Il bytecode Kitten

Fausto Spoto

23 febbraio 2005

Risultato della generazione del codice intermedio

```
load 1 int
iconst 1
if_cmple int
```

yes:

```
iconst 1
return int
```

no:

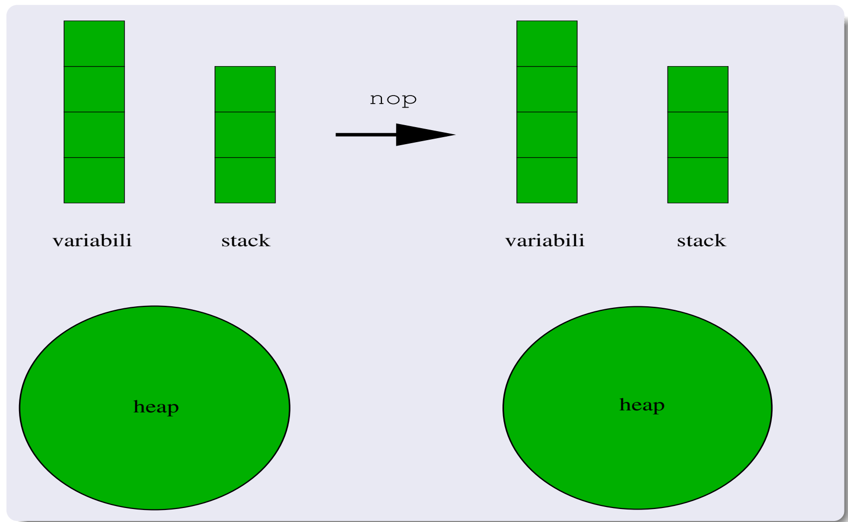
```
load 0 Fibonacci
load 1 int
iconst 1
sub int
virtualcall Fibonacci.fib(int)
load 0 Fibonacci
load 1 int
iconst 2
sub int
virtualcall Fibonacci.fib(int)
add int
return int
```

Il codice intermedio: bytecode Kitten

- Un grafo di blocchi di codice
- Ciascun blocco contiene una sequenza di bytecode Kitten
- Ci sono 51 bytecode Kitten
- Si tratta di una semplificazione strutturata del Java bytecode
- Il bytecode Kitten è eseguito da una Kitten Virtual Machine il cui stato è composto da
  1. un insieme di variabili locali $l_0, l_1 \ldots l_{max}$ che possono contenere valori primitivi o puntatori a oggetti
  2. uno stack di calcolo $s_{top}, s_{top-1} \ldots s_0$ che può contenere valori primitivi o puntatori a oggetti
  3. una memoria o heap che contiene oggetti
- Ciascun bytecode definisce una trasformazione di stato
- Per gestire la ricorsione, esiste anche uno stack di attivazione di coppie: $\langle$variabili locali,stack di calcolo$\rangle$

Il bytecode nop

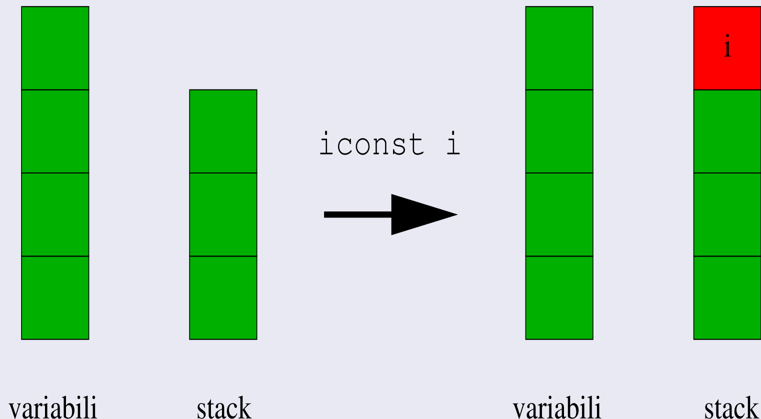
Indichiamo in verde ciò che non cambia



Il bytecode `iconst`

Non indichiamo più lo heap se non cambia

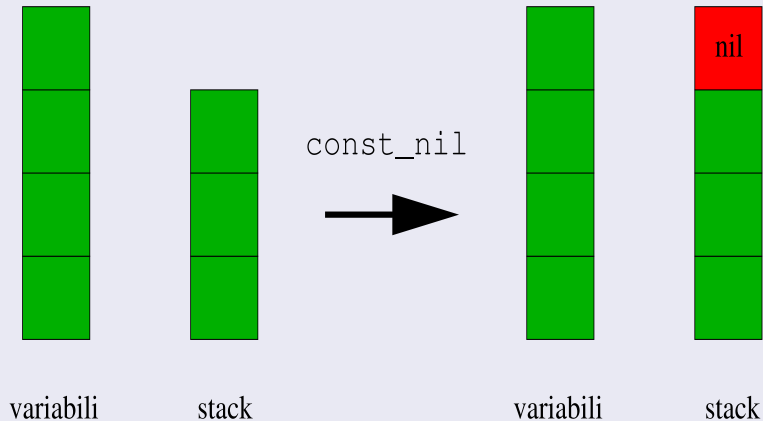
Carica una costante intera in cima allo stack



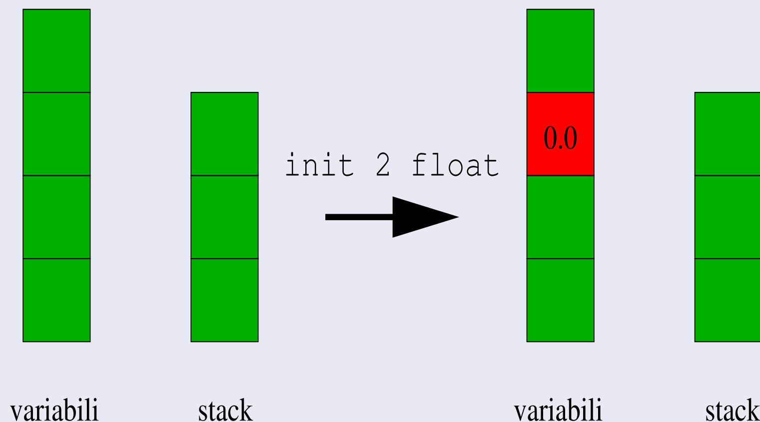
Esistono anche `bconst`, `cconst` ed `fconst`

