LEZIONI 13-14
BINDING IN JAVA

Ingegneria del Software e Progettazione Web

Università degli Studi di Tor Vergata - Roma

Guglielmo De Angelis
guglielmo.deangelis@isti.cnr.it

linguaggio java

<< … un linguaggio semplice e familiare, orientato agli oggetti, robusto, sicuro, architettura neutrale, portabile, ad alte prestazioni, interpretato, multithreaded e dinamico … >>

[J. Gosling e H. McGilton, “The Java Language Environment. A White Paper.” Maggio 1996]

linguaggio java

<< … un linguaggio semplice e familiare, orientato agli oggetti, robusto, sicuro, architettura neutrale, portabile, ad alte prestazioni, interpretato, multithreaded e ***dinamico*** … >>

[J. Gosling e H. McGilton, "The Java Language Environment. A White Paper." Maggio 1996]

sono molteplici gli aspetti che caratterizzano JAVA come un linguaggio DINAMICO

linguaggio java

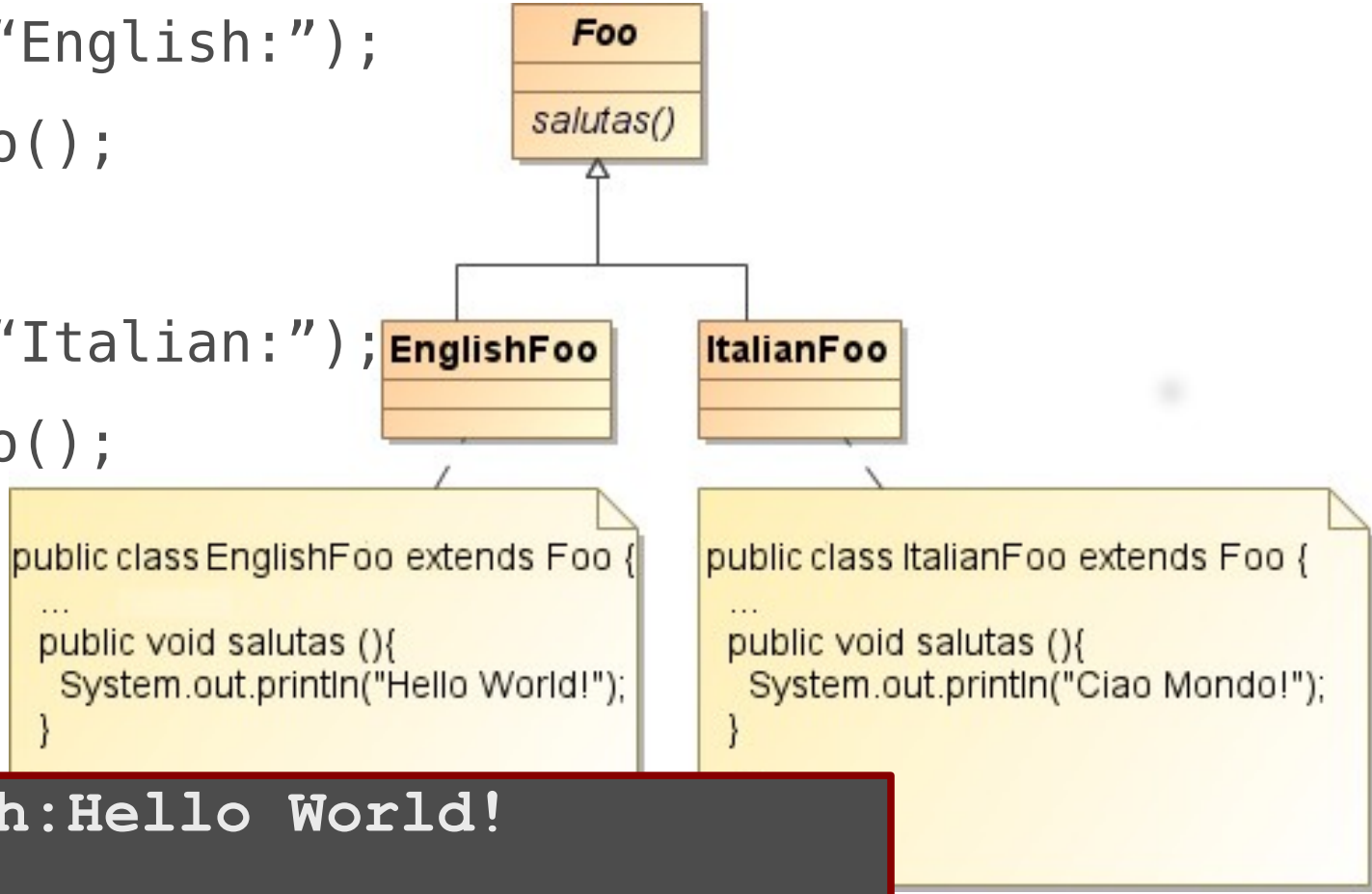
<< … un linguaggio semplice e familiare, orientato agli oggetti, robusto, sicuro, architettura neutrale, portabile, ad alte prestazioni, interpretato, multithreaded e ***dinamico*** … >>

[J. Gosling e H. McGilton, "The Java Language Environment. A White Paper." Maggio 1996]

- C++ soffre del "***constant recompilation problem***"
  - le funzioni/attributi/costanti sono linkati con specifici indirizzi nei file compilati, e non da nomi simbolici
  - ricompilare anche le classi che riferiscono una classe modificata
- C++ soffre del "***fragile base-class problem***" (come definito nel Java White Paper)
  - *"Any time you add a new method or a new instance variable to a class, any and all classes that reference that class will require a recompilation, or they'll break"*
- Il compilatore Java usa riferimenti simbolici e non riferimenti numerici
- L'interprete Java risolve i nomi quando le classi sono linkate

generalizzazione e polimorfismo

```
Foo f;
if (<test>){
  System.out.print(“English:”);
  f = new EnglishFoo();
} else {
  System.out.print(“Italian:”);
  f = new ItalianFoo();
}
f.salutas();
```




```
<test>   : English:Hello World!
!<test>  : Italian:Ciao Mondo!
```

binding a metodi

- binding: l'associazione tra l'invocazione di una operazione con l'effettivo metodo da eseguire
- early binding: effettuato a tempo di compilazione/linking prima dell'esecuzione del programma
  - static binding (simbolico in JAVA, vedi slide 4)
- late binding: l'associazione viene effettuata a tempo di esecuzione basandosi sull'effettivo tipo dell'oggetto coinvolto nell'esecuzione
  - dynamic binding
  - runtime binding

binding a metodi

- nei linguaggi che supportano il binding dinamico
  - il compilatore non può (o meglio non vuole) inferire l'esatto tipo degli oggetti
  - si utilizzano meccanismi per recuperare il tipo degli oggetti a run-time ed invocare il metodo corretto
- non esiste una strategia univoca per implementare binding dinamico
  - in generale, una parte delle informazioni sul tipo di dato sono incluse nella rappresentazione interna degli oggetti

