

# Numero variabile di argomenti

- Altrimenti noto come meccanismo dei varargs.
- Esempio: per la classe **Numero** definiamo un metodo **sommaA** il cui prototipo è:

**Numero sommaA(Numero x)**

e produce la somma dell'oggetto destinatario del messaggio con **x**.

- Vogliamo sovraccaricare questo metodo con un altro con un numero variabile di argomenti.

# Numero variabile di argomenti

- Per realizzare un overloading con numero variabile di argomenti, scriveremo:

```
public Numero sommaA(Numero... args)
{ ..... }
```

- In questo modo viene accettato un numero variabile di **args**, tutti di tipo **Numero**. Dentro il corpo del metodo, **args** va trattato come un array (cioè è di tipo **Numero[]**).
- Analogo a quanto capita per il **main** dove si usa un array di **String**.

# Numero variabile di argomenti

- Diventa quindi ammesso scrivere:

```
x.sommaA();
```

```
x.sommaA(new Numero(2));
```

```
x.sommaA(new Numero(2), new Numero(6));
```

```
x.sommaA(new Numero(2), new Numero(6),  
new Numero(17));
```

```
...
```

I vincoli di questo sistema sono che vale per un solo tipo (qui **Numero**) e la parte variabile dev'essere l'ultima nella firma del metodo.

# Generare numeri casuali

- ❑ I calcolatori non sono bravi a generare numeri casuali. Quello che fanno è usare funzioni complicate e quindi difficilmente predicibili per estrarre sequenze **pseudo-casuali**.
- ❑ Java può creare un oggetto della classe **Random**, contenuta nel package **java.util**, per generare simili sequenze.
- ❑ Per creare questo oggetto, occorre un numero (che è il seme di generazione della sequenza). È il parametro del costruttore, di tipo long.

# Generare numeri casuali

- Ad esempio:

```
Random generator = new Random(4815162342) ;
```

- Esiste anche un costruttore senza parametri

```
Random generator2 = new Random() ;
```

- usa un numero “segreto” deciso da chi ha implementato la JVM.
- Un modo comune di avere questo numero usa il metodo `currentTimeMillis()` della classe `System` del package `java.lang` (con prototipo `public static long currentTimeMillis()` che ritorna il “tempo corrente” in millisecondi.

# Generare numeri casuali

- Una volta creato il generatore, si può invocare il suo metodo **nextInt** (ritorna il “prossimo” valore della sequenza pseudorandom).
  - ▣ è sovraccaricato: a un parametro intero **x**, genera un intero tra 0 e **x-1**; senza parametri, tra 0 e  $2^{32}$
- Esiste anche il metodo **nextDouble** (genera il prossimo valore **double** anziché intero).
  - ▣ a seconda dell'applicazione, il risultato poi va scalato per avere un valore sensato

# Generare numeri casuali

- Un programma per lanciare 10 dadi e visualizzare a video il risultato.

```
import java.util.*;
public class Lancia10Dadi
{ public static void main (String[] args)
  { Random dGen =
      new Random(System.currentTimeMillis());
    System.out.println("Lancio 10 dadi:");
    int dado;
    for (int i=0; i<10; i++)
    { dado = dGen.nextInt(6) + 1;
      System.out.println(""+dado);
    }
  }
}
```

# Input e output

- La gestione di input e output in Java avviene tramite la classe pubblica **System**
- È una classe **final** contenuta nel package `java.lang` quindi automaticamente importata.
- Questa classe ha tre oggetti di interesse:
  - ▣ **System.in** che prende input (da tastiera)
  - ▣ **System.out** che produce output (a video)
  - ▣ **System.err** che genera messaggi di errore
  - ▣ e anche un metodo, **System.exit(int nc)** che fa terminare l'esecuzione con codice `nc`.

# Input e output

- **`System.in`, `System.out`, `System.err`**, richiamano (anche nei nomi) i flussi standard **`stdin`, `stdout`, `stderr`** visti in C.
- Ma sono diversi da C, dove **`stdin`, `stdout`, `stderr`** sono tipicamente `#definiti` come 0, 1, 2 e quindi sono alias per valori numerici.
  - ▣ Nessuna capacità aggiuntiva, solo descrittori
- In Java invece sono **oggetti!**
  - ▣ Possiedono quindi proprietà e metodi.

