Programmazione 2

Lorenzo Pernigoni

1 Prima parte del programma

OGGETTO Struttura dati con **variabili d'istanza** (private, stato dell'oggetto) e **metodi** (solitamente tutti pubblici). Ogni oggetto è un'istanza di una classe.

CLASSE È un costrutto usato come modello per creare oggetti.
Costruttori - eseguiti al momento della creazione di un oggetto.
Variabili d'istanza - tipo e valori iniziali dello stato locale degli oggetti.
Metodi - operazioni che possono essere eseguite.

INTERFACCIA Definisce i metodi che un oggetto mette a disposizione.

INCAPSULAMENTO Non rendere visibile all'esterno l'implementazione dell'oggetto (Information Hiding).

METODO STATICO Appartiene alla classe e non può accedere a variabili d'istanza (non c'è this). In una classe è semplicemente una funzione che preso un parametro ritorna un risultato. In pratica è il meccanismo che mi permette di avere lo stile imperativo o funzionale nella programmazione a oggetti.

VARIABILI STATICHE Variabili accessibili tramite il nome della classe. Non possono essere inizializzate nel costruttore dato che non possono essere associate a istanze della classe. Sono accessibili solo da metodi statici. Sono globali a tutte le istanze della classe.

RIFERIMENTO Valore che identifica un determinato oggetto. In pratica è come se fosse un puntatore all'oggetto ma mentre in C si espone la sua natura di indirizzo con l'aritmetica dei puntatori, in Java non si dice come sono implementati i riferimenti, sono solo valori che identificano univocamente oggetti.
Pointer equality (o1 == o2) è vero se i due oggetti denotano lo stesso riferimento.
Deep equality (o1.equals(o2)) è vero se le due variabili denotano due oggetti identici.

EREDITARIETÀ Una superclasse generalizza una sottoclasse fornendo un comportamento che è condiviso dalle sottoclassi.

Se B è un sottotipo di A allora ogni oggetto che soddisfa l'interfaccia di B soddisfa anche l'interfaccia di A.

Un'istanza del sottotipo: soddisfa le proprietà del supertipo; può avere maggiori vincoli di quella del supertipo.

Permette il **polimorfismo del sottotipo**: grazie all'ereditarietà una variabile può assumere tipi (di classi) differenti.

CASTING B sottoclasse di A.
Upcasting - un oggetto di tipo B può essere legato a una variabile di tipo A.
Downcasting - un oggetto di tipo A può essere legato a una variabile di tipo B.

```
class Vehicle { ... }
class Car extends Vehicle { ... }
Vehicle v = new Car();          // upcasting (implicito)
Car c = (Car) new Vehicle();    // downcasting (esplicito)
```

TIPO STATICO E DINAMICO

- **Statico** è il tipo di una classe o interfaccia determinato dal compilatore. È l'espressione che descrive il valore calcolato in base alla struttura statica del programma.
- **Dinamico** è il tipo della classe di cui l'oggetto è istanza.

Nell'upcasting dell'esempio di prima il tipo statico è Vehicle mentre il tipo dinamico è Car.

ECCEZIONI Modo per gestire situazioni anomale a tempo di esecuzione.

- **Throw** (sollevare)

  ```
  if(myObj.equals(null)) throw new NullPointerException();
  ```

  L'argomento di throw è un oggetto che è sottotipo di Throwable.

  Try-Catch (catturare)

  ```
  try { <codice> } catch(IOException e) { <gestione eccezione> }
  ```

  Unchecked comportano la terminazione del programma; non obbligano il programmatore a gestirle.

  Checked nella dichiarazione del metodo:

  ```
  public void MyMethod throws Exception { ... }
  ```

  Eccezioni non fatali da gestire tramite la clausola try-catch.

DISPATCH

- **Static dispatch** (C++) l'operazione di individuazione del metodo da invocare è fatta staticamente (a tempo di compilazione).
- **Dynamic dispatch** (Java) il metodo da invocare durante l'esecuzione dipende dal tipo corrente associato all'oggetto (quindi dal tipo dinamico).