… in Java

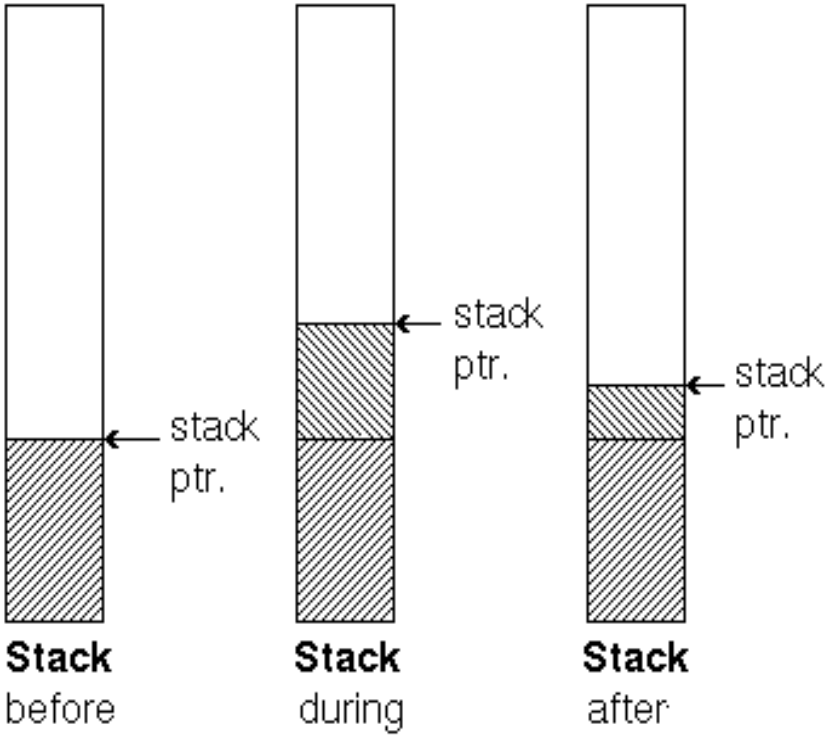
- le invocazioni di operazioni vengono associate ai rispettivi metodi per mezzo di *late binding*
- (principalmente) fanno eccezione i metodi dichiarati:
  - static
  - final
  - private

… in Java

- le invocazioni di operazioni vengono associate ai rispettivi metodi per mezzo di late binding
- (principalmente) fanno eccezione i metodi dichiarati:
  - static
    - hanno ambito di classe
    - il riferimento al metodo è sempre noto a tempo di compilazione
  - final
    - metodi che non possono essere ulteriormente specializzati
    - è impedito l'overriding sul metodo
    - si sta dichiarando esplicitamente al compilatore di disabilitare il binding dinamico
  - private
    - metodi ad uso esclusivo della classe che li dichiara
    - il contesto di accesso a compile-time coincide sempre quello a run-time
    - in Java i metodi privati sono implicitamente final

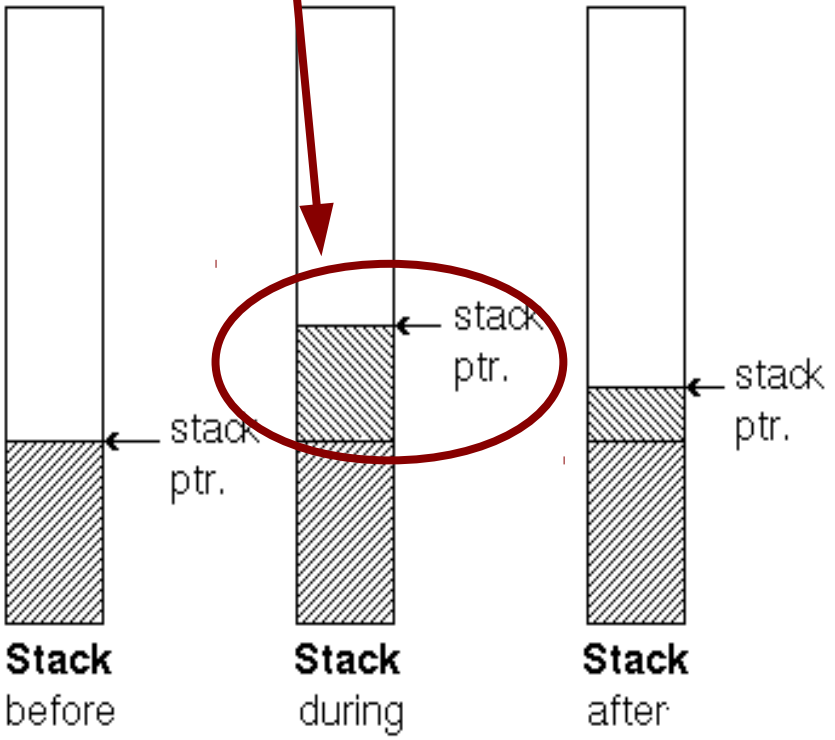
PARENTESI : STACK di Programma

- nei linguaggi di programmazione: area di memoria utilizza per mantenere lo stato locale ad una computazione
- a seguito della chiamata di una sotto-funzione, viene allocata una nuova area in STACK dove vengono (principalmente) salvati: i parametri formali (inizializzati ai valori dei parametri attuali) , le variabili locali, il valore di ritorno computato dalla sotto-funzione
  - stack PRIMA della chiamata a sotto-funzione
  - stack DURANTE della chiamata a sotto-funzione
  - stack DOPO la chiamata a sotto-funzione



