Conversii, conversii explicite(casting), tablouri.

Tipuri referinţă

Tipurile referinţă sunt folosite pentru a referi un obiect din interiorul unui alt obiect. În acest mod putem înlănţui informaţiile aflate în memorie.

Tipurile referinţă au, la fel ca şi toate celelalte tipuri o valoare implicită care este atribuită automat oricărei variabile de tip referinţă care nu a fost iniţializată. Această valoare implicită este definită de către limbajul Java prin cuvântul rezervat **null**.

Puteţi înţelege semnificaţia referinţei nule ca o referinţă care nu trimite nicăieri, a cărei destinaţie nu a fost încă fixată.

Simpla declaraţie a unei referinţe nu duce automat la rezervarea spaţiului de memorie pentru obiectul referit. Singura rezervare care se face este aceea a spaţiului necesar memorării referinţei în sine. Rezervarea obiectului trebuie făcută explicit în program printr-o expresie de alocare care foloseşte cuvântul rezervat **new**.

O variabilă de tip referinţă nu trebuie să trimită pe tot timpul existenţei sale către acelaşi obiect în memorie. Cu alte cuvinte, variabila îşi poate schimba locaţia referită în timpul execuţiei.

Clasa de memorare

Fiecare variabilă trebuie să aibă o anumită clasă de memorare. Această clasă ne permite să aflăm care este intervalul de existenţă şi vizibilitatea unei variabile în contextul execuţiei unui program.

Este important să înţelegem exact această noţiune pentru că altfel vom încerca să referim variabile înainte ca acestea să fi fost create sau după ce au fost distruse sau să referim variabile care nu sunt vizibile din zona de program în care le apelăm. Soluţia simplă de existenţă a tuturor variabilelor pe tot timpul execuţiei este desigur afară din discuţie atât din punct de vedere al eficienţei cât şi a eleganţei şi stabilităţii codului.

Variabile locale

Aceste variabile nu au importanţă prea mare în contextul întregii aplicaţii, ele servind la rezolvarea unor probleme locale. Variabilele locale sunt declarate, rezervate în memorie şi utilizate doar în interiorul unor blocuri de instrucţiuni, fiind distruse automat la ieşirea din aceste blocuri. Aceste variabile sunt vizibile doar în interiorul blocului în care au fost create şi în subblocurile acestuia.

Variabile statice

Variabilele statice sunt în general legate de funcţionalitatea anumitor clase de obiecte ale căror instanţe folosesc în comun aceste variabile. Variabilele statice sunt create atunci când codul specific clasei în care au fost declarate este încărcat în memorie şi nu sunt distruse decât atunci când acest cod este eliminat din memorie.

Valorile memorate în variabile statice au importanţă mult mai mare în aplicaţie decât cele locale, ele păstrând informaţii care nu trebuie să se piardă la dispariţia unei instanţe a clasei. De exemplu, variabila în care este memorat numărul de picioare al obiectelor din clasa Om nu

trebuie să fie distrusă la dispariția unei instanțe din această clasă. Aceasta din cauză că şi celelalte instanțe ale clasei folosesc aceeaşi valoare. Şi chiar dacă la un moment dat nu mai există nici o instanță a acestei clase, numărul de picioare ale unui Om trebuie să fie accesibil în continuare pentru interogare de către celelalte clase.

Variabilele statice nu se pot declara decât ca variabile ale unor clase şi conțin în declarație cuvântul rezervat **static**. Din cauza faptului că ele aparțin clasei şi nu unei anumite instanțe a clasei, variabilele statice se mai numesc uneori şi *variabile de clasă*.

Variabile dinamice

Un alt tip de variabile sunt variabilele a căror perioadă de existență este stabilită de către programator. Aceste variabile pot fi alocate la cerere, dinamic, în orice moment al execuției programului. Ele vor fi distruse doar atunci când nu mai sunt referite de nicăieri.

La alocarea unei variabile dinamice, este obligatoriu să păstrăm o referință către ea într-o variabilă de tip referință. Altfel, nu vom putea accesa în viitor variabila dinamică. În momentul în care nici o referință nu mai trimite către variabila dinamică, de exemplu pentru că referința a fost o variabilă locală şi blocul în care a fost declarată şi-a terminat execuția, variabila dinamică este distrusă automat de către sistem printr-un mecanism numit *colector de gunoaie*.