TIPI DI DATO ASTRATTO L'utente non conosce l'implementazione del tipo. L'unico modo che ha di accedere a quel tipo è tramite le operazioni che esso mette a disposizione. Non si tiene conto della rappresentazione ma delle funzionalità offerte.

Osservatori - metodi che prelevano delle parti dello stato mascherato dell'astrazione e le mettono a disposizione dell'utente.

Creatori e Produttori - .

Mutators - metodi che modificano la struttura del tipo di dato.

TDA Immutable non ha i mutators.

TDA Mutable ha i mutators dato che opera per effetti laterali.

GENERICI Un generics è uno strumento che permette la definizione di un tipo parametrizzato, che viene esplicitato successivamente in fase di compilazione secondo la necessità; i generics permettono di definire delle astrazioni sui tipi definiti nel linguaggio.

Convenzioni: <T> per Type, <E> per Element, <K> per Key, <V> per Value.

```
<T extends SuperType>                    // upper bound
<T extends ClassA & InterfB & InterfC>   // multiple upper bounds
<T super SubType>                        // lower bound
```

- **Covarianza, Controvarianza, Invarianza**

  A(T) è un tipo definito usando il tipo T e "<:" sta per sottotipo:

  A è covariante se T <: S ⇒ A(T) <: A(S).

  A è controvariante se T <: S ⇒ A(S) <: A(T).

  A è invariante se non è né covariante né controvariante.

- **Regole di Java**

  Se T2 è sottotipo di T3 allora T1<T2> non è sottotipo di T1<T3>.

  La nozione di sottotipo è invariante per le calssi generiche.

  e.g. Integer è sottotipo di Number. ArrayList<E> è sottotipo di List<E> che a sua volta è sottotipo di Collection<E>. List<Integer> non è sottotipo di List<Number>. Questa decisione è stata presa per garantire la **retrocompatibilità**.

- **Java Array**

  Se T1 è sottotipo di T2 allora T1[ ] è sottotipo di T2[ ]. È **covariante!** Questa scelta era stata fatta prima dei generici ed è stata mantenuta.

WILDCARD

Non esiste una compatibilità generale fra i tipi parametrici. Se si è alla ricerca della compatibilità, bisogna prendere in considerazione casi specifici e tipi di parametri di singoli metodi. Alla normale notazione dei tipi generici List<T> *, usata per creare oggetti, si affianca una nuova notazione, pensata per esprimere i tipi accettabili come parametri in singoli metodi. Si parla quindi di tipi parametrici varianti, in Java detti wildcard.*

List<? extends T> cattura le proprietà dei List<X> in cui X estende T, si usa per specificare tipi che possono essere solo letti (covariante).

List<? super T> cattura le proprietà dei List<X> in cui X è esteso da T, si usa per specificare tipi che possono essere solo scritti (controvariante).

List<?> cattura tutti i List<T> senza distinzioni, si usa per specificare i tipi che non consentono né letture né scritture (invariante).

TYPE ERASURE Tutti i tipi generici sono trasformati in Object nel processo di compilazione per motivi di **retrocompatibilità**. A runtime hanno tutti lo stesso tipo.

```
List<String> lst1 = new ArrayList<String>();
List<Integer> lst2 = new ArrayList<Integer>();
lst1.getClass() == lst2.getClass()                    // TRUE
```

SPECIFICA

- **Precondizione** proprietà che devono valere prima dell'invocazione di un metodo.

  @requires

- **Postcondizione** proprietà che devono valere dopo l'invocazione di un metodo.

  @modifies oggetti modificati durante l'esecuzione del metodo.

  @throws eccezioni che potrebbero essere sollevate.

  @effects proprietà dello stato finale.