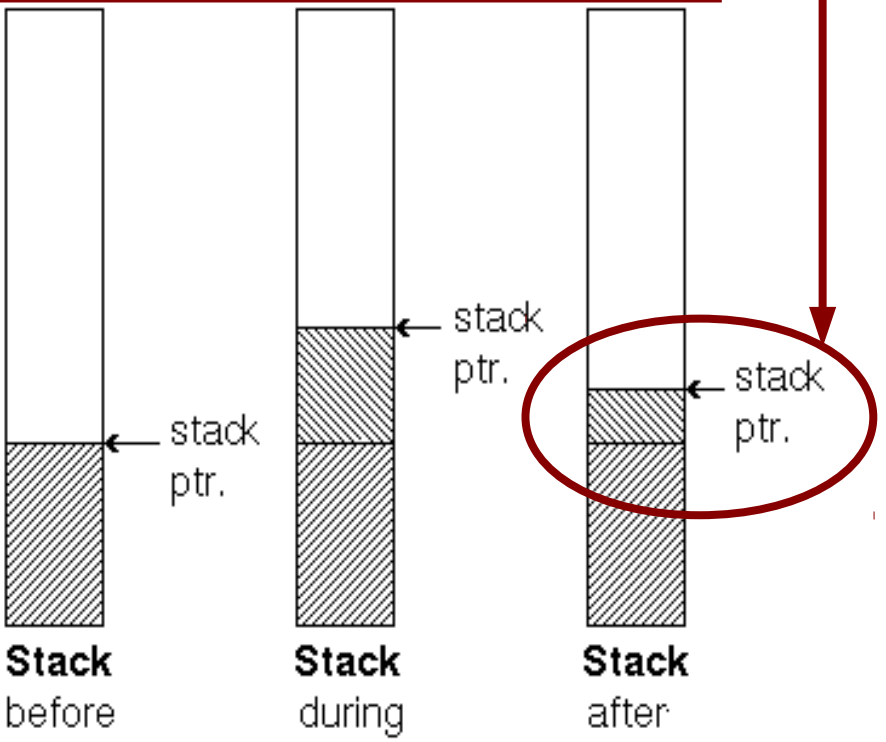
PARENTESI : STACK di Programma

- nei linguaggi di programmazione: area di memoria utilizza per mantenere lo stato locale ad una computazione
- a seguito della chiamata di una sotto-funzione, viene allocata una nuova area in STACK dove vengono (principalmente) salvati: i parametri formali (inizializzati ai valori dei parametri attuali) , le variabili locali, il valore di ritorno computato dalla sotto-funzione
  - stack PRIMA della chiamata a sotto-funzione
  - stack DURANTE della chiamata a sotto-funzione
  - stack DOPO la chiamata a sotto-funzione



PARENTESI : STACK di Programma

- nei linguaggi di programmazione: area di memoria utilizza per mantenere lo stato locale ad una computazione
- a seguito della chiamata di una sotto-funzione, viene allocata una nuova area in STACK dove vengono (principalmente) salvati: i parametri formali (inizializzati ai valori dei parametri attuali) , le variabili locali, il valore di ritorno computato dalla sotto-funzione
  - stack PRIMA della chiamata a sotto-funzione
  - stack DURANTE della chiamata a sotto-funzione
  - stack DOPO la chiamata a sotto-funzione



PARENTESI : HEAP di Programma

- nei linguaggi di programmazione: area di memoria utilizza per lo più per mantenere dati di una applicazione.
- le principali caratteristiche sono
  - i dati sono globalmente accessibili per mezzo di un puntatore di riferimento (e.g. memorizzato in una area STACK)
  - la gestione di questa area di memoria è principalmente delegata al programmatore
    - l'allocazione della memoria è esplicita (i.e. in Java attraverso l'operatore "`new`")
    - i dati contenuti in HEAP vi permangono finché:
      - non ci sia una esplicita deallocazione della memoria (i.e. come nel caso del C o C++ con gli operatori "`free`"/"`dispose`")
      - non venga avviata una politica implicita di deallocazione della memoria (i.e. esecuzione del garbage collector in Java)
      - … banalmente : il programma termina la sua esecuzione

JVM sotto il cofano … – 1

- JVM alloca e gestisce diversi tipologie di aree di memoria. In particolare:
  - STACK di programma
    - uno per ogni thread in esecuzione sulla JVM
    - contiene variabili locali, risultati parziali sull'esecuzione di un metodo, e riferimento di ritorno al metodo invocante
  - HEAP di programma
    - unico e condiviso da tutti i thread in esecuzione sulla JVM
    - è l'area di memoria a run-time dove sono allocate le istanze delle classi e gli array di dati (i.e. memoria dinamica)
- per ogni classe instanziata a run-time la JVM mantiene un "*run-time constant pool*"
  - costanti, riferimenti a campi/metodi definiti a tempo di compilazione, riferimenti da definire a run-time
- JVM method area
  - zona di memoria condivisa da tutti i thread
  - per ogni classe mantiene il *run-time constant pool*, il codice di metodi e costruttori, dati relativi a campi e metodi, etc.
  - l'indicazione generale della specifica è di includerlo in HEAP

JVM sotto il cofano … – 2

- (principalmente) la JVM implementa l'invocazione di metodi attraverso le istruzioni
  - `invokevirtual`
  - `invokestatic`
  - `invokespecial`

JVM sotto il cofano … – 2

- (principalmente) la JVM implementa l'invocazione di metodi attraverso le istruzioni
  - `invokevirtual`

JVM sotto il cofano … – 2.1

- (principalmente) la JVM implementa l'invocazione di metodi attraverso le istruzioni
  - `invokevirtual` : viene consultato il *run-time constant pool* accedendo in base al tipo del corrente oggetto (sullo STACK), ed usando parametri attuali

```
Object x;
…
x.equals("hello");
```

configurazione dello STACK (vedi slide successive)

```
aload_1
ldc "hello"

invokevirtual java/lang/Object/equals(Ljava/lang/Object;)Z
```

  - `invokestatic`
  - `invokespecial`

JVM sotto il cofano … – 2.1

- (principalmente) la JVM implementa l'invocazione di metodi attraverso le istruzioni
  - `invokevirtual` : viene consultato il *run-time constant pool* accedendo in base al tipo del corrente oggetto (sullo STACK), ed usando parametri attuali

```
Object x;
…
x.equals("hello");
```

configurazione dello STACK (vedi slide successive)

```
aload_1
ldc "hello"

invokevirtual java/lang/Object/equals(Ljava/lang/Object;)Z
```

allocazione dell'oggetto `x` sullo STACK

allocazione del parametro attuale sullo STACK

  - `invokestatic`
  - `invokespecial`

JVM sotto il cofano … – 2.1

- (principalmente) la JVM implementa l'invocazione di metodi attraverso le istruzioni
  - `invokevirtual` : viene consultato il *run-time constant pool* accedendo in base al tipo del corrente oggetto (sullo STACK), ed usando parametri attuali

```
Object x;
…
x.equals("hello");
```

se l'oggetto riferito da `x` è stato soggetto a override su `equals`, allora il metodo eseguito è quello dato dal tipo effettivo dell'istanza nello STACK, non quello di `Object`

configurazione dello STACK (vedi slide successive)

```
aload_1
ldc "hello"

invokevirtual java/lang/Object/equals(Ljava/lang/Object;)Z
```

allocazione dell'oggetto `x` sullo STACK

allocazione del parametro attuale sullo STACK

il risultato dell'invocazione è messo sullo STACK

  - `invokestatic`
  - `invokespecial`

JVM sotto il cofano … – 2.2

- (principalmente) la JVM implementa l'invocazione di metodi attraverso le istruzioni
  - `invokevirtual`
  - `invokestatic` : viene consultato il run-time constant pool accedendo in base alla classe referenziata, ed usando parametri attuali

