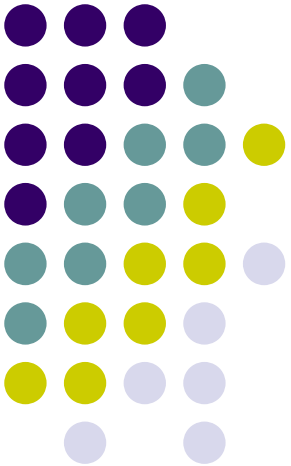
Università di Napoli Federico II - Dip. di Ingegneria Elettrica e delle Tecnologie dell'Informazione
Corso di Laurea in Ingegneria Informatica

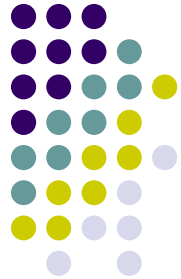
DIETI

Corso di Programmazione

Gestione della memoria in Java



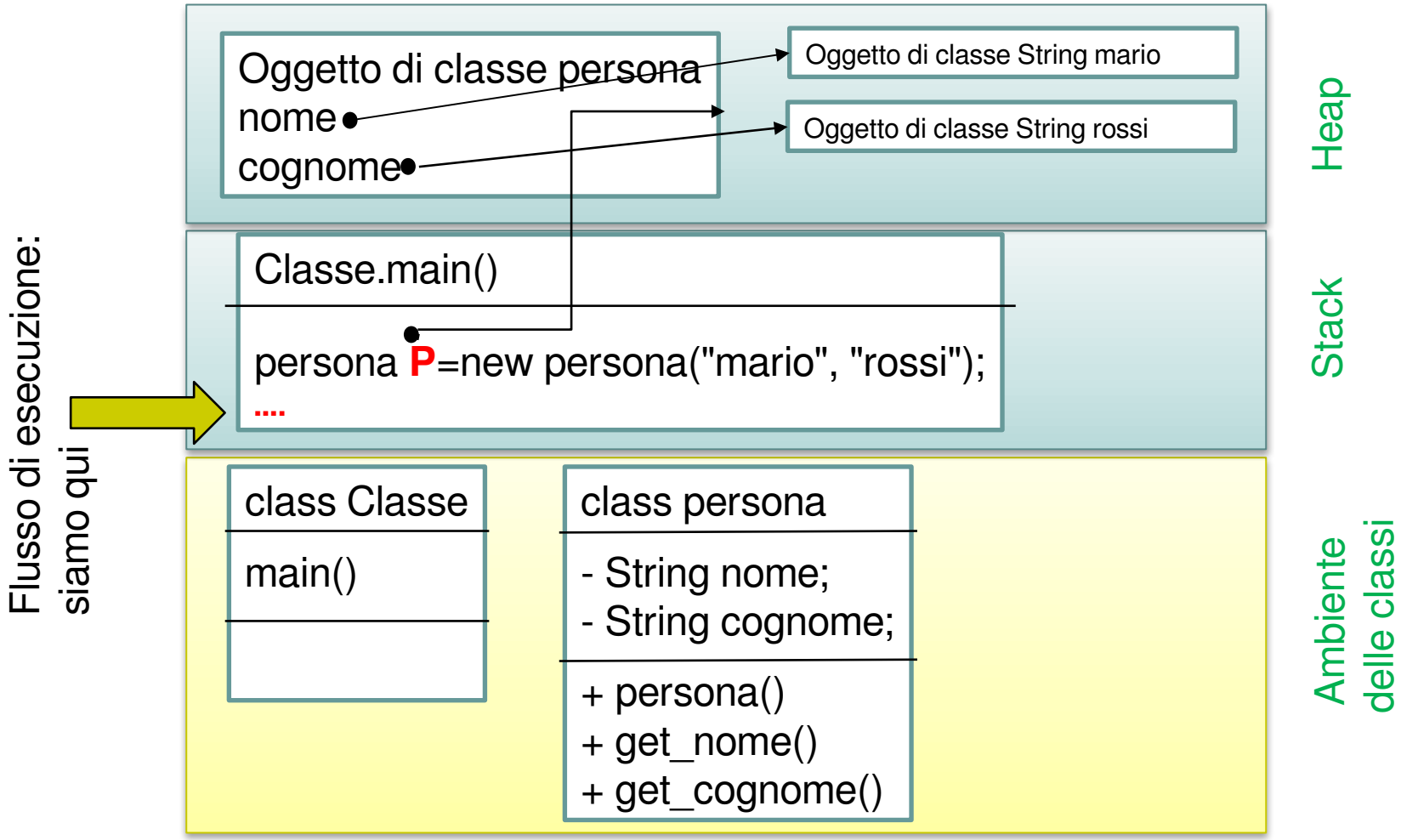
JVM: gestione della memoria



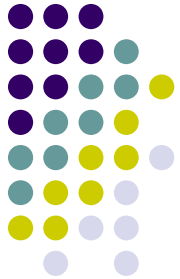
- *La memoria usata dalla JVM e concettualmente divisa in tre parti*
  - *Ambiente delle classi: area di memoria in cui vengono caricate (allocate) tutte le classi che costituiscono il programma*
  - *Stack: area di memoria in cui vengono caricati (allocati) i record di attivazione dei metodi, e quindi tutte le variabili locali*
  - *Heap: area di memoria in cui vengono caricati (allocati) tutti i vari oggetti creati nel programma, man mano che vengono creati.*

Esempio

- Supponiamo che a tempo di esecuzione **si stia eseguendo il metodo main** all'interno del quale sia stato instanziato un oggetto P di una classe Persona. L'allocazione di memoria nella JVM è mostrata in figura.

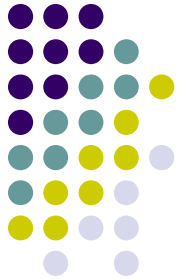


JVM: gestione della memoria



