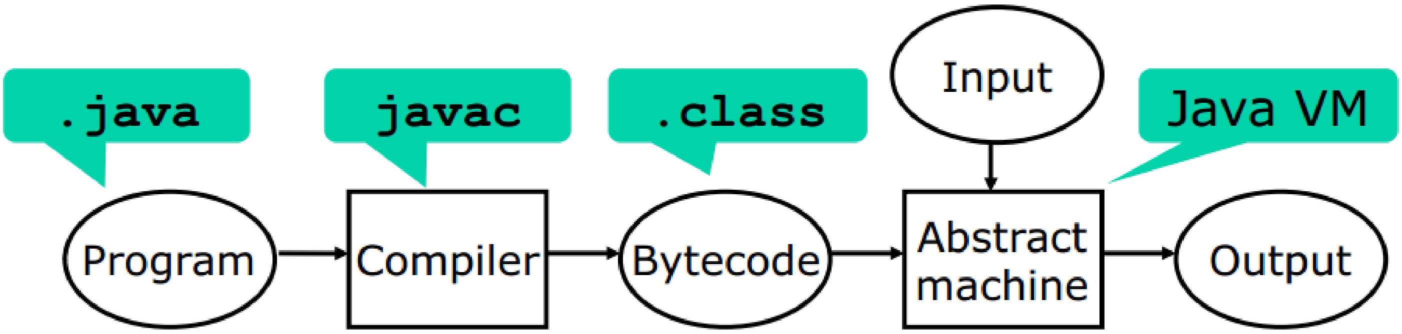
内容

- Java虚拟机 Java Virtual Machine
- .NET公共语言运行时（CLI）.NET Common Language Infrastructure (CLI)
- 垃圾回收（GC）技术Garbage collection (GC) techniques
    - 引用计数 Reference-counting
    - 标记扫描 Mark-sweep
    - 双空间停止和复制 Two-space stop and copy
    - JVM和.NET中的垃圾回收器 The garbage collectors in JVM and.NET
- List-C，一个带有堆和GC的Micro-C版本 List-C, a version of Micro-C with a heap and GC

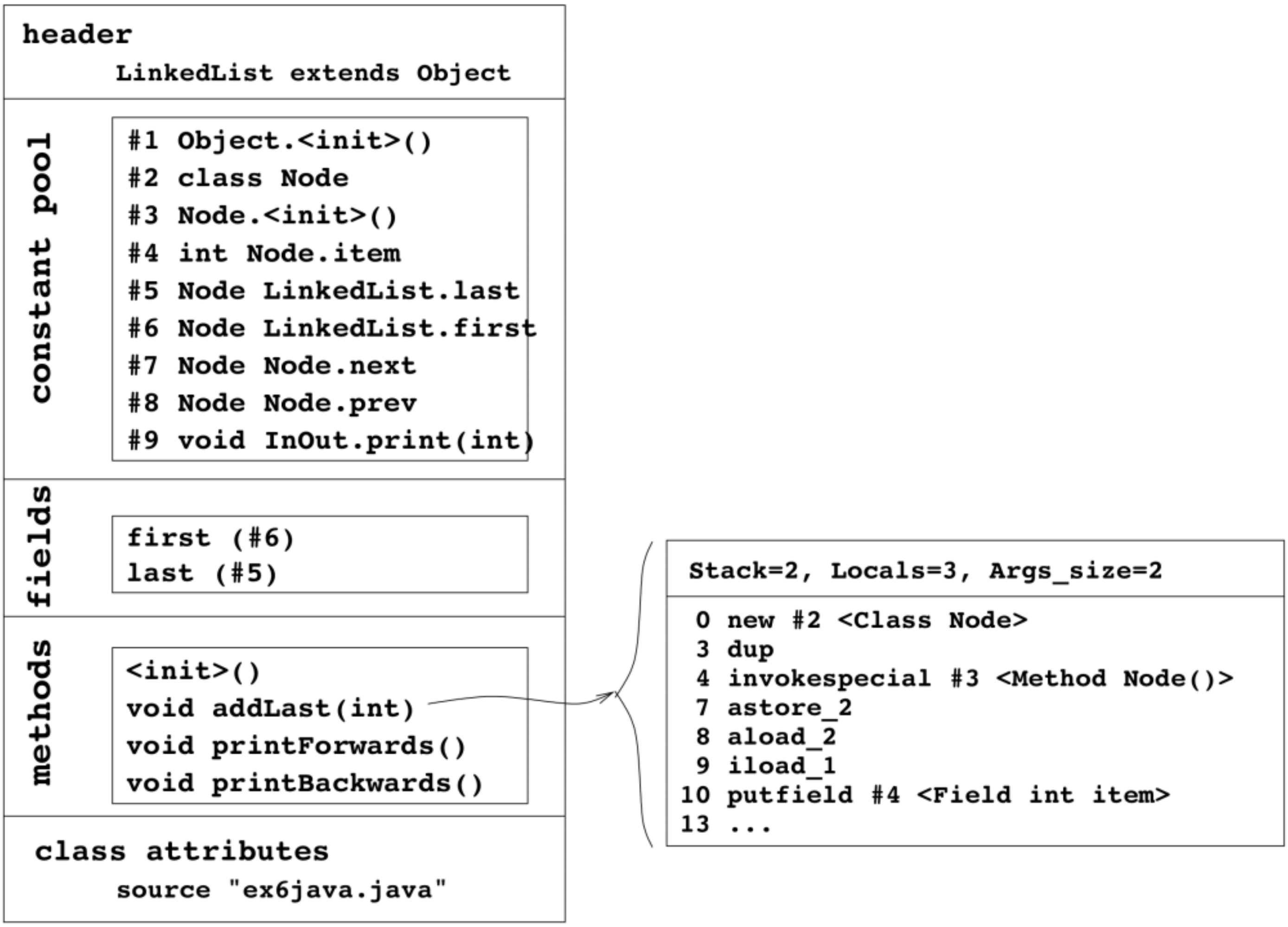


示例程序

```
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文件

LinkedList.class




