

APPUNTI DI JAVA - RECAP

1. Classi e Oggetti

Il codice mostra la definizione di una classe (Lesson01HelloWorld). Una classe è un modello che definisce la struttura e il comportamento di un oggetto. In questo caso, la classe contiene il metodo `main`.

Metodo Main

Il `main` è il punto di ingresso di un programma Java. Ha la firma `public static void main(String[] args)` e viene eseguito automaticamente all'avvio dell'applicazione. Il parametro `String[] args` consente di passare argomenti da linea di comando.

Console e Output

Il codice utilizza `Console.print()` per stampare del testo. Questo dimostra il concetto di **I/O (Input/Output)** - la comunicazione tra il programma e l'utente attraverso la console.

Stringhe Letterali

"Hello world" è una stringa letterale (literal string) - una sequenza di caratteri scritta direttamente nel codice e racchiusa tra doppi apici.

Librerie e Import

Siamo vivi per usarci →

In java le classi si usano a vicenda.

L'istruzione `import com.generation.library.Console;` mostra come importare classi da librerie esterne, estendendo le funzionalità base del linguaggio.

Commenti

Il codice utilizza commenti (//) per documentare il codice e spiegare i concetti, evidenziando l'importanza della documentazione nel codice.

Espressione: Definizione e Concetti

Un'**espressione** è qualunque costrutto di codice che **valuta a un valore**. Ha sempre un valore e un tipo.

Caratteristiche Principali

Restituisce un valore: L'elemento distintivo di un'espressione è che produce sempre un risultato che può essere assegnato, confrontato o utilizzato in altre operazioni.

Può contenere: variabili, costanti, operatori, chiamate a metodi, operazioni aritmetiche, logiche, ecc.

Esempi di Espressioni

- `5 + 3` → restituisce 8
- `x * 2` → restituisce il valore di `x` moltiplicato per 2
- `"Hello" + " world"` → restituisce "Hello world" (concatenazione stringhe)
- `Console.print("text")` → restituisce un valore (in questo caso `void`, ma è comunque un'espressione)
- `x > 5` → restituisce `true` o `false` (valore booleano)
- `Math.sqrt(16)` → restituisce 4.0

Cosa NON è un'Espressione

- **Dichiarazioni:** `int x;` (non restituisce valore)
- **Assegnazioni pure:** `x = 5;` (in molti linguaggi, anche se in alcuni lo sono)
- **Strutture di controllo:** `if, for, while` (non restituiscono valori)

Nel Contesto del Codice Java

Nel programma mostrato, `Console.print("Hello world")` è un'espressione che chiama un metodo e restituisce un valore (in questo caso `void`).

`String fullName = "Ferdinando" + " " + "Primerano";` → Espressione di tipo `String`

diventa: "Ferdinando Primerano" e il valore verrà salvato in `fullName`.

Variabile: Definizione e Concetti

Una **variabile** è un contenitore nominato che memorizza un valore in memoria durante l'esecuzione del programma.

Caratteristiche Principali

Nome: identificatore univoco per accedere al valore (es. `x`, `nome`, `età`)

Tipo: definisce il genere di dato che può contenere (es. `int`, `String`, `boolean`)

Valore: il dato effettivamente memorizzato, che può cambiare durante l'esecuzione

Indirizzo di memoria: ogni variabile occupa uno spazio specifico nella memoria del computer

Dichiarazione di una Variabile

`tipo nomeVariabile;`

Esempi:

- `int x;` → dichiara una variabile intera
- `String nome;` → dichiara una variabile stringa
- `boolean attivo;` → dichiara una variabile booleana

Inizializzazione

`tipo nomeVariabile = valore;`

Esempi:

- `int x = 10;`
- `String messaggio = "Hello";`
- `boolean acceso = true;`

Ciclo di Vita di una Variabile

1. **Dichiarazione:** la variabile viene creata
2. **Inizializzazione:** le viene assegnato un valore iniziale
3. **Utilizzo:** il valore viene letto o modificato
4. **Distruzione:** quando esce dall'ambito (scope), la memoria viene liberata

Scope (Ambito) → Lo scope definisce dove una variabile è accessibile:

- **Variabile locale:** dichiarata dentro un metodo, accessibile solo in quel metodo
- **Variabile di istanza:** dichiarata nella classe, accessibile in tutta la classe
- **Variabile di classe:** dichiarata con `static`, condivisa da tutte le istanze

Dichiarazione vs Assegnamento

Dichiarazione → Crea una variabile e le assegna uno spazio in memoria, specificandone il tipo.