- ***Nell'ambiente delle classi*** *vengono memorizzati il codice dei metodi e le variabili statiche di tutte le classi del programma (condivise dai vari oggetti della classe e utilizzabili anche in assenza di oggetti)*
- ***Nello stack*** *vengono memorizzate le variabili locali dei metodi in esecuzione:*
  - *per le variabili di tipi primitivi viene memorizzato il valore*
  - *per gli oggetti viene memorizzato un riferimento*
- ***Nell'heap*** *per ogni oggetto creato vengono memorizzate le variabili d'istanza (ossia, le variabili non statiche)*
  - *ogni oggetto nell'heap contiene anche il nome della classe di appartenenza*

JVM: gestione della memoria



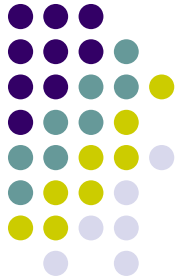
In JAVA quindi ogni volta che si crea un'istanza di una classe (***new()***) si richiede viene allocata memoria nello heap (della JVM)

Quali soluzioni adottare ?

1) alla C/C++...tutto a carico del programmatore (`malloc()-free(), new()-delete()`)

2) tutto a carico dell'ambiente run-time...
...ma le prestazioni ?

JVM: gestione della memoria



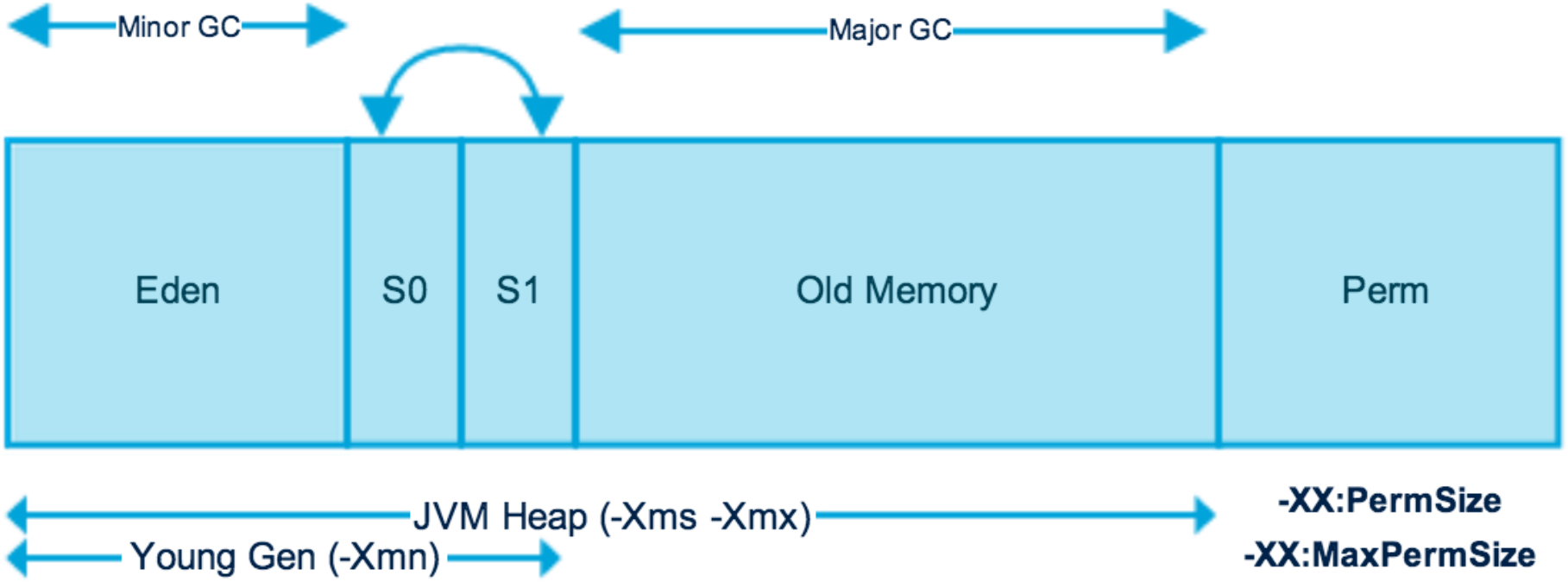
In JAVA è stato implementato un'apposito "modulo" che, in maniera automatica, recupera la memoria degli oggetti **non più utilizzati** (cioè che non sono più riferiti)