```
Generated by
javac ex6java.java

Shown by
javap -c -v LinkedList
```

一些JVM字节码指令 Some JVM bytecode instructions

类Kind	示例说明Example instructions
push constant	iconst, ldc, aconst_null, …
arithmetic	iadd, isub, imul, idiv, irem, ineg, iinc, fadd, …
load local variable	iload, aload, fload, …
store local variable	istore, astore, fstore, …
load array element	iaload, baload, aaload, …

类Kind	示例说明Example instructions
stack manipulation	swap, pop, dup, dup_x1, dup_x2, …
load field	getfield, getstatic
method call	invokestatic, invokevirtual, invokespecial
method return	return, ireturn, areturn, freturn, …
unconditional jump	goto
conditional jump	ifeq, ifne, iflt, ifle, …; if_icmpeq, if_icmpne, …
object-related	new, instanceof, checkcast
Type prefixes: i=int, a=object, f=float, d=double, s=short, b=byte, …	

JVM 字节码验证 JVM bytecode verification

字节码在加载时，在执行之前验证JVM字节码：The JVM bytecode is verified at loadtime, before execution:

- 指令执行时，堆栈上操作数和局部变量的类型必须正确　An instruction must work on stack operands and local variables of the correct type
- 方法中使用的局部变量，堆栈与其声明相符合　A method must use no more local variables and no more local stack positions than it claims to
- 对于字节码中的每一个点，本地堆栈都是相同的深度　For every point in the bytecode, the local stack has the same depth whenever that point is reached
- 方法抛出的异常正确 A method must throw no more exceptions than it admits to
- 方法的执行必须以return或throw指令结束 The execution of a method must end with a return or throw instruction, not `fall off the end'
- 方法执行不得使用双字值（例如，long）的一半作为单字值（int）Execution must not use one half of a two-word value (e.g. a long) as a one-word value (int)

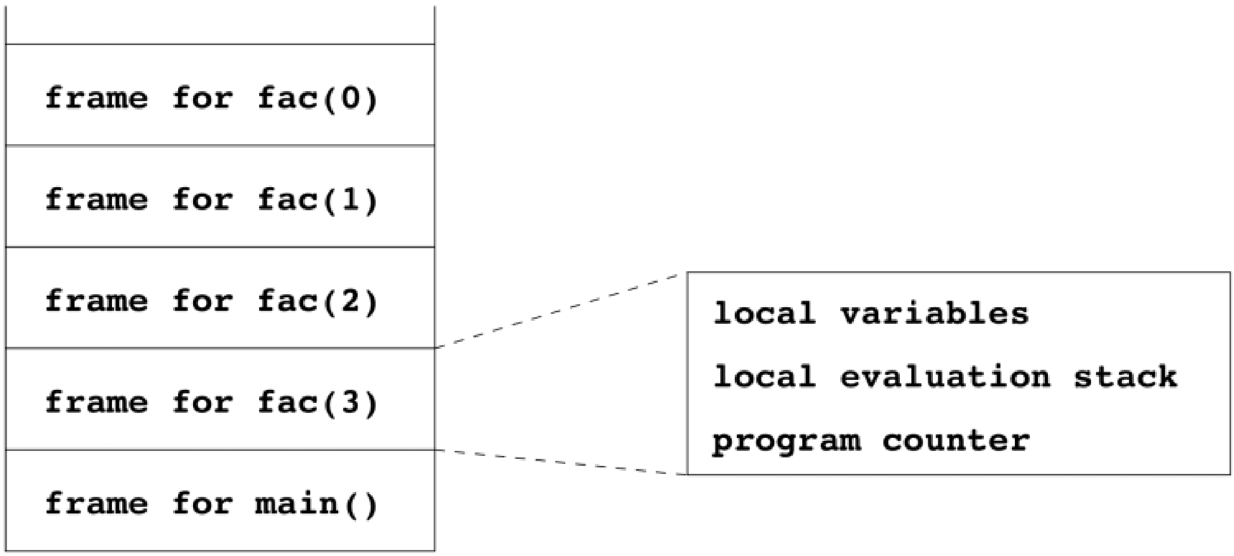
其他JVM运行时检查 Additional JVM runtime checks

- 数组边界检查arr[i] Array-bounds checks on arr[i]
- 数组赋值检查：数组A[]只能存储A的子类型 Array assignment checks: Can store only sub types of A into an A[] array
- 空引用检查（引用为null或指向对象/数组）Null-reference check (a reference is null or points to an object or array, because no pointer arithmetics)
- 类型转换检查：无法进行任意转换对象类之间Checked casts: Cannot make arbitrary conversions between object classes
- 内存分配成功或抛出异常 Memory allocation succeeds or throws exception
- 没有手动内存释放或重用 No manual memory deallocation or reuse
- JVM程序无法读取或覆盖任意内存 Bottom line: A JVM program cannot read or overwrite arbitrary memory

- 更好的调试，更好的安全性 Better debugging, better security
- 没有缓冲区溢出攻击，蠕虫等 No buffer overflow attacks, worms, etc as in C/C++

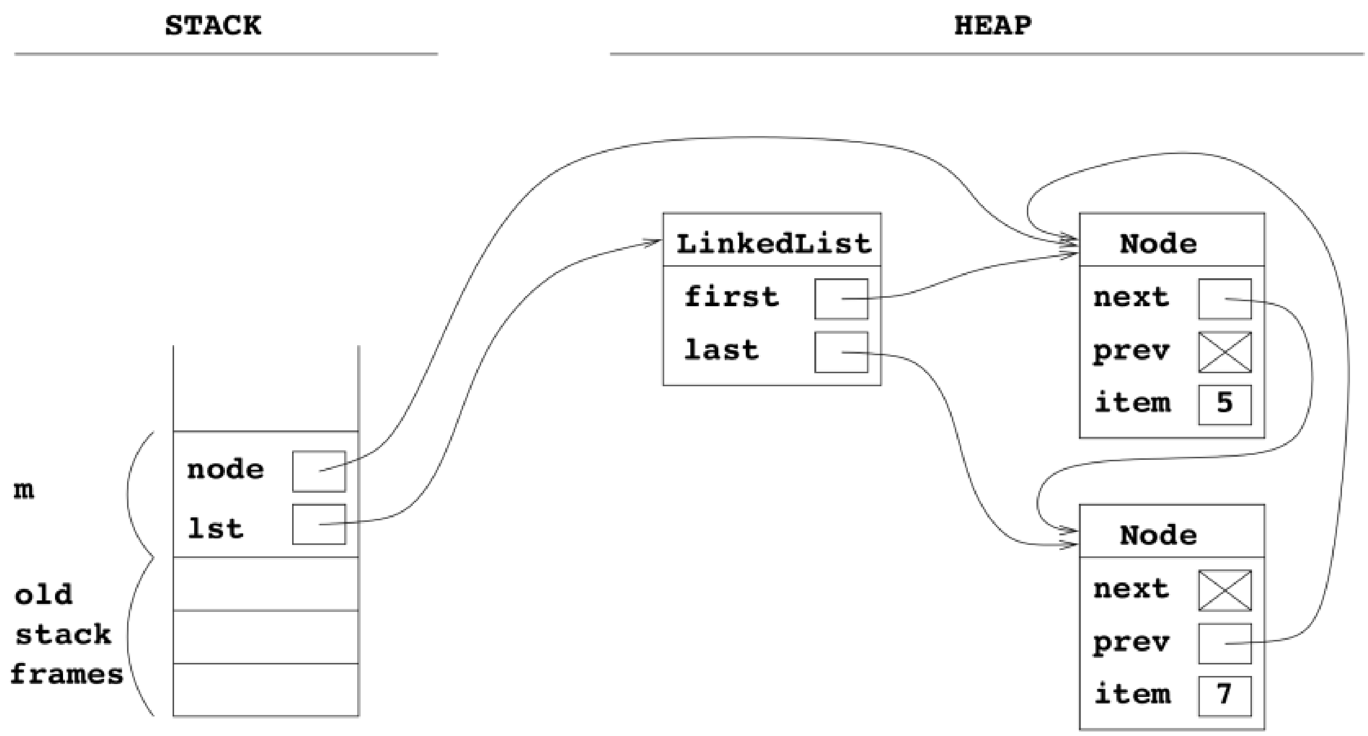
JVM运行时堆栈 The JVM runtime stacks

- 每个线程一个运行时栈 One runtime stack per thread
  - 每个函数调用的活动记录 Contains activation records, one for each active function call
  - 每个活动记录都有程序计数器，局部变量和中间结果 Each activation record has program counter, local variables, and local stack for intermediate results



运行时状态 Example JVM runtime state

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



.NET公共语言运行时 The .NET Common Language Infrastructure (CLI, CLR)

- 与JVM相同的哲学和设计 Same philosophy and design as JVM
- 一些改进：Some improvements:
  - 标准化的字节码汇编（文本）格式 Standardized bytecode assembly (text) format
  - 更好的版本控制，强名 Better versioning, strongnames, …
  - 设计为多种源语言的目标 Designed as target for multiple source languages (C#, VB.NET, JScript, Eiffel, F#, Python, Ruby, …)
  - 用户定义的值类型（struct） User-defined value types (structs)
  - 尾调用优化 Tail calls to support functional languages
  - 泛型字节码 True generic types in bytecode: safer, more efficient, and more complex
- `.exe`文件 = `stub + 字节码` The .exe file = stub + bytecode
- 标准化`Ecma-335` Standardized as Ecma-335

.NET CLI字节码指令 Some .NET CLI bytecode instructions

Kind	Example instructions
push constant	ldc.i4, ldc.r8, ldnull, ldstr, ldtoken

Kind	Example instructions
arithmetic	add, sub, mul, div, rem, neg; add.ovf, sub.ovf, …
load local variable	ldloc, ldarg
store local variable	stloc, starg
load array element	ldelem.i1, ldelem.i2, ldelem.i4, ldelem.r8
stack manipulation	pop, dup
load field	ldfld, ldstfld
method call	call, calli, callvirt
method return	ret
unconditional jump	br
conditional jump	brfalse, brtrue; beq, bge, bgt, ble, blt, …; bge.un …
object-related	newobj, isinst, castclass
Type suffixes: i1=byte, i2=short, i4=int, i8=long, r4=float, r8=double, …	

Java/C# 字节码 From Java and C# to bytecode

- Java /C#/C 程序ex13：Consider the Java/C#/C program ex13:

```
//java
static void Main(string[] args) {
   int n = int.Parse(args[0]);
   int y;
   y = 1889;
   while (y < n) {
     y = y + 1;
     if (y % 4 == 0 && (y % 100 != 0 || y % 400 == 0))
        InOut.PrintI(y);
   }
   InOut.PrintC(10);
}
```

- 编译和反编译：Let us compile and disassemble it twice:
    - javac ex13.java
    - javap –c ex13
    - csc /o ex13.cs

- ildasm /text ex13.exe

JVM 字节码 .Net字节码

```
00 aload_0                          |              0000 ldarg.0
01 iconst_0                         |              0001 ldc.i4.0
02 aaload                           |              0002 ldelem.ref
03 invokestatic parseInt            |              0003 call Parse

06 istore_1                         |              0008 stloc.0
07 sipush 1889                      |              0009 ldc.i4 0x761
10 istore_2                         |              000e stloc.1
11 iload_2                          |              000f br 003b
12 iload_1                          |              0014 ldloc.1
13 if_icmpge 48                     |              0015 ldc.i4.1
16 iload_2                          |              0016 add
17 iconst_1                         |              0017 stloc.1
18 iadd                             |              0018 ldloc.1
19 istore_2                         |              0019 ldc.i4.4
20 iload_2                          |              001a rem
21 iconst_4                         |              001b brtrue 003b
22 irem                             /              0020 ldloc.1
23 ifne 11                          |              0021 ldc.i4.s 100
26 iload_2                          |              0023 rem
27 bipush 100                       |              0024 brtrue 0035
29 irem                             /              0029 ldloc.1
30 ifne 41                          |              002a ldc.i4 0x190
33 iload_2                          |              002f rem
34 sipush 400                       |              0030 brtrue 003b
37 irem                             /              0035 ldloc.1
38 ifne 11                          |              0036 call PrintI
41 iload_2                          |              003b ldloc.1
42 invokestatic printi              |              003c ldloc.0
45 goto 11                          |              003d blt 0014
48 bipush 10                        |              0042 ldc.i4.s 10
50 invokestatic printc              |              0044 call PrintC
53 return                           |              0049 ret
```

练习Ten-minute exercise

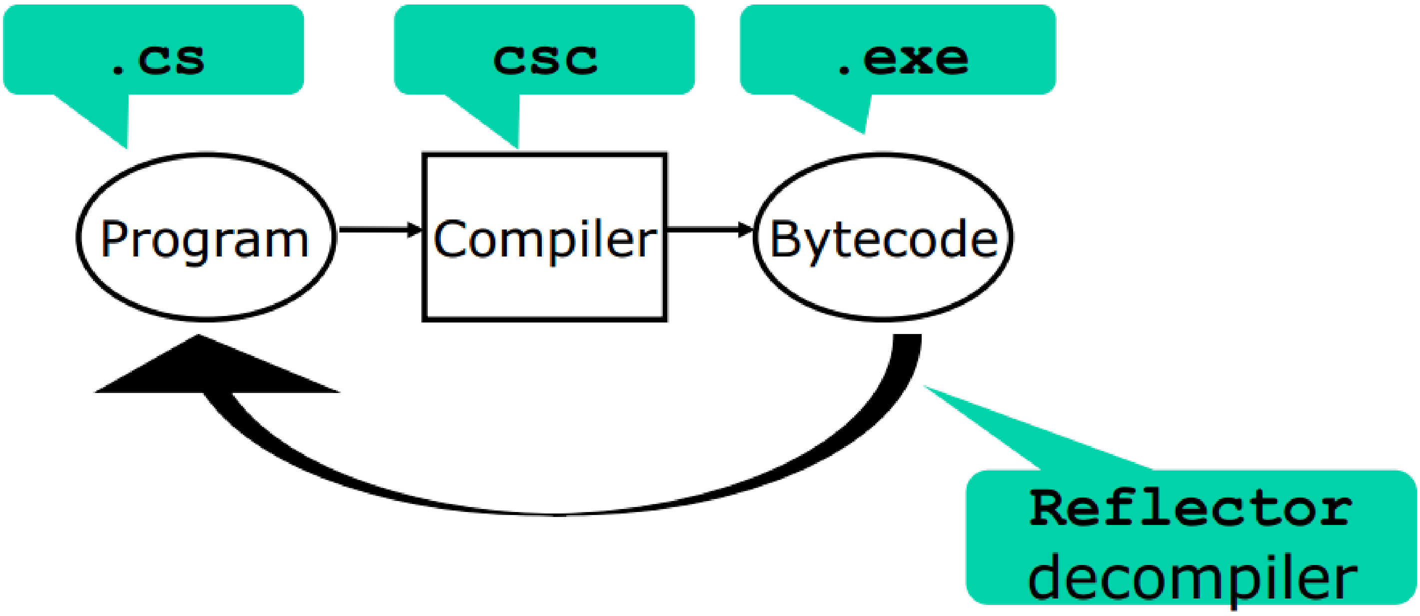
查看上面的字节码，并

- JVM /.NET For both the JVM and the .NET columns
  - 绘制箭头以指示跳转位置 Draw arrows to indicate where jumps go
  - 标出字节码连续的代码块 Draw blocks around the bytecode segments 对应ex13.java和ex13.cs程序中的表达式和语句 corresponding to expressions and statements in the ex13.java and ex13.cs programs

元数据和反编译器 Metadata and decompilers

- `.class`和`.exe` 文件包含`元数据`：`字段，方法，类的名称，类型` The .class and .exe files contains metadata : names and types of fields, methods, classes
- 可以将字节码`反编译`成程序 One can decompile bytecode into programs:

- 保护知识产权困难 Bad for protecting your secrets (intellectual property)
- 可使用字节码混淆器对抗反编译 Bytecode obfuscators make decompilation harder



.Net VM 支持泛型 .NET CLI has generic types, JVM doesn’t