Sintassi:

`tipo nomeVariabile;`

Esempi:

```
int x;  
String nome;  
boolean attivo;
```

Caratteristiche: Avviene una sola volta per variabile , Allocata memoria per la variabile , Dopo la dichiarazione, la variabile ha un valore indeterminato (non inizializzato) , In Java, le variabili non inizializzate hanno un valore di default (0 per int, null per String, false per boolean)

Assegnamento → Attribuisce un valore a una variabile già dichiarata (o contemporaneamente dichiarata).

Sintassi:

```
nomeVariabile = valore;
```

Esempi:

```
x = 5;  
nome = "Marco";  
attivo = true;
```

Caratteristiche: Può avvenire più volte , Modifica il contenuto della variabile , La variabile deve essere già dichiarata , Sovrascrive il valore precedente

Dichiarazione e Assegnamento Combinati → Spesso avvengono insieme:

```
int x = 5; // dichiarazione + assegnamento  
String nome = "Ana"; // dichiarazione + assegnamento
```

Differenza Pratica

```
int eta; // solo dichiarazione - eta non ha ancora valore utile  
eta = 25; // assegnamento - ora eta contiene 25  
eta = 30; // nuovo assegnamento - ora eta contiene 30
```

In sintesi: **dichiarazione** = creazione della variabile | **assegnamento** = attribuzione di un valore

Concordanza di Tipo: Approfondimento Teorico

Definizione → La concordanza di tipo (type agreement o type compatibility) è la proprietà che governa se un valore di un certo tipo di dato può essere utilizzato in un contesto che richiede un tipo di dato diverso. È il meccanismo che determina la validità di un'assegnazione, di un'operazione o di un passaggio di parametri dal punto di vista del sistema di tipi.

Sistema di Tipi e Compatibilità

Un **sistema di tipi** è l'insieme di regole che definisce quali operazioni sono lecite tra valori di tipi diversi. La concordanza di tipo è il criterio attraverso il quale il sistema di tipi valuta la correttezza di un'operazione.

In linguaggi **fortemente tipizzati** come Java, la concordanza di tipo è verificata dal compilatore prima dell'esecuzione del programma (**type checking statico**). Se la concordanza non esiste, il compilatore rifiuta il codice e genera un errore.

Concordanza Stretta vs Concordanza Debole

Concordanza Stretta → Due tipi hanno concordanza stretta quando sono **identici**. Un valore di tipo `int` può essere assegnato a una variabile di tipo `int`, e nessun'altra operazione è lecita senza una conversione esplicita. Questo approccio massimizza la sicurezza di tipo ma riduce la flessibilità.

Concordanza Debole → Due tipi hanno concordanza debole quando sono **compatibili secondo regole predefinite** del linguaggio, anche se non identici. Ad esempio, un valore di tipo `int` può essere assegnato a una variabile di tipo `long` senza conversione esplicita, perché esiste una relazione di compatibilità implicita tra i due tipi. Questo approccio bilancia sicurezza e flessibilità.

Gerarchia di Tipi e Concordanza per Ereditarietà

Nei linguaggi orientati agli oggetti, esiste una gerarchia di tipi basata sull'ereditarietà. Un sottotipo (classe derivata) è sempre concordante con il suo supertipo (classe base).

