

**Programming Languages, F2003**  
**Lecture 10, Wednesday 9 April 2003**

- The Java Virtual Machine (JVM), a real-world abstract machine
- Compiling micro-C to JVM code
- Microsoft's Common Language Runtime (CLR, .NET)
- The stack, the heap, and garbage collection
- The garbage collectors of Moscow ML and of the Sun Hotspot JVM

**Java Virtual Machine instructions**

- push constant onto local stack: `bipush, sipush, iconst, ldc, aconst_null, ...`
  - arithmetic: `iadd, isub, imul, idiv, irem, ineg, linc, fadd, fsub, ...`
  - load local variable onto local stack: `iload, aload, fload, ...`
  - store local variable from local stack: `istore, astore, fstore, ...`
  - load array element onto local stack: `iaload, baload, aaload, faload, ...`
  - stack manipulation: `swap, pop, dup, dup_x1, dup_x2, ...`
  - load field onto local stack: `getfield, getstatic`
  - method call: `invokestatic, invokevirtual, invokespecial`
  - method return: `return, ireturn, areturn, freturn, ...`
  - unconditional jumps: `goto`
  - conditional jumps (compare to 0): `ifeq, ifne, iflt, ifle, ifgt, ifge`
  - conditional jumps (compare two values): `if_icmpeq, if_icmpne, ...`
  - object-related: `new, instanceof, checkcast`
- Instruction type prefixes: `i=int, a=object, f=float, d=double, s=short, b=byte, ...`

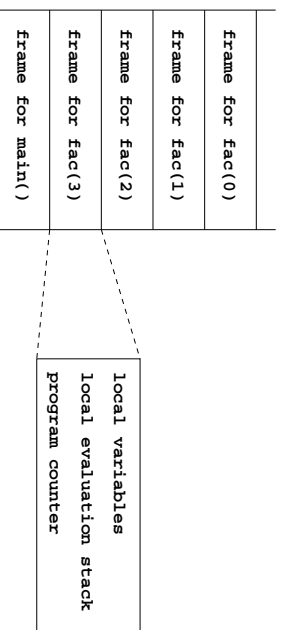
**The Java Virtual Machine (JVM), a stack abstract machine**

Compiled Java programs are executed by the Java Virtual Machine (JVM):

```
java Factorial 3
```

The JVM is a stack machine.

Every thread of execution in the JVM has its own stack, containing stack frames:



A stack frame represents one method that has been called but has not yet returned.

The stack frame contains the method's local variables and parameters, and its local expression evaluation stack.

**JVM class files**

JVM code and metadata (names and types of fields, methods, etc) are stored in *class files* \*.class

A class file contains:

- the name and package of the class, the superclass, superinterfaces, and access flags (`public, ...`);
- the *field declarations* and *method declarations* of the class;
- the *constant pool* containing field descriptions, method descriptions, string constants, etc.

For each method declaration (and similarly for field declarations), the class file describes:

- the *name* of the method, the *signature* of the method, and its *access modifiers* (`static, public, ...`);
- the *attributes*, including the code for the method, and the exceptions thrown by the method.

The code for a method is stored in attribute `CODE`; it includes:

- the maximal depth of the local stack in a stack frame for the method;
- the number of local variables in the method;
- the bytecode itself, as a list of JVM instructions;
- the *exception handlers*, that is, try-catch blocks, of the method body.

Each handler describes the bytecode range covered by the handler, its entry, and its exception class.

#### A Java example file (oc/ex5.java)

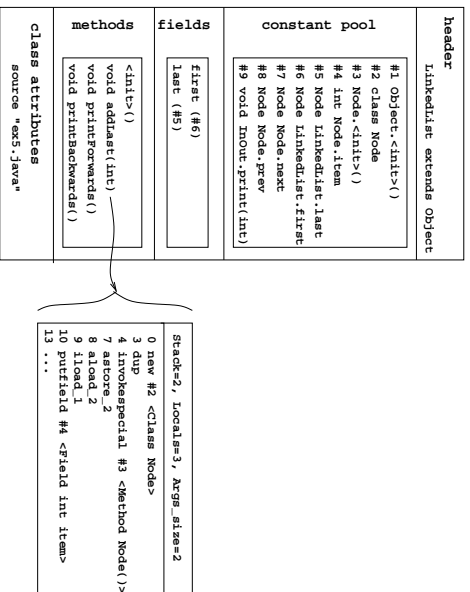
```
class Node extends Object {
    Node next;
    Node prev;
    int item;
}

class LinkedList extends Object {
    Node first, last;

    void addLast(int item) {
        Node node = new Node();
        node.item = item;
        if (this.last == null) {
            this.first = node;
            this.last = node;
        } else {
            this.last.next = node;
            node.prev = this.last;
            this.last = node;
        }
    }

    void printForwards() { ... }
    void printBackwards() { ... }
}
```