# System.out

- Abbiamo già visto `System.out` e i suoi metodi `print()` e `println()`.
- L'oggetto `System.out` è più precisamente un oggetto di tipo `PrintStream`, con modificatori

`public,`

`static,`

e `final.`

Il suo valore può essere settato al più una volta

Può essere acceduto ovunque perché membro `public` di una classe `public`

È un membro di classe, non di un'istanza specifica: ogni modifica è generale

# System.err

- Anche **System.err** è **public**, **static**, e **final**. È di tipo **PrintStream**, quindi ha gli stessi metodi di **System.out**.
  - ▣ Motivo: tenere separati i messaggi d'errore (es.: output a stampante, lasciando **System.err** a video per “La stampante ha finito la carta”)

```
public double tech(double x, double y)
{ if (x < 0)
  { System.err.println("x errato, uso 0");
    return 0.0;
  }
  return Math.pow(x,y) ;
}
```

# System.err

- È logico abbinare **System.err** alla stampa di messaggi di errore se s'intercetta un'eccezione.

```
public class Prudente
{ public void unMetodo(int parametro)
  { try
    { unOggetto.faQualcosa(parametro);
    }
    catch (Exception e)
    { System.err.println("Non riesco");
    }
  }
}
```

# System.in

- Anche **System.in** è **public**, **static**, e **final**. È invece di tipo **InputStream**.
- Questa classe ha capacità molto limitate. Ha infatti un unico metodo chiamato **read()** che restituisce un **byte** letto dal flusso d'ingresso.
- Per leggere valori più elaborati bisognerebbe:
  - ▣ leggere un sufficiente numero di byte (anche i caratteri, che usano Unicode, sono 2 byte)
  - ▣ fare una conversione di tipo (trattando con cura i problemi che possono presentarsi)

# System.in

- Conviene usare un metodo più comodo, in cui le funzionalità più standard sono già implementate
  - ▣ tipicamente si usano classi del package `java.io`
- Ad esempio, è molto comune l'uso della classe **BufferedReader** e del suo metodo **readLine()**
  - ▣ Questo metodo legge una **stringa** da un buffer
  - ▣ Sicuramente più comodo che non gestire un byte alla volta, ma richiede comunque conversioni
  - ▣ Il tutto viene implementato tramite alcuni passaggi intermedi

# Leggere input

- Trasformare un flusso di byte letti uno alla volta (operazione fornita dall'oggetto **System.in**) in un flusso (bufferizzato) di caratteri letti una stringa alla volta comporta:
  - va creato un nuovo oggetto che legga caratteri anziché byte; a tale scopo esiste la classe **InputStreamReader**.
  - va creato un nuovo oggetto che legga stringhe anziché caratteri (fino all'end-of-line); a tale scopo esiste la classe **BufferedReader**.

# Leggere input

- La classe **InputStreamReader** è in grado di leggere caratteri anziché byte, convertendoli usando uno specifico set di caratteri.
  - ▣ si può usare Unicode, che tipicamente è il set predefinito, oppure usare un altro set
- Per creare un oggetto **InputStreamReader** tipicamente si usa un costruttore con un parametro di tipo **InputStream**
  - ▣ di solito questo è proprio **System.in**

```
InputStreamReader cs = new InputStreamReader(System.in) ;
```

# Leggere input

- La classe **BufferedReader** mette a disposizione un metodo **readLine()**.
  - ▣ per usarlo, bisogna avere un oggetto di questa classe, che viene costruito ad esempio a partire da un oggetto di tipo **InputStreamReader**
- ▣ dato che l'oggetto **cs** di tipo **InputStreamReader** è inutile, si poteva anche renderlo anonimo

```
BufferedReader in = new BufferedReader(cs) ;
```

```
BufferedReader in =  
new BufferedReader(new InputStreamReader(System.in)) ;
```

# Leggere input

- Si può anche seguire un altro ordine
  - ▣ prima bufferizzare poi tradurre in stringhe; nel mezzo si usa la classe **BufferedReader**

```
BufferedReader in =  
new BufferedReader(new BufferedReader(System.in)) ;
```

- ▣ L'oggetto di tipo **BufferedReader** usa un costruttore con un parametro di tipo **InputStream**, cioè **System.in**
- ▣ L'oggetto di tipo **BufferedReader** viene creato con un altro costruttore che ha sempre un parametro ma di tipo **BufferedReader**

# Leggere input

- Una volta creato un oggetto in grado di leggere “meglio” lo standard input, possiamo invocare il metodo `readLine()`.

```
String str = new String();  
str = in.readLine(); //str = stringa letta
```

- ▣ Raggiunta la fine input, `readLine()` ritorna `null`.
- La classe **BufferedReader** ha uno scopo più generale che leggere solo da standard input.
  - ▣ Potrebbe leggere anche da altri input, e nel farlo potrebbero verificarsi errori.