Questo principio, noto come Principio di Sostituzione di Liskov, stabilisce che un oggetto di una classe derivata può essere utilizzato ovunque sia atteso un oggetto della classe base. La concordanza è garantita dalla relazione di ereditarietà: il sottotipo "è un" supertipo.

Concatenazione di Stringhe: Approfondimento Teorico

Definizione → La concatenazione è l'operazione di unire due o più stringhe in una singola stringa. È una delle operazioni fondamentali sulla manipolazione di dati testuali nei linguaggi di programmazione.

Natura dell'Operatore +

L'operatore `+` è un operatore **polimorfo** o **sovraffunzionato**, ovvero il suo comportamento varia in base ai tipi di dato degli operandi:

- Con operandi numerici: esegue un'operazione aritmetica (addizione)
- Con almeno un operando di tipo stringa: esegue una concatenazione

Conversione di Tipo Implicita

Quando uno degli operandi è una stringa e l'altro non lo è, il linguaggio esegue una **conversione di tipo implicita** (coercion). Il valore non-stringa viene automaticamente convertito nella sua rappresentazione testuale prima di essere concatenato.

Questo meccanismo consente l'integrazione trasparente tra tipi di dato diversi senza richiedere conversioni esplicite da parte del programmatore.

Associatività e Valutazione

L'operatore `+` è **associativo da sinistra a destra**. In un'espressione con più operandi, l'operazione viene valutata sequenzialmente dal primo operando verso l'ultimo, producendo risultati intermedi che diventano operandi per la successiva operazione. Questo significa che l'ordine degli operandi influenza il risultato quando si mischiano stringhe e numeri, poiché il tipo dell'operando di sinistra determina come viene interpretato l'operando di destra.

Conversione Implicita a Stringa: Approfondimento Teorico

Definizione → La conversione implicita a stringa è il processo automatico mediante il quale il linguaggio trasforma un valore di tipo non-stringa nella sua rappresentazione testuale quando tale valore viene utilizzato in un contesto che richiede una stringa.

Meccanismo di Conversione

Quando un operatore richiede operandi di un tipo specifico, il linguaggio attiva un meccanismo di **coercion** (coercione di tipo). Nel caso della concatenazione con stringhe, il compilatore riconosce che uno degli operandi è una stringa e automaticamente converte l'altro operando nel suo equivalente testuale.

Questo è un esempio di **conversione di tipo automatica** che non richiede un'azione esplicita del programmatore, diversamente dalla **conversione esplicita** (casting) dove lo sviluppatore specifica manualmente la conversione.

Rappresentazione Testuale → Ogni tipo di dato ha una rappresentazione testuale canonica:

- Un intero 2025 diventa la stringa "2025"

- Un numero decimale 19.99 diventa la stringa "19.99"
- Un booleano true diventa la stringa "true"
- Un oggetto invoca il suo metodo `toString()` per ottenere la rappresentazione testuale

La conversione non è una semplice traduzione simbolica, ma il risultato di una funzione di trasformazione che mappa il valore nel dominio delle stringhe.

Contesto di Tipo

La conversione avviene per **inferenza di tipo** dal contesto. Quando il compilatore incontra l'operatore + con almeno un operando di tipo String, determina che l'intero risultato dell'operazione deve essere una stringa, e quindi applica le conversioni necessarie agli altri operandi.

Questo è un esempio del principio di **type inference** (inferenza di tipo), dove il sistema di tipi deduce quale conversione è appropriata analizzando il contesto dell'espressione.

Fondamento nel Metodo `toString()`

Teoricamente, ogni conversione implicita a stringa è equivalente a una chiamata al metodo `toString()` dell'oggetto. Il linguaggio fornisce implementazioni predefinite di `toString()` per i tipi primitivi che restituiscono la loro rappresentazione testuale standard. Questo meccanismo mostra come la conversione di tipo non sia una primitiva del linguaggio, ma si basi su interfacce e metodi definiti nel sistema di tipi.