- **Barriera di astrazione** meccanismo che ci permette di usare un programma senza conoscere il codice sorgente.
- **Invariante di rappresentazione** : Object → boolean

  Ci dice se la rappresentazione che abbiamo scelto è ben formata ovvero se rispetta i vincoli che volevamo dall'astrazione.

  Guida per chi implementa, modifica o verifica l'implementazione delle astrazioni: nessun oggetto deve violare il rep invariant.
- **Funzione di astrazione** : Object → abstractValue

  Come interpretare la struttura dati concreta dell'implementazione. È definita solo sui valori che rispettano il rep invariant.

  Guida per chi implementa o modifica l'astrazione: ogni operazione deve fare la cosa giusta con la rappresentazione concreta.
- **checkRep()** progettare codice in modo tale che alla chiamata del metodo si verifica il rep invariant e le precondizioni; all'uscita del metodo si verifica il rep invariant e le postcondizioni.

ESPORRE LA RAPPRESENTAZIONE C'è il rischio di dare in mano al cliente l'accesso diretto alle strutture dati. L'uso di *private* potrebbe non bastare, si potrebbe avere un aliasing delle strutture mutabili all'interno e all'esterno dell'astrazione.

Per evitare l'esposizione della rappresentazione una prima tecnica è quella di fare copie dei dati che oltrepassano la barriera di astrazione (caso di dati **mutable**).

Copy in - parametri che diventano valori della rappresentazione.

Copy out - risultati che sono parte dell'implementazione.

Una shallow copy non è sufficiente, a causa dell'aliasing si opta per una deep copy.

```
class Line {
    private Point s, e;
    public Line(Point s, Point e) {                    // copy in
        this.s = new Point(s.get(x), s.get(y));
        this.e = new Point(e.get(x), e.get(y));
    }
    public Point getStart() {                          // copy out
        return new Point(this.s.get(x), this.s.get(y));
    } }
```

L'alternativa è rendere i dati **immutable**. I vantaggi sono che l'aliasing non è un problema, non serve fare copie, rep invariant non può essere rotto. Lo svantaggio è che si possono comunque derivare informazioni su elementi della struttura.

OPEN / CLOSED Una classe è chiusa per modifiche perché la definizione delle astrazioni non cambia; però può essere estesa tramite sottoclassi, quindi è aperta per estensioni.

PRINCIPIO DI SOSTITUZIONE DELLA LISKOV Se una classe B estende una classe A vorrei usare la classe B in tutti i posti in cui poteva essere usata A senza che chi usa A si accorga che sta usando B.

In altre parole **se B è un sottotipo di A allora B può sempre sostituire A; ogni istanza di B può stare in tutti i posti dove può stare un'istanza di A**.

Variabili e metodi *private* non sono visibili a B, per fare altrimenti bisogna usare *protected*.

(1) Si applica a gerarchie di ereditarietà. (2) Si deve essere sicuri che le classi derivate estendono le classi di base senza cambiare il loro comportamento generale. (3) L'utente non si deve accorgere delle differenze tra tipo e sottotipo.

- **Regola della segnatura** gli oggetti del sottotipo devono avere tutti i metodi del supertipo; le segnature dei metodi del sottotipo devono essere compatibili con le segnature dei corrispondenti metodi del supertipo (tutto garantito dal compilatore Java).
- **Regola dei metodi** le chiamate dei metodi del sottotipo devono comportarsi come le chiamate dei corrispondenti metodi del supertipo.

```
/* REQUIRES: this non e' vuoto.
   EFFECTS: aggiunge 0 a this. */
public void addZero();
```

Precondizione indebolita : $\text{pre}_{\text{super}} \Rightarrow \text{pre}_{\text{sub}}$

```
/* EFFECTS: aggiunge 0 a this. */
public void addZero();              // sottotipo
```

Postcondizione rafforzata : $\text{pre}_{\text{super}} \wedge \text{post}_{\text{sub}} \Rightarrow \text{post}_{\text{super}}$

```
/* EFFECTS: se this non e' vuoto aggiunge 0 a this
   altrimenti aggiunge 1 a this. */
public void addZero();              // sottotipo
```

- **Regola delle proprietà** il sottotipo deve preservare tutte le proprietà che possono essere provate sugli oggetti del supertipo. Le proprietà sono dichiarate nell'OVERVIEW del supertipo.