Colectorul de gunoaie poate porni din inițiativa sistemului sau din inițiativa programatorului la momente bine precizate ale execuției.

Pentru a rezerva spațiu pentru o variabilă dinamică este nevoie să apelăm la o *expresie de alocare* care foloseşte cuvântul rezervat **new**. Această expresie alocă spațiul necesar pentru un anumit tip de valoare. De exemplu, pentru a rezerva spațiul necesar unui obiect de tip Minge, putem apela la sintaxa:

```
Minge mingeaMea = new Minge();
```

iar pentru a rezerva spațiul necesar unui tablou de referințe către obiecte de tip Minge putem folosi declarația:

```
Minge echipament[] = new Minge[5];
```

Am alocat astfel spațiu pentru un tablou care conține 5 referințe către obiecte de tip Minge. Pentru alocarea tablourilor conținând tipuri primitive se foloseşte aceeaşi sintaxă. De exemplu, următoarea linie de program alocă spațiul necesar unui tablou cu 10 întregi, creând în acelaşi timp şi o variabilă referință spre acest tablou, numită *numere*:

```
int numere[] = new int[10];
```

Modelul memorie al obiectelor

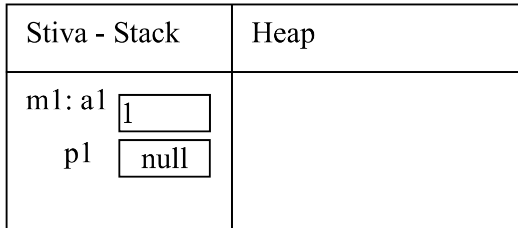
Structural, alocarea memoriei program în cadrul unui program Java divizează memoria în două regiuni:

- zonă de tip stivă – **F**irst **I**n **L**ast **O**ut – FILO – în care sunt memorate declarațiile variabilelor locale din cadrul metodelor şi al blocurilor.
- zonă heap – în care sunt memorate efectiv valorile atribuite referințelor din cadrul zonei stack. Această zonă este defapt un tablou uriaş de celule de memorie în care sunt efectiv stocate datele. O alocare de tip new, are ca efect nu numai introducerea numelui referință

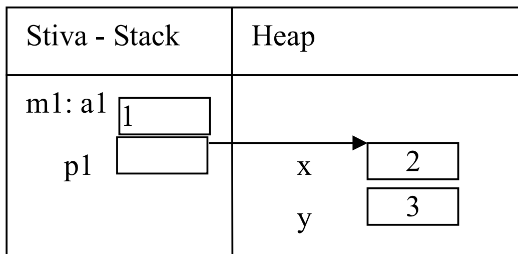
în cadrul zonei stack, dar şi rezervarea locaţiilor indicate de parametrii folosiţi în cadrul declaraţiei new, în cadrul zonei heap.

Spre exemplificare vom utiliza programul următor:

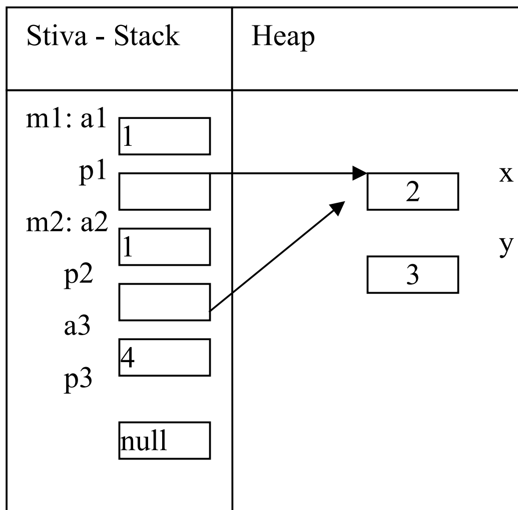
```
void m1() {
int a1 =1;
Point2D p1;
```




```
p1 = new Point2D(2,3);
```




```
m2(a1,p1);
a1=8;
}
void m2(int a2, Point p2){
int a3 = 4;
Point2D p3;
```




```
p3 = new Point2D(5,6);
a2=7;
}
```

Tablouri de variabile

Tablourile servesc, după cum spuneam, la memorarea secvenţelor de elemente de acelaşi tip. Tablourile unidimensionale au semnificaţia vectorilor de elemente. Se poate întâmpla să lucrăm şi cu tablouri de referinţe către tablouri, în acest caz modelul fiind acela al unei matrici bidimensionale. În fine, putem extinde definiţia şi pentru mai mult de două dimensiuni.