Implicazioni Semantiche

La conversione implicita risolve un **conflitto di tipo**: l'operatore richiede operandi compatibili, ma gli operandi forniti sono di tipo differente. La conversione automatica consente al programma di proseguire senza errore, applicando una trasformazione semanticamente ragionevole. Tuttavia, questa comodità ha un costo: il programmatore deve comprendere quali conversioni avvengono implicitamente per evitare risultati inattesi.

Approfondimento Teorico: Operatori Booleani e Logica

1. Espressioni Booleane: Fondamenti Algebrici

Un'espressione booleana è un'espressione che appartiene al dominio dell'**algebra booleana**, un sistema algebrico dove i soli valori possibili sono due: vero e falso (true e false).

L'**algebra booleana**, sviluppata da George Boole nel XIX secolo, fornisce un framework matematico per ragionare su proposizioni logiche. Nel contesto della programmazione, un'espressione booleana è la materializzazione computazionale di una proposizione logica.

Una proposizione logica è un'affermazione che può essere valutata come vera o falsa, ma non può essere entrambi contemporaneamente (**principio di non-contraddizione**). Ogni espressione booleana rappresenta una proposizione il cui valore di verità è determinato dal contesto (i valori delle variabili coinvolte).

2. Operatori di Confronto: Relazioni tra Valori

Un **operatore di confronto** è un operatore binario che stabilisce una **relazione** tra due operandi e restituisce un valore booleano indicante se la relazione sussiste.

Gli operatori di confronto implementano **relazioni ordinali** sul dominio dei numeri:

- > (maggiore): relazione di ordine stretto
- >= (maggiore o uguale): relazione di ordine debole
- < (minore): relazione di ordine stretto inverso
- <= (minore o uguale): relazione di ordine debole inversa
- == (uguale): relazione di equivalenza
- != (diverso): negazione della relazione di equivalenza

Una **relazione di equivalenza** (==) è riflessiva (un valore è uguale a se stesso), simmetrica (se A=B allora B=A) e transitiva (se A=B e B=C allora A=C).

Una **relazione di ordine** è transitiva (se $A > B$ e $B > C$ allora $A > C$) e antisimmetrica (non può valere simultaneamente $A > B$ e $B > A$).

Il risultato di un confronto è un valore booleano che testimonia l'esito della valutazione della relazione.

3. Operatori Logici: Connettivi Proposizionali

Gli **operatori logici** sono operatori che combinano proposizioni logiche (espressioni booleane) secondo le regole della **logica proposizionale**, creando proposizioni composte.

Operatore AND (&&)

L'operatore AND è un **connettivo logico** che implementa la **congiunzione** in logica proposizionale. Se P e Q sono due proposizioni, allora "P AND Q" è vera se e solo se sia P che Q sono vere.

Formalmente, la tabella di verità di AND è:

P	Q	P AND Q
T	T	T
T	F	F
F	T	F
F	F	F

La congiunzione è **commutativa** ($P \text{ AND } Q = Q \text{ AND } P$) e **associativa** ($(P \text{ AND } Q) \text{ AND } R = P \text{ AND } (Q \text{ AND } R)$), proprietà che riflettono la struttura dell'algebra booleana.

Operatore OR (||)

L'operatore OR implementa la **disgiunzione** (inclusiva) in logica proposizionale. Se P e Q sono due proposizioni, allora "P OR Q" è vera se e solo se almeno una tra P e Q è vera.

Formalmente, la tabella di verità di OR è:

P	Q	P OR Q
T	T	T
T	F	T
F	T	T
F	F	F

La disgiunzione è anch'essa **commutativa** e **associativa**.

Nota che si tratta di disgiunzione **inclusiva**: quando entrambe le proposizioni sono vere, il risultato è vero. Esiste anche la disgiunzione esclusiva (XOR), dove il risultato è vero solo se esattamente una delle proposizioni è vera, ma non è rappresentata da un operatore standard in Java.

Operatore NOT (!)

L'operatore NOT implementa la **negazione** logica. Se P è una proposizione, allora "NOT P" è vera se P è falsa, e falsa se P è vera.