Il ***garbage collector***

Quindi in JAVA non esiste più il problema, tipico del C e C++, della "perdita di memoria" dovuto a...

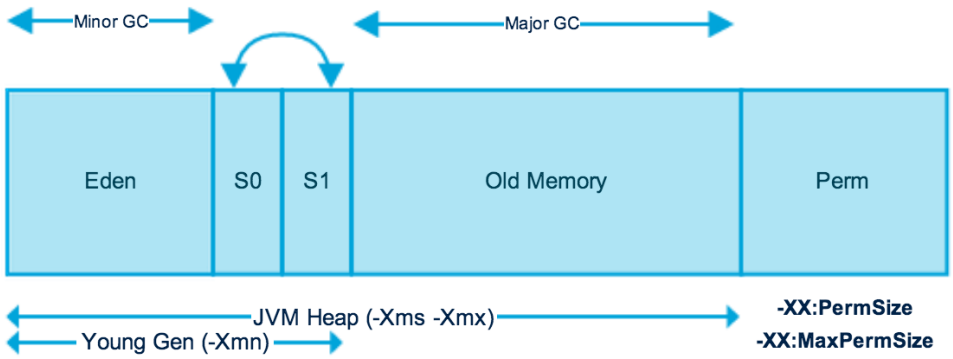
...programmatori un po' distratti !

JVM Memory Model



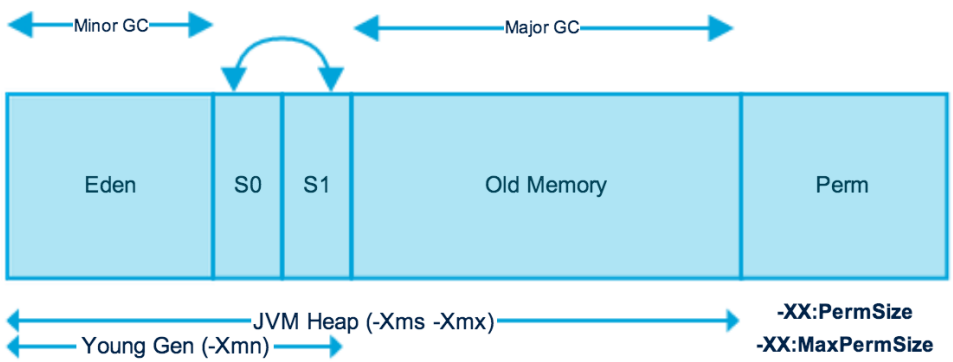
- La memoria della JVM è divisa in parti separate.
- A livello generale, la memoria Heap JVM è fisicamente divisa in due parti:
  - Young Generation e Old Generation.

Young Generation



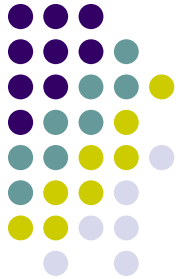
- È dove nascono tutti i nuovi oggetti.
- Quando è piena, viene eseguito il Garbage Collector detto *Minor GC*.
- Divisa in tre parti: Eden Memory e due spazi Survivor Memory.
- La maggior parte degli oggetti appena creati si trovano nello spazio della memoria dell'Eden.
- Quando lo spazio dell'Eden è pieno di oggetti, viene eseguito il Minor GC e tutti gli oggetti sopravvissuti vengono spostati in uno degli spazi sopravvissuti.
- Il Minor GC controlla anche gli oggetti sopravvissuti e li sposta nell'altro spazio sopravvissuto. Quindi alla volta, uno degli spazi sopravvissuti è sempre vuoto.
- Gli oggetti sopravvissuti dopo molti cicli di GC vengono spostati nello spazio di memoria di Old Generation. Di solito, ciò avviene impostando una soglia per l'età degli oggetti della Young Generation prima che diventino idonei a essere promossi alla Old generation.