Declaraţia variabilelor de tip tablou

Pentru a declara variabile de tip tablou, trebuie să specificăm tipul elementelor care vor umple tabloul şi un nume pentru variabila referinţă care va păstra trimiterea către zona de memorie în care sunt memorate elementele tabloului.

Deşi putem declara variabile referinţă către tablou şi separat, de obicei declaraţia este făcută în acelaşi timp cu alocarea spaţiului ca în exemplele din paragraful anterior.

Sintaxa Java permite plasarea parantezelor drepte care specifică tipul tablou înainte sau după numele variabilei. Astfel, următoarele două declaraţii sunt echivalente:

```
int[] numere;
int numere[];
```

Dacă doriţi să folosiţi tablouri cu două dimensiuni ca matricile, puteţi să declaraţi un tablou de referinţe către tablouri cu una dintre următoarele trei sintaxe echivalente:

```
float[][] matrice;
float[] matrice[];
float matrice[][];
```

De precizat că şi în cazul dimensiunilor multiple, declaraţiile de mai sus nu fac nimic altceva decât să rezerve loc pentru o referinţă şi să precizeze numărul de dimensiuni. Alocarea spaţiului pentru elementele tabloului trebuie făcută explicit.

Despre rezervarea spaţiului pentru tablourile cu o singură dimensiune am vorbit deja. Pentru tablourile cu mai multe dimensiuni, rezervarea spaţiului se poate face cu următoarea sintaxă:

```
byte [][]octeti = new byte[23][5];
```

În expresia de alocare sunt specificate în clar numărul elementelor pentru fiecare dimensiune a tabloului.

Iniţializarea tablourilor.

Limbajul Java permite şi o sintaxă pentru iniţializarea elementelor unui tablou. Într-un astfel de caz este rezervat automat şi spaţiul de memorie necesar memorării valorilor iniţiale. Sintaxa folosită în astfel de cazuri este următoarea:

```
char []caractere = { a, b, c, d };
```

Acest prim exemplu alocă spaţiu pentru patru elemente de tip caracter şi iniţializează aceste elemente cu valorile dintre acolade. După aceea, creează variabila de tip referinţă numită *caractere* şi o iniţializează cu referinţa la zona de memorie care păstrează cele patru valori.

Iniţializarea funcţionează şi la tablouri cu mai multe dimensiuni ca în exemplele următoare:

```
int [][]numere = {
{ 1, 3, 4, 5 },
{ 2, 4, 5 },
{ 1, 2, 3, 4, 5 }
};
double [][][]reali = {
{ { 0.0, -1.0 }, { 4.5 } },
{ { 2.5, 3.0 } }
};
```

După cum observaţi numărul iniţializatorilor nu trebuie să fie acelaşi pentru fiecare element.

Lungimea tablourilor

Tablourile Java sunt alocate dinamic, ceea ce înseamnă că ele îşi pot schimba dimensiunile pe parcursul execuţiei. Pentru a afla numărul de elemente dintr-un tablou, putem apela la următoarea sintaxă:

```
float []tablou = new float[25];
int dimensiune = tablou.length;
// dimensiune primeşte valoarea 25
```

sau

```
float [][]multiTablou = new float[3][4];
int dimensiune1 = multiTablou[2].length;
// dimensiune1 primeşte valoarea 4
int dimensiune2 = multiTablou.length;
// dimensiune2 primeşte valoarea 3
```

Referirea elementelor din tablou

Elementele unui tablou se pot referi prin numele referinţei tabloului şi indexul elementului pe care dorim să-l referim. În Java, primul element din tablou este elementul cu numărul 0, al doilea este elementul numărul 1 şi aşa mai departe.

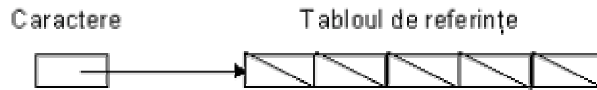
Sintaxa de referire foloseşte parantezele pătrate **[** şi **]**. Între ele trebuie specificat indexul elementului pe care dorim să-l referim. Indexul nu trebuie să fie constant, el putând fi o expresie de complexitate oarecare.