Formalmente, la tabella di verità di NOT è:

P	NOT P
T	F
F	T

La negazione è l'operatore logico **involutorio**: $\text{NOT}(\text{NOT}(P)) = P$.

Leggi della Logica Proposizionale

Gli operatori logici obbediscono a un insieme di **leggi algebriche** che permettono di trasformare e semplificare espressioni booleane:

Leggi di De Morgan

Queste leggi descrivono come la negazione interagisce con AND e OR:

- $\text{NOT}(P \text{ AND } Q) = (\text{NOT } P) \text{ OR } (\text{NOT } Q)$

- $\text{NOT}(P \text{ OR } Q) = (\text{NOT } P) \text{ AND } (\text{NOT } Q)$

Queste leggi consentono di eliminare negazioni applicate a espressioni composte, trasformandole in espressioni equivalenti.

Leggi di Idempotenza

- $P \text{ AND } P = P$
- $P \text{ OR } P = P$

Una proposizione combinata con se stessa tramite AND o OR produce la proposizione originale.

Leggi di Identità

- $P \text{ AND true} = P$
- $P \text{ OR false} = P$

La vera è l'elemento identità per AND, la falsa è l'elemento identità per OR.

Leggi di Dominanza

- $P \text{ AND false} = \text{false}$
- $P \text{ OR true} = \text{true}$

La falsa è l'elemento assorbente per AND, la vera è l'elemento assorbente per OR.

Legge del Terzo Escluso

- $P \text{ OR } (\text{NOT } P) = \text{true}$

Una proposizione o la sua negazione deve essere vera (non esiste una "terza" possibilità).

Legge della Non-Contraddizione

- $P \text{ AND } (\text{NOT } P) = \text{false}$

Una proposizione e la sua negazione non possono essere entrambe vere.

Confronto di Identità vs Equivalenza

Identità (Riferimento): Operatore ==

L'operatore == confronta i **riferimenti** (indirizzi di memoria) di due oggetti. Due variabili hanno lo stesso riferimento se puntano allo stesso oggetto in memoria.

Nel caso di stringhe, due stringhe create separatamente con lo stesso contenuto avranno riferimenti diversi:

```
String s1 = new String("hello");
String s2 = new String("hello");
s1 == s2 // false (riferimenti diversi)
```

Questo confronto implementa la **relazione di identità**: due entità sono identiche se sono lo stesso oggetto.

Equivalenza (Contenuto): Metodo equals()

Il metodo equals() confronta il **contenuto** di due oggetti. Per le stringhe, restituisce true se hanno la stessa sequenza di caratteri, indipendentemente da dove sono memorizzate:

```
String s1 = new String("hello");
String s2 = new String("hello");
s1.equals(s2) // true (contenuto identico)
```

Questo confronto implementa una **relazione di equivalenza semantica**: due entità sono equivalenti se hanno lo stesso significato, indipendentemente dalla loro identità fisica.

La distinzione è fondamentale: l'identità è una proprietà intima dell'oggetto (la sua locazione in memoria), mentre l'equivalenza è una proprietà semantica (il suo significato o valore).

Precedenza degli Operatori: Ordine di Valutazione

La **precedenza degli operatori** è una regola sintattica che determina l'ordine in cui gli operatori vengono valutati in un'espressione priva di parentesi esplicite.

In Java, la precedenza segue questo ordine (dal più alto al più basso):

1. **Operatori di accesso e invocazione:** ., (), []
2. **Operatori aritmetici:** *, /, % (moltiplicazione/divisione)
3. **Operatori aritmetici:** +, - (addizione/sottrazione)
4. **Operatori di confronto:** <, >, <=, >=
5. **Operatori di uguaglianza:** ==, !=
6. **Operatore AND logico:** &&
7. **Operatore OR logico:** ||

La precedenza è una convenzione che evita la necessità di scrivere parentesi esplicite in ogni espressione. Tuttavia, le parentesi hanno la precedenza massima e consentono di sovvertire l'ordine naturale.