Old Generation

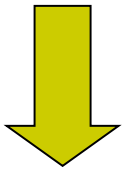


- Gli oggetti sopravvissuti dopo molti cicli di GC vengono spostati nello spazio di memoria di Old Generation.
- Di solito, ciò avviene impostando una soglia per l'età degli oggetti della Young Generation prima che diventino idonei a essere promossi alla Old generation.
- La Old Generation Memory contiene gli oggetti che hanno vissuto a lungo *(long-lived)* e sono sopravvissuti dopo molti round di Minor GC.
- Di solito, il garbage collector viene eseguito nella Old Generation Memory quando questa è piena.
- La Garbage Collection della Old Generation si chiama Major GC e solitamente richiede più tempo.

JVM: gestione della memoria

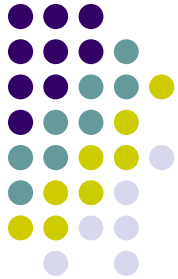


Prima che il “garbage collector” possa liberare la memoria occupata da un oggetto non più referenziato, invoca un particolare metodo: **finalize()** (… è un metodo della classe Object)



Ogni oggetto, dunque, deve fornire un servizio, **finalize()**, per il rilascio di tutte le risorse da lui acquisite durante la sua vita.
Tutti gli oggetti, in quanto sottoclassi della classe Object, hanno un’implementazione di default del metodo *finalize()*.

