Fondamenti di Informatica (Elettronici)

THINK JULIA – Capitolo 15

2 dicembre 2020

15. Strutture e oggetti [1]

[1] Tratto da https://benlauwens.github.io/ThinkJulia.jl/latest/book.html, disponibile sotto Licenza 'Creative Commons Attribution-NonCommercial 3.0 Unported'.

A questo punto sappiamo come usare le funzioni per organizzare il codice e i tipi predefiniti per organizzare i dati.

Il passaggio successivo consiste nell'apprendere come creare i propri tipi per organizzare sia il codice che i dati.

Questo è un argomento importante; ci vorranno alcuni capitoli per arrivarci.

Section 1

Tipi compositi

Introduzione

Abbiamo utilizzato molti dei tipi predefiniti di Julia; ora definiremo un nuovo tipo.

Come esempio, creeremo un tipo chiamato `Point` che rappresenta un punto nello spazio 2D.

Nella notazione matematica, i punti sono spesso scritti tra parentesi con una virgola che separa le coordinate.

Ad esempio, "`(0, 0)`" rappresenta l'origine e "`(x, y)`" rappresenta il punto "`x`" unità in orizzontale e "`y`" unità in verticale a partire dall'origine.

Introduzione

Abbiamo utilizzato molti dei tipi predefiniti di Julia; ora definiremo un nuovo tipo.

Come esempio, creeremo un tipo chiamato `Point` che rappresenta un punto nello spazio 2D.

Nella notazione matematica, i punti sono spesso scritti tra parentesi con una virgola che separa le coordinate.

Ad esempio, "`(0, 0)`" rappresenta l'origine e "`(x, y)`" rappresenta il punto "`x`" unità in orizzontale e "`y`" unità in verticale a partire dall'origine.

Ci sono diversi modi in cui potremmo rappresentare i punti in Julia:

- Potremmo memorizzare le coordinate separatamente in due variabili, `x` e `y`.

Introduzione

Abbiamo utilizzato molti dei tipi predefiniti di Julia; ora definiremo un nuovo tipo.

Come esempio, creeremo un tipo chiamato `Point` che rappresenta un punto nello spazio 2D.

Nella notazione matematica, i punti sono spesso scritti tra parentesi con una virgola che separa le coordinate.

Ad esempio, "`(0, 0)`" rappresenta l'origine e "`(x, y)`" rappresenta il punto "`x`" unità in orizzontale e "`y`" unità in verticale a partire dall'origine.

Ci sono diversi modi in cui potremmo rappresentare i punti in Julia:

- Potremmo memorizzare le coordinate separatamente in due variabili, `x` e `y`.
- Potremmo memorizzare le coordinate come elementi in un array o in una tupla.

Introduzione

Abbiamo utilizzato molti dei tipi predefiniti di Julia; ora definiremo un nuovo tipo.

Come esempio, creeremo un tipo chiamato `Point` che rappresenta un punto nello spazio 2D.

Nella notazione matematica, i punti sono spesso scritti tra parentesi con una virgola che separa le coordinate.

Ad esempio, "`(0, 0)`" rappresenta l'origine e "`(x, y)`" rappresenta il punto "`x`" unità in orizzontale e "`y`" unità in verticale a partire dall'origine.

Ci sono diversi modi in cui potremmo rappresentare i punti in Julia:

- Potremmo memorizzare le coordinate separatamente in due variabili, `x` e `y`.
- Potremmo memorizzare le coordinate come elementi in un array o in una tupla.
- Potremmo creare un nuovo tipo per rappresentare i punti come oggetti.

Introduzione

Abbiamo utilizzato molti dei tipi predefiniti di Julia; ora definiremo un nuovo tipo.

Come esempio, creeremo un tipo chiamato `Point` che rappresenta un punto nello spazio 2D.

Nella notazione matematica, i punti sono spesso scritti tra parentesi con una virgola che separa le coordinate.

Ad esempio, "`(0, 0)`" rappresenta l'origine e "`(x, y)`" rappresenta il punto "`x`" unità in orizzontale e "`y`" unità in verticale a partire dall'origine.

Ci sono diversi modi in cui potremmo rappresentare i punti in Julia:

- Potremmo memorizzare le coordinate separatamente in due variabili, `x` e `y`.
- Potremmo memorizzare le coordinate come elementi in un array o in una tupla.
- Potremmo creare un nuovo tipo per rappresentare i punti come oggetti.

