteBuffer

- **These give you fun, unsafe ways of coercing bytes of one primitive type to another type, by writing data as one type and reading them as another.**

- **But actually this isn't the interesting bit—this was always possible with the old java.io DataStream's.**

- **The interesting bit is that the new ByteBuffer class has a method that allows you to set the byte order…**

View Buffers

- **ByteBuffer has no methods for bulk transfer of arrays other than type byte[].**

- **Instead, create a view of (a portion of) a ByteBuffer as any other kind of typed buffer, then use the bulk transfer methods on that view. Following methods of ByteBuffer create views:**

  - CharBuffer asCharBuffer()

  - IntBuffer asIntBuffer()

  - …

  - To create a view of just a portion of a ByteBuffer, set position and limit appropriately beforehand—the created view only covers the region between these.

  - You cannot create views of typed buffers other than ByteBuffer.

  - You can create another buffer that represents a subsection of any buffer (without changing element type) by using the slice() method.
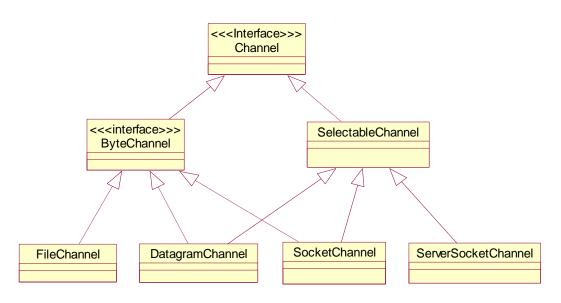
View Buffers

- **For example, writing an array of floats to a byte buffer, starting at the current position:**
  - float [] array ;
  - …
  - FloatBuffer floatBuf = byteBuf.asFloatBuffer() ;
  - floatBuf.put(array) ;

Channels

- **A channel is a new abstraction in java.nio.**

  – In the package java.nio.channels.

- **Channels are a high-level version of the file-descriptors familiar from POSIX-compliant operating systems.**

  – So a channel is a handle for performing I/O operations and various control operations on an open file or socket.

- **For those familiar with conventional Java I/O, java.nio associates a channel with any RandomAccessFile, FileInputStream, FileOutputStream, Socket, ServerSocket or DatagramSocket object.**

  – The channel becomes a peer to the conventional Java handle objects; the conventional objects still exist, and in general retain their role—the channel just provides extra NIO-specific functionality.

- **NIO buffer objects can written to or read from channels directly**

Simplified Channel Hierarchy

<<<Interface>>>
Channel

<<<interface>>>
ByteChannel

SelectableChannel

FileChannel

DatagramChannel

SocketChannel

ServerSocketChannel

Some of the "inheritance" arcs here are indirect: we missed out some interesting intervening classes and interfaces.

Opening Channels

- **Socket channel classes have static factory methods called open(), e.g.:**
  - SocketChannel sc = SocketChannel.open() ;
  - Sc.connect(new InetSocketAddress(hostname, portnumber)) ;
- **File channels cannot be created directly; first use conventional Java I/O mechanisms to create a FileInputStream, FileOutputStream, or RandomAccessFile, then apply the new getChannel() method to get an associated NIO channel, e.g.:**
  - RandomAccessFile raf = new RandomAccessFile(filename, "r") ;
  - FileChannel fc = raf.getChannel() ;

Using Channels

- **Any channel that implements the ByteChannel interface — i.e. all channels except ServerSocketChannel — provide a read() and a write() instance method:**
  - int read(ByteBuffer dst)
  - int write(ByteBuffer src)
  - These may look reminiscent of the read() and write() system calls in UNIX:
    - int read(int fd, void* buf, int count)
    - int write(int fd, void* buf, int count)
  - The Java read() attempts to read from the channel as many bytes as there are remaining to be written in the dst buffer. Returns number of bytes actually read, or -1 if end-of-stream. Also updates dst buffer position.
  - Similarly write() attempts to write to the channel as many bytes as there are remaining in the src buffer. Returns number of bytes actually read, and updates src buffer position.