# Leggere input

- In effetti è possibile che la lettura fallisca.
- In tal caso viene generata un'eccezione, di tipo `IOException`.
- È bene prevedere la possibilità che la lettura fallisca, e cosa fare in caso di eccezione.

```
try
{ str = in.readLine();
}
catch (IOException e)
{ SoluzioneAllEccezione();
}
```

# Leggere input

```
import java.io.*;
public class LeggiStringa
{ public static void main (String[] args)
  { BufferedReader in = new BufferedReader(new
    InputStreamReader(System.in)) ;
    String str = new String() ;
    System.out.print("Come ti chiami?") ;
    try
    { str = in.readLine() ;
    } catch(IOException e)
    { System.err.println("Errore lettura nome") ;
      System.exit(-1) ;
    }
    System.out.println("Lieto di conoscerti, " +str) ;
    ....
  }
```

# Leggere input

- Se però l'input da leggere non fosse una stringa, ma per esempio un numero intero, bisognerebbe usare il metodo `parseInt()`.
- Ricordiamo che è un metodo statico della classe involucro `Integer`.

```
String s; int n;  
s = in.readLine();  
n = Integer.parseInt(s);
```

- C'è anche `parseDouble` della classe involucro `Double` per convertire a `double`.

# Leggere input

```
.....
    System.out.println("Lieto di conoscerti, "+nome);
    System.out.println("Quanti anni hai?");
    try
    { str = in.readLine();
    } catch(IOException e)
    { System.err.println("Errore lettura età");
      System.exit(-1);
    }
    try
    { int age = Integer.parseInt(str);
    } catch(NumberFormatException e)
    { System.err.println("Dovevi mettere un int!");
      System.exit(-1);
    }
    System.out.println("'" + age + " primavera? Però!");
    ...
```

# Leggere input

- ❑ **parseInt** può lanciare un'eccezione (di tipo **NumberFormatException**) se non gradisce la stringa (numero non valido).
- ❑ Per input più complicati si può usare la classe **StringTokenizer** contenuta in **java.util**.
  - ▣ I suoi oggetti sono creati con un costruttore a due parametri di tipo **String**  
**StringTokenizer(String stringa, String delim)**
  - ▣ Gli oggetti sono in grado di leggere **stringa** come insieme di token intervallati da delimitatori **delim**

# Leggere input

- Un oggetto di tipo `StringTokenizer` ha a disposizione i metodi **`nextToken()`** che legge in chiamate successive il token in testa alla sequenza e **`hasMoreTokens()`** che dice se ne ha ancora.

```
seq = "5 45 800 3 122";  
StringTokenizer st =  
    new StringTokenizer(seq, " ");  
while (st.hasMoreTokens)  
{ System.out.println(st.nextToken());  
}
```

# Leggere input

- Se siamo sicuri che la stringa è fatta solo da interi, possiamo darli in pasto a `parseInt()`.

```
seq = "5 45 800 3 122"; int somma;  
StringTokenizer st =  
    new StringTokenizer(seq, " ");  
while (st.hasMoreTokens() )  
{  somma += Integer.parseInt(st.nextToken() ) ;  
}
```

...ma se ci sbagliamo verrà lanciata una eccezione di tipo `NumberFormatException`.

# Leggere input

- In realtà gli oggetti di tipo **StringTokenizer** **violano l'incapsulamento!**
- Infatti, i valori dovrebbero essere acceduti da metodi get, e fissati da metodi set.
- Il metodo **nextToken()** fa entrambe le cose:
  - ▣ preleva un oggetto dalla sequenza (get)
  - ▣ ma lo elimina anche dalla sequenza (set); in alternativa lo si può anche vedere come effetto collaterale.