L'espressione age >= 18 && age <= 70 è valutata come (age >= 18) && (age <= 70) perché i confronti hanno precedenza più alta di AND.

Short-Circuit Evaluation: Valutazione Pigra

La **valutazione pigra** (lazy evaluation) o **short-circuit evaluation** è un meccanismo di ottimizzazione dove il compilatore/interprete evita di valutare operandi che non sono necessari per determinare il risultato finale.

Per l'operatore AND (&&):

- Se il primo operando è falso, il risultato è necessariamente falso, indipendentemente dal secondo operando
- Quindi il secondo operando non viene valutato

Per l'operatore OR (||):

- Se il primo operando è vero, il risultato è necessariamente vero, indipendentemente dal secondo operando
- Quindi il secondo operando non viene valutato

Questa strategia ha due implicazioni importanti:

Efficienza: Evita la valutazione di espressioni potenzialmente costose. Ad esempio, se la prima condizione in un AND è falsa, non è necessario eseguire un calcolo complesso nella seconda condizione.

Correttezza semantica: Previene errori che sarebbero causati dalla valutazione del secondo operando. Se il secondo operando contiene un'operazione che causerebbe un'eccezione (ad esempio, una divisione per zero), la valutazione pigra garantisce che tale eccezione non si verifichi se non necessaria.

Nota che in Java gli operatori & e | (versioni non-short-circuit) valutano sempre entrambi gli operandi, a differenza di && e ||.

Composizionalità: Costruzione di Espressioni Complesse

La **composizionalità** è il principio per cui operazioni semplici possono essere combinate per formare operazioni più complesse, e il significato (semantica) dell'operazione complessa è determinato dal significato delle operazioni semplici e dal modo in cui sono combinate. Nel contesto delle espressioni booleane, questo significa che un'espressione complessa come:

```
(profession.equals("teacher") || profession.equals("soldier")) && age >= 18 && !origin.equals("north")
```

è composta da operazioni più semplici:

1. `profession.equals("teacher")` → booleano
2. `profession.equals("soldier")` → booleano
3. `age >= 18` → booleano
4. `origin.equals("north")` → booleano

Queste sono combinate tramite operatori logici, e il risultato finale è determinato sistematicamente dalla combinazione dei risultati intermedi secondo le regole della logica proposizionale.

Questa composizionalità è essenziale per la leggibilità e la manutenibilità del codice: permette di scrivere espressioni che riflettono la struttura logica del problema.

Letterali e Costanti: Valori Fissi nel Codice

Un **letterale** è una rappresentazione sintattica diretta di un valore nel codice sorgente. Non è una variabile (il cui valore può cambiare) né una costante (un identificatore che si riferisce a un valore), ma la diretta iscrizione del valore stesso.

I letterali stringhe ("teacher", "north") sono token sintattico che il compilatore converte in oggetti stringa in fase di compilazione o runtime. Il compilatore riconosce i delimitatori (virgolette) e il contenuto tra di essi come una stringa letterale.

La **semantica** di un letterale è il valore che rappresenta. Un letterale stringa ha sempre lo stesso significato: la sequenza di caratteri che contiene.

Assegnamento e Concordanza di Tipo Booleana

L'assegnamento di un valore a una variabile richiede **concordanza di tipo**: il tipo dell'espressione deve essere compatibile con il tipo dichiarato della variabile.

Nel caso di `boolean working = age >= 18 && age <= 70;`, la concordanza è garantita:

- L'espressione `age >= 18 && age <= 70` valuta sempre a un tipo booleano
- La variabile `working` è dichiarata di tipo boolean
- La concordanza è perfetta: non è necessaria nessuna conversione

Questo assegnamento trasferisce il valore booleano (il risultato della valutazione dell'espressione) nella variabile, che da quel punto in poi contiene quel valore fino a che non viene sovrascritta.