ITERATORI Astrazione che permette di estrarre uno alla volta gli elementi di una collezione senza esporne la rappresentazione. Sono oggetti di classi che implementano la seguente interfaccia.

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove(); }

// tipico uso
    Iterator<Integer> itr = myIntCollection.iterator();
    while(itr.hasNext()) { int x = itr.next(); ... }
```

Inner class - si può dichiarare una classe all'interno di un'altra classe. La inner class è il meccanismo di generazione dell'iteratore per la classe esterna.

```
public class Primes implements Iterable<Integer> {
    public Iterator<Integer> iterator() {
        return new PrimeGen();
    }
    private static class PrimeGen implements Iterator<Integer> {
        ... // inner class statica
    } }
```

2 Seconda parte del programma

ASPETTI SIGNIFICATIVI

Astrazione sui dati - meccanismo che permette di nascondere le scelte di implementazione dei dati in modo da essere usati come se fossero primitivi.

Astrazione sul controllo - il linguaggio presenta delle primitive linguistiche che permettono a chi programma di definire il proprio meccanismo di esecuzione.

Astrazione di modularità - come fare un programma mettendo insieme vari pezzi.

MACCHINA ASTRATTA Sistema virtuale che rappresenta il comportamento di una macchina fisica individuando precisamente l'insieme delle risorse necessarie per l'esecuzione di programmi.

Per noi è una collezione di strutture dati e algoritmi in grado di memorizzare ed eseguire programmi. Le componenti sono: **interprete**, **memoria** (dati e programmi), **controllo**, **operazioni primitive**.

INTERPRETE *Programma in grado di eseguire altri programmi a partire direttamente dal codice sorgente senza compilazione, eseguendo cioè le istruzioni traducendole di volta in volta in istruzioni del linguaggio macchina.*

Programma che prende in ingresso l'albero di sintassi astratta di un programma e lo esegue ispezionandolo per vedere cosa deve essere fatto.

VALUTAZIONE

Eager - valutare un'espressione non appena viene legata a una variabile. Utilizzata principalmente nei linguaggi imperativi dove l'ordine di esecuzione è definito implicitamente dall'organizzazione del codice sorgente. **Lazy -** valutare un'espressione solo quando si chiede il valore di un'espressione dipendente da essa.

RUNTIME SUPPORT Collezione di strutture dati e sottoprogrammi caricati sulla macchina ospite per permettere l'esecuzione del codice prodotto dal compilatore.

SINTASSI La **sintassi concreta** di un linguaggio di solito è definita da una grammatica libera da contesto. La **sintassi astratta** è una rappresentazione lineare dell'albero sintattico dove gli operatori sono nodi dell'albero e gli operandi sono rappresentati dai sottoalberi.

SEMANTICA STATICA Proprietà del programma che si manifesta senza doverlo eseguire. Proprietà controllata e verificata dal compilatore o da altri strumenti (variabili non definite, programma corretto rispetto ai tipi, programma non contiene dead code).

TABELLA DEI SIMBOLI Struttura di supporto utilizzata da compilatori e interpreti per l'analisi statica. Ogni identificatore o simbolo nel codice sorgente di un programma è associato a informazioni relative alla sua dichiarazione.

Nel caso del nostro semplice linguaggio la tabella è costituita dall'insieme dei nomi delle variabili dichiarate nel programma. $\Gamma = \{x, y, ..., w\}$

SEMANTICA DINAMICA Specifica le regole di esecuzione del programma. Può includere effetti laterali e la produzione di valori. Noi usiamo la **semantica operazionale** che consiste nel definire la semantica in termini della macchina astratta.

Sistema di transizione - definisce la valutazione step-by-step di un programma. Insieme di **stati** S della macchina astratta; insieme I di stati **iniziali**; insieme F di stati **finali**; **relazione di transizione**, $\rightarrow \subseteq S \times S$, descrive l'effetto di un singolo passo di valutazione.

AMBIENTE Con binding si intende un'associazione nome/valore. L'ambiente è una collezione di binding, astrattamente è una funzione Ide $\rightarrow$ Value + Unbound.

DESCRITTORE Struttura che contiene la descrizione del tipo, usato quando si applica un'operazione al dato. Serve per controllare che il tipo del dato sia quello previsto dall'operazione (**type checking dinamico**) e per selezionare l'operatore giusto.

TYPE CHECKING

- Informazione sui tipi è nota completamente a tempo di compilazione (OCaml).

  Si possono eliminare i descrittori. **Type checking statico** in quanto è effettuato totalmente dal compilatore.

- Informazione sui tipi è nota solo a tempo si esecuzione (JavaScript).

  Servono descrittori per tutti i tipi di dato. **Type checking dinamico** in quanto è effettuato totalmente a tempo di esecuzione.

- Informazione sui tipi è nota solo parzialmente a tempo di compilazione (Java).

  I descrittori contengono solo l'informazione dinamica. Il type checking è effettuato in parte dal compilatore e in parte dal runtime support.

SCOPING

- **Scoping statico** la visibilità delle variabili dipende dalla struttura sintattica del programma, si guarda il codice precedente. (x = 1)
- **Scoping dinamico** si va a vedere l'ultima dichiarazione eseguita prima di invocare la funzione, non si amalgama bene con l'analisi statica. (x = 10)