Il bytecode const_nil

Carica nil in cima allo stack



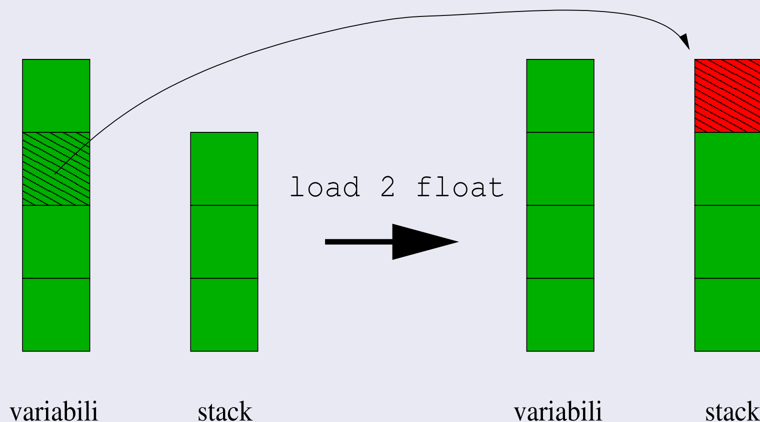
Il bytecode `init`



`init 2 float` significa: *inizializza la variabile 2 al valore di default per i float*

Il bytecode `load`

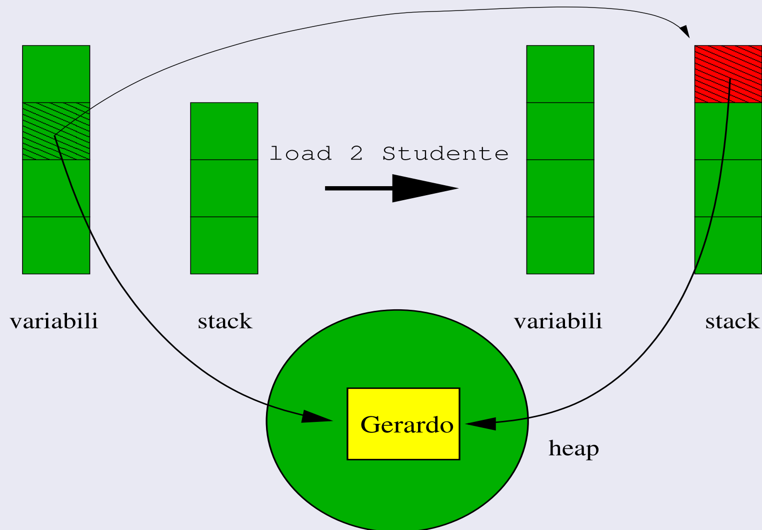
copia una variabile locale in cima allo stack



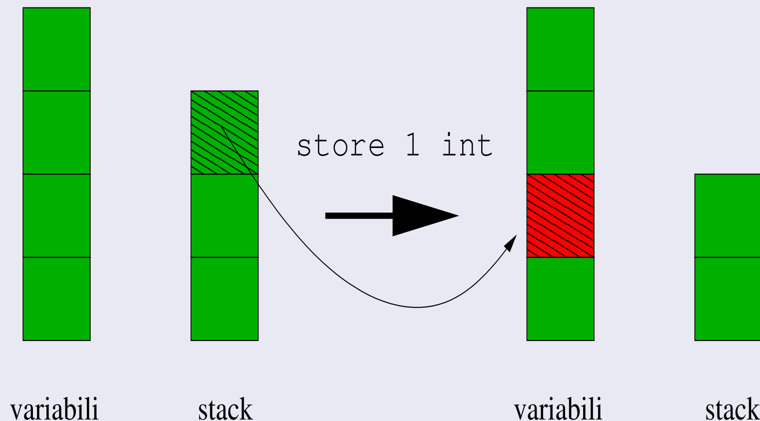
`load 2 float` significa: *copia in cima allo stack il valore float contenuto della variabile 2*

Il bytecode `load`

Copiando un tipo classe se ne copia il puntatore!



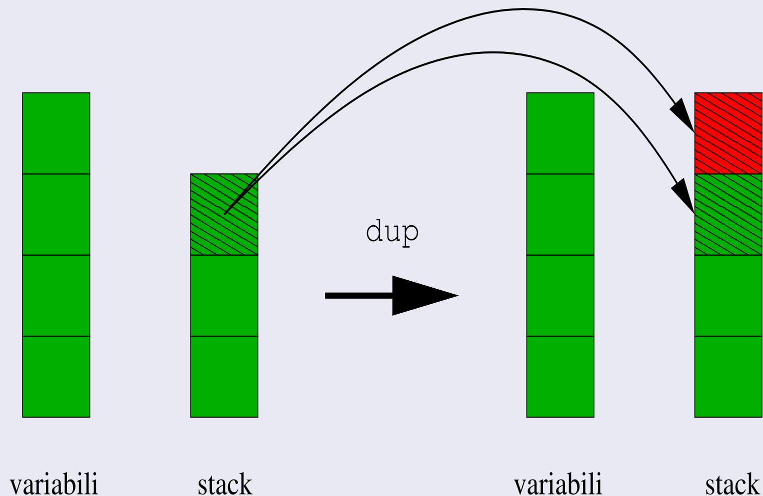
Il bytecode `store`



`store 1 int` significa: *muovi nella variabile 1 il valore intero che sta in cima allo stack*