Memoria Heap e Stack Java

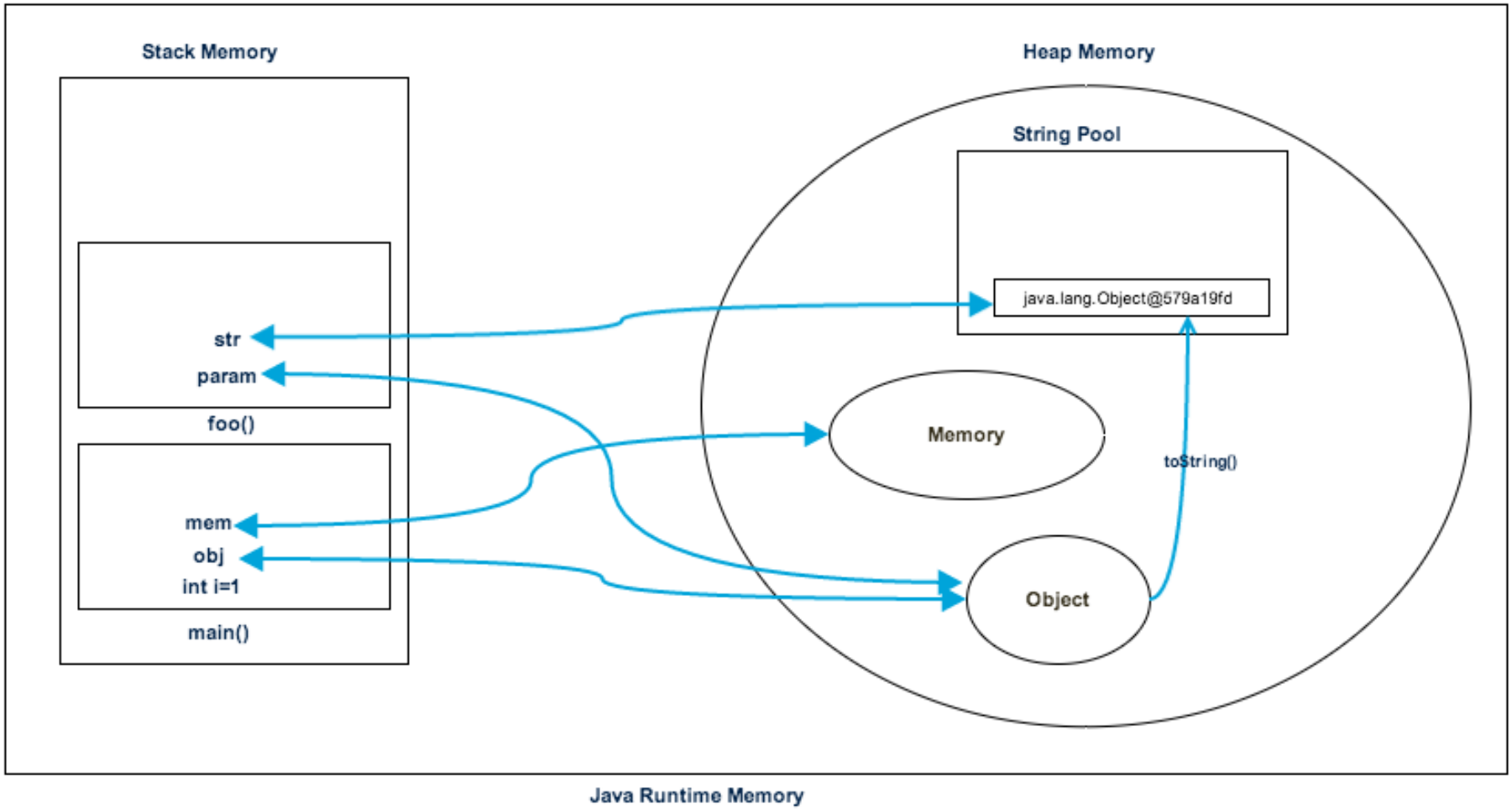


Analizziamo l'utilizzo della memoria Heap e Stack con un semplice programma.

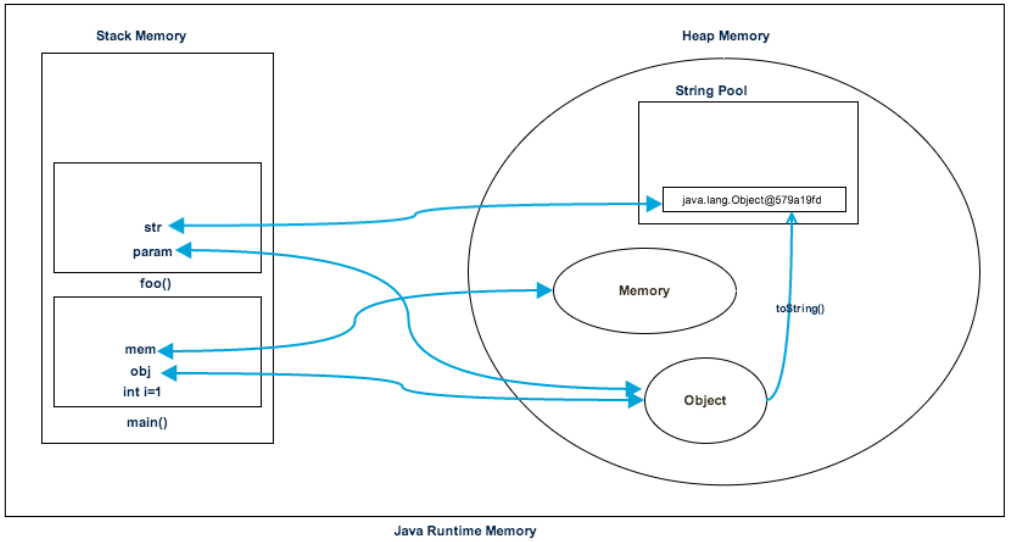
```
package com.journaldev.test;
public class Memory {
	public static void main(String[] args) { // Line 1
		int i=1; // Line 2
		Object obj = new Object(); // Line 3
		Memory mem = new Memory(); // Line 4
		mem.foo(obj); // Line 5
	} // Line 9

	private void foo(Object param) { // Line 6
		String str = param.toString(); // Line 7
		System.out.println(str);
	} // Line 8
}
```

https://www.digitalocean.com/community/tutorials/java-heap-space-vs-stack-memory#heap-and-stack-memory-in-java-program

Memoria Heap e Stack Java



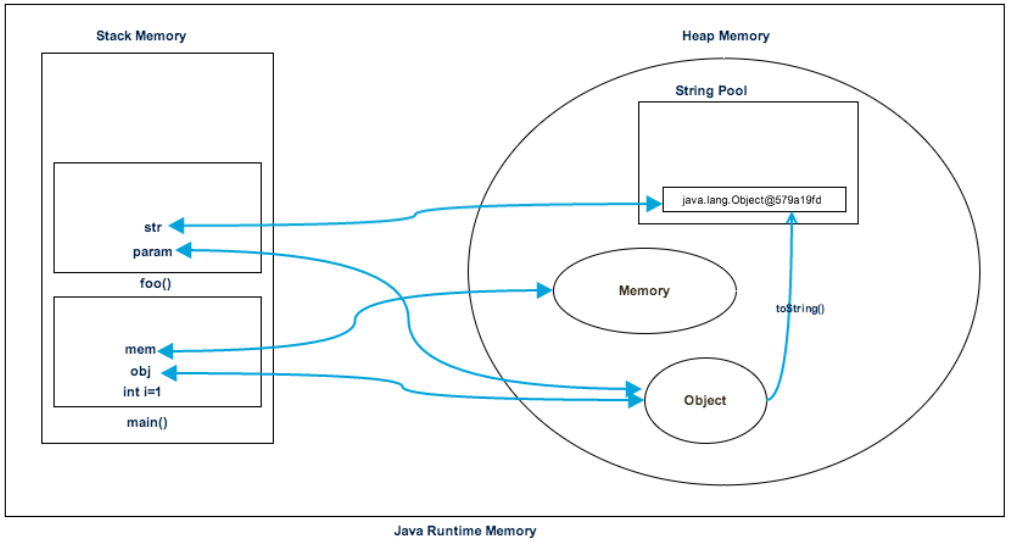
```
package com.journaldev.test;
public class Memory {
        public static void main(String[] args) { // Line 1
                int i=1; // Line 2
                Object obj = new Object(); // Line 3
                Memory mem = new Memory(); // Line 4
                mem.foo(obj); // Line 5
        } // Line 9

        private void foo(Object param) { // Line 6
                String str = param.toString(); // Line 7
                System.out.println(str);
        } // Line 8
}
```