```
class CircularQueue<T> {              //Source;  generics
  private readonly T[] items;
  public CircularQueue(int capacity) {
     this.items = new T[capacity];
  }
  public T Dequeue() { ... }
  public void Enqueue(T x) { ... }
}
```

```
.class CircularQueue`1<T> ... {      //  .NET CLI;  generics
  .field private initonly !T[] items
  ...
  .method !T      Dequeue() { ... }
  .method void Enqueue(!T x) { ... }
}
```

```
class CircularQueue ... {                    // JVM; no generics
  public java.lang.Object dequeue(); ...
  public void enqueue(java.lang.Object); ...
}
```

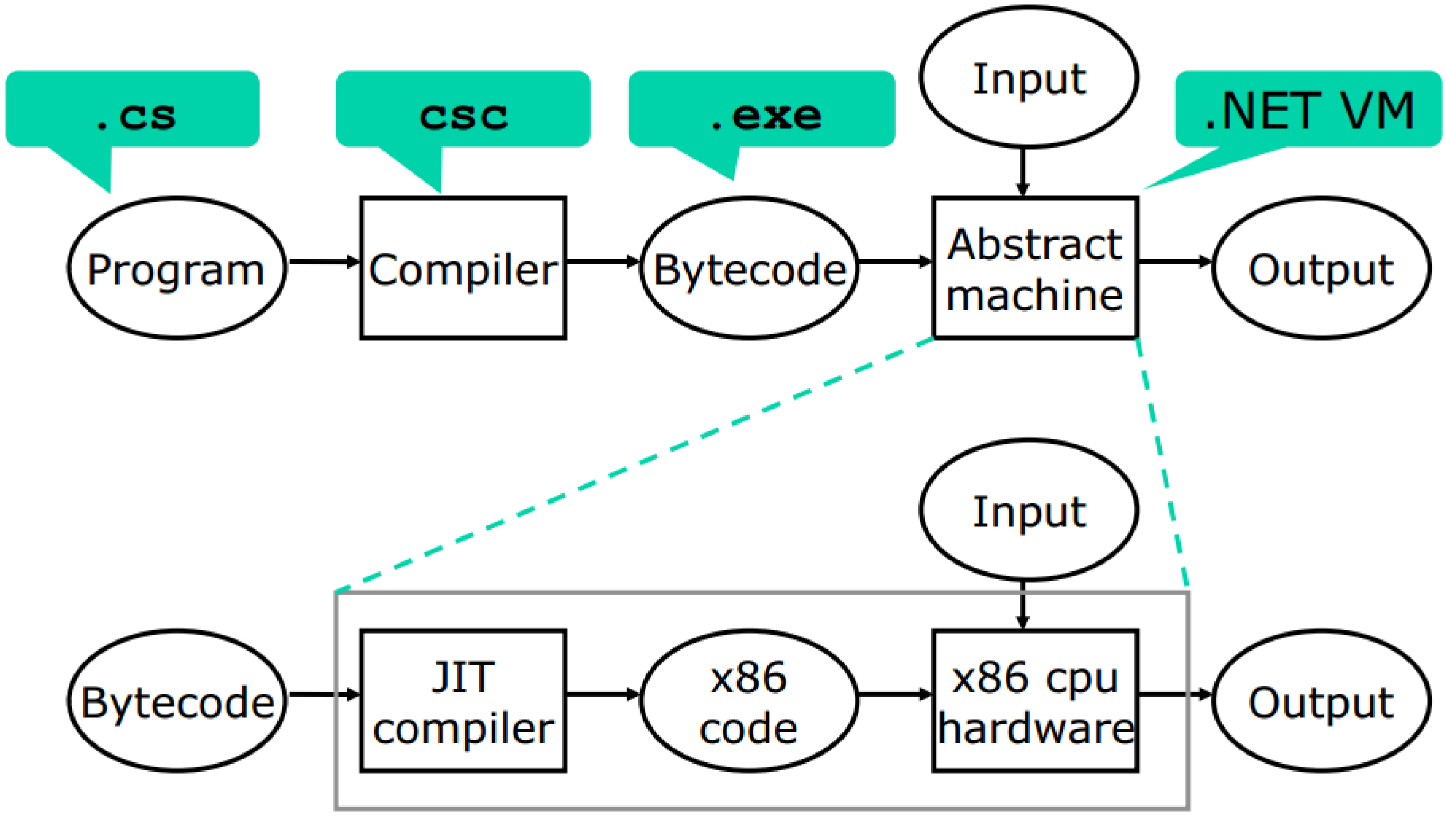
Java如何实现泛型 Consequences for Java

- Java编译器替换T The Java compiler replaces T
  - 用 `Object`类型代替 在C 中的`T` with Object in C
  - Mytype in C
- 以下特性在`Java`无法使用，但在`C#` 中可以： So this doesn't work in Java, but works in C#:
  - 类型转换 Cast: `(T)e`
  - 实例检查：`e instance of T` Instance check: (e instanceof T)
  - 反射：`T.class` Reflection: T.class
  - 在gen类的不同类型实例上的重载： Overload on different type instances of gen class:
  - 数组创建：arr = new T [10] Array creation: arr=new T[10] 因此Java版本的CircularQueue 必须使用 So Java versions of CircularQueue must use ArrayList ，而不是T [] ArrayList, not T[]