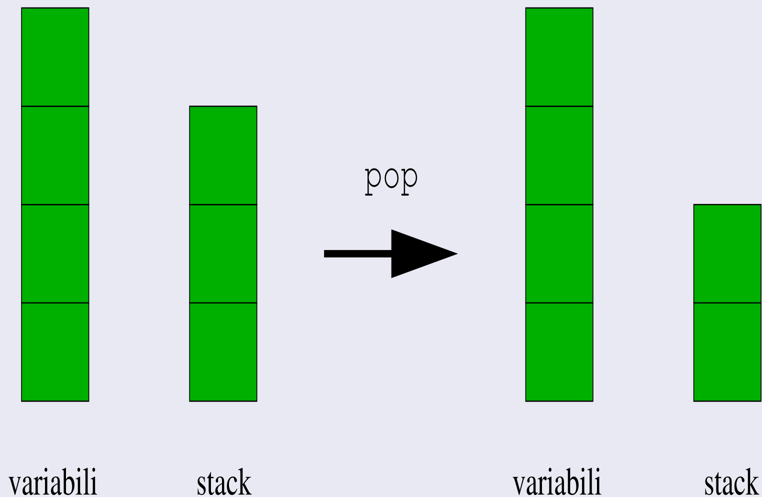
Il bytecode dup

duplica la cima dello stack



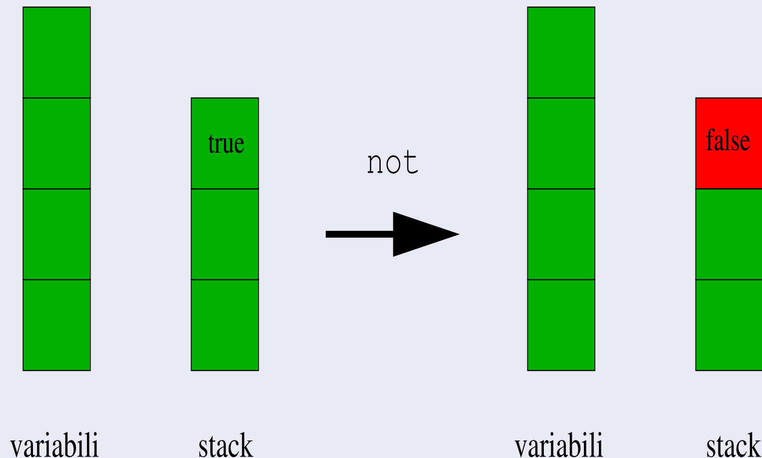
Il bytecode pop

rimuove la cima dello stack



Il bytecode `not`

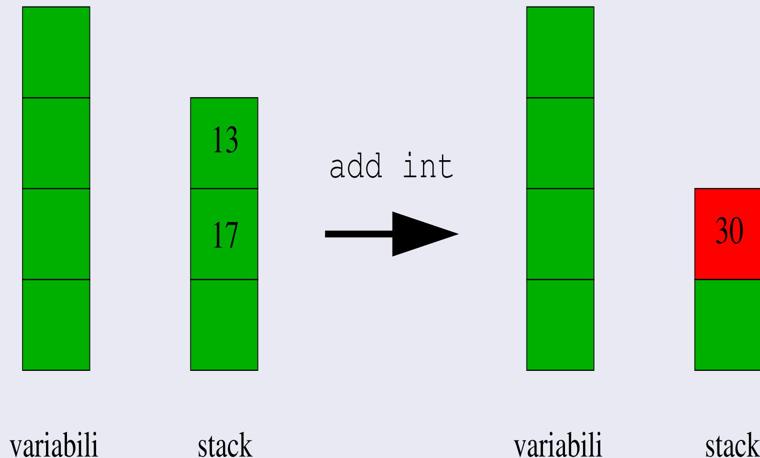
nega il booleano che sta in cima allo stack



Esiste anche `neg` che nega un intero

Il bytecode `add`

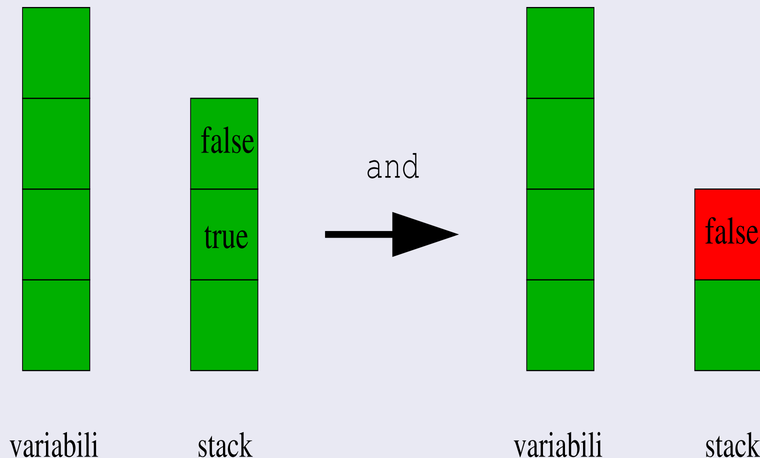
Somma i due interi che stanno in cima allo stack



Esistono anche `sub`, `mul` e `div`

Il bytecode and

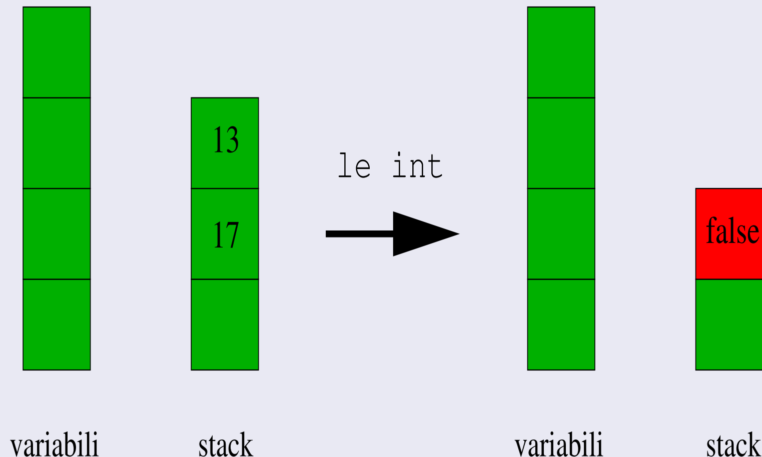
Carica sullo stack il risultato dell'and logico fra i due booleani in cima allo stack