Non appena eseguiamo il programma, vengono caricate tutte le classi Runtime nello spazio Heap.

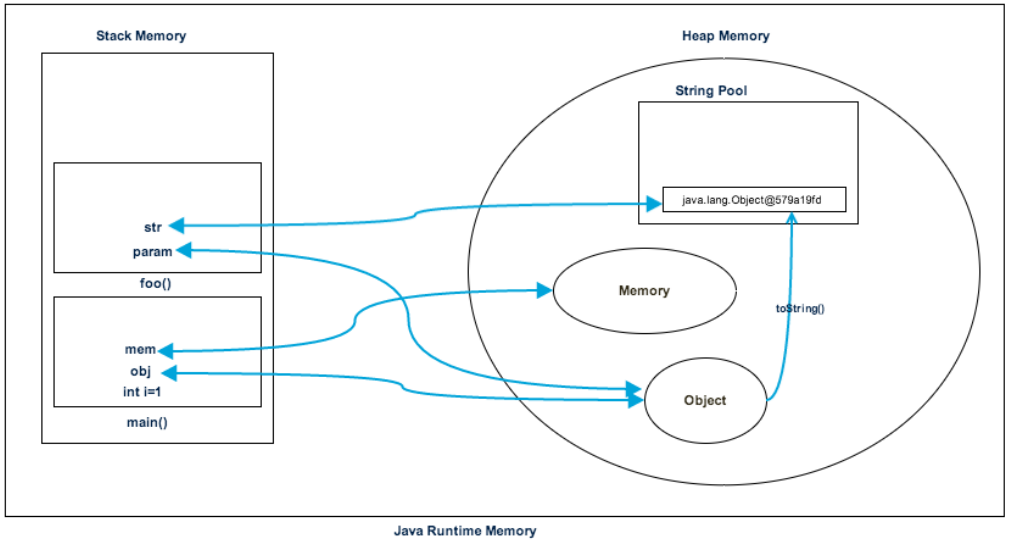
Quando il metodo main() viene trovato alla riga 1, la Java Runtime crea memoria stack che sarà utilizzata dal metodo main().

```
package com.journaldev.test;
public class Memory {
        public static void main(String[] args) { // Line 1
                int i=1; // Line 2
                Object obj = new Object(); // Line 3
                Memory mem = new Memory(); // Line 4
                mem.foo(obj); // Line 5
        } // Line 9

        private void foo(Object param) { // Line 6
                String str = param.toString(); // Line 7
                System.out.println(str);
        } // Line 8
}
```



Stiamo creando una variabile locale primitiva alla riga 2, quindi viene creata e archiviata nella memoria stack del metodo main().

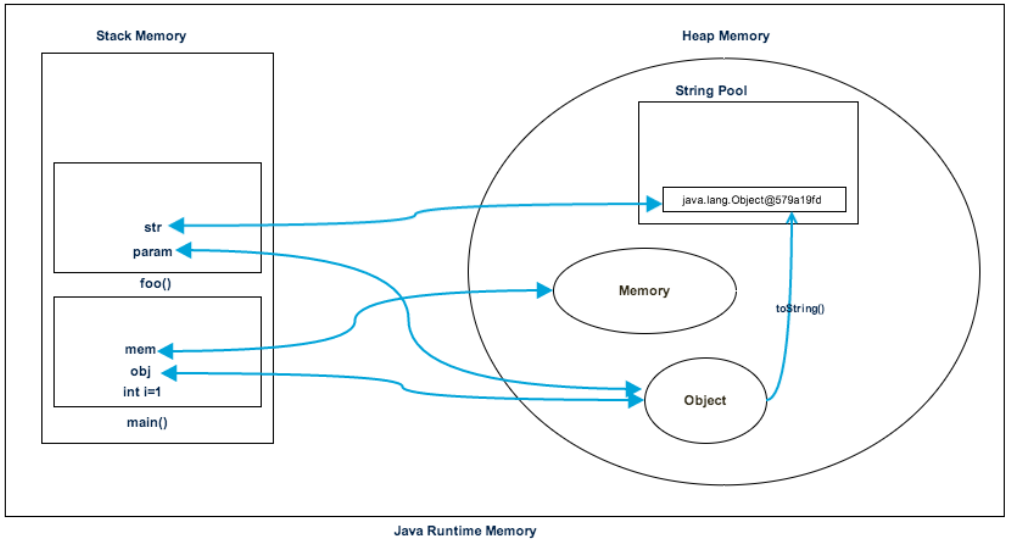
```
package com.journaldev.test;
public class Memory {
        public static void main(String[] args) { // Line 1
                int i=1; // Line 2
                Object obj = new Object(); // Line 3
                Memory mem = new Memory(); // Line 4
                mem.foo(obj); // Line 5
        } // Line 9

        private void foo(Object param) { // Line 6
                String str = param.toString(); // Line 7
                System.out.println(str);
        } // Line 8
}
```