```
let x = 1 in
    let f = fun (y : int) -> x + y in
        let x = 10 in f 3;;
```

CALL/RETURN DI SOTTOPROGRAMMI

Chiamante (1) Crea un'istanza del record di attivazione. (2) Salva lo stato dell'unità corrente di esecuzione. (3) Fa il passaggio dei parametri. (4) Inserisce il punto di ritorno. (5) Trasferisce il controllo al chiamato.

Chiamato - prologo (1) Salva il valore corrente dell'Environment Pointer (EP) e lo memorizza nel record di attivazione. (2) Definisce il nuovo valore di EP. (3) Alloca le variabili locali.

Chiamato - epilogo (1) Eventuale passaggio di valori. (2) Il valore calcolato dalla funzione è trasferito al chiamante. (3) Ripristina le info di controllo ovvero il vecchio valore

di EP salvato nel record di attivazione. (4) Ripristina lo stato di esecuzione del chiamante. (5) Trasferisce il controllo al chiamante.

INLINE-BLOCK

- **Record di attivazione** contiene control link, variabili locali e risultati intermedi.

  Control link - indirizzo di base (puntatore di catena dinamica) dell'AR precedente nello stack.

  Push AR - il valore di EP diventa il valore del control link del nuovo AR; modifica EP a puntare al nuovo AR.

  Pop AR - il valore del nuovo EP è ottenuto seguendo il control link.

- **Static scope -** i riferimenti non locali si risolvono nel più vicino blocco esterno nella struttura del programma.

  Dynamic scope - i riferimenti non locali si risolvono nell'AR precedente sullo stack.

  Nel caso di inline-block, scoping statico e dinamico **coincidono**.

FUNZIONI E PROCEDURE

- **Record di attivazione** contiene control link, parametri, indirizzo di ritorno, variabili locali e risultati intermedi, valore restituito, indirizzo del risultato di ritorno.

  Indirizzo di ritorno - indirizzo dell'istruzione da eseguire quando viene restituito il controllo al chiamante.

  Indirizzo del risultato di ritorno - indirizzo nell'AR del chiamante dove memorizzare il risultato.

- **Passaggio di parametri** y := x; L-value è la locazione, R-value è il contenuto della locazione.

  Per riferimento - memorizzare L-value nel record di attivazione; il corpo della funzione può modificare il valore del parametro attuale; aliasing tra parametro formale e attuale.

  Per valore - memorizzare R-value nel record di attivazione.

In C e Java si ha solo il passaggio per valore. Il passaggio per riferimento è simile al meccanismo che ho quando passo un puntatore o un riferimento a un oggetto perché il valore del puntatore o del riferimento è una locazione.

ANALISI STATICA Permette di tradurre ogni occorrenza di un nome x in una coordinata di distanza statica. x = <livello, offset> Il livello è il livello lessicale al quale appare la dichiarazione di x, l'offset è un indirizzo che identifica in modo univoco la posizione per x.

Modo molto efficiente per risolvere i riferimenti non locali senza dover fare ricerche con i nomi a runtime. Pago il costo di un'analisi statica nel back-end del compilatore ma ho un overhead minore a runtime.

(1) Ogni volta che l'analisi statica entra in un nuovo scope crea una nuova tabella dei simboli per tale scope. (2) Appena incontra le dichiarazione nello scope, inserisce le info nella tabella corrente. (3) Quando incontra un riferimento a una variabile, cerca nella tabella dello scope corrente. Se la tabella corrente non contiene una dichiarazione per il nome allora controlla la tabella per lo scope precedente.