#### JVM class file LinkedList.class



The contents of a Java class file may be inspected using javap:  
javap -c -verbose LinkedList

#### Java bytecode verification

The JVM bytecode must be *verified* before execution, requiring for instance:

- all instructions must work on stack operands and local variables of the correct type;
- a method must use no more local variables, and no more local stack positions, than it claims to;
- for every given point in the bytecode, the local stack has the same fixed depth whenever that point is reached;
- a method must throw no more exceptions than it claims to;
- the execution of a method must end with a return or throw instruction, not fall off the end;
- execution must not use one half of a two-word value (a long or double) as a one-word value (int).

#### Generating JVM class files from SML: The SML-JVM toolkit

Manual generation of correct JVM class files is very cumbersome.

The SML-JVM toolkit helps generating binary JVM class files in SML.

```
fun jvmcompile2File program (classname : string) =
  let open Classdecl Localvar Label Bytecode
      ...
      (* The method public static void main(String[] args) { ... } *)
      val main : method_decl =
          let val locals0
              val (locals1, args) = nextVar1 locals0
              val (code, locals2) = cProgram args locals1 program
              in
                  {flags = [ACCpublic, ACCstatic],
                   name = "main",
                   msig = (TvmType.Tarray (TvmType.Tclass stringClass)), NONE,
                   attrs = [CODE {attrs = [],
                                stack = Stackdepth.maxdepth code [],
                                locals = Localvar.count locals2,
                                code = code,
                                hdls = []}]}
              end
          end
      val myClass = { mdecls = [main, ...], ... } : class_decl
  in ... Classfile.emlt ... myClass ... end
```

### Last lecture's backwards compiler from micro-C to stack machine code (jimp/contcomp.sml)

```

fun addCst i C =
  case (i, C) of
  | (0, EQ)      => addNOT C1
  | (0, ADD)     => C1 => C1
  | (0, SUB)     => C1 => C1
  | (0, NOT)     => C1 => addCST 1 C1
  | (_, NOT)    => C1 => addCST 0 C1
  | (_, MUL)    => C1 => C1
  | (1, DIV)    => C1 => C1
  | (_, INCSPL m) => C1 => if m < 0 then addINCSPL (m+1) C1
                        else CSTI 1 :: C
  | (0, IFZERO lab :: C1) => addGOTO lab C1
  | (_, IFZERO lab :: C1) => C1
  | (_, IFNZRO lab :: C1) => addGOTO lab C1
  | (_, IFNZRO lab :: C1) => CSTI 1 :: C
  | _            =>
    and cExpr (e : expr) (env : envv) (C : instr list) : instr list =
      case e of
      | Access acc      => cAccess acc env (LDI :: C)
      | ...             => addCST i C
      | Cst (CstI i)    => addCST i C
      | ...

```

### Compiling integer comparisons in the JVM

In the JVM, integer comparisons (<, <=, =, !=, >=, >) can be made only in conditional jumps.

There are one-argument comparisons (compare to zero) and two-argument comparisons (compare two numbers).

There are comparisons for less, equal, greater, and their negations.

All in all there are  $2 \cdot 3 \cdot 3 = 12$  different comparisons: `if_cmp_eq`, `if_eq`, `if_cmp_ne`, `if_ne`, ...

This complicates the code generation for an integer comparison:

- if used in an `if- or while`-condition, generate a conditional jump;
- if used in a composite logical expression (`&&` or `||`); sometimes generate a conditional jump, sometimes not;
- if assigned to a variable, a 0 (false) or 1 (true) must be generated on the stack;
- if one operand of a comparison is 0, use a one-argument comparison, otherwise a two-argument comparison;
- the usual optimizations should apply if the comparison is followed by a negation, or jump, or ...

Handling all these special cases requires a systematic approach.