Strutture di Controllo: Approfondimento Teorico

Fondamenti Teorici del Controllo di Flusso

Le strutture di controllo sono costrutti che determinano l'ordine di esecuzione delle istruzioni in un programma. Senza di esse, un programma eseguirebbe sempre le istruzioni in sequenza lineare, rendendo impossibile implementare logiche condizionali e iterative.

Il flusso di controllo è il percorso che l'esecuzione del programma segue attraverso il codice. Le strutture di controllo modificano questo percorso in base a condizioni valutate a runtime.

Sequenza: Esecuzione Lineare

La sequenza è la struttura di controllo più primitiva e rappresenta l'ordine naturale di esecuzione delle istruzioni.

In termini di semantica operazionale, la sequenza è definita come:

```
esecuzione(istruzione1; istruzione2; istruzione3)
= esecuzione(istruzione1) seguito da esecuzione(istruzione2) seguito da
  esecuzione(istruzione3)
```

Ogni istruzione modifica lo stato del programma (i valori delle variabili in memoria), e la successiva istruzione opera sullo stato risultante.

Proprietà Algebriche

La sequenza non è commutativa: l'ordine importa. $A; B$ non è equivalente a $B; A$ perché la seconda istruzione dipende dallo stato creato dalla prima.

Formalmente: $A; B \neq B; A$ (non commutatività)

Selezione: Il Paradosma della Decisione

La selezione introduce il concetto di biforcazione condizionale nel flusso di controllo. Un programma che contiene solo sequenza è deterministico nel suo percorso: seguirà sempre lo stesso cammino. La selezione rende il programma non-deterministico dal punto di vista del codice, ma deterministico dal punto di vista dell'esecuzione: il percorso dipende dai dati.

Struttura Condizionale come Mapping

Una struttura condizionale può essere rappresentata matematicamente come:

```
se (P) allora esegui B1 altrimenti esegui B2  
=  
if P then B1 else B2  
=  
stato_finale = P ? B1(stato_iniziale) : B2(stato_iniziale)
```

Dove P è una proposizione logica (espressione booleana) e $B1, B2$ sono blocchi di istruzioni (funzioni che trasformano lo stato).

Completezza di Espressività

Con sole sequenze e selezioni, è possibile implementare qualunque algoritmo computabile (questo è correlato alla tesi di Church-Turing e alla nozione di Turing completezza).

Struttura if: Esecuzione Condizionata

Semantica Formale

```
if (P) {  
    B  
} = se P valuta a true, allora esegui B; altrimenti non eseguire nulla =  
stato_finale = P ? B(stato_iniziale) : stato_iniziale
```

Il blocco B è eseguito condizionatamente: la sua esecuzione dipende dalla verità di P .

Proprietà Logiche

Un if senza $else$ rappresenta un'implicazione logica:

- Se P è falso, il blocco non viene eseguito
- Se P è vero, il blocco deve essere eseguito

Questo è equivalente a: "Non fare nulla se falso, altrimenti prosegui con il blocco".

5. Struttura if-else: Bivalenza Decisionale

Semantica Formale

```
if (P)  
    B1  
else  
    B2  
= stato_finale = P ? B1(stato_iniziale) : B2(stato_iniziale)
```

L' if - $else$ introduce una dicotomia: esattamente uno dei due blocchi viene eseguito, a seconda del valore di verità di P .

Proprietà Algebriche

L'if-else è simmetrico rispetto alla negazione:

```
if (P)
    B1
else
    B2  ≡  if (!P) B2 else B1
```

Questa equivalenza mostra che la struttura è simmetrica: invertire la condizione e scambiare i blocchi produce il medesimo risultato.

Principio di Esclusione Mutua

La proprietà fondamentale dell'if-else è che i due blocchi sono in esclusione mutua: non possono essere eseguiti contemporaneamente. Esattamente uno è sempre eseguito.