Esiste anche or

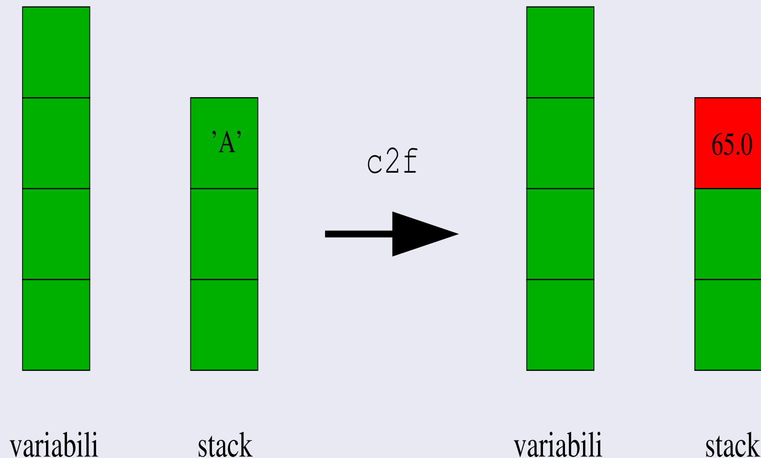
Il bytecode `le`

Carica sullo stack il risultato booleano del confronto ($\leq$) fra i due elementi che stanno in cima allo stack



Esistono anche `eq` (che confronta anche puntatori), `ge` ed `lt`

Il bytecode c2f



Esistono anche c2i, f2c, f2i, i2c ed i2f

Esempi

4*4

```
iconst 4
iconst 4
mul int
```

4*4

```
iconst 4
dup
mul int
```

3+(4*4)

```
iconst 3
iconst 4
iconst 4
mul int
add int
```

3+(4*4)>5

```
iconst 3
iconst 4
iconst 4
mul int
add int
iconst 5
gt int
```

incrementa l_2 (float)

```
load 2 float
fconst 1.0
add float
store 2 float
```

Esempi

$l_2 + 5 < l_1$ (float)

```
load 2 float
iconst 5
i2f
add float
load 1 float
le float
```

scambia l_1 ed l_2 (int)

```
load 1 int
load 2 int
store 1 int
store 2 int
```

l_1 diventa ($l_1 = l_2$)

```
load 1 int
load 2 int
eq int
store 1 boolean
```

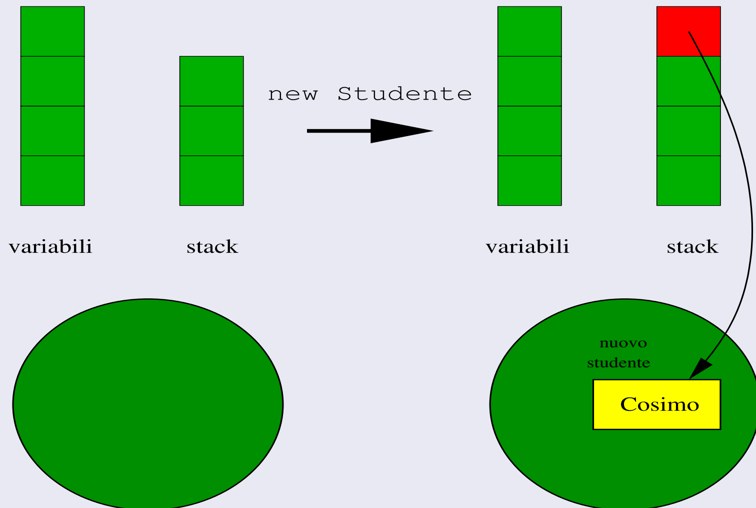
Tipo di una variabile?

Una variabile locale può contenere valori di tipo diverso in tempi diversi!

Comunque non sfrutteremo questa possibilità

Il bytecode `new`

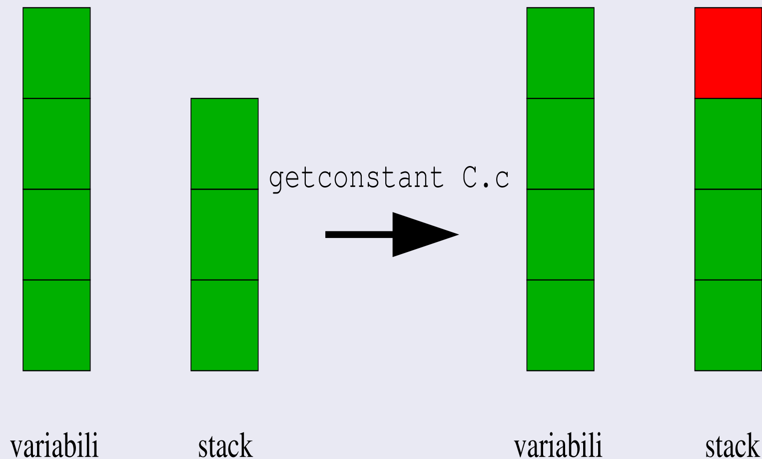
Carica in cima allo stack un puntatore a un nuovo oggetto



Non chiama alcun costruttore! Esiste anche `newstring`

Il bytecode getconstant

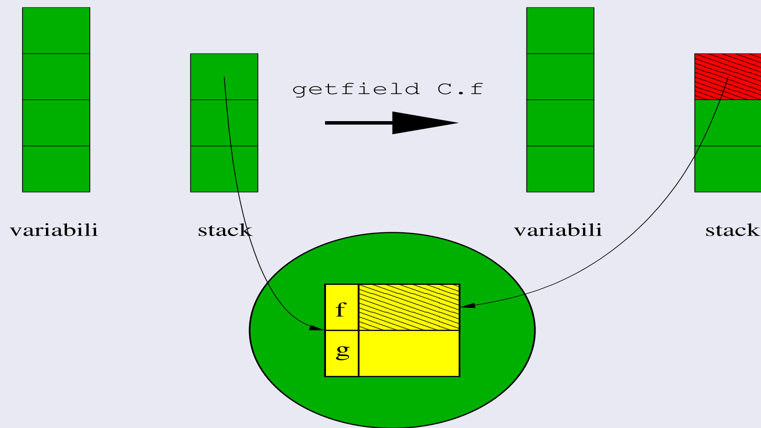
Carica in cima allo stack una costante di una classe