SCOPING STATICO

Control link - puntatore all'AR che era in testa alla pila. **Static link -** puntatore all'AR che contiene il blocco più vicino che racchiude la dichiarazione del codice in esecuzione. In C non abbiamo funzioni nested (sono globali) quindi non serve lo static link.

Analisi - control link memorizza il flusso dinamico di esecuzione; static link dipende dalla struttura sintattica del programma.

Static depth - profondità statica della dichiarazione.
SD(Chiamante) = n, SD(Chiamato) = m ⇒ il chiamante deve fare n - m passi lungo la catena statica per definire il valore del puntatore della catena statica del chiamato.

CHIUSURA Valore di una funzione. Quando il parametro formale viene invocato si alloca sullo stack l'AR della funzione e si mette come valore del puntatore di catena statica il puntatore a env_dichiarazione. **closure = <env_dichiarazione, codice_fun>**

In altre parole il compilatore, nel caso di programmazione funzionale, associa alle funzioni un parametro implicito che è l'ambiente dove sono state dichiarate.

Nel caso di scoping dinamico abbiamo bisogno di chiusure solo nel caso in cui si abbiano funzioni come parametro o risultato perché si fa sempre riferimento all'ambiente del chiamante.

SCOPING DINAMICO Si usa solo il control link. Non si possono fare controlli prima di mandare il programma in esecuzione, quindi si avrà un overhead sia di tempo che di spazio (per i nomi) a runtime per il controllo dei tipi.

FUNZIONE COME RISULTATO Bisogna congelare l'ambiente dove la funzione è "dichiarata". Funzione "dichiarata" dinamicamente: può avere variabili non locali, restituisce una chiusura <env, code>, l'AR a cui punta env non può essere distrutto finché la funzione può essere usata (**retention**).

CLASSI E OGGETTI L'istanziazione (attivazione) di una classe avviene attraverso la chiamata del costruttore con la quale si passano alla classe eventuali parametri attuali e restituisce un oggetto. L'ambiente e la memoria locale dell'oggetto sono creati dalla valutazione delle dichiarazioni: di variabili che definiscono le istanze dell'oggetto (se ci sono allora l'oggetto ha memoria quindi uno stato modificabile); di funzioni che definiscono i metodi.

Tabella dei metodi - si associa all'oggetto un puntatore alla tabella dei metodi (o dispatch vector) che contiene il binding dei metodi (associazione nome/codice). Così riusciamo a utilizzare le informazioni sull'oggetto a runtime per individuare il codice del metodo giusto da eseguire.

Implementazione dei metodi - un metodo è eseguito come una funzione (vedi sopra). L'oggetto è un parametro implicito, quando un metodo è invocato gli viene passato anche un puntatore all'oggetto sul quale viene invocato; durante l'esecuzione del metodo il puntatore è il **this** del metodo.

JVM Il **Java Runtime** contiene la JVM, le librerie standard (API Java) e un launcher per eseguire i programmi già compilati in bytecode (linguaggio macchina della JVM).

La **JVM** è la macchina astratta di Java. È composta da loader, verifier, linker e interprete del bytecode.