```
void put(CircularQueue<Double> cqd) { ... }
void put(CircularQueue<Integer> cqd) { ... }
```

JIT编译 Just-in-time (JIT) compilation

- 字节码编译到机器码（例如x86） Bytecode is compiled to real (e.g. x86) machine
- 代码在运行时速度同C/C++相当 code at runtime to get speed comparable to C/C++



即时编译 Just-in-time compilation

- 查看.NET JIT后的代码 How to inspect .NET JITted code

```
// C#
static double Sqr(double x) {
    return x * x;
}
csc /debug /o Square.cs
```

```
    IL_0000:   ldarg.0
    IL_0001:   ldarg.0                        CLI
    IL_0002:   mul
    IL_0003:   ret
```

```
JIT compiler      x86
```

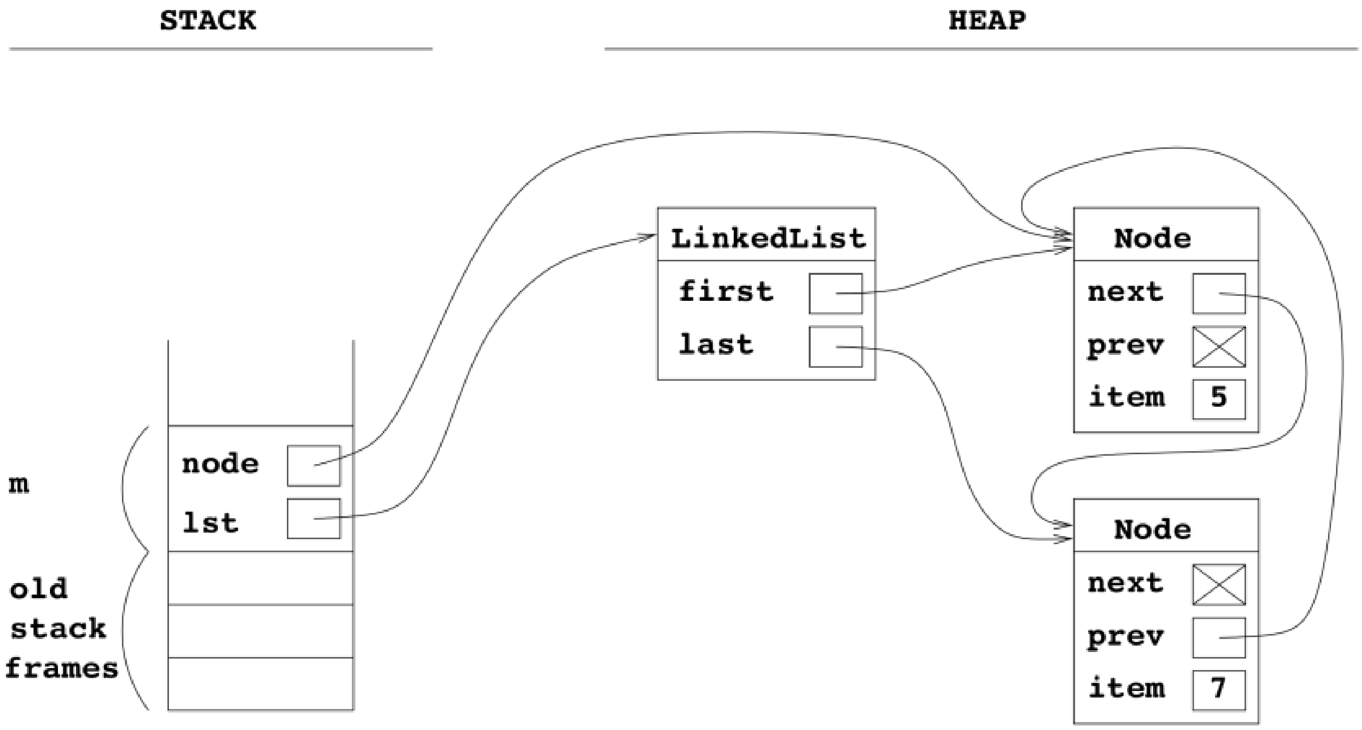
```
                              00 pushl    %ebp
                              01 movl     %esp,%ebp
                              03 subl     $0x08,%esp
                              06 fldl     0x08(%ebp)
 Mono 3.2.3 MacOS 32 bit      09 fldl     0x08(%ebp)
                              0c fmulp    %st,%st(1)
                              0e leave            ==>    movl    %ebp,%esp
 mono -optimize=-inline       0f ret                     popq    %ebp
       -v -v Square.exe
```

垃圾回收Garbage collection

- 引用计数 Reference counting
- 标记清除 Mark-sweep
- 双空间停止复制，压缩 Two-space stop-and-copy, compacting
- 多代GC Generational
- 保守GC Conservative

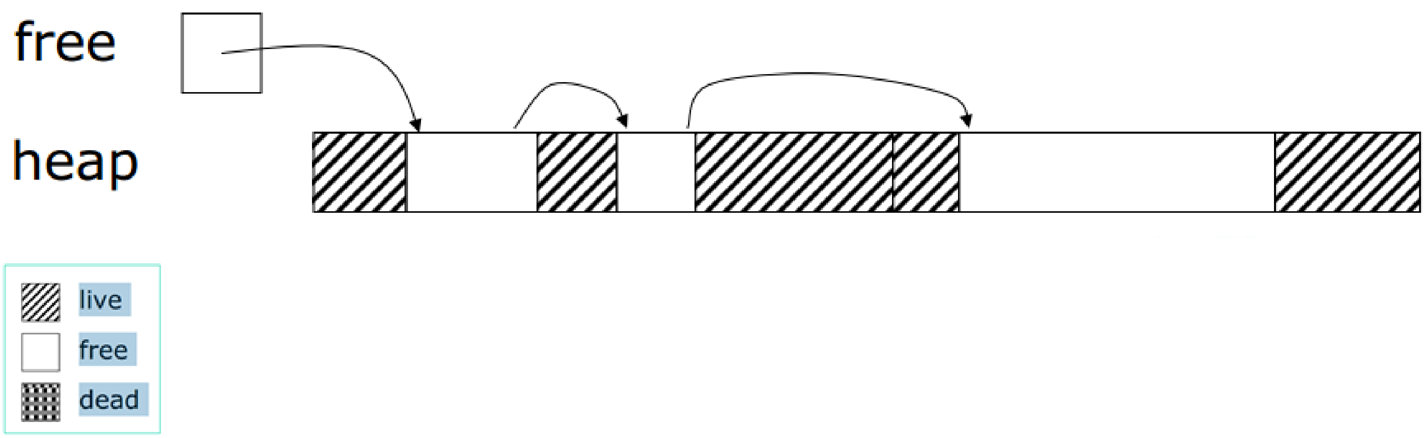
堆上的对象关系图 The heap as a graph

- node = object，edge = referenceThe heap is a graph : node=object, edge=reference
- 如果从`根 roots`可达，对象是活的 An object is live if reachable from roots
- 垃圾回收 `根 roots` = 运行时堆栈，寄存器，全局数据区 Garbage collection roots = stack elements



空闲表 The freelist

- freelist是一个可用内存的链表：A freelist is a linked list of free heap blocks:



- 从freelist分配：Allocation from freelist:
  - 搜索一个足够大的空闲块 Search for a large enough free block
  - 如果没有找到，进行垃圾回收 If none found, do garbage collection
  - 再次尝试搜索 Try the search again
  - 如果失败，则报内存溢出 If it fails, we are out of memory

引用计数 Reference counting with freelist

- 每个对象有一个`字段`，记录被引用的数量 Each object knows the number of references to it
- 从freelist分配对象后 Allocate objects from the freelist
- 如果执行`x = o` 语句; 的运行时系统 After assignment x=o; the runtime system
    - 增加`对象o`的计数 Increments the count of object o
    - 减少`x原来引用的对象`的计数 Decrements the count of x’s old reference (if any)
    - 如果`原对象`计数变为零，If that count becomes zero,
        - 把该对象`回收`，放在freelist上 put that object on the freelist
        - `递归`地递减`原对象`指向对象的计数 recursively decrement count of all objects it points to
- 优点 Good
    - 易于实现 Simple to implement
- 缺点 Bad
    - 引用计数字段在`每个对象`中占用一定空间 Reference count field takes space in every object
    - 引用计数`更新和检查`需要时间 Reference count updates and checks take time
    - `级联的递减`导致系统停顿 A cascade of decrements takes long time, gives long pause
    - 无法释放循环引用 Cannot deallocate cyclic structures

标记清除 Mark-sweep with freelist

- 从freelist分配对象 Allocate objects from the freelist
- GC阶段1：`标记阶段` GC phase 1: mark phase
    - 假设所有对象都`不可达(标记为白色对象)` Assume all objects are white to begin with
    - 查找所有从堆栈`可到达`的对象，标记为`黑色` Find all objects that are reachable from the stack, and color them black
- GC阶段2：`清除阶段` GC phase 2: sweep phase
    - 扫描`整个堆`，将所有剩下的`白色对象`回收掉，将`保留对象(黑色)`标记为白色 Scan entire heap, put all white objects on the freelist, and color black objects white
- 优点 Good
    - 实现简单 Rather simple to implement
- 缺点 Bad
    - `清除阶段`必须查看`整个堆`，即使是死对象; Sweep must look at entire heap, also dead objects; 当许多频繁分配的`小对象`时候，效率低下 inefficient when many small objects die young
    - 容易导致内存碎片 Risk of heap fragmentation

停止-拷贝 `S&C` C: Two-space stop and copy

- 将堆分为`to-space`和`from-space`两个部分 Divide heap into to-space and from-space
- 在`from-space` 中分配对象 Allocate objects in from-space
- 若在`from-space`分配不成功，移动所有`from-space`可到达的对象到 `to-space` When full, recursively move all reachable objects from from-space to the empty to-space
- 交换`from-space` `to-space` Swap (empty) from-space with to-space
- 优点 Good
    - 只需要扫描`活对象` Need only to look at live objects
    - 缓存友好，局部引用 Good reference locality and cache behavior