Introduzione

Abbiamo utilizzato molti dei tipi predefiniti di Julia; ora definiremo un nuovo tipo.

Come esempio, creeremo un tipo chiamato `Point` che rappresenta un punto nello spazio 2D.

Nella notazione matematica, i punti sono spesso scritti tra parentesi con una virgola che separa le coordinate.

Ad esempio, "`(0, 0)`" rappresenta l'origine e "`(x, y)`" rappresenta il punto "`x`" unità in orizzontale e "`y`" unità in verticale a partire dall'origine.

Ci sono diversi modi in cui potremmo rappresentare i punti in Julia:

- Potremmo memorizzare le coordinate separatamente in due variabili, `x` e `y`.
- Potremmo memorizzare le coordinate come elementi in un array o in una tupla.
- Potremmo creare un nuovo tipo per rappresentare i punti come oggetti.

La creazione di un nuovo tipo è più complicata rispetto alle altre opzioni, ma presenta vantaggi che saranno presto evidenti.

`struct`: tipo composito definito dal programmatore

Un tipo composto definito dal programmatore è anche chiamato `struct`.

La definizione di `struct` per un `Point` ha questo aspetto:

```
struct Point
    x
    y
end
```

L'intestazione indica che la nuova `struct` è chiamata `Point`.

Il corpo definisce gli attributi o campi della struttura. La struttura "`Point`" ha due campi: "`x`" e "`y`".

Costruttore `struct`

Una `struct` è come una fabbrica per creare oggetti.

Per creare un punto si chiama `Point` come se fosse una funzione avente come argomenti i valori dei campi.

Quando "`Point`" viene utilizzato come funzione, viene chiamato costruttore.

```
julia> p = Point(3.0, 4.0)
Point(3.0, 4.0)
```

Il valore restituito è un riferimento a un oggetto `Point`, che assegniamo a `p`.

La creazione di un nuovo oggetto è chiamata istanza e l'oggetto è una istanza del tipo.

Diagramma degli oggetti e selettori

Quando stampi un'istanza, Julia ti dice a quale tipo appartiene e quali sono i valori degli attributi.

Ogni oggetto è un'istanza di qualche tipo, quindi "oggetto" e "istanza" sono intercambiabili.

Ma in questo capitolo usiamo "istanza" per indicare che stiamo parlando di un tipo definito dal programmatore.

Un diagramma di stato che mostra un oggetto e i suoi campi è chiamato diagramma degli oggetti:

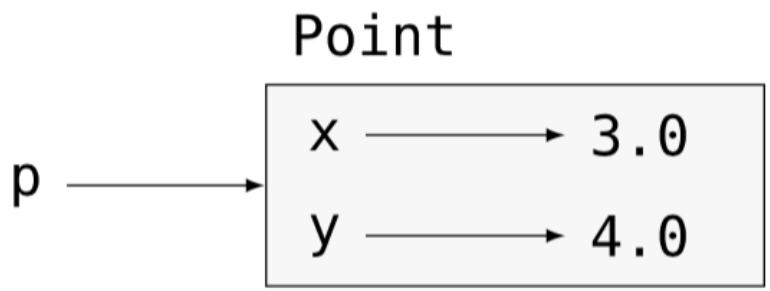

Figure 20. Object diagram

Section 2

Le strutture sono immutabili

Notazione Dot (.)

Puoi ottenere i valori dei campi usando la notazione "." (dot):

```
julia> x = p.x
3.0
julia> p.y
4.0
```

L'espressione "`p.x`" significa: vai all'oggetto a cui si riferisce `p` e ottieni il valore di "`x`".

Nell'esempio, assegniamo quel valore a una variabile denominata "`x`". Non c'è conflitto tra la variabile "`x`" e il campo "`x`".

Oggetto `struct` immutabile per impostazione predefinita

È possibile utilizzare la notazione dot come parte di qualsiasi espressione. Per esempio:

```
julia> distance = sqrt(p.x^2 + p.y^2)
5.0
```

Le strutture (oggetti `struct`) sono comunque immutabili di default; dopo la costruzione i campi non possono cambiare valore:

```
julia> p.y = 1.0
ERROR: setfield! immutable struct of type Point cannot be changed
```

All'inizio può sembrare strano, ma presenta diversi vantaggi:

- Può essere più efficiente.