C.c è una ConstantEntry che identifica la costante c della classe C

Il bytecode `getfield`

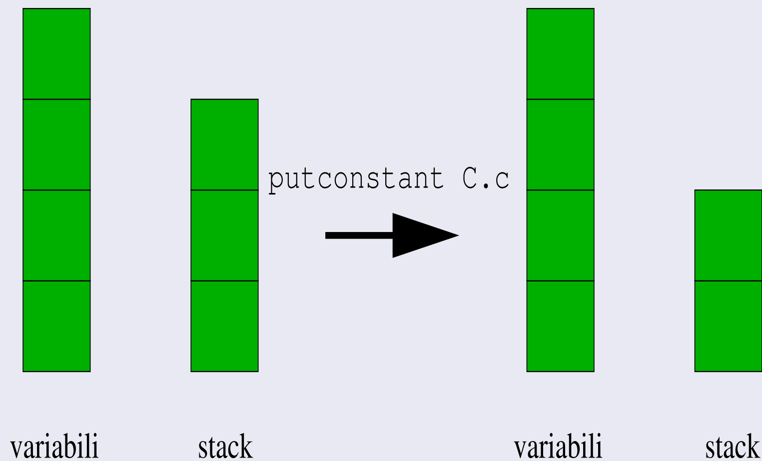
Carica in cima allo stack un campo di una classe



- `C.f` è una `FieldEntry` che identifica il campo `f` della classe `C`
- la cima dello stack non deve essere `nil`

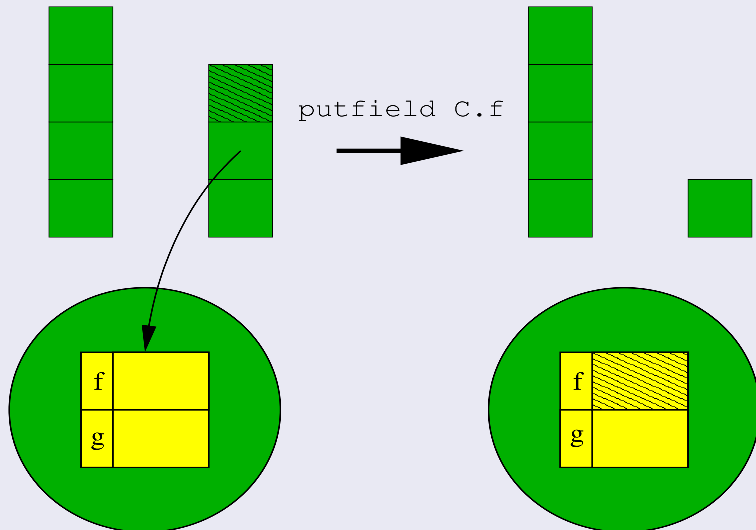
Il bytecode `putconstant`

Copia l'elemento in cima allo stack dentro una costante



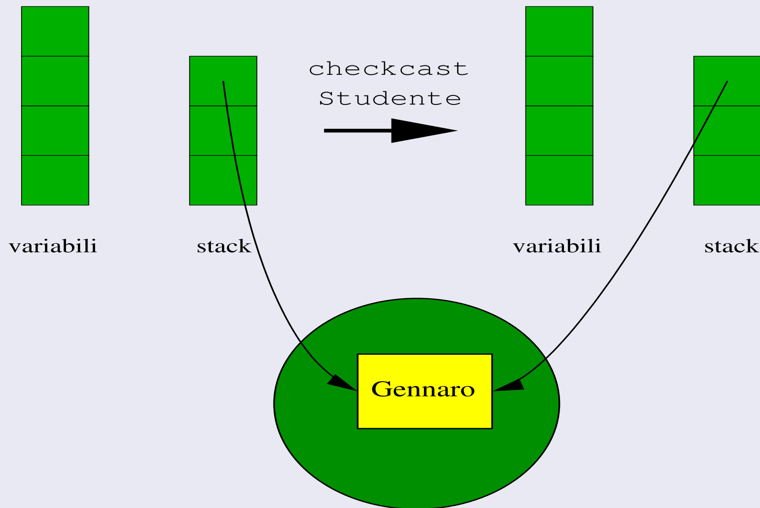
Il bytecode putfield

Copia la cima dello stack dentro un campo di un oggetto



Il bytecode checkcast

Controlla se l'oggetto in cima allo stack è del tipo specificato



Se il controllo fallisce, la computazione si ferma

Esempi

Crea uno studente e ne legge la matricola

```
new Studente
<chiama un costruttore>
getfield matricola
```

Incrementa l'età dello studente in l_1

```
load 1 Studente
load 1 Studente
getfield Studente.età
iconst 1
add int
putfield Studente.età
```

Copia l'età di l_1 nell'età di l_2

```
load 2 Studente
load 1 Studente
getfield Studente.età
putfield Studente.età
```

Codice che si blocca in fase di esecuzione

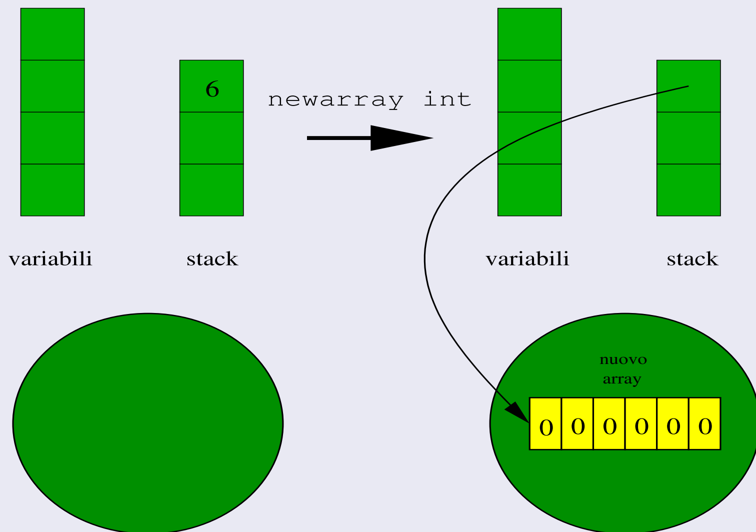
```
new Persona
<chiama un costruttore>
checkcast Studente
```

Codice che si blocca in fase di esecuzione

```
const_nil
getfield Studente.età
```