    - 整理内存，没有内存碎片 Compacts the live objects: no fragmentation

- 缺点 Bad
    - 需要`两倍`内存 Uses twice as much memory as maximal live object size
    - 在移动对象时需`更新`引用 Needs to update references when moving objects
    - 移动大对象`很慢`（例如，数组） Moving a large object (e.g. an array) is slow
    - 当堆接近满时`非常慢`（复制操作太多）Very slow (much copying) when heap is nearly full

多代垃圾回收D: Generational garbage collection

- 大`多数`对象`生命周期短` Observation: Most objects die young
- 将堆分成`年轻`和`老化`两种 Divide heap into young (nursery) and old generation
- 只在 `年轻`堆中分配对象 Allocate in young generation
- `年轻`堆满了后，将`没有被回收的对象 live object`移动到`老化`堆中。 When full, move live objects to old gen. (minor GC)
- 当`老化`堆满，执行`（主要）GC` When old gen. full, perform a (major) GC there
- `年轻`堆 `老化`堆 GC 算法不同
    - minor GC `S&C` 运行快
    - major GC `M&S`
- 优点 Good
    - 回收大量垃圾快Recovers much garbage fast
- 缺点 Bad
    - 可能遭受老一代的分裂（如果标记清除）May suffer fragmentation of old generation (if mark-sweep)
    - 需要对字段分配进行写入屏障测试：Needs a write barrier test on field assignments: 在赋值o.f = y后，o在old和y在young，After assignment o.f=y where o in old and y in young, 需要记住，y是活的need to remember that y is live

保守垃圾回收器 Conservative garbage collectors

- 堆栈上的值 `0xFFFFFFFA`是`int`还是`堆引用` ? Is 0xFFFFFFFA on the stack an int or a heap ref?
- 如果GC不知道，它必须采用保守机制 If the GC doesn’t know, it must be conservative :
    - 假设是对象的引用 Assume it could be a reference to an object
- 保守回收 C / C ++库存在 Conservative collectors exist as C/C++ libraries
- 优点 Good
    - 可以作为 C和C++ 程序库 Can be added to C and C++ programs as a library
    - 支持指针算术运算 Works even with pointer arithmetics
- 缺点 Bad
    - 不可预测的内存泄漏 Unpredictable memory leaks
    - 不能压缩：更新"引用" Cannot be compacting: updating a "reference" that is 可能出错 actually a customer number leads to madness

并发垃圾回收Concurrent garbage collection

- 在多CPU的机器上，让一个cpu运行GC In a multi-cpu machine, let one cpu run GC
- 复杂 Complicated
  - 分配对象时的竞争条件 Race conditions when allocating objects
  - 移动对象时的竞争条件 Race conditions when moving objects
- 通常在"GC安全"点挂起线程 Typically suspends threads at "GC safe" points
  - 可能降低并发性（因为一个线程可能需要很长时间才能达到安全点） May considerably reduce concurrency (because one thread may take long to reach a safe point)

在主流虚拟机中的GC GC in mainstream virtual machines

- Sun / Oracle Hotspot JVM（客户端+服务器）Sun/Oracle Hotspot JVM (client+server)
  - 三代 Three generations
  - 当`gen0`已满，将活对象移动到`gen1`When gen. 0 is full, move live objects to gen. 1
  - `gen1`使用双空间`停止和复制`GC; Gen. 1 uses two-space stop-and-copy GC; when objects get 当对象获取被移到 `gen2` old they are moved to gen. 2
  - `gen2` 使用标记清除与压缩 en. 2 uses mark-sweep with compaction
- IBM JVM（用于例如Websphere服务器）IBM JVM (used in e.g. Websphere server)
  - 高并发多代; 参见David Bacon的文章Highly concurrent generational; see David Bacon’s paper
- Microsoft .NET（桌面+服务器）Microsoft .NET (desktop+server)
  - `三代` 小堆+大堆 Three generation small-obj heap + large-obj heap
  - 当`gen0`已满，移至`gen1` When gen. 0 is full, move to gen. 1
  - 当`gen1`已满，移至`gen2` When gen. 1 is full, move to gen. 2
  - `gen2`使用`标记清除` Gen. 2 uses mark-sweep with occasional compaction
- Mono .NET实现Mono .NET implementation
  - Boehm的保守回收（2012年5月标准） Boehm’s conservative collector (still standard May 2012)
  - 新的二代（停止和复制加M-S或S&C） New two-generational (stop-and-copy plus M-S or S-&-C)

其他GC相关主题 Other GC-related topics

- `大对象空间`,大数组和其他长寿命对象可以单独存储 Large object space : Large arrays and other long-lived objects may be stored separately
- `弱引用` ：不保持对象活性的`引用` Weak reference: A reference that cannot itself keep an object live
- `Finalizer` 当对象被收集时（例如关闭文件）`执行的代码` Finalizer : Code that will be executed when an object dies and gets collected (e.g. close file)
  - Finalizer 可能重新`激活对象` Resurrection : A finalizer may make a dead object live again
- `固定` 当Java/C# 导出引用到C/C++代码，对象必须被固定，如果GC移动对象，会导致引用出错 Pinning : When Java/C# exports a reference to C/C++ code, the object must be pinned; if GC moves it, the reference will be wrong

GC stress (StringConcatSpeed.java)

- 哪个更好？What do these loops do? Which is better?