Iată câteva exemple:

```
int []tablou = new int[10];
tablou[3] = 1;
// al patrulea element primeşte valoarea 1
float [][]reali = new float[3][4];
reali[2][3] = 1.0f;
// al patrulea element din al treilea tablou
// primeşte valoarea 1
```

În cazul tablourilor cu mai multe dimensiuni, avem în realitate tablouri de referințe la tablouri. Asta înseamnă că dacă considerăm următoarea declarație:

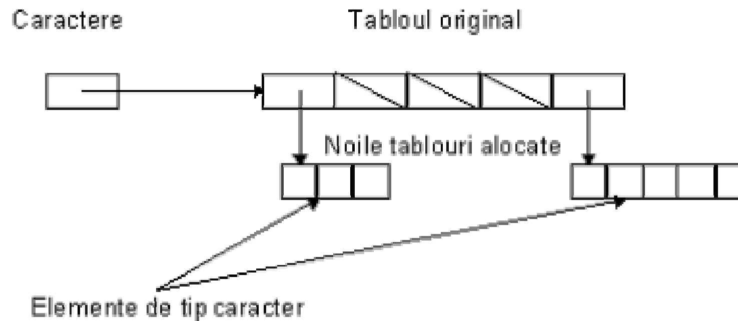
```
char [][]caractere = new char [5][];
```



Elementele tabloului sunt de tip referință, inițializate implicit la valoarea null.

Variabila referință numită *caractere* conține deocamdată un tablou de 5 referințe la tablouri de caractere. Cele cinci referințe sunt inițializate cu **null**. Putem inițializa aceste tablouri prin atribuiri de expresii de alocare:

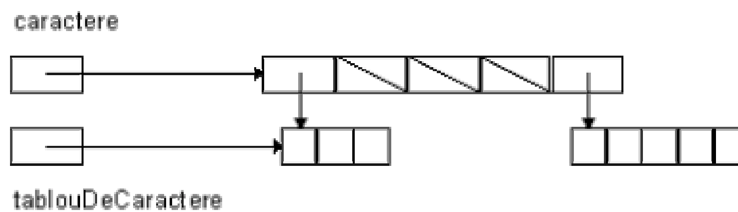
```
caractere[0] = new char [3];
caractere[4] = new char [5];
```



Noile tablouri sunt referite din interiorul tabloului original. Elementele noilor tablouri sunt caractere.

La fel, putem scrie:

```
char []tablouDeCaractere = caractere[0];
```



Variabilele de tip referință *caractere[0]* şi *tablouDeCaractere* trimit spre acelaşi tablou rezervat în memorie.

Variabila *tablouDeCaractere* trimite către acelaşi tablou de caractere ca şi cel referit de primul element al tabloului referit de variabila *caractere*.

Să mai precizăm că referirea unui element de tablou printr-un index mai mare sau egal cu lungimea tabloului duce la oprirea execuției programului cu un mesaj de eroare de execuție corespunzător.

Alocarea şi eliberarea tablourilor

Despre alocarea tablourilor am spus deja destul de multe. În cazul în care nu avem inițializatori, variabilele sunt inițializate cu valorile implicite definite de limbaj pentru tipul corespunzător. Aceasta înseamnă că, pentru tablourile cu mai multe dimensiuni, referințele sunt inițializate cu **null**.

Pentru eliberarea memoriei ocupate de un tablou, este suficient să tăiem toate referințele către tablou. Sistemul va sesiza automat că tabloul nu mai este referit şi mecanismul colector de gunoaie va elibera zona. Pentru a tăia o referință către un tablou dăm o altă valoare variabilei care referă tabloul. Valoarea poate fi **null** sau o referință către un alt tablou.

De exemplu:

```
float []reali = new float[10];
?
reali = null; // eliberarea tabloului
sau
reali = new float[15]; // eliberarea în alt fel
sau
{
float []reali = new float[10];
?
}// eliberare automată, variabila reali a fost
// distrusă la ieşirea din blocul în care a
// fost declarată, iar tabloul de 10 flotanți
// nu mai este referit
```

Conversii

Operațiile definite în limbajul Java au un tip bine precizat de argumente. Din păcate, există situații în care nu putem transmite la apelul acestora exact tipul pe care compilatorul Java îl aşteaptă. În asemenea situații, compilatorul are două alternative: fie respinge orice operație cu argumente greşite, fie încearcă să convertească argumentele către tipurile necesare. Desigur, în cazul în care conversia nu este posibilă, singura alternativă rămâne prima.