Poiché stiamo creando un oggetto nella terza riga, questo viene creato nella memoria heap e la memoria stack ne contiene il riferimento.

Un processo simile si verifica quando creiamo l'oggetto Memory nella quarta riga.

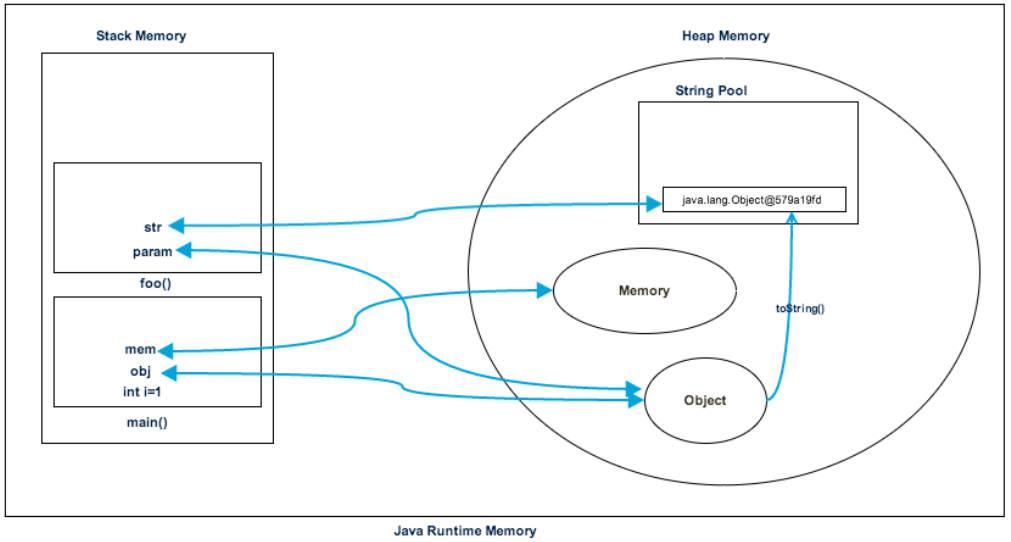
```
package com.journaldev.test;
public class Memory {
        public static void main(String[] args) { // Line 1
                int i=1; // Line 2
                Object obj = new Object(); // Line 3
                Memory mem = new Memory(); // Line 4
                mem.foo(obj); // Line 5
        } // Line 9

        private void foo(Object param) { // Line 6
                String str = param.toString(); // Line 7
                System.out.println(str);
        } // Line 8
}
```



Quando chiamiamo il metodo foo() nella quinta riga, viene creato un blocco in cima allo stack per essere utilizzato dal metodo foo().

Poiché Java è pass-by-value, viene creato un nuovo riferimento a Object nel blocco dello stack foo() nella sesta riga.