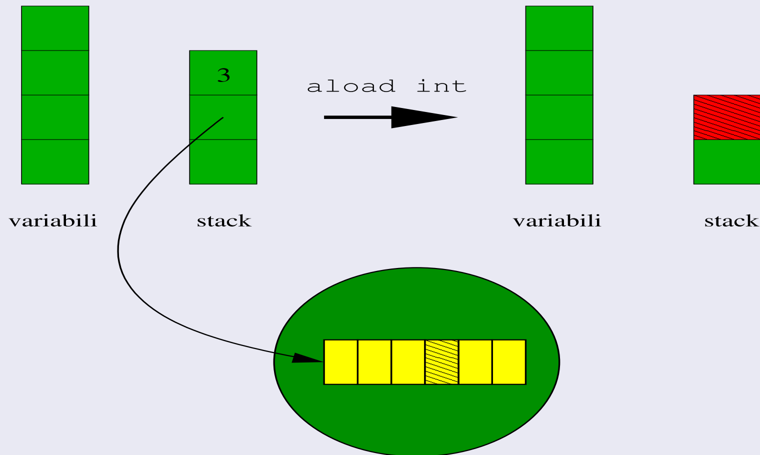
Il bytecode `newarray`

Crea un array del tipo specificato



Il bytecode `aload`

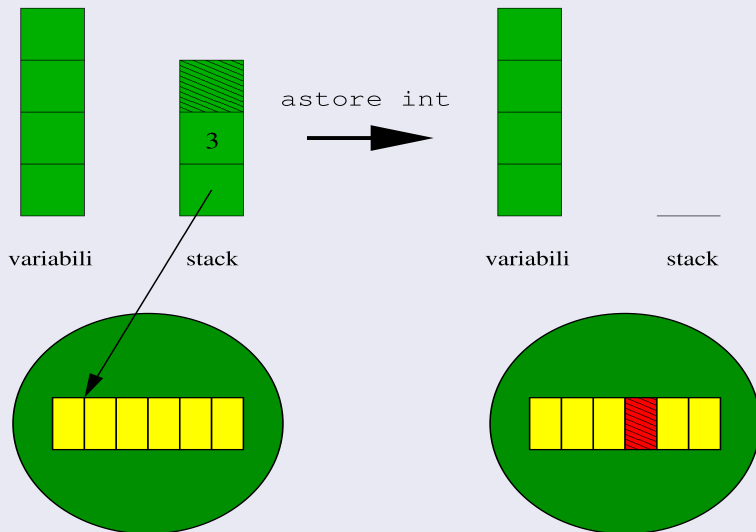
Carica sullo stack un elemento di un array



- `aload` *t* lavora su un `array of` *t*
- se l'indice non è valido, l'esecuzione si blocca

Il bytecode `astore`

Copia la cima dello stack dentro un elemento di un array



Esempi

Crea un array di 3 float e lo inizializza

```
iconst 3
newarray float
dup
iconst 0
fconst 3.1415
astore float
dup
iconst 1
fconst 3.1415
astore float
dup
iconst 2
fconst 3.1415
astore float
```

Legge l_2[1] (float)

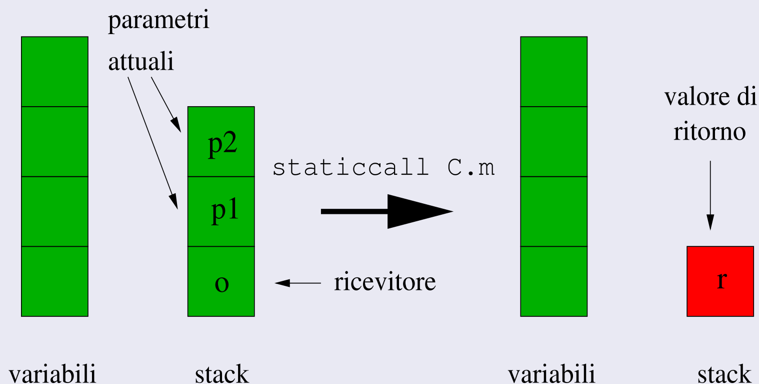
```
load 2 array of float
iconst 1
aload float
```

Incrementa l_2[1] (float)

```
load 2 array of float
iconst 1
load 2 array of float
iconst 1
aload float
fconst 1.0
add float
astore float
```

Il bytecode staticcall

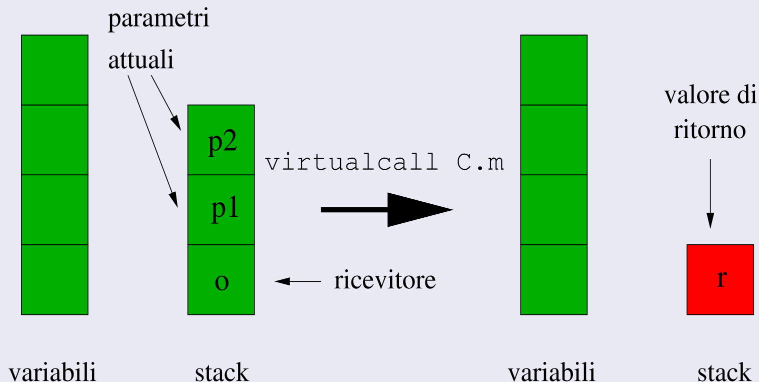
Esegue il metodo specificato



- `C.m` è una `CodeEntry` che specifica il metodo chiamato
- `r` è il suo risultato (non esiste se `C.m` ritorna `void`)
- Si noti che le variabili locali non vengono modificate

Il bytecode `virtualcall`

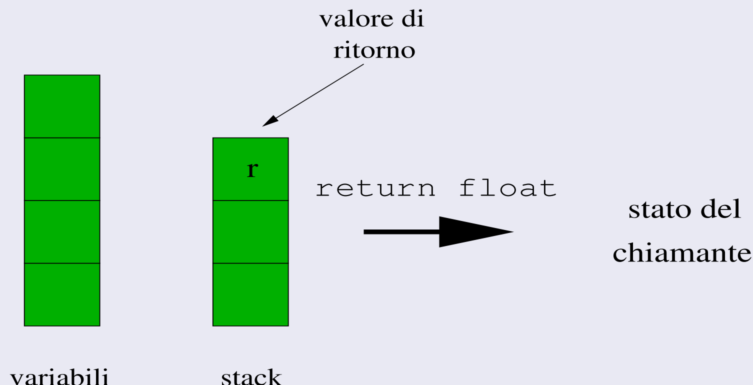
Esegue il metodo selezionato



- `C.m` dà la segnatura del metodo da chiamare
- Il metodo chiamato è selezionato cercando un metodo compatibile con `C.m` nella classe di *o*, oppure nella sua superclasse, oppure nella sua super-superclasse...