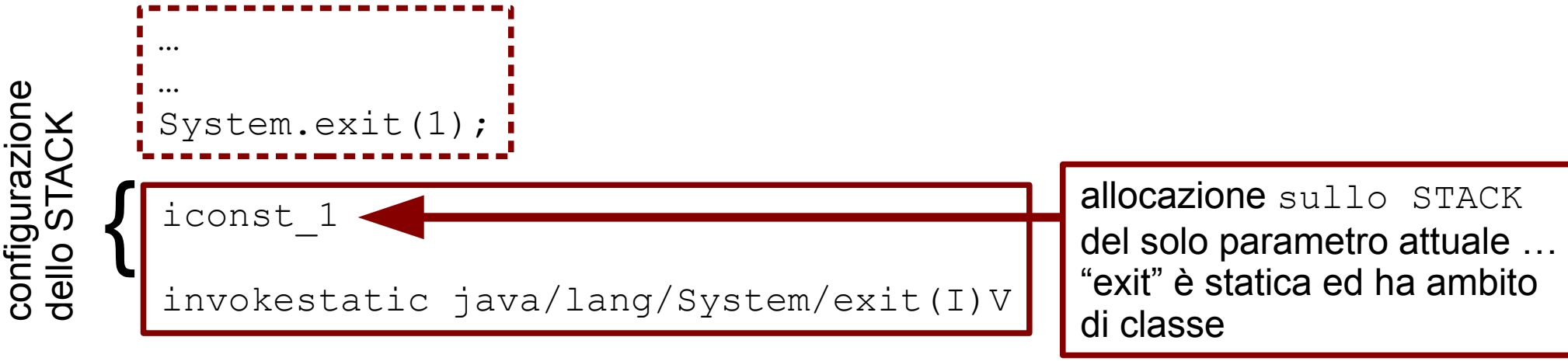
```
…
…
System.exit(1);
```

```
iconst_1

invokestatic java/lang/System/exit(I)V
```

  - `invokespecial`

JVM sotto il cofano … – 2.2

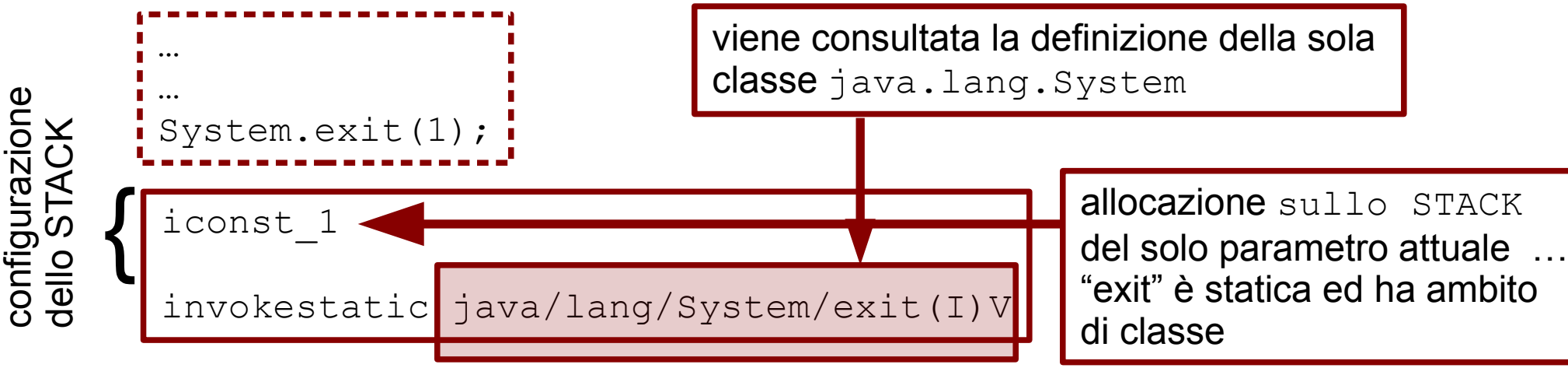
- (principalmente) la JVM implementa l'invocazione di metodi attraverso le istruzioni
  - `invokevirtual`
  - `invokestatic` : viene consultato il run-time constant pool accedendo in base alla classe referenziata, ed usando parametri attuali



  - `invokespecial`

JVM sotto il cofano … – 2.2

- (principalmente) la JVM implementa l'invocazione di metodi attraverso le istruzioni
  - `invokevirtual`
  - `invokestatic` : viene consultato il run-time constant pool accedendo in base alla classe referenziata, ed usando parametri attuali



  - `invokespecial`

JVM sotto il cofano … – 2.3

- (principalmente) la JVM implementa l'invocazione di metodi attraverso le istruzioni
  - `invokevirtual`
  - `invokestatic`
  - `invokespecial` : è un ibrido tra le precedenti soluzioni, ed è usato per gestire casi particolari (e.g. `main`, metodi `private` o `final`)
    - viene consultato il run-time constant pool accedendo in base alla classe referenziata, ed usando parametri attuali (come nel caso di `invokestatic`)
    - si tiene traccia dell'istanza nello STACK (come nel caso di `invokevirtual`)

JVM sotto il cofano … – 3

- Java dinamico → le invocazioni codificate per la JVM sono simboliche
  - i parametri delle istruzioni “*invoke*” sono riferimenti che identificano univocamente un metodo
  - la prima volta che la JVM incontra una istruzione di “*invoke*”, risolve i riferimenti
- invocazione di metodi
  - `invokevirtual`: il riferimento all'oggetto sul quale viene invocato il metodo è allocato nello STACK insieme ai parametri attuali. Il riferimento `this` è implicitamente passato al contesto di esecuzione del metodo
  - `invokestatic`: nello STACK vengono allocati solo i parametri attuali che devono essere passati al metodo
  - `invokespecial`: il riferimento all'oggetto sul quale viene invocato il metodo è allocato nello STACK insieme ai parametri attuali. Il riferimento `this` è implicitamente passato al contesto di esecuzione del metodo

esempio

(credits Jonathan Shewchuk)

esempio

(credits Jonathan Shewchuk)

```
public class Fraction {
    private long numerator;
    private long denominator;

    public Fraction(long numerator, long denominator){
        this.numerator = numerator;
        this.denominator = denominator;
    }

    public Fraction(Fraction original) {
        this(original.numerator, original.denominator);
        }

    public void divide(Fraction f) {
        this.numerator = this.numerator * f.denominator;
        this.denominator = this.denominator * f.numerator;
        this.reduce();
    }

    public Fraction quotient(Fraction f) {
        Fraction result = new Fraction(this);
        result.divide(f);
        return result;
    }


    static private long gcd(long a, long b) {
        while (b > 0) {
            long remainder = a % b;
            a = b;
            b = remainder;
        }
        return a;
    }

    private void reduce() {
        long divisor = gcd(this.numerator, this.denominator);
        this.numerator = this.numerator / divisor;
        this.denominator = this.denominator / divisor;
        if (this.denominator < 0) {
            this.numerator = -this.numerator;
            this.denominator = -this.denominator;
        }
    }

}
```

esempio

(credits Jonathan Shewchuk)