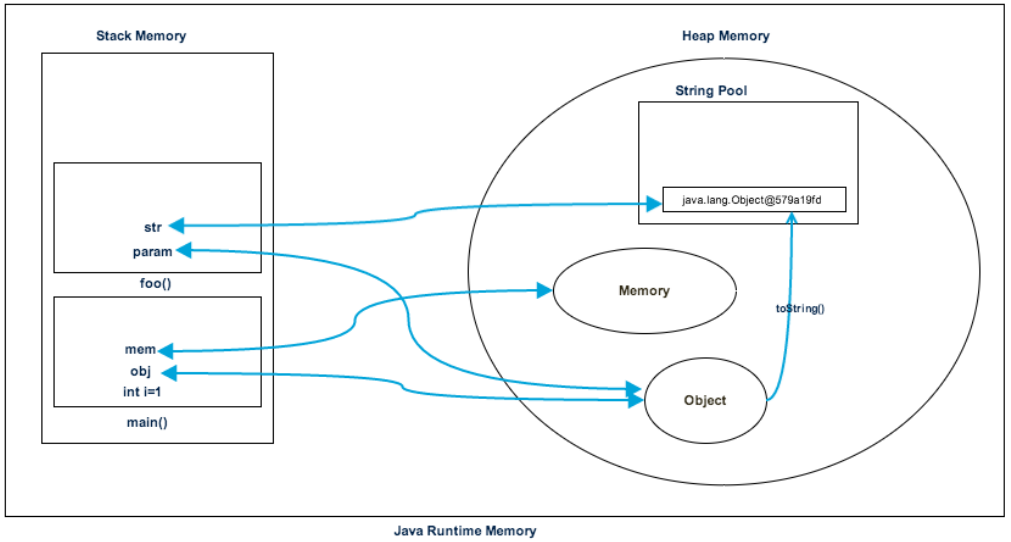
```
package com.journaldev.test;
public class Memory {
        public static void main(String[] args) { // Line 1
                int i=1; // Line 2
                Object obj = new Object(); // Line 3
                Memory mem = new Memory(); // Line 4
                mem.foo(obj); // Line 5
        } // Line 9

        private void foo(Object param) { // Line 6
                String str = param.toString(); // Line 7
                System.out.println(str);
        } // Line 8
}
```



Una stringa viene creata nella settima riga. Questa va nello String Pool nello spazio heap e viene creato un riferimento nello spazio dello stack foo() per essa.

Lo String Pool in Java è un pool di stringhe archiviate nella memoria heap Java.

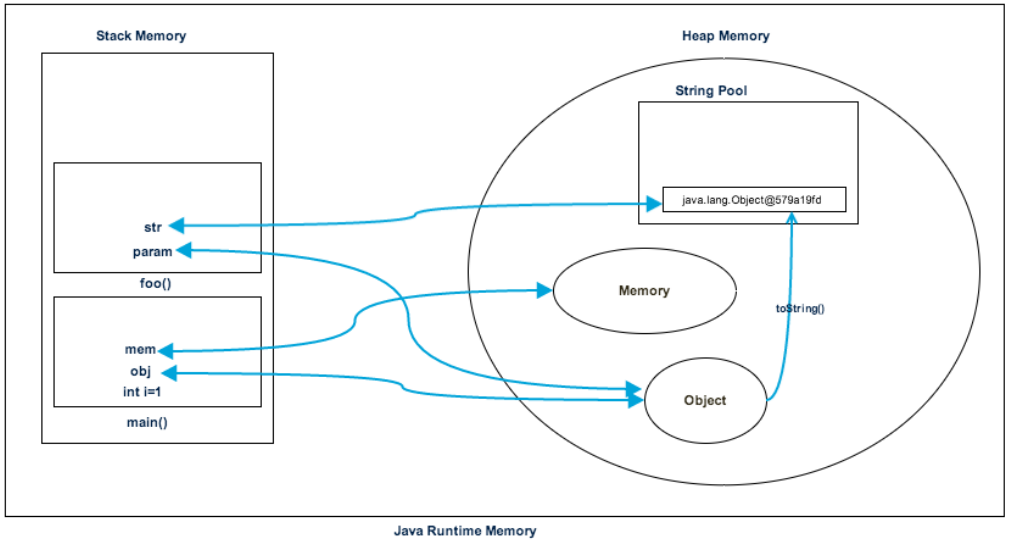
```
package com.journaldev.test;
public class Memory {
        public static void main(String[] args) { // Line 1
                int i=1; // Line 2
                Object obj = new Object(); // Line 3
                Memory mem = new Memory(); // Line 4
                mem.foo(obj); // Line 5
        } // Line 9

        private void foo(Object param) { // Line 6
                String str = param.toString(); // Line 7
                System.out.println(str);
        } // Line 8
}
```



L'esecuzione del metodo foo() termina all'ottava riga.

In questo momento il blocco di memoria allocato per foo() nello stack diventa libero.

```
package com.journaldev.test;
public class Memory {
        public static void main(String[] args) { // Line 1
                int i=1; // Line 2
                Object obj = new Object(); // Line 3
                Memory mem = new Memory(); // Line 4
                mem.foo(obj); // Line 5
        } // Line 9

        private void foo(Object param) { // Line 6
                String str = param.toString(); // Line 7
                System.out.println(str);
        } // Line 8
}
```



Nella riga 9, il metodo main() termina e la memoria dello stack creata per il metodo main() viene distrutta.

Inoltre, il programma termina su questa riga, quindi Java Runtime libera tutta la memoria e termina l'esecuzione del programma.