CLASSLOADER All'inizio carica la classe Object (superclasse di ogni classe) e la classe che contiene il main. Il class loader, quando trova il riferimento a una classe non ancora caricata, la carica insieme a tutte quelle che stanno sopra di essa nella gerarchia.

ROOT SET Contiene le variabili statiche e quelle allocate sul runtime stack che puntano a oggetti sullo heap. Pertanto anche tutte le variabili locali del main che puntano allo heap appartengono al root set. Questa informazione è usata dal GC per determinare i dati ancora attivi, ovvero quelli raggiungibili anche indirettamente dal root set seguendo i puntatori.

GESTIONE DELLA MEMORIA

Static area - dimensione fissa, contenuti allocati a tempo di compilazione.
Runtime stack - dimensione variabile, record di attivazione.
Heap - supporto all'allocazione di oggetti e strutture dati dinamiche (malloc, new).

GARBAGE COLLECTOR Modalità automatica di gestione della memoria, mediante la quale si liberano porzioni di memoria non più utilizzate dalle applicazioni. Il GC esonera il programmatore dall'eseguire manualmente l'allocazione e la deallocazione di aree di memoria, eliminando alcuni bug: puntatori pendenti, doppia deallocazione, memory leak.

Reference counting - si aggiunge un contatore di riferimenti alle celle (numero di cammini di accesso attivi verso la cella). Overhead a runtime dato dalla gestione dei contatori. Il riuso delle celle libere è immediato, basta controllare se il contatore è 0. Non permette di gestire strutture dati con cicli interni.

Modello a grafo della memoria - si determina il root set, cioè l'insieme dei dati sicuramente attivi. Il Java root set comprende variabili statiche e variabili allocate sul runtime stack. Per ogni struttura dati allocata (sullo stack o heap) occorre sapere dove ci possono essere puntatori a elementi dello heap (info presente sui descrittori di tipo). I dati attivi raggiungibili corrispondono alla chiusura transitiva del grafo a partire dalle radici, ovvero tutti i dati raggiungibili dal root set seguendo i puntatori.

Cella - blocco di memoria sullo heap. Una cella è **live** sse appartiene ai dati attivi raggiungibili. Una cella è **garbage** se non è live. Il **GC** individua le celle garbage e le rende riutilizzabili inserendole nella lista libera.

Mark-and-sweep - ogni cella prevede un bit di marcatura. **Marking**: si parte dal root set e si marcano le celle live. **Sweep**: tutte le celle non marcate sono garbage, sono restituite alla lista libera e si resetta il bit di marcatura. Opera correttamente sulle strutture circolari, nessun overhead di spazio ma non interviene sulla frammentazione.

Algoritmo di Cheney (Copying collection) - suddivide lo heap in due parti: from-space e to-space. Solo una parte dello heap è attiva. Quando viene attivato il GC le celle live vengono copiate nella porzione dello heap non attiva; alla fine dell'operazione di copia la parte non attiva diventa attiva e viceversa. Le celle nella parte non attiva vengono restituite alla lista libera in un unico blocco evitando problemi di frammentazione.

GC Generazionale - si divide lo heap in un insieme di generazioni, il GC opera su quelle più giovani. Most cells that die, die young.
In Java si hanno tre generazioni, la 0 e la 2 copy collection, la 1 mark-and-sweep.

JIT COMPILER Compilazione effettuata durante l'esecuzione del programma piuttosto che precedentemente.