# Gestione della memoria

- Java usa gli oggetti, perciò gestisce la memoria per due principali scopi:
  - ▣ memorizzare gli oggetti in sé (il loro stato)
  - ▣ memorizzare informazioni per l'esecuzione di metodi e costruttori
- Nota: gli oggetti vengono memorizzati una volta (e poi eventualmente aggiornati), mentre i metodi tutte le volte che vengono eseguiti (i parametri possono cambiare)

# Gestione della memoria

- Se si considera la memoria suddivisa in aree (fisicamente, pensiamo a gruppi adiacenti di byte) una certa area può essere
  - ▣ allocata, ovvero destinata a uno scopo specifico
  - ▣ deallocata, ovvero liberata da un'allocazione
- In Java la gestione della memoria è automatica, non come in C dove è lasciata al programmatore.

# Gestione della memoria

- La gestione della memoria secondo Java è dinamica, nel senso che:
  - ▣ un oggetto riceve un'allocazione di spazio in memoria solo quando effettivamente richiesto, e questo viene deallocato quando si deduce che l'oggetto non serve più
  - ▣ un metodo riceve un'allocazione di spazio in memoria (per lui, ma anche per le sue variabili locali) quando viene eseguito. Al termine dell'esecuzione, questa memoria viene liberata.

# Gestione della memoria

- Durante l'esecuzione, la JVM organizza la memoria in tre blocchi principali:
  - ▣ memoria statica
  - ▣ heap
  - ▣ stack
- La **memoria statica** è una parte della memoria allocabile destinata a contenere i campi statici. Si può sapere a priori quanta ne serve (lo si sa già anche al lato compilatore).

# Gestione della memoria

- Lo **heap** (letteralmente “mucchio”) è utilizzato per gli oggetti. Viene riservata un’area tutte le volte che si usa **new** per creare un oggetto.
  - ▣ Java lo usa anche per memorizzare una classe tutte le volte che viene riferita/usata quella classe, prima ancora che venga creato un oggetto.
  - ▣ La liberazione di memoria dallo heap avviene quando la JVM deduce che un oggetto non serve più. Questo intervento avviene tipicamente in automatico, ma si può anche forzare.

# Gestione della memoria

- Lo **stack** contiene i dati (parametri espliciti / impliciti, variabili) dei metodi in esecuzione.
  - ▣ Deve essere uno stack per corrispondere alla pila dei record di attivazione. La sua struttura evolve a seconda dei metodi eseguiti (ignoti a priori dal compilatore).
  - ▣ La pila dei record di attivazione contiene tutti i dati necessari al metodo attualmente in esecuzione (che potrebbe avere visibilità, anche ricorsiva, su metodi precedentemente chiamati).

# Gestione della memoria

- Nel record di attivazione sono compresi:
  - ▣ le variabili locali (compresi i parametri attuali)
  - ▣ il riferimento a **this** (se il metodo non è statico)
  - ▣ il punto di ritorno (quando il metodo termina, viene rimandato il controllo al punto da cui era stato passato)
- Sempre quando il metodo termina, questo record viene cancellato dallo stack.
- Adotta una politica LIFO, quindi è uno stack.

# Gestione della memoria

- Politica Last-In-First-Out: il metodo attualmente in esecuzione è quello in cima.
  - Se questo invoca un altro metodo passandogli il controllo, il suo record di attivazione viene messo sopra (è stato l'ultimo a cominciare, ma sarà anche il primo a finire)
  - Quando quello in cima finisce, il controllo passa a quello sottostante, è lui che l'aveva invocato. Questo diventa il metodo in cima alla pila.

# Garbage collector

- Non esiste un modo esplicito per eliminare un oggetto e cancellare ogni riferimento alla sua area di memoria. Lo fa la JVM, quando è sicura che quell'oggetto è davvero inutile.
- Un oggetto è inutile se è accessibile, cioè non esiste più nessun riferimento verso di lui.
- La JVM si preoccupa di cercare e distruggere oggetti inutili. Soprattutto lo fa quando è necessario, ad es. perché la memoria è finita.

# Garbage collector

- Questo meccanismo automatico è detto **garbage collector** (=raccoglitore di spazzatura)
- È una routine della JVM che si occupa di individuare gli oggetti non più accessibili e liberarne l'area di memoria.
- Anche se non esiste un comando free o delete, è comunque possibile rendere un oggetto non più referenziato se si assegna la sua variabile di riferimento al valore **null**.