### A backwards compiler from micro-C to JVM bytecode (jimp/jvmcomp.sml)

It compiles only a subset of micro-C: no pointers (because of verification), only one function: `main`.

In the compiler, we replace our homegrown stack machine instructions with JVM instructions.

The compiler uses SML representations `Jiadd`, ... of the JVM instructions `iadd`, ...:

```

fun addCst i C =
  case (i, C) of
  | (0, Jiadd)  => C1
  | (0, Jisub) => C1 => C1
  | (1, Jimul) => C1 => C1
  | (1, Jidiv) => C1 => C1
  | (_, Jipop)  => C1 => C1
  | (0, Jifeq lab :: C1) => addGoto lab C1
  | (_, Jifeq lab :: C1) => C1
  | (0, Jifne lab :: C1) => C1
  | (_, Jifne lab :: C1) => addGoto lab C1
  | _          => intConst 1 :: C;

fun cExpr (e : expr) (env : envv) (C : instrs) : instrs =
  case e of
  | Access(AccVar x) => JIload (cAccess x env) :: C
  | ...              => addCst i C
  | ...

```

### Calling the comparison compiler mknumtest from the expression compiler

```

fun cExpr e (env : envv) (C : jvm_instr list) : jvm_instr list =
  case e of
  | ...
  | Prim2(ope, e1, e2) =>
      (case ope of
      | "==" => mknumtest env e1 e2 EQUAL true C
      | "!=" => mknumtest env e1 e2 EQUAL false C
      | "<"  => mknumtest env e1 e2 LESS true C
      | ">"  => mknumtest env e1 e2 LESS false C
      | ">=" => mknumtest env e1 e2 LESS true C
      | ">"  => mknumtest env e1 e2 GREATER true C
      | "<=" => mknumtest env e1 e2 GREATER false C
      | _    =>
          (cExpr e1 env
           (cExpr e2 env
            (case ope of
            | "*" => JImul :: C
            | "+" => Jiadd :: C
            | "-" => Jisub :: C
            | "/" => Jidiv :: C
            | "%" => Jirem :: C
            | _   => raise Fail "unknown primitive 2")))))

```

### The ultimate comparison compiler

```
fun mkxumtest env e1 e2 cmp sign C =
  let val l1test = ((Jiflt, Jifge), (* Compare one operand to zero *)
    (Jifeq, Jifne),
    (Jifgt, Jiflle))
    val i2test = ((Jif_cmplt, Jif_cmpge), (* Compare two operands *)
    (Jif_cmpeq, Jif_cmpne),
    (Jif_cmpgt, Jif_cmplle))
    fun rev LESS = GREATER | ... (* Reverse comparison *)
    fun select ((lt, ge), (eq, ne), (gt, lle)) ord b =
      case ord of
        LESS => if b then lt else ge
      | EQUAL => if b then eq else ne
      | GREATER => if b then gt else lle
    fun genTest b l1l C2 = (* Optimize if one operand is zero *)
      case (e1, e2) of
        (_, Cst (CstI 0)) =>
          cExpr e1 env (select l1test cmp b l1l :: C2)
        | (Cst (CstI 0), _) =>
          cExpr e2 env (select i1test (rev cmp) b l1l :: C2)
        | _ =>
          cExpr e1 env (cExpr e2 env
            (select i2test cmp b l1l :: C2))
    in
      case C of
        Jifne l1l :: C1 => genTest sign l1l C1
      | Jifeq l1l :: C1 => genTest (not sign) l1l C1
      | _ => ... push 0 or 1 onto stack depending on comparison ...
    end
```

FIG

### The quality of the generated code: example `imp/ex1.3.c`

```
void main(int n) {
  int y;
  y = 1889;
  while (y < n) {
    y = y + 1;
    if (y % 4 == 0 && y % 100 != 0 || y % 400 == 0)
      print y;
  }
}
```

The code generated by our `jvncomp` is the same as that generated by `javac` from `imp/ex1.3.java`

### Speed of the generated JVM code

Loop executing 20 million times (file `imp/ex8.c`) and *n* queens problem (file `imp/ex11.c`):

Target	20 mill loop	12 queens
Interpreting stack machine code	8.90	29.90
Executing JVM code	0.47	1.91

Why is it so much slower to execute Machine Interpreting `ex11.out`, than to execute `Ex11?`