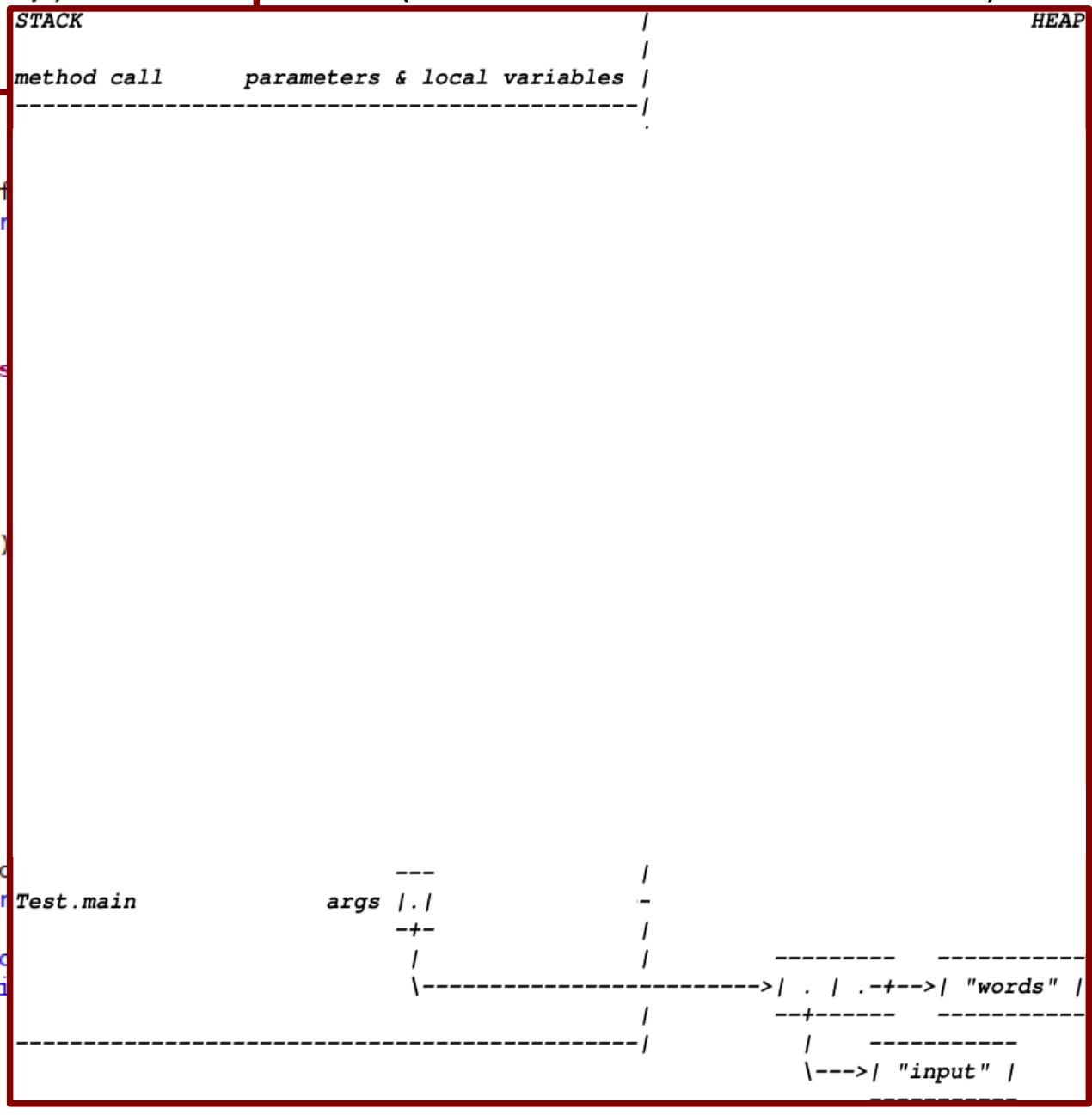
```
public class Fraction {
public class Test {
  public static void main(String[] args) {
    Fraction f1 = new Fraction(3, 5);
    Fraction f2 = new Fraction(9, 10);
    Fraction f3 = f1.quotient(f2);
  }
}
    }

    public void divide(Fraction f) {
        this.numerator = this.numerator * f.denominator;
        this.denominator = this.denominator * f.numerator;
        this.reduce();
    }

    public Fraction quotient(Fraction f) {
        Fraction result = new Fraction(this);
        result.divide(f);
        return result;
    }


    static private long gcd(long a, long b) {
        while (b > 0) {
            long remainder = a % b;
            a = b;
            b = remainder;
        }
        return a;
    }

    private void reduce() {
        long divisor = gcd(this.numerator, this.denominator);
        this.numerator = this.numerator / divisor;
        this.denominator = this.denominator / divisor;
        if (this.denominator < 0) {
            this.numerator = -this.numerator;
            this.denominator = -this.denominator;
        }
    }

}
```

esempio

(credits Jonathan Shewchuk)

```
public class Test {
  public static void main(String[] args) {
    Fraction f1 = new Fraction(3, 5);
    Fraction f2 = new Fraction(9, 10);
    Fraction f3 = f1.quotient(f2);
  }
}
```

```
public class Fraction {
    }

    public void divide(Fraction f) {
        this.numerator = this.numerator * f
        this.denominator = this.denominator
        this.reduce();
    }

    public Fraction quotient(Fraction f) {
        Fraction result = new Fraction(this
        result.divide(f);
        return result;
    }


    static private long gcd(long a, long b)
        while (b > 0) {
            long remainder = a % b;
            a = b;
            b = remainder;
        }
        return a;
    }

    private void reduce() {
        long divisor = gcd(this.numerator,
        this.numerator = this.numerator / d
        this.denominator = this.denominator
        if (this.denominator < 0) {
            this.numerator = -this.numerato
            this.denominator = -this.denomi
        }
    }

}
```