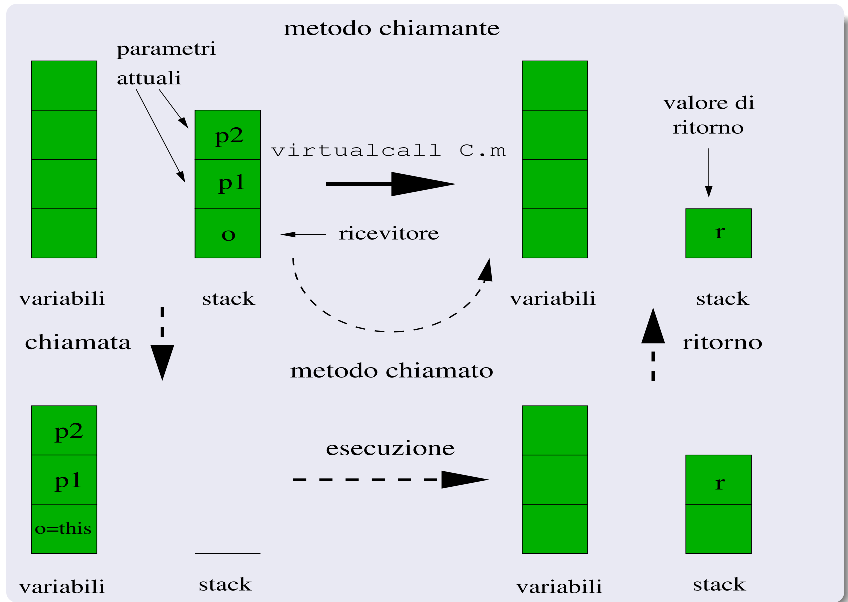
Il bytecode `return`

Ritorna al metodo chiamante



- Nel caso di `return void`, il valore *r* non esiste
- Il valore *r* può non essere l'unico elemento dello stack

Interazione chiamante/chiamato



Esempi

Chiamata a un costruttore

```
new Studente
dup
staticcall Studente.<init>()
```

Chiamata a un costruttore con parametri

```
new Studente
dup
newstring “vr012345”    // matricola
iconst 45               // età
staticcall Studente.<init>(String,int)
```

Esempi

Un costruttore per `Studente`

```
Studente.<init>():
return
```

Un altro costruttore per `Studente`

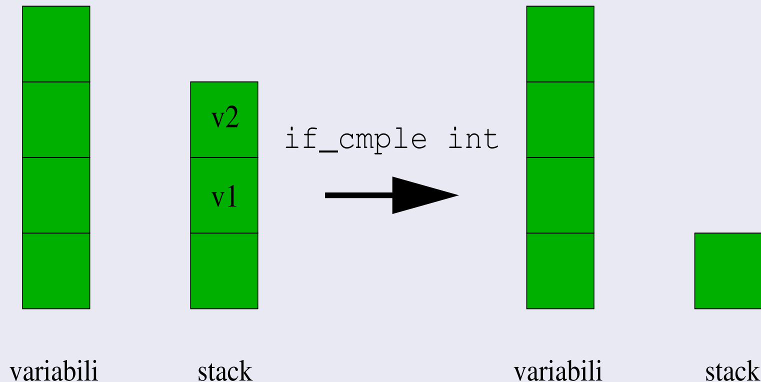
```
Studente.<init>(String,int):
load 0 Studente // carica this
load 1 String   // carica il primo parametro
putfield Studente.matricola
load 0 Studente // carica this
load 1 int      // carica il secondo parametro
putfield Studente.età
return
```

Blocchi di codice

- i bytecode precedenti vengono inseriti dentro dei blocchi detti basic blocks
- un blocco può essere legato ad altri blocchi da archi orientati, formando quindi un grafo di blocchi
  0. blocco finale è un blocco senza archi uscenti. Il suo ultimo bytecode è un `return`
  1. blocco linkato è un blocco che ha un unico arco uscente. Il suo ultimo bytecode non è condizionale né un `return`
  2. blocco condizionale è un blocco con due archi uscenti. Il suo ultimo bytecode è un bytecode condizionale
- uno dei blocchi è etichettato come iniziale

I bytecode condizionali

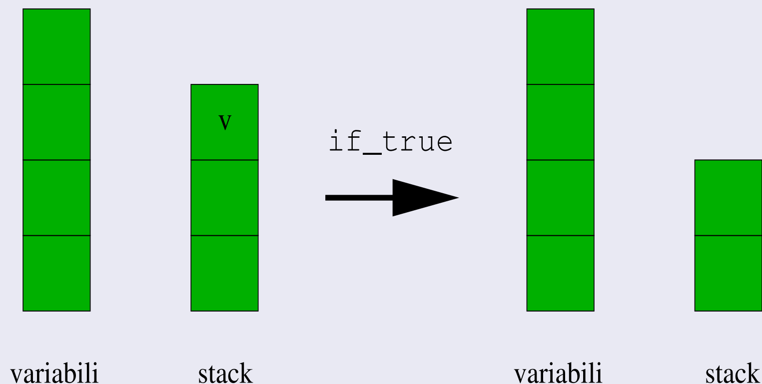
Il bytecode `if_cmple`



- va verso la diramazione yes se $v_1 \leq v_2$
- va verso la diramazione no se $v_1 > v_2$
- esistono anche `if_cmplt`, `if_cmpge`, `if_cmpgt`, `if_cmpeq` ed `if_cmpne`