FIG

### Interpretive overhead

The Machine-`.java` stack machine interpreter spends most of the time figuring out what to do:

```
for (i;) {
  if (trace)
    printspc(s, bp, sp, p, pc);
  switch (plpc++) {
  case CST:
    s[sp+1] = p[pc++]; sp++; break;
  case ADD:
    s[sp-1] = s[sp-1] + s[sp]; sp--; break;
  case SUB:
    s[sp-1] = s[sp-1] - s[sp]; sp--; break;
  case MUL:
    s[sp-1] = s[sp-1] * s[sp]; sp--; break;
  case DIV:
    s[sp-1] = s[sp-1] / s[sp]; sp--; break;
  case MOD:
    s[sp-1] = s[sp-1] % s[sp]; sp--; break;
  case EQ:
    s[sp-1] = (s[sp-1] == s[sp]) ? 1 : 0; sp--; break;
  case LT:
    s[sp-1] = (s[sp-1] < s[sp]) ? 1 : 0; sp--; break;
  }
}
```

FIG

### The stack, the heap, and garbage collection

Stack allocation is efficient: allocate = increment stack pointer; deallocate = decrement stack pointer.

But a stack works only if lifetimes are (1) predictable, and (2) nested last-in-first-out.

Some values (arrays, objects, strings, closures, ...) may have unpredictable lifetime.

The JVM allocates all objects, strings, and arrays in a heap.

A *heap* is a collection of values with addresses. No relation to algorithms (priority queues).

The running program can allocate values in the heap but cannot explicit deallocate them.

Deallocation is done automatically by a *garbage collector*.

The garbage collector recycles all memory that is not reachable from the *root set*: stack frames and registers.

The running program is sometimes called the *mutator*, in contrast to the *collector*.

Garbage collection has been used since 1960 for functional, object-oriented, and scripting languages.

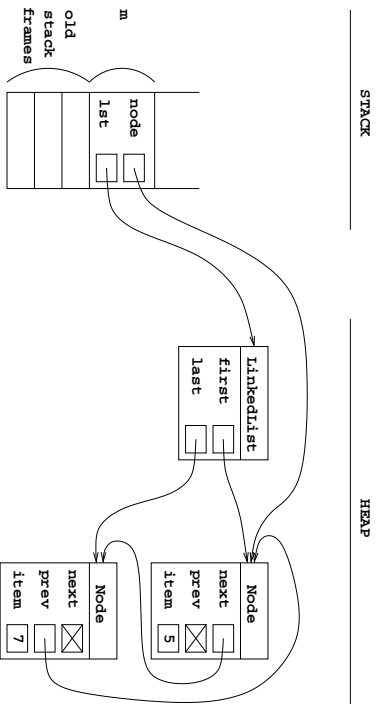
Java brought garbage collection into mainstream programming.

Pascal, C, C++ are not designed to work with a garbage collector, but can be used with *conservative* collectors.

FIG

### JVM runtime state: frame stack and heap (oo/ex5.java)

```
void m() {
    LinkedList lst = new LinkedList();
    lst.addLast(5);
    lst.addLast(7);
    Node node = lst.first;
}
```



IT-C

### Another stack machine: Microsoft Common Language Runtime (CLR)

Common Language Runtime (CLR) is part of Microsoft .Net; version 1.0 January 2002; version 1.1 April 2003.

Goals of the CLR:

- provide a safe (managed) execution environment with verification and garbage collection (like the JVM);
- support a range of languages: C#, Visual Basic, JavaScript, Standard ML, Eiffel#, ... (unlike the JVM);
- support integration of languages: a C# class may extend a Visual Basic class, etc;
- support call-by-reference parameter passing in addition to call-by-value;
- support value types: structs allocated in the stack (whereas objects are allocated in the heap);
- allow unmanaged, unverified execution to support C and C++ pointer arithmetics etc (unlike the JVM).

IT-C

### Getting and using C#Common Language Runtime

The .NET Framework SDK implementation of CLR can be downloaded from [msdn.microsoft.com/net/](http://msdn.microsoft.com/net/). It is free but requires MS Windows NT/2000/XP.

A shared source version of CLR for FreeBSD (Unix) and MS Windows is available from Microsoft.

An open source implementation of CLR and C# is available from the Mono project ([www.go-mono.com](http://www.go-mono.com)).

To compile and run C# program ex13.cs:

```
csc ex13.cs
ex13 2050
```

The CLR code of a program is bundled in an assembly (a so-called .exe file), not in several .class files.