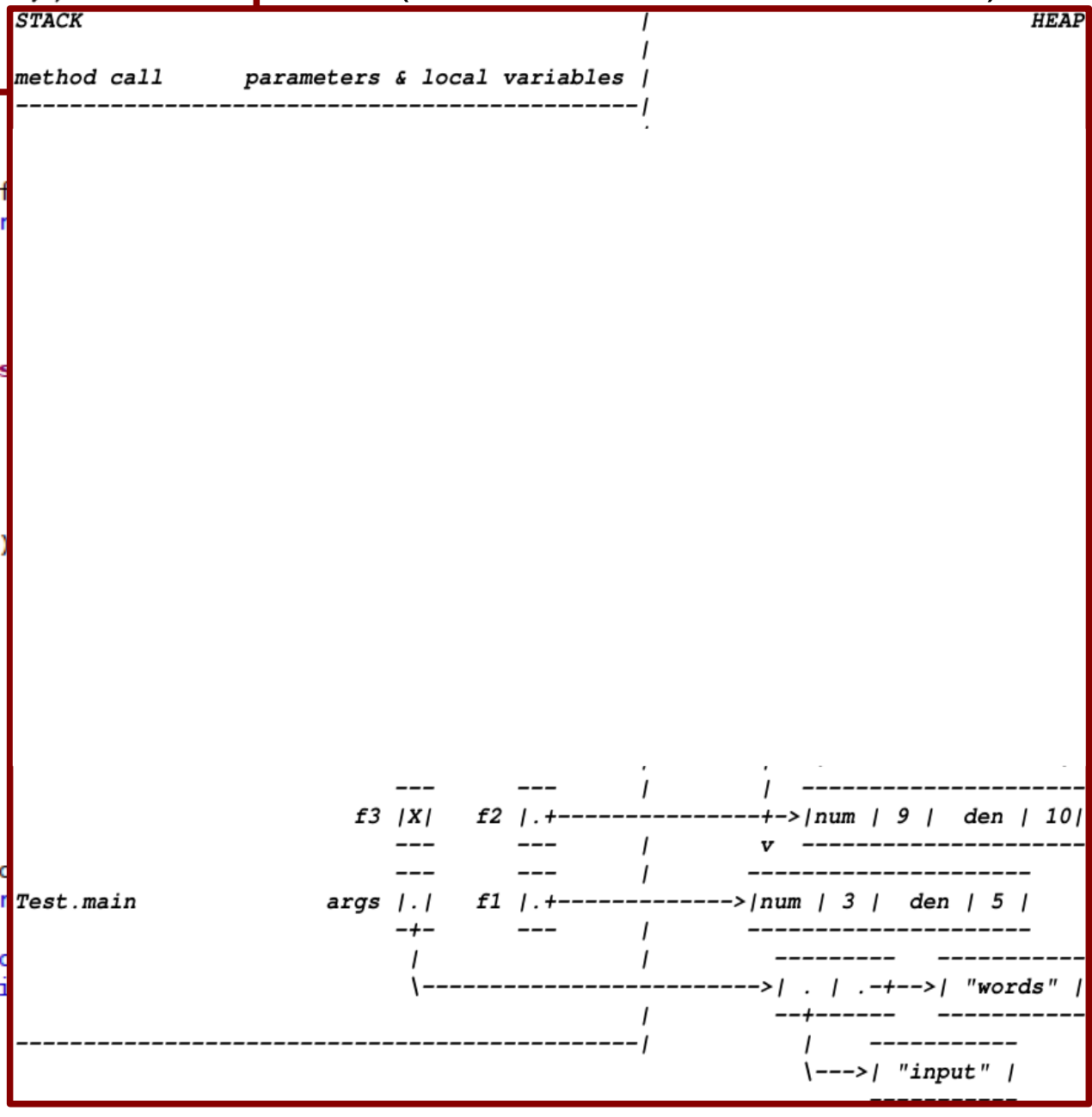


esempio

(credits Jonathan Shewchuk)

```
public class Test {
  public static void main(String[] args) {
    Fraction f1 = new Fraction(3, 5);
    Fraction f2 = new Fraction(9, 10);
    Fraction f3 = f1.quotient(f2);
  }
}
```

```
public class Fraction {
    }

    public void divide(Fraction f) {
        this.numerator = this.numerator * f
        this.denominator = this.denominator
        this.reduce();
    }

    public Fraction quotient(Fraction f) {
        Fraction result = new Fraction(this
        result.divide(f);
        return result;
    }


    static private long gcd(long a, long b)
        while (b > 0) {
            long remainder = a % b;
            a = b;
            b = remainder;
        }
        return a;
    }

    private void reduce() {
        long divisor = gcd(this.numerator,
        this.numerator = this.numerator / d
        this.denominator = this.denominator
        if (this.denominator < 0) {
            this.numerator = -this.numerato
            this.denominator = -this.denomi
        }
    }

}
```