Oggetto `struct` immutabile per impostazione predefinita

È possibile utilizzare la notazione dot come parte di qualsiasi espressione. Per esempio:

```
julia> distance = sqrt(p.x^2 + p.y^2)
5.0
```

Le strutture (oggetti `struct`) sono comunque immutabili di default; dopo la costruzione i campi non possono cambiare valore:

```
julia> p.y = 1.0
ERROR: setfield! immutable struct of type Point cannot be changed
```

All'inizio può sembrare strano, ma presenta diversi vantaggi:

- Può essere più efficiente.
- Non è possibile violare gli invarianti forniti dai costruttori del tipo (vedere Costruttori).

Oggetto `struct` immutabile per impostazione predefinita

È possibile utilizzare la notazione dot come parte di qualsiasi espressione. Per esempio:

```
julia> distance = sqrt(p.x^2 + p.y^2)
5.0
```

Le strutture (oggetti `struct`) sono comunque immutabili di default; dopo la costruzione i campi non possono cambiare valore:

```
julia> p.y = 1.0
ERROR: setfield! immutable struct of type Point cannot be changed
```

All'inizio può sembrare strano, ma presenta diversi vantaggi:

- Può essere più efficiente.
- Non è possibile violare gli invarianti forniti dai costruttori del tipo (vedere Costruttori).
- È più facile ragionare sul codice che utilizza oggetti immutabili.

Section 3

Strutture mutabili

Tipi compositi mutabili

Dove richiesto, i tipi compositi modificabili possono essere dichiarati con la parola chiave `mutable struct`. Ecco la definizione di un punto mutabile:

```
mutable struct MPoint
    x
    y
end
```

Puoi assegnare valori a un'istanza di una `struttura mutabile` usando la notazione punto:

```
julia> blank = MPoint(0.0, 0.0)
MPoint(0.0, 0.0)
julia> blank.x = 3.0
3.0
julia> blank.y = 4.0
4.0
```

Section 4

Rettangoli

Alternative di progetto

A volte è ovvio quali dovrebbero essere i campi di un oggetto, ma altre volte bisogna prendere decisioni.

Ad esempio, immagina di progettare un tipo per rappresentare i rettangoli.

Quali campi useresti per specificare la posizione e le dimensioni di un rettangolo?

- Puoi ignorare l'angolo; per mantenere le cose semplici,

Alternative di progetto

A volte è ovvio quali dovrebbero essere i campi di un oggetto, ma altre volte bisogna prendere decisioni.

Ad esempio, immagina di progettare un tipo per rappresentare i rettangoli.

Quali campi useresti per specificare la posizione e le dimensioni di un rettangolo?

- Puoi ignorare l'angolo; per mantenere le cose semplici,
- supponi che il rettangolo sia verticale o orizzontale.

Alternative di progetto

A volte è ovvio quali dovrebbero essere i campi di un oggetto, ma altre volte bisogna prendere decisioni.

Ad esempio, immagina di progettare un tipo per rappresentare i rettangoli.

Quali campi useresti per specificare la posizione e le dimensioni di un rettangolo?

- Puoi ignorare l'angolo; per mantenere le cose semplici,
- supponi che il rettangolo sia verticale o orizzontale.

Alternative di progetto

A volte è ovvio quali dovrebbero essere i campi di un oggetto, ma altre volte bisogna prendere decisioni.

Ad esempio, immagina di progettare un tipo per rappresentare i rettangoli.

Quali campi useresti per specificare la posizione e le dimensioni di un rettangolo?

- Puoi ignorare l'angolo; per mantenere le cose semplici,
- supponi che il rettangolo sia verticale o orizzontale.

Ci sono almeno due possibilità:

- È possibile specificare un (punto d') angolo del rettangolo (o il centro), la larghezza e l'altezza.

Alternative di progetto

A volte è ovvio quali dovrebbero essere i campi di un oggetto, ma altre volte bisogna prendere decisioni.

Ad esempio, immagina di progettare un tipo per rappresentare i rettangoli.

Quali campi useresti per specificare la posizione e le dimensioni di un rettangolo?

- Puoi ignorare l'angolo; per mantenere le cose semplici,
- supponi che il rettangolo sia verticale o orizzontale.