To disassemble the CLR code in file ex13.exe:

```
ildasm /text ex13.exe
```

Let us compare the JVM code for file imp/ex13.java to the CLR code for imp/ex13.cs.

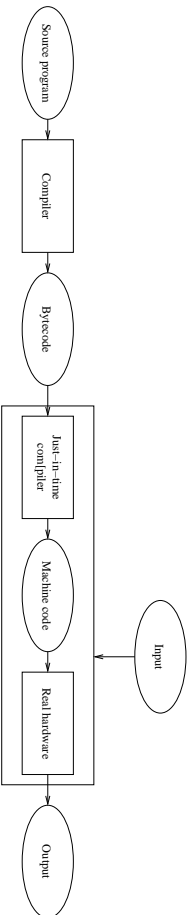
IT-C

0 aload_0	IL_0000: ldarg.0	args
1 iconst_0	IL_0001: ldc.i4.0	
2 aaload	IL_0002: ldlem.ref	args[0]
3 invokestatic #2 (...)	IL_0003: call	parse int
6 istore_1	IL_0008: stloc.0	n = ...
7 sipush 1889	IL_0009: ldc.i4	
10 istore_2	IL_000e: stloc.1	Y = 1889;
11 goto 43	IL_000f: br.s	while (...)
14 aload_2	IL_0011: ldloc.1	
15 iconst_1	IL_0012: ldc.i4.1	
16 iadd	IL_0013: add	Y = Y + 1;
17 istore_2	IL_0014: stloc.1	
18 aload_2	IL_0015: ldloc.1	
19 iconst_4	IL_0016: ldc.i4.4	
20 irem	IL_0017: rem	Y % 4 == 0
21 ifne 31	IL_0018: brtrue.s	
24 aload_2	IL_001a: ldloc.1	
25 bipush 100	IL_001b: ldc.i4.s	100
27 irem	IL_001d: rem	
28 ifne 39	IL_001e: brtrue.s	Y % 100 != 0
31 aload_2	IL_0020: ldloc.1	
32 sipush 400	IL_0021: ldc.i4	0x190
35 irem	IL_0026: rem	
36 ifne 43	IL_0027: brtrue.s	Y % 400 == 0
39 aload_2	IL_0029: ldloc.1	
40 invokestatic #3 (...)	IL_002a: call	print Y
43 aload_2	IL_002f: ldloc.1	
44 aload_1	IL_0030: ldloc.0	(Y < n)
45 if_cmpit 14	IL_0031: blt.s	
48 bipush 10	IL_0033: ldc.i4.s	10
50 invokestatic #4 (...)	IL_0035: call	newline
53 return	IL_003a: ret	return

IT-C

### Just-in-time compilation

The virtual machines (Sun HotSpot JVM and the CLR) do not execute the bytecode one instruction at a time. They compile the bytecode to machine code (after loading and verification):



This is called *just-in-time* compilation; it may even wait until the second time a loop is executed.

One can inspect the x86 machine code generated by the CLR just-in-time compiler:

```
ildasm /out:ex13.11 ex13.exe  
ilasm /debug ex13
```

Then use the CLR debugger `dbgclr` to look at the CLR code and its translation into x86 code.

IT-C

### Garbage collection techniques

#### Mark-sweep garbage collection and the freelist

The heap consists of used and free blocks.

The free blocks are linked together in a *freelist*.

**Allocation:** search for a large enough free block on the freelist.

**Garbage collection** is done in two phases:

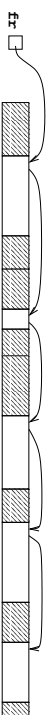
- Mark phase: find and mark all values that are reachable from the root set.
- Sweep phase: put unmarked values on the freelist.

Advantages: simple to implement; values do not move.

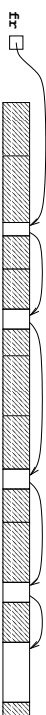
Disadvantages: heap may become fragmented; searching for a large enough free block may be slow.

IT-C

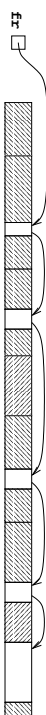
### Mark-sweep garbage collection



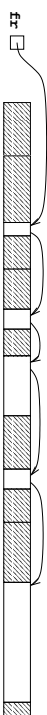
After some more allocation



After mark phase



After sweep phase



IT-C

#### Two-space stop-and-copy garbage collection

The heap is divided into two equally large spaces: *from-space* and *to-space*.