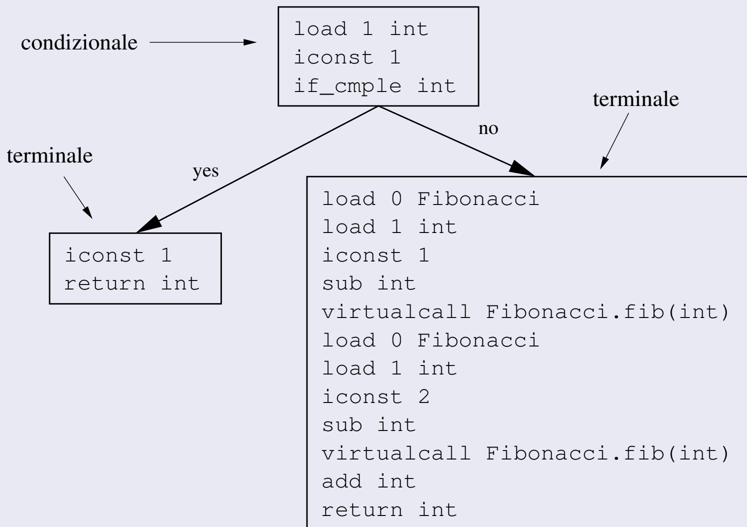
I bytecode condizionali

Il bytecode `if_true`

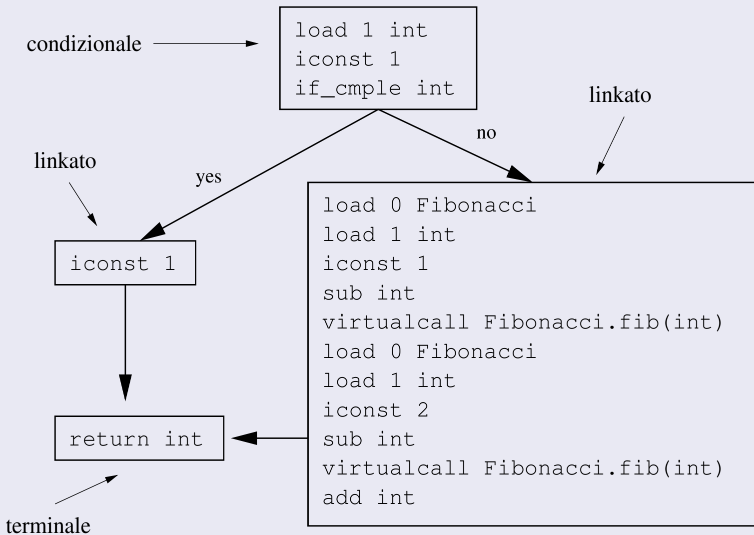


- va verso la diramazione yes se $v =$ `true`
- va verso la diramazione no se $v =$ `false`

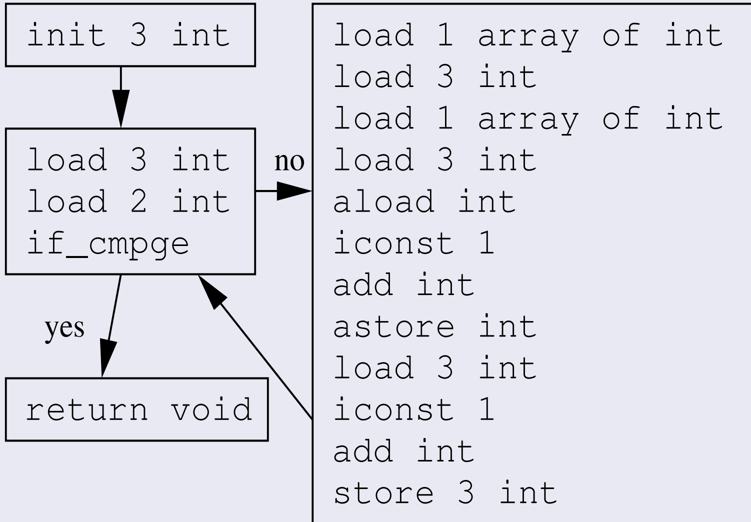
Esempio: Fibonacci



Oppure:



Esempio: incremento di tutti gli elementi di un array



Esempio: somma degli elementi di una lista di interi

```
int Sum.sum(Lista lista)
```

```
load 1 Lista
const_nil
if_cmpeq Lista
```

yes:

```
iconst 0
return int
```

no:

```
load 1 Lista
virtualcall Lista.getHead()
load 0 Sum
load 1 Lista
virtualcall Lista.getTail()
virtualcall Sum.sum(Lista)
add int
return int
```

Esercizi

Scrivere in bytecode Kitten:

1. un metodo che calcola 2^n
2. un metodo che calcola n^n
3. un metodo che calcola $n^n + n$
4. un metodo che incrementa di 1 i valori di una lista
5. un metodo che somma a ogni elemento di una lista gli elementi che lo seguono
6. un metodo che calcola la somma degli elementi di un array di array di dimensione $n \times n$

La classe Bytecode/Bytecode.java

I bytecode sono organizzati in una lista

```
public abstract class Bytecode {
 private Bytecode next, last;
 protected Bytecode(Bytecode next) {
  this.next = next;
  if (next != null) this.last = next.last;
  else this.last = this;
 }
 public Bytecode append(Bytecode other) {
  last.next = other;
  last = other.last;
  return this;
 }
}
```

Esempi di bytecode

Bytecode/NOP.java

```
public class NOP extends Bytecode {
 public NOP(Bytecode next) { super(next); }
 public String toString() { return “nop”; }
}
```

Bytecode/LOAD.java

```
public class LOAD extends Bytecode {
 private int varNum;
 private Type type;
 public LOAD(int varNum,Type type,Bytecode next) {
  super(next);
  this.varNum = varNum;
  this.type = type; }
 public String toString() {
  return “load ” + varNum + type; }
}
```

I basic block

```
public abstract class CodeBlock {
 protected Bytecode bytecode;
 protected CodeBlock(Bytecode bytecode) {
  this.bytecode = bytecode; }
}
```

```
public class LinkedCodeBlock extends CodeBlock {
 private CodeBlock follow;
 public LinkedCodeBlock
    (Bytecode bytecode, CodeBlock follow) {
  super(bytecode);
  this.follow = follow; }
 public void linkTo(CodeBlock follow) {
  this.follow = follow; }
}
```

```
public class ConditionalCodeBlock
    extends CodeBlock {
 private ConditionalBytecode condition;
 private CodeBlock yes, no;
 public ConditionalCodeBlock
  (Bytecode bytecode,
   ConditionalBytecode condition,
   CodeBlock yes, CodeBlock no) {
  super(bytecode); this.condition = condition;
  this.yes = yes; this.no = no; }
}
```

```
public class FinalCodeBlock extends CodeBlock {
 public FinalCodeBlock(Bytecode bytecode) {
  super(bytecode); }
}
```