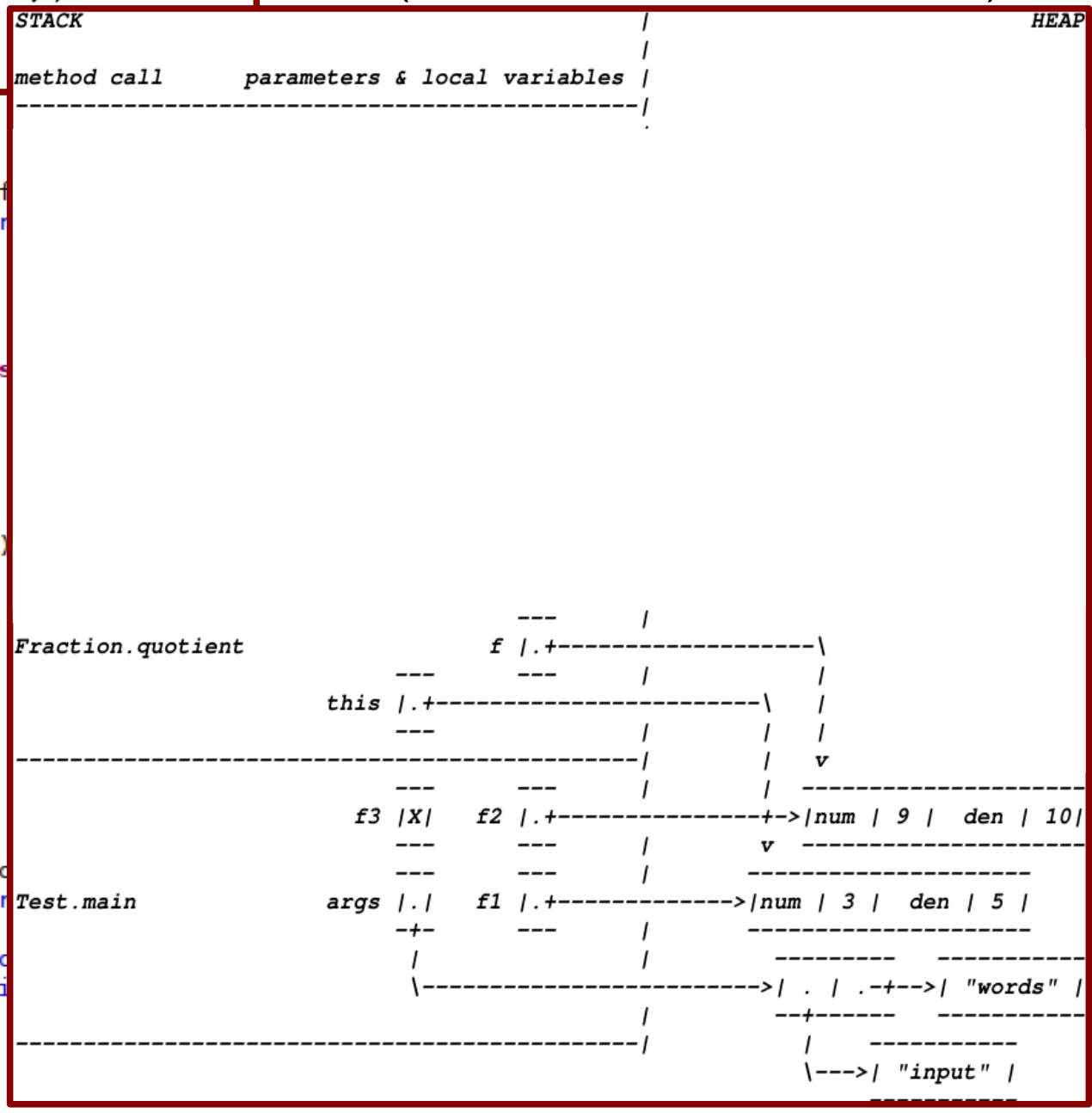


esempio

(credits Jonathan Shewchuk)

```
public class Test {
  public static void main(String[] args) {
    Fraction f1 = new Fraction(3, 5);
    Fraction f2 = new Fraction(9, 10);
    Fraction f3 = f1.quotient(f2);
  }
}
```

```
public class Fraction {
    }

    public void divide(Fraction f) {
        this.numerator = this.numerator * f
        this.denominator = this.denominator
        this.reduce();
    }

    public Fraction quotient(Fraction f) {
        Fraction result = new Fraction(this
        result.divide(f);
        return result;
    }


    static private long gcd(long a, long b)
        while (b > 0) {
            long remainder = a % b;
            a = b;
            b = remainder;
        }
        return a;
    }

    private void reduce() {
        long divisor = gcd(this.numerator,
        this.numerator = this.numerator / d
        this.denominator = this.denominator
        if (this.denominator < 0) {
            this.numerator = -this.numerato
            this.denominator = -this.denomi
        }
    }

}
```

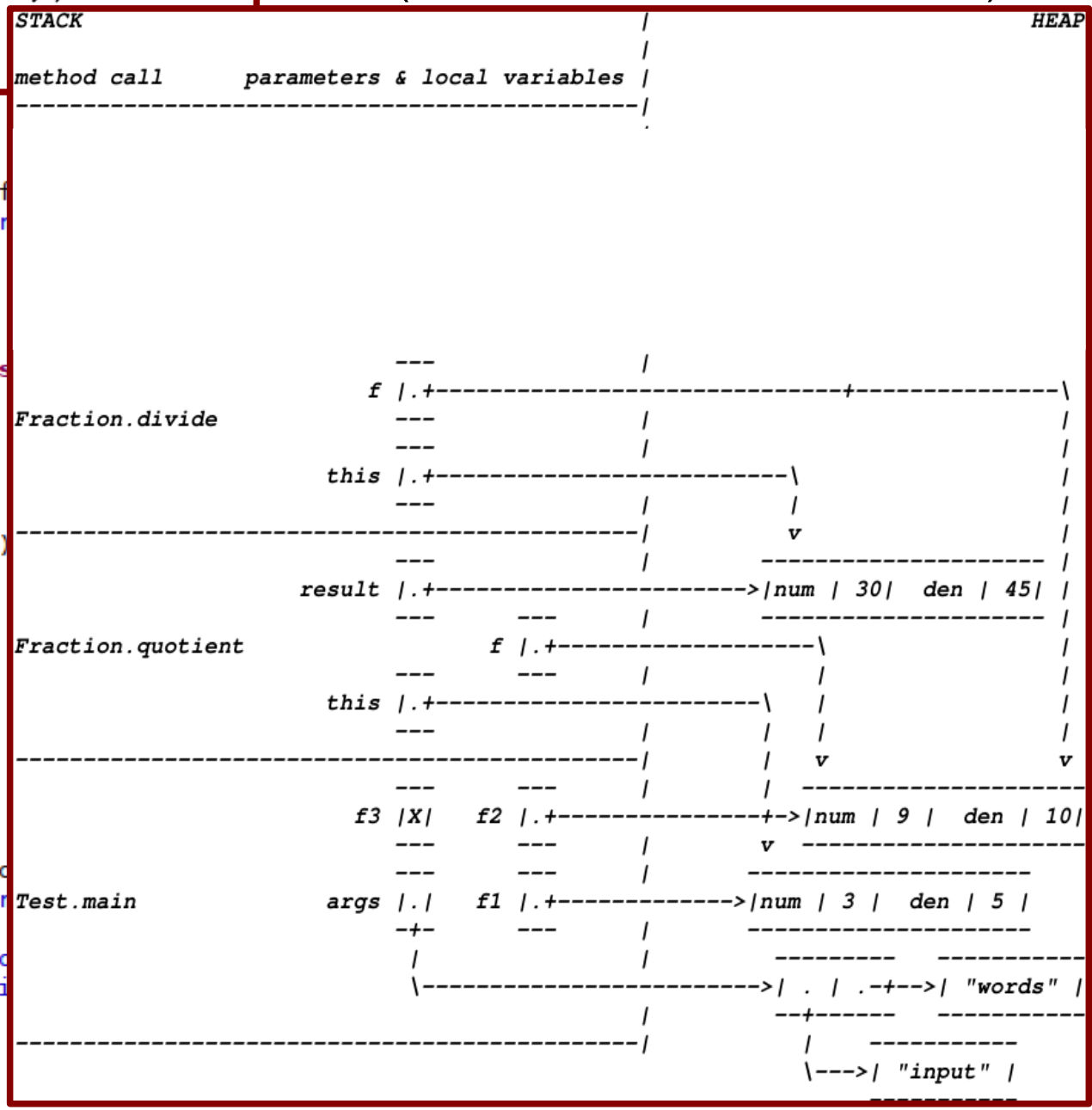


esempio

(credits Jonathan Shewchuk)

```
public class Test {
  public static void main(String[] args) {
    Fraction f1 = new Fraction(3, 5);
    Fraction f2 = new Fraction(9, 10);
    Fraction f3 = f1.quotient(f2);
  }
}
```

```
    }

    public void divide(Fraction f) {
        this.numerator = this.numerator * f
        this.denominator = this.denominator
        this.reduce();
    }

    public Fraction quotient(Fraction f) {
        Fraction result = new Fraction(this
        result.divide(f);
        return result;
    }


    static private long gcd(long a, long b)
        while (b > 0) {
            long remainder = a % b;
            a = b;
            b = remainder;
        }
        return a;
    }

    private void reduce() {
        long divisor = gcd(this.numerator,
        this.numerator = this.numerator / d
        this.denominator = this.denominator
        if (this.denominator < 0) {
            this.numerator = -this.numerato
            this.denominator = -this.denomi
        }
    }

}
```