**Allocation:** allocate in the free part of from-space.

**Garbage collection:**

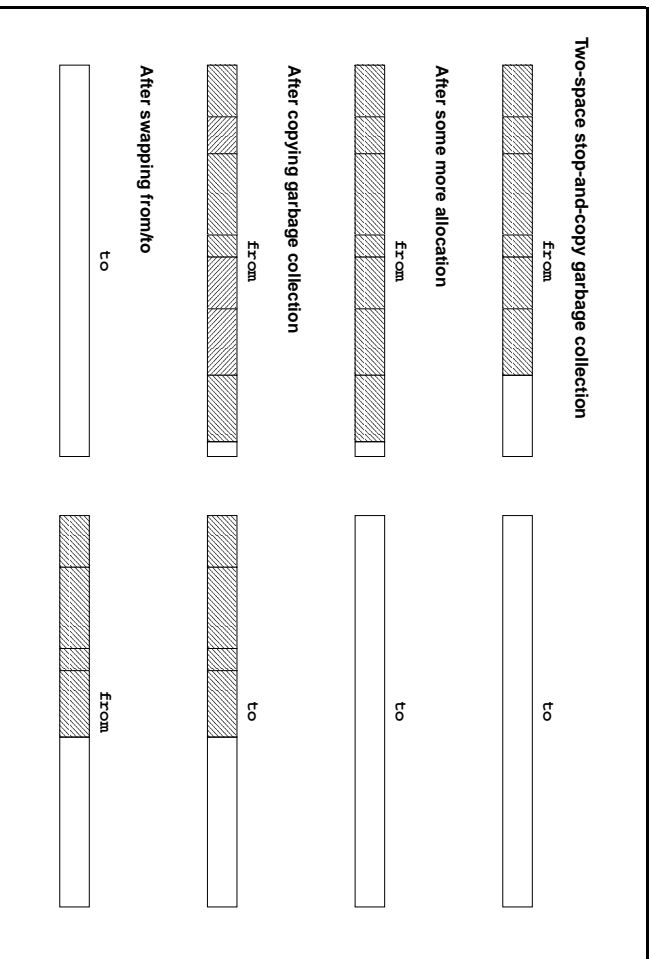
All values reachable from the root set are moved from from-space to to-space.

Then the roles of from-space and to-space are swapped.

Advantages: copying will compact the live values; no fragmentation; fast if there is much memory available.

Disadvantages: can use at most half the available memory; values are moved; very slow if there is little memory.

IT-C



**Generational garbage collection**

Observation: Most values die young; old values most likely will live for yet some time. It's wasteful to move all the old values, just to reclaim the space left by young values.

Solution: Divide the heap into several *generations*.

Allocate in generation 1, the youngest one.

Garbage collection in generation  $n$  moves live values to generation  $n + 1$ .

Consequences:

- Only a few values survive and need to be moved in every 'minor' garbage collection.
- Older generations need not be collected very often.

A complication: assignment to a field may create a reference from an old generation into a younger one. Such references must be taken into account when collecting the younger generation.

**The garbage collector in Moscow ML**

The garbage collector was written for Caml Light by Damien Doligez, INRIA, France.

The heap has two generations.

The garbage collector copies from generation 1 and does incremental mark-sweep collection in generation 2.

Garbage collection in generation 1 (a minor collection) moves live values to generation 2.

Garbage collection in generation 2 is done by *incremental* mark-sweep.

The mark phase and the sweep phase are divided into slices.

For every minor collection, a slice of the mark phase or a slice of the sweep phase is done in generation 2.

The division into slices reduces garbage collection pauses and improves real-time response.

A somewhat complicated storage invariant is maintained by the garbage collector.

**The garbage collector in Sun Hotspot since JDK 1.3.1**

The JVM heap has three generations.

A minor collection copies from generation 1 to generation 2.

Generation 2 consists of two semi-spaces with stop-and-copy garbage collection.

Values are moved to generation 3 when they are old enough.

Generation 3 uses (non-incremental) mark-sweep with compaction.

Generation 3 collections can be made incremental by passing the option `-Xincgc` to `java`.

The behaviour of the JVM's garbage collector can be studied using option `java -verbose:gc`