```
StringBuilder buf
   = new StringBuilder();
for (int i=0; i<n; i++)
   buf.append(ss[i]);
res = buf.toString();
```

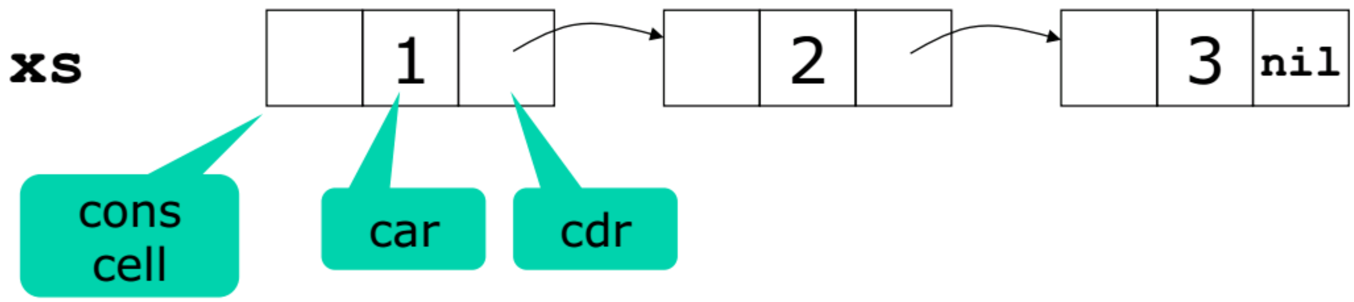
```
String res = "";
for (int i=0; i<n; i++)
   res += ss[i];
```

List-C 语言 New: List-C and the list machine

- list-c = micro-C with Lisp / Scheme

```
void main(int n) {
   dynamic xs;
   xs = nil;
   while (n>0) {
     xs = cons(n,xs);
     n = n - 1;
   }
   printlist(xs);
}
```