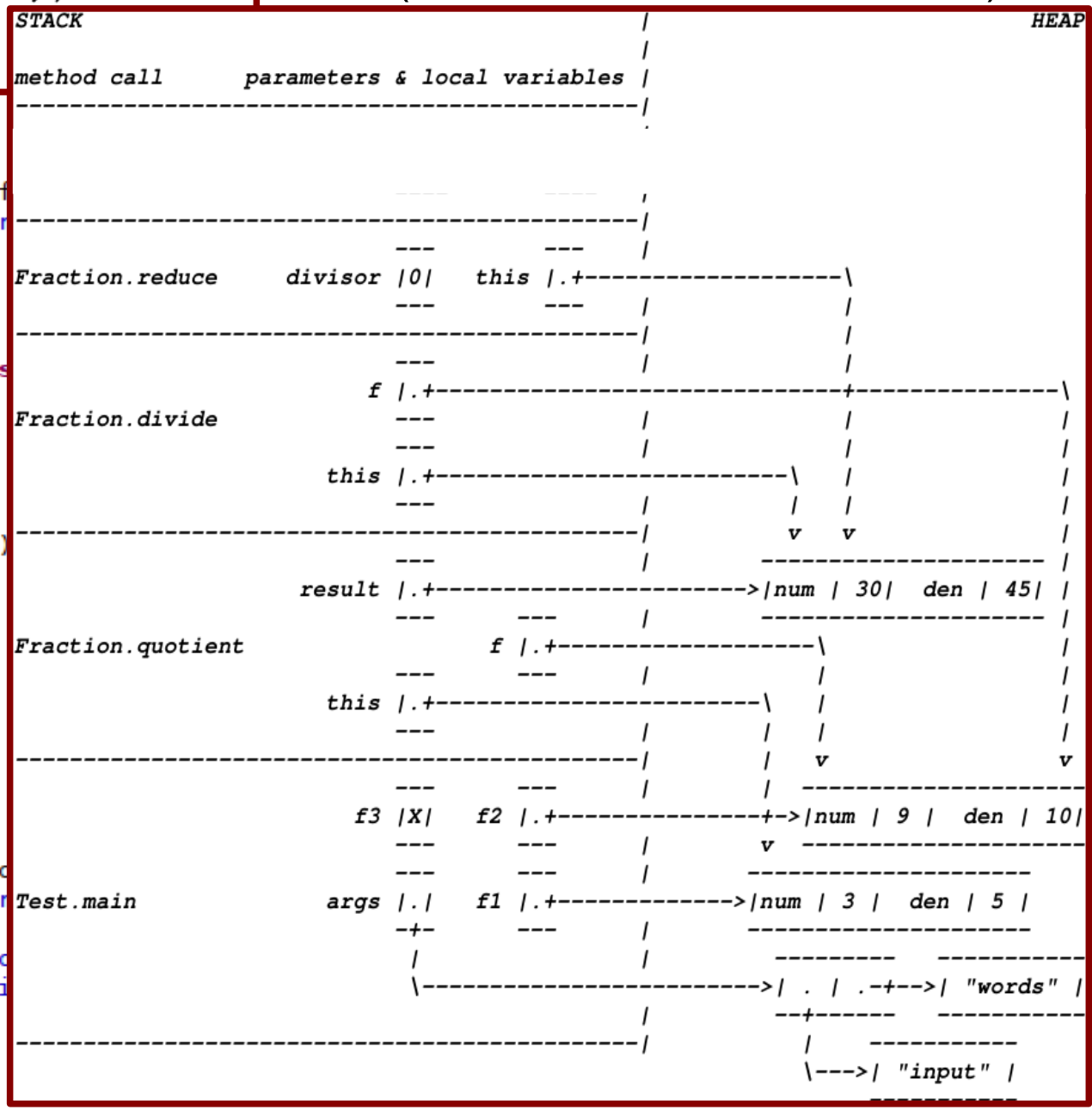


esempio

(credits Jonathan Shewchuk)

```
public class Test {
  public static void main(String[] args) {
    Fraction f1 = new Fraction(3, 5);
    Fraction f2 = new Fraction(9, 10);
    Fraction f3 = f1.quotient(f2);
  }
}
```

```
public class Fraction {
    }

    public void divide(Fraction f) {
        this.numerator = this.numerator * f
        this.denominator = this.denominator
        this.reduce();
    }

    public Fraction quotient(Fraction f) {
        Fraction result = new Fraction(this
        result.divide(f);
        return result;
    }


    static private long gcd(long a, long b)
        while (b > 0) {
            long remainder = a % b;
            a = b;
            b = remainder;
        }
        return a;
    }

    private void reduce() {
        long divisor = gcd(this.numerator,
        this.numerator = this.numerator / d
        this.denominator = this.denominator
        if (this.denominator < 0) {
            this.numerator = -this.numerato
            this.denominator = -this.denomi
        }
    }

}
```



esempio

(credits Jonathan Shewchuk)

```
public class Test {
  public static void main(String[] args) {
    Fraction f1 = new Fraction(3, 5);
    Fraction f2 = new Fraction(9, 10);
    Fraction f3 = f1.quotient(f2);
  }
}
```

```
public class Fraction {
    }

    public void divide(Fraction f) {
        this.numerator = this.numerator * f
        this.denominator = this.denominator
        this.reduce();
    }

    public Fraction quotient(Fraction f) {
        Fraction result = new Fraction(this
        result.divide(f);
        return result;
    }


    static private long gcd(long a, long b)
        while (b > 0) {
            long remainder = a % b;
            a = b;
            b = remainder;
        }
        return a;
    }

    private void reduce() {
        long divisor = gcd(this.numerator,
        this.numerator = this.numerator / d
        this.denominator = this.denominator
        if (this.denominator < 0) {
            this.numerator = -this.numerato
            this.denominator = -this.denomi
        }
    }

}
```