Example: Copying one Channel to Another

- **This example assumes a source channel src and a destination channel dest:**

  - ByteBuffer buffer = ByteBuffer.allocateDirect(BUF_SIZE) ;

  - while(src.read(buffer) != -1) {

  -     buffer.flip() ;   // Prepare read buffer for "draining"

  -     while(buffer.hasRemaining())

  -        dest.write(buffer) ;

  -     buffer.clear() ;   // Empty buffer, ready to read next chunk.

  - }

Example: Copying one Channel to Another

- – Note a write() call (or a read() call) may or may not succeed in transferring whole buffer in a single call. Hence need for inner while loop.

- – Example introduces two new methods on Buffer: hasRemaining() returns true if position < limit; clear() sets position to 0 and limit to buffer's capacity.

- – Because copying is a common operation on files, FileChannel provides a couple of special methods to do just this:
  - long transferTo(long position, long count, WriteableByteChannel target)
  - long transferFrom(ReadableByteChannel src, long position, long count)

Memory-Mapped Files

- **In modern operating systems one can exploit the virtual memory system to map a physical file into a region of program memory.**

  - Once the file is mapped, accesses to the file can be extremely fast: one doesn't have to go through read() and write() system calls.

  - One application might be a Web Server, where you want to read a whole file quickly and send it to a socket.

  - Problems arise if the file structure is changed while it is mapped—use this technique only for fixed-size files.

- **This low-level optimization is now available in Java. FileChannel has a method:**

  - MappedByteBuffer map(MapMode mode, long position, long size)

  - mode should be one of MapMode.READ_ONLY, MapMode.READ_WRITE, MapMode.PRIVATE.

  - The returned MappedByteBuffer can be used wherever an ordinary ByteBuffer can.

Scatter/Gather

- **Often called vectored I/O, this just means you can pass an array of buffers to a read or write operation; the overloaded channel instance methods have signatures:**

  - long read(ByteBuffer [] dsts)

  - long read(ByteBuffer [] dsts, int offset, int length)

  - long write(ByteBuffer [] srcs)

  - long write(ByteBuffer [] srcs, int offset, int length)

- **The first form of read() attempts to read enough data to fill all buffers in the array, and divides it between them, in order.**

- **The first form of write() attempts to concatenate the remaining data in all buffers and write it.**

  - The arguments offset and length select a subset of buffers from the arrays (not, say, an interval within buffers).

SocketChannels

- **As mentioned at the beginning of this section, socket channels are created directly with their own factory methods**

  - If you want to manage a socked connection as a NIO channel this is the only option. Creating NIO socket channel implicitly creates a peer java.net socket object, but (contrary to the situation with file handles) the converse is not true.

- **As with file channels, socket channels can be more complicated to work with than the traditional java.net socket classes, but provide much of the hard-boiled flexibility you get programming sockets in C.**

- **The most notable new facilities are that now socket communications can be non-blocking, they can be interrupted, and there is a selection mechanism that allows a single thread to do multiplex servicing of any number of channels.**

Basic Socket Channel Operations

- **Typical use of a server socket channel follows a pattern like:**

```
ServerSocketChannel ssc = ServerSocketChannel.open() ;
ssc.socket().bind( new InetSocketAddress(port) ) ;
while(true) {
        SocketChannel sc = ssc.accept() ;
        … process a transaction with client through sc …}
```

- **The client does something like:**

```
SocketChannel sc = SocketChannel.open() ;
sc.connect( new InetSocketAddr(serverName, port) ) ;
… initiate a transaction with server through sc …
```

- **The elided code above will typically be using read() and write() calls on the SocketChannel to exchange data between client and server.**

  – So there are four important operations: accept(), connect(), write(), read() .