```
void printlist(dynamic xs) {
  while (xs) {
     print car(xs);
     xs = cdr(xs);
  }
}
```



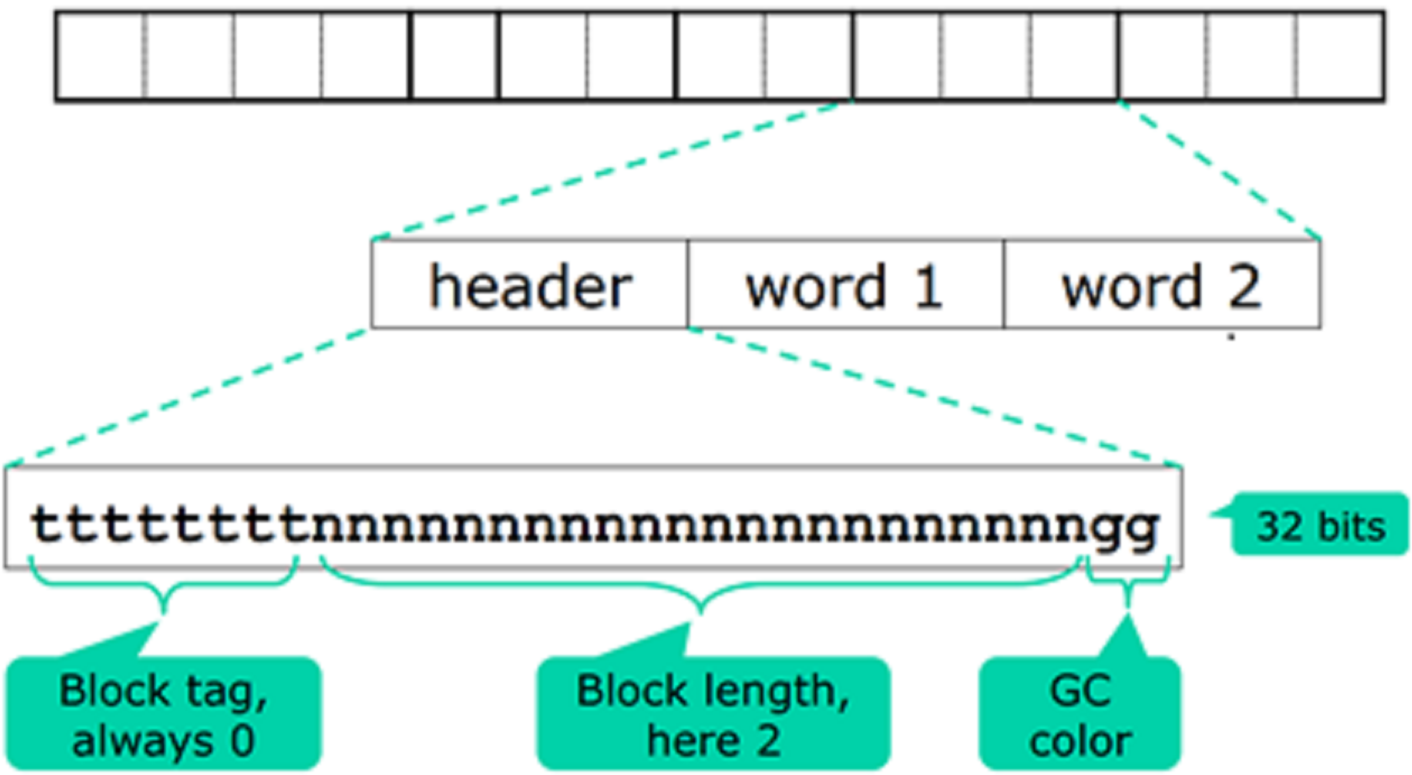
List machine instructions

- List机器= micro-C abstract machine + 加上六个指令：plus six extra instructions:
  - `NIL`：在堆栈上放置nil引用 NIL: Put nil reference on stack
  - `CONS`：在堆上分配两个块,在堆栈放对两个块引用 put CONS: Allocate two-word block on heap, put reference to it on stack
  - `CAR，CDR`：访问首块或尾块的值 CAR, CDR: Access word 1 or 2 of block
  - `SETCAR，SETCDR`：设置首块或尾块的值 SETCAR, SETCDR: Set word 1 or 2 of block

	Instr	St before		St after	Effect
26	NIL	s	$\Rightarrow$	s, nil	Load *nil* reference
27	CONS	s, v_1, v_2	$\Rightarrow$	s, p	Create cons cell $p \mapsto (v_1, v_2)$ in heap
28	CAR	s, p	$\Rightarrow$	s, v_1	Component 1 of $p \mapsto (v_1, v_2)$ in heap
29	CDR	s, p	$\Rightarrow$	s, v_2	Component 2 of $p \mapsto (v_1, v_2)$ in heap
30	SETCAR	s, p, v	$\Rightarrow$	s	Set component 1 of $p \mapsto$ _ in heap
31	SETCDR	s, p, v	$\Rightarrow$	s	Set component 2 of $p \mapsto$ _ in heap

堆的结构The structure of the list machine heap

- 堆由32位（4字节）字组成 The heap consists of 32-bit (4-byte) words
- 堆被块覆盖 The heap is covered by blocks



垃圾回收位gg Garbage collection bits gg

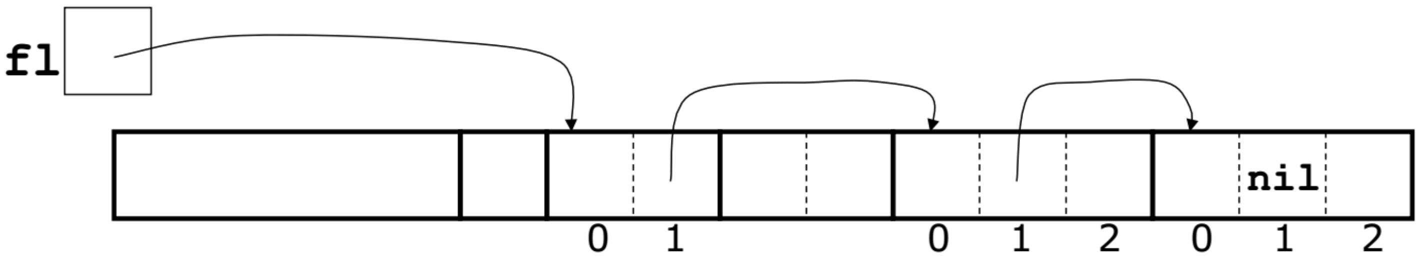
Bits	Color	Meaning
00	white	After mark phase: Not reachable可以回收

Bits	Color	Meaning
		from stack; may be collected
01	grey	During mark phase: Reachable, 可达对象
		referred-to blocks not yet marked
10	black	After mark phase: Reachable from 不可回收对象
		stack; cannot be collected
11	blue	On freelist, or is orphan block 已经回收对象

- 标记 ===> `涂黑`所有可达的块 The mark phase paints all reachable blocks black
- 清理 ===> `涂白`回收后剩下的黑色块; The sweep phase paints black blocks white;
- 空闲表中的块`涂蓝` freelist paints white blocks blue and puts them on freelist

空闲表 孤儿 The freelist; orphans

- freelist上的所有块都是蓝色的（gg = 11） All blocks on the freelist are blue (gg=11)
- 空闲表中，字1 包含对下一个freelist的引用 或nil： Word 1 contains a reference to the next freelist element, or nil:



- 长度为零的块是孤儿 A block of length zero is an orphan
- 它只包含一个头 It consists of a header only

区分整数和引用Distinguishing integers and references

- 需要区别整数和引用 For exact garbage collection we need to distinguish integers from references
- 办法：Old trick:
  - 使所有堆块从地址`倍数为4`开始; 二进制形式`xxxxxx00` Make all heap blocks begin on address that is a multiple of 4; in binary it has form xxxxxx00
  - 将整数`n`表示为2n + 1，因此为整数Represent integer n as 2n+1, so the integer’s 表示形式为`xxxxxxx1` representation has form xxxxxxx1
- 测试`IsInt（v）`： (v)&1==1 Test for IsInt(v): (v)&1==1
- 标记一个int：((v)<<1)|1 Tagging an int: ((v)<<1)|1
- 取消标记int： (v)>>1 Untagging an int: (v)>>1

示例list-C程序，ex30.lc An example list-C program, ex30.lc

- 每次迭代分配cell Each iteration allocates a cons cell that dies
- 没有垃圾回收器的程序 很快就耗尽了内存 Without a garbage collector the program soon runs out of memory

```
void main(int n) {
    dynamic xs;                  // Allocate cons cell    in heap
    while (n>0) {
        xs = cons(n, 22);       // Assignment causes previous xs value to die
        print car(xs);
        n = n - 1;
    }
}
```

- 任务：实现垃圾收集器：标记扫描，停止和复制 Your task in BOSC: Implement garbage collectors: mark-sweep, and stop-and-copy
- `listmachine.c`只在32位机器编译，试试改成支持64位？

Reading and homework

- 本周讲座：This week’s lecture:
  - PLC第9章和第10章 PLC chapters 9 and 10
  - Sun Microsystems：内存管理 Sun Microsystems: Memory Management in the Java Hotspot虚拟机 Java Hotspot Virtual Machine
  - David Bacon，IBM：实时垃圾回收 David Bacon, IBM: Realtime garbage collection
  - 练习9.1，练习9.2 练习9.3

可选练习9.3 Alternative exercise 9.3

```
class SentinelLockQueue implements Queue {
  private static class Node {
    final int item;
    volatile Node next;
    public Node(int item, Node next) {
      this.item = item;
      this.next = next;
    }
  }
  private final Node dummy = new Node(-444, null);
  private Node head = dummy, tail = dummy;
  public synchronized boolean put(int item) {
    Node node = new Node(item, null);
    tail.next = node;
    tail = node;
    return true;
  }
  public synchronized int get() {
    if (head.next == null)
      return -999;
    Node first = head;
    head = first.next;
    return head.item;
  }
}
```

- SentinelLockQueue 类包含内存管理问题 Class SentinelLockQueue contains a memory management problem
- 运行它，看看会发生什么 Run it and see what happens
- 找出问题是什么，解释一下，并修复 Find out what the problem is, explain it, and fix it
- 更正代码，使之运行没有错误 The corrected code should run to completion without error
- 源代码在QueueWithMistake.java Source code and more explanation is in file QueueWithMistake.java