Struttura if-else if-else: Partizione dello Spazio

Semantica Formale

```
if (P1)
    B1
else if (P2)
    B2
else if (P3)
    B3
else
```

B4

Questa struttura partitiona lo spazio delle possibilità in più regioni mutualmente esclusive:

- Regione 1: P1 è vero
- Regione 2: P1 è falso AND P2 è vero
- Regione 3: P1 è falso AND P2 è falso AND P3 è vero
- Regione 4: P1 è falso AND P2 è falso AND P3 è falso

Ordine di Valutazione

Le condizioni sono valutate sequenzialmente: P1 è valutato prima, poi P2, poi P3, ecc. Non appena una condizione risulta vera, il blocco corrispondente è eseguito e la valutazione termina.

Questo è un aspetto del corto-circuito già visto negli operatori logici.

Struttura switch: Dispatching su Valore

Semantica Formale → Lo switch è un caso speciale di selezione multipla dove il criterio di selezione è l'uguaglianza di un valore con più casi:

```
switch (E) {
    case v1:
        B1;
        break;
    case v2:
        B2;
        break;
    ...
    default:
        Bd;
}
```

È semanticamente equivalente a:

```
if (E == v1)
    B1
else if (E == v2)
    B2
...
else
    Bd
```

Ruolo di break

L'istruzione break è essenziale perché introduce una **semantica di esclusione**. Senza break, avviene il **fall-through**: l'esecuzione continua al case successivo anche se il valore non corrisponde.

Formalmente: **Con break**: l'esecuzione termina dopo il blocco , **Senza break**: l'esecuzione prosegue al prossimo case (comportamento non intenzionale nella maggior parte dei casi)

Limitazioni Teoriche

Lo switch funziona solo su valori discreti e **confrontabili per uguaglianza**. Non può essere usato per confronti relazionali ($>$, $<$, \geq , \leq). Questo è un vincolo sia sintattico che semantico del costrutto.

8. Operatore Ternario: Selezione come Espressione

Differenza Fondamentale da if-else

L'if-else è un **statement** (istruzione), mentre l'operatore ternario è un'**expression** (espressione).

Un'istruzione **modifica lo stato** del programma ma non restituisce un valore. Un'espressione **restituisce sempre un valore** che può essere usato in un contesto più ampio.

Semantica Formale

```
(P) ? E1 : E2
=
se P è vero, restituisci E1; altrimenti restituisci E2
=
valore = P ? E1(stato) : E2(stato)
```

A differenza dell'if-else dove i blocchi sono statement, in E1 e E2 sono **espressioni** che valutano a un valore.

Proprietà Algebriche

L'operatore ternario è **associativo a destra**:

```
a ? b : c ? d : e
=
a ? b : (c ? d : e)
```

Questo significa che i ternari annidati vengono associati dal fondo verso l'alto.

Limitazioni di Leggibilità

Sebbene teoricamente il ternario possa essere annidato indefinitamente, da un punto di vista **pratico e di leggibilità**, ternari annidati diventano rapidamente incomprensibili.

Il **principio di parsimonia cognitiva** suggerisce di usare ternari solo per selezioni semplici e binarie.

Completezza e Espressività

Teorema di Böhm-Jacopini → Il **teorema di Böhm-Jacopini** afferma che qualunque algoritmo computabile può essere espresso usando solo:

1. **Sequenza**
2. **Selezione** (if-else)
3. **Iterazione** (while)

Questo significa che le altre strutture (switch, operatore ternario) sono **zucchero sintattico** (syntactic sugar): offrono comodità e leggibilità, ma non aggiungono potenza computazionale.

Flusso di Controllo come Grafo

Il flusso di controllo di un programma può essere rappresentato come un **grafo diretto** dove:

- **Nodi** rappresentano blocchi di istruzioni
- **Archi** rappresentano transizioni di controllo (salti)

Una struttura if-else crea un **nodo di decisione** con due archi uscenti (uno per il ramo true, uno per il ramo false).

Uno switch crea un nodo di decisione con molteplici archi uscenti. Una sequenza è semplicemente una catena lineare di nodi. Questa rappresentazione grafica è fondamentale per analizzare la **complessità di controllo** di un programma.