Other Features of Channels

- **File channels provide a quite general file locking facility. This is presumably important to many applications (database applications), but less obviously so to HPC operations, so we don't discuss it here.**

- **There is a DatagramChannel for sending UDP–style messages. This may well be important for high performance communications, but we don't have time to discuss it.**

- **There is a special channel implementation representing a kind of pipe, which can be used for inter-thread communication.**

An example

一个使用 NIO 进行文件复制的例子如下，它展示了通过 NIO 进行文件读取和文件写入操作：

```
public static void nioCopyFile(String resource, String destination)
        throws IOException {
    FileInputStream fis = new FileInputStream(resource);
    FileOutputStream fos = new FileOutputStream(destination);
    FileChannel readChannel = fis.getChannel();            //读文件通道
    FileChannel writeChannel = fos.getChannel();           //写文件通道
    ByteBuffer buffer = ByteBuffer.allocate(1024);         //读入数据缓存
    while (true) {
        buffer.clear();
        int len = readChannel.read(buffer);                //读入数据
        if (len == -1) {
            break;                                         //读取完毕
        }
        buffer.flip();
        writeChannel.write(buffer);                        //写入文件
    }
    readChannel.close();
    writeChannel.close();
}
```

A brief, sad history of I/O in Java

Release, Year	Changes
JDK 1.0, 1996	`java.io.InputStream/OutputStream` – byte-based
JDK 1.1, 1997	`java.io.Reader/Writer` – char-based wrappers
J2SE 1.4, 2002	`java.nio.Channel/Buffer` – "Flexible" + select/poll, mmap
J2SE 5.0, 2004	`java.util.Scanner, String.printf/format` – Formatted
Java 7, 2011	`java.nio.file Path/Files` – file systems `java.nio.AsynchronousFileChannel` - *Real* async I/O
Java 8, 2014	`Files.lines` – lambda/stream integration
3d party, 2014	`com.squareup.okio.Buffer` – "Modern"

1: Stream

```java
/**
 * Reads all lines from a text file and prints them.
 * Uses Java 1.0-era (circa 1996) Streams to read the file.
 */
public class StreamCat {
    public static void main(String[] args) throws IOException {
        DataInputStream dis = new DataInputStream(
            new FileInputStream(args[0]));
         // Don't do this! DataInputStream.readLine is DEPRECATED!
        String line;
        while ((line = dis.readLine()) != null)
            System.out.println(line);
    }
}
```

2: Reader

```java
/**
 * Reads all lines from a text file and prints them.
 * Uses Java 1.1-era (circa 1997) Streams to read the file.
 */
public class ReaderCat {
    public static void main(String[] args) throws IOException {
        try (BufferedReader rd = new BufferedReader(
                new FileReader(args[0]))) {
            String line;
            while ((line = rd.readLine()) != null) {
                System.out.println(line);
                // you could also wrap System.out in a PrintWriter
            }
        }
    }
}
```

3: Nio

```java
/**
 * Reads all lines from a text file and prints them.
 * Uses nio FileChannel and ByteBuffer.
 */
public class NioCat {
    public static void main(String[] args) throws IOException {
        ByteBuffer buf = ByteBuffer.allocate(512);
        try (FileChannel ch = FileChannel.open(Paths.get(args[0]),
                StandardOpenOption.READ)) {
            int n;
            while ((n = ch.read(buf)) > -1) {
                System.out.print(new String(buf.array(), 0, n));
                buf.clear();
            }
        }
    }
}
```

4: Scanner

```java
/**
 * Reads all lines from a text file and prints them
 * Uses Java 5 scanner.
 */
public class ScannerCat {
    public static void main(String[] args) throws IOException {
        try (Scanner s = new Scanner(new File(args[0]))) {
            while (s.hasNextLine())
                System.out.println(s.nextLine());
        }
    }
}
```

5: Lines

```java
/**
 * Reads all lines from a text file and prints them. Uses Files,
 * Java 8-era Stream API (not IO Streams!) and method references.
 */
public class LinesCat {
    public static void main(String[] args) throws IOException {
        Files.lines(Paths.get(args[0])).forEach(System.out::println);
    }
}
```

Java I/O Recommendations

- **Everyday use – Buffered{Reader, Writer}**

- **Casual use - Scanner**
  - Easy but not general and swallows exceptions

- **Stream integration – Files.lines**
  - No parallelism support yet

- **Async – java.nio.AsynchronousFileChannel**

- **Many niche APIs, e.g. mem mapping, line numbering**
  - Search them out as needed

- **Consider Okio if third party API allowed**

The end

May 9, 2018