Ci sono almeno due possibilità:

- È possibile specificare un (punto d') angolo del rettangolo (o il centro), la larghezza e l'altezza.
- È possibile specificare due punti agli angoli opposti.

Implementazione scelta

A questo punto è difficile dire se una dei due sia migliore dell'altra, quindi implementeremo la prima, solo come esempio.

```
"""
Represents a rectangle.
fields: width, height, corner.
"""
struct Rectangle
    width
    height
    corner
end
```

La `docstring` elenca i campi: `width` e `height` sono numeri; `corner` è un oggetto "`Point`" che specifica la posizione del vertice inferiore sinistro.

Istanziazione di oggetti rettangolari

Per rappresentare un `rectangle`, devi istanziare un oggetto `Rectangle`:

```
julia> origin = MPoint(0.0, 0.0)
MPoint(0.0, 0.0)
julia> box = Rectangle(100.0, 200.0, origin)
Rectangle(100.0, 200.0, MPoint(0.0, 0.0))
```

Il diagramma dell'oggetto mostra lo "stato" di questo oggetto.

Un oggetto che è un "campo" (field) di un altro oggetto è incorporato.

Poiché l'attributo `corner` si riferisce a un oggetto modificabile, quest'ultimo viene disegnato all'esterno dell'oggetto `Rectangle`.

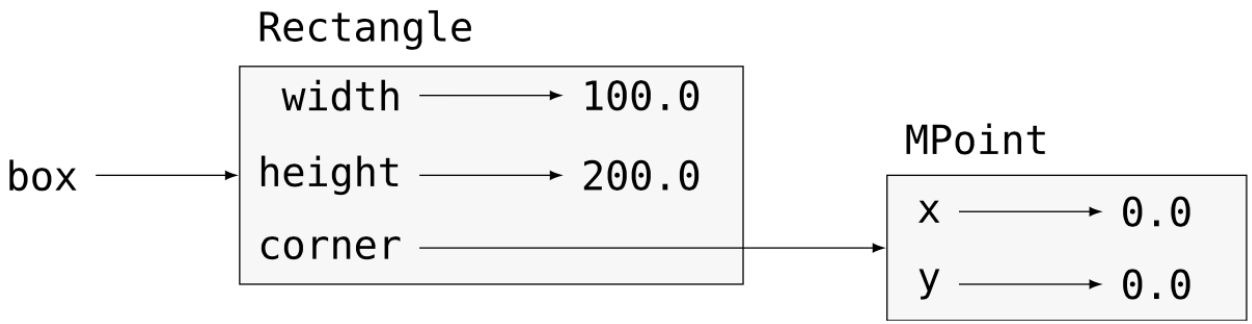

Figure 21. Object diagram

Section 5

Istanze come argomenti

Passaggio di istanza come argomento

Puoi passare un'istanza come argomento nel solito modo. Per esempio:

```
function printpoint(p)
    println("($(p.x), $(p.y))")
end
```

`printpoint` accetta un `Point` come argomento e lo visualizza in notazione matematica.

Per invocarlo, puoi passare "p" come argomento:

```
julia> p = (3.0, 4.0);
julia> printpoint(p)
(3.0, 4.0)
```

Esercizio 15-1

Scrivi una funzione chiamata `distancebetweenpoints` che prenda due punti come argomenti e restituisca la distanza tra loro.

Se un oggetto `mutable struct` viene passato a una funzione come argomento, la funzione può modificare i campi dell'oggetto.

Ad esempio___, `movepoint!`___ prende un oggetto mutabile `Point` e due numeri, `dx` e `dy`, e somma i numeri rispettivamente agli attributi `x` e `y` del `Point`:

```
function movepoint!(p, dx, dy)
    p.x += dx
    p.y += dy
    nothing
end
```

Ecco un esempio che dimostra l'effetto:

```
julia> origin = MPoint(0.0, 0.0)
MPoint(0.0, 0.0)
julia> movepoint!(origin, 1.0, 2.0)
julia> origin
MPoint(1.0, 2.0)
```

All'interno della funzione, `p` è un alias per `origin`, quindi quando la funzione modifica `p`, anche `origin` cambia.

Un oggetto immutabile non può essere modificato

Il passaggio di un oggetto immutabile `Point` a `movepoint!` provoca un errore:

```
julia> movepoint!(p, 1.0, 2.0)
ERROR: setfield! immutable struct of type Point cannot be char
```

È tuttavia possibile modificare il valore di un attributo modificabile di un oggetto immutabile.