# Garbage collector

- Il garbage collector tipicamente entra in campo solo se la memoria disponibile è esaurita.
- Il metodo **gc** della classe **Runtime** “forza” l'utilizzo del garbage collector.
  - ▣ In realtà suggerisce semplicemente alla JVM che potrebbe essere una buona idea girare il g.c.
  - ▣ Può essere invocato come **System.gc()**, una scorciatoia per **Runtime.getRuntime().gc()**.
  - ▣ Il suo utilizzo dipende molto dal sistema e dall'applicazione che si utilizza.

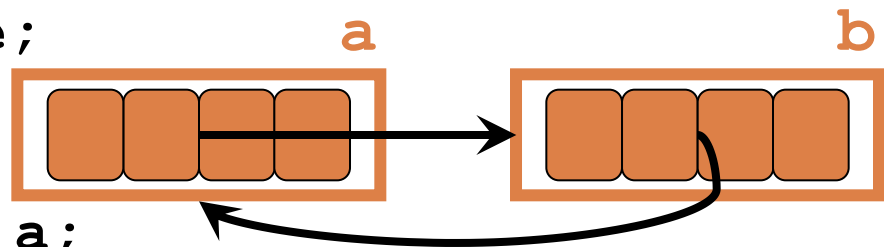
# Garbage collector

- Confronto tra stack e heap:
  - ▣ lo stack è ben ordinato, lo heap è disorganizzato
- In altre parole, lo stack non presenta “buchi” nell’allocazione, perché, in virtù del LIFO, si sa sempre dove prendere la memoria che serve
- Nello heap si allocano blocchi di memoria di varie dimensioni, secondo le richieste. Questi poi si liberano e possono rimanere piccole aree vuote inutilizzabili: **frammentazione**.

# Garbage collector

- Il garbage collector è in grado di eliminare questi buchi, ovvero **deframmentare** lo heap.
- Però il garbage collector non risolve tutto
  - ▣ per esempio non può nulla contro i riferimenti incrociati (**a** riferisce **b** e viceversa); in questo caso non è in grado di accorgersi che le aree di memoria di **a** e **b** potrebbero essere liberate

```
class Person
{ public Person Madre;
  .... }
Person a,b;
a.Madre = b; b.Madre = a;
```



# Il metodo **finalize**

- Generalmente, l'intervento del garbage collector passa sotto silenzio. Potrebbe però essere utile sapere quando interviene.
- Per ogni classe, si può definire in questo caso un metodo **finalize()**
  - ▣ Questo metodo viene eseguito dal garbage collector prima di cancellare l'oggetto
  - ▣ Va chiamato così per motivi di ereditarietà. In realtà esiste (viene da **java.lang**) per tutte le classi, ma non fa nulla: si può però riscriverlo.

# Il metodo **finalize**

- Il metodo **finalize()** può servire quando l'oggetto è in contatto con realtà non-Java.
  - ▣ Se ho un oggetto che è una connessione I/O, prima di scartarlo conviene accertarsi che sia chiusa, e se non lo è chiuderla.
  - ▣ Notare che se **finalize()** solleva delle eccezioni, queste non vengono ritornate al chiamante (vanno intercettate subito).

# Il metodo `finalize`

- Prima di finire l'esecuzione, la JVM esegue i metodi `finalize()` di tutte le classi.
- Normalmente non c'è garanzia su quando viene eseguito `finalize()`, però esiste un metodo di `System` (in realtà della classe `Runtime`) chiamato `runFinalization()` ed esegue il metodo `finalize()` di tutti gli oggetti che aspettano di essere finalizzati.

# Il metodo `finalize`

- Il metodo `finalize()` consente anche di **resuscitare** un oggetto. Questo avviene semplicemente se `finalize()` esegue un assegnamento del riferimento all'oggetto a un'altra variabile riferimento dello stesso tipo.
  - ▣ comportamento formalmente giusto ma discutibile
  - ▣ l'oggetto può essere resuscitato una sola volta; alla successiva invocazione del garbage collector il `finalize()` non verrà più eseguito
  - ▣ se si vuole insistere si può però clonare l'oggetto resuscitato (il clone potrà essere poi resuscitato)