În multe situații însă, conversia este posibilă. Să luăm de exemplu tipurile întregi. Putem să convertim întotdeauna un întreg scurt la un întreg. Valoarea rezultată va fi exact aceeaşi. Conversia inversă însă, poate pune probleme dacă valoarea memorată în întreg depăşeşte capacitatea de memorare a unui întreg scurt.

În afară de conversiile implicite, pe care compilatorul le hotărăşte de unul singur, există şi conversii explicite, pe care programatorul le poate forța la nevoie. Aceste conversii efectuează de obicei operații în care există pericolul să se piardă o parte din informații. Compilatorul nu poate hotărî de unul singur în aceste situații.

Conversiile implicite pot fi un pericol pentru stabilitatea aplicației dacă pot să ducă la pierderi de informații fără avertizarea programatorului. Aceste erori sunt de obicei extrem de greu de depistat.

În fiecare limbaj care lucrează cu tipuri fixe pentru datele sale există conversii imposibile, conversii periculoase şi conversii sigure. Conversiile imposibile sunt conversiile pe care limbajul nu le permite pentru că nu ştie cum să le execute sau pentru că operația este prea periculoasă. De exemplu, Java refuză să convertească un tip primitiv către un tip referință. Deşi s-ar putea

imagina o astfel de conversie bazată pe faptul că o adresă este în cele din urmă un număr natural, acest tip de conversii sunt extrem de periculoase, chiar şi atunci când programatorul cere explicit această conversie.

Conversii de extindere a valorii

În aceste conversii valoarea se reprezintă într-o zonă mai mare fără să se piardă nici un fel de informații. Iată conversiile de extindere pe tipuri primitive:

- **byte** la **short**, **int**, **long**, **float** sau **double**
- **short** la **int**, **long**, **float** sau **double**
- **char** la **int**, **long**, **float** sau **double**
- **int** la **long**, **float** sau **double**
- **long** la **float** sau **double**
- **float** la **double**

Să mai precizăm totuşi că, într-o parte din aceste cazuri, putem pierde din precizie. Această situație apare de exemplu la conversia unui **long** într-un **float**, caz în care se pierd o parte din cifrele semnificative păstrându-se însă ordinul de mărime. De altfel această observație este evidentă dacă ținem cont de faptul că un **long** este reprezentat pe 64 de biți în timp ce un **float** este reprezentat doar pe 32 de biți.

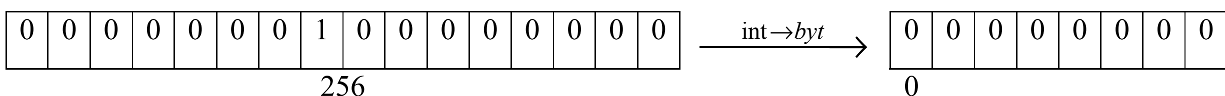
Precizia se pierde chiar şi în cazul conversiei **long** la **double** sau **int** la **float** pentru că, deşi dimensiunea zonei alocată pentru cele două tipuri este aceeaşi, numerele flotante au nevoie de o parte din această zonă pentru a reprezenta exponentul. În aceste situații, se va produce o rotunjire a numerelor reprezentate.

Conversii de trunchiere a valorii

Convențiile de trunchiere a valorii pot produce pierderi de informație pentru că ele convertesc tipuri mai bogate în informații către tipuri mai sărace. Conversiile de trunchiere pe tipurile elementare sunt următoarele:

- **byte** la **char**
- **short** la **byte** sau **char**
- **char** la **byte** sau **short**
- **int** la **byte**, **short** sau **char**
- **long** la **byte**, **short char**, sau **int**
- **float** la **byte**, **short**, **char**, **int** sau **long**
- **double** la **byte**, **short**, **char**, **int**, **long** sau **float**.