Ad esempio, "`moverectangle!`" Ha come argomenti un oggetto "`Rectangle`" e due numeri, `dx` e `dy`, e usa "`movepoint!`" per spostare il "`corner`" del rettangolo:

```
function moverectangle!(rect, dx, dy)
    movepoint!(rect.corner, dx, dy)
end
```

Non riassegnare un campo mutabile di un oggetto immutabile

Ora `p` in `movepoint!` è un alias per `rect.corner`, quindi quando viene modificato `p`, cambia anche `rect.corner`:

```
julia> box
Rectangle(100.0, 200.0, MPoint(0.0, 0.0))
julia> moverectangle!(box, 1.0, 2.0)
julia> box
Rectangle(100.0, 200.0, MPoint(1.0, 2.0))
```

Non riassegnare un campo mutabile di un oggetto immutabile

Ora `p` in `movepoint!` è un alias per `rect.corner`, quindi quando viene modificato `p`, cambia anche `rect.corner`:

```
julia> box
Rectangle(100.0, 200.0, MPoint(0.0, 0.0))
julia> moverectangle!(box, 1.0, 2.0)
julia> box
Rectangle(100.0, 200.0, MPoint(1.0, 2.0))
```

AVVERTIMENTO

Non è possibile riassegnare un attributo modificabile di un oggetto immutabile:

```
julia> box.corner = MPoint(1.0, 2.0)
ERROR: setfield! immutable struct of type Rectangle cannot be
```

Section 6

Istanze come valori di ritorno

Le funzioni possono restituire istanze.

Ad esempio, la funzione `findcenter` accetta un rettangolo come argomento e restituisce un punto che contiene le coordinate del centro del rettangolo:

```
function findcenter(rect)
    Point(rect.corner.x + rect.width / 2, rect.corner.y + rect
end
```

L'espressione `rect.corner.x` significa "Vai all'oggetto a cui si riferisce" `rect` "e seleziona il campo denominato" `corner` "; quindi"vai a quell'oggetto e seleziona il campo denominato" `x`.

Ecco un esempio che passa "box" come argomento e assegna il "Point" risultante a "center":

```
julia> center = findcenter(box)
Point(51.0, 102.0)
```

Section 7

Copia

Aliasing vs deepcopy

L'aliasing può rendere un programma difficile da leggere perché le modifiche in un punto potrebbero avere effetti imprevisti in un altro.

È difficile tenere traccia di tutte le variabili che potrebbero fare riferimento a un determinato oggetto.

La copia di un oggetto è spesso un'alternativa all'aliasing.

Julia fornisce una funzione chiamata deepcopy che può duplicare qualsiasi oggetto:

```
julia> p1 = MPoint(3.0, 4.0)
MPoint(3.0, 4.0)
julia> p2 = deepcopy(p1)
MPoint(3.0, 4.0)
julia> p1 === p2
false
julia> p1 == p2
false
```

Per gli oggetti mutabili controlla l'identità, non l'equivalenza

L'operatore $\equiv$ indica che `p1` e `p2` non sono lo stesso oggetto, che è quello che ci aspettavamo.

Ma potresti aspettarti che "==" restituisca `true` perché questi punti contengono gli stessi dati.

In tal caso, rimarrai deluso nell'apprendere che per gli oggetti modificabili, il comportamento predefinito dell'operatore "==" è lo stesso dell'operatore "==="; controlla l'identità dell'oggetto, non l'equivalenza dell'oggetto.

Questo perché per i tipi compositi mutabili, Julia non sa cosa dovrebbe essere considerato equivalente. Almeno non ancora.

Per gli oggetti mutabili controlla l'identità, non l'equivalenza

L'operatore $\equiv$ indica che `p1` e `p2` non sono lo stesso oggetto, che è quello che ci aspettavamo.

Ma potresti aspettarti che "==" restituisca `true` perché questi punti contengono gli stessi dati.

In tal caso, rimarrai deluso nell'apprendere che per gli oggetti modificabili, il comportamento predefinito dell'operatore "==" è lo stesso dell'operatore "==="; controlla l'identità dell'oggetto, non l'equivalenza dell'oggetto.

Questo perché per i tipi compositi mutabili, Julia non sa cosa dovrebbe essere considerato equivalente. Almeno non ancora.

Esercizio 15-2

Crea un'istanza `Point`, creane una "copia" e controlla l'equivalenza e l'eguaglianza di entrambe. Il risultato può sorprenderti ma spiega perché aliasing non è un problema per un oggetto immutabile.

Section 8

Debugging

Con gli oggetti è probabile che incontri alcune nuove eccezioni

Quando inizi a lavorare con gli oggetti, potresti incontrare alcune nuove eccezioni.

Se provi ad accedere a un campo che non esiste, ottieni:

```
julia> p = Point(3.0, 4.0)
Point(3.0, 4.0)
julia> p.z = 1.0
ERROR: type Point has no field z Stacktrace:
    [1] setproperty!(::Point, ::Symbol, ::Float64) at ./sysimg.jl:19
    [2] top-level scope at none:0
```

Se non sei sicuro di quale tipo sia un oggetto, puoi chiedere:

```
julia> typeof(p)
Point
```

Puoi anche usare `isa` (`is-a` infisso) per verificare se un oggetto è un'istanza di un tipo:

```
julia> p isa Point
true
```

Funzioni `fieldnames` e `isdefined`

Se non sei sicuro che un oggetto abbia un particolare attributo, puoi usare la funzione predefinita `fieldnames`:

```
julia> fieldnames(Point) (:x, :y)
```

oppure la funzione `isdefined` :

```
julia> isdefined(p, :x)
true
julia> isdefined(p, :z)
false
```

Il primo argomento può essere qualsiasi oggetto; il secondo argomento è un simbolo, `:` seguito dal nome del campo (rappresenta un "simbolo" julia).

Section 9

Glossario

Glossario

struct Un tipo composito.

Glossario

struct Un tipo composito.

costruttore Una funzione con lo stesso nome di un tipo che crea istanze del tipo.

Glossario

struct Un tipo composito.

costruttore Una funzione con lo stesso nome di un tipo che crea istanze del tipo.

istanza (esempio) Un oggetto che appartiene a un tipo.

Glossario

struct Un tipo composito.

costruttore Una funzione con lo stesso nome di un tipo che crea istanze del tipo.

istanza (esempio) Un oggetto che appartiene a un tipo.

istanziare Per creare un nuovo oggetto.

Glossario

struct Un tipo composito.

costruttore Una funzione con lo stesso nome di un tipo che crea istanze del tipo.

istanza (esempio) Un oggetto che appartiene a un tipo.

istanziare Per creare un nuovo oggetto.

attributo o campo Uno dei valori denominati associati a un oggetto.

Glossario

struct Un tipo composito.

costruttore Una funzione con lo stesso nome di un tipo che crea istanze del tipo.

istanza (esempio) Un oggetto che appartiene a un tipo.

istanziare Per creare un nuovo oggetto.

attributo o campo Uno dei valori denominati associati a un oggetto.

oggetto incorporato Un oggetto memorizzato come campo di un altro oggetto.

Glossario

struct Un tipo composito.

costruttore Una funzione con lo stesso nome di un tipo che crea istanze del tipo.

istanza (esempio) Un oggetto che appartiene a un tipo.

istanziare Per creare un nuovo oggetto.

attributo o campo Uno dei valori denominati associati a un oggetto.

oggetto incorporato Un oggetto memorizzato come campo di un altro oggetto.

copia profonda Per copiare il contenuto di un oggetto, nonché qualsiasi oggetto incorporato e qualsiasi oggetto incorporato in essi e così via; implementato dalla funzione `deepcopy`.

Glossario

struct Un tipo composito.

costruttore Una funzione con lo stesso nome di un tipo che crea istanze del tipo.

istanza (esempio) Un oggetto che appartiene a un tipo.

istanziare Per creare un nuovo oggetto.

attributo o campo Uno dei valori denominati associati a un oggetto.

oggetto incorporato Un oggetto memorizzato come campo di un altro oggetto.

copia profonda Per copiare il contenuto di un oggetto, nonché qualsiasi oggetto incorporato e qualsiasi oggetto incorporato in essi e così via; implementato dalla funzione `deepcopy`.

diagramma degli oggetti Un diagramma che mostra gli oggetti, i loro campi e i valori dei campi.

Section 10

Esercizi

aaaa

aaaa