În cazul conversiilor de trunchiere la numerele cu semn, este posibil să se schimbe semnul pentru că, în timpul conversiei, se îndepărtează pur şi simplu octeții care nu mai încap şi poate rămâne primul bit diferit de vechiul prim bit. Copierea se face începând cu octeții mai puțin semnificativi iar trunchierea se face la octeții cei mai semnificativi.



Prin *octeții cei mai semnificativi* ne referim la octeții în care sunt reprezentate cifrele cele mai semnificative. *Cifrele cele mai semnificative* sunt cifrele care dau ordinul de mărime al numărului. De exemplu, la numărul 123456, cifrele cele mai semnificative sunt primele, adică: 1, 2, etc. La acelaşi număr*, cifrele cele mai puțin semnificative* sunt ultimele, adică: 6, 5, etc.

Conversii pe tipuri referință

Conversiile tipurilor referință nu pun probleme pentru modul în care trebuie executată operația din cauză că, referința fiind o adresă, în timpul conversiei nu trebuie afectată în nici un fel această adresă. În schimb, se pun probleme legate de corectitudinea logică a conversiei. De exemplu, dacă avem o referință la un obiect care nu este tablou, este absurd să încercăm să convertim această referință la o referință de tablou. Limbajul Java defineşte extrem de strict conversiile posibile în cazul tipurilor referință pentru a salva programatorul de eventualele necazuri care pot apare în timpul execuției. Iată conversiile posibile:

- O referință către un obiect aparținând unei clase *C* poate fi convertit la o referință către un obiect aparținând clasei *S* doar în cazul în care *C* este chiar *S* sau *C* este derivată direct sau indirect din *S*.
- O referință către un obiect aparținând unei clase *C* poate fi convertit către o referință de interfață *I* numai dacă clasa *C* implementează interfața *I*.
- O referință către un tablou poate fi convertită la o referință către o clasă numai dacă clasa respectivă este clasa **Object**.
- O referință către un tablou de elemente ale cărui elemente sunt de tipul *T1* poate fi convertită la o referință către un tablou de elemente de tip *T2* numai dacă *T1* şi *T2* reprezintă acelaşi tip primitiv sau *T2* este un tip referință şi *T1* poate fi convertit către *T2*.

Valorile de tip primitiv nu pot fi atribuite variabilelor de tip referință. La fel, valorile de tip referință nu pot fi memorate în variabile de tip primitiv. În ceea ce priveşte tipurile referință între ele, următorul tabel defineşte situațiile în care conversiile sunt posibile la atribuirea unei valori de tipul *T* la o variabilă de tipul *S*:

	T este o clasă care nu este finală	T este o clasă care este finală	T este o interfață	T = B[] este un tablou cu elemente de tipul B
S este o clasă care nu este finală	T trebuie să fie subclasă a lui S	T trebuie să fie o subclasă a lui S	eroare la compilare	S trebuie să fie Object
S este o clasă care este finală	T trebuie să fie aceeaşi clasă ca şi S	T trebuie să fie aceeaşi clasă ca şi S	eroare la compilare	eroare la compilare
S este o interfață	T trebuie să implementeze interfața S	T trebuie să implementeze interfața S	T trebuie să fie o subinterfață a lui S	eroare la compilare
S = A[] este un tablou cu elemente de tipul A	eroare la compilare	eroare la compilare	eroare la compilare	A sau B sunt acelaşi tip primitiv sau A este un tip referință şi B poate fi atribuit lui A

Conversiile posibile la atribuirea unei valori de tipul T la o variabilă de tipul S.

Conversii explicite

Conversiile de tip cast, sau *casturile,* sunt apelate de către programator în mod explicit. Sintaxa pentru construcția unui cast este scrierea tipului către care dorim să convertim în paranteze în fața valorii pe care dorim să o convertim. Forma generală este:

(*Tip*) *Valoare*

Conversiile posibile în acest caz sunt mai multe decât conversiile implicite la atribuire pentru că în acest caz programatorul este prevenit de eventuale pierderi de date el trebuind să apeleze conversia explicit.

Dar, continuă să existe conversii care nu se pot apela nici măcar în mod explicit, după cum am explicat înainte.

În cazul conversiilor de tip cast, orice valoare numerică poate fi convertită la orice valoare numerică.

În continuare, valorile de tip boolean nu pot fi convertite la nici un alt tip.

Nu există conversii între valorile de tip referință şi valorile de tip primitiv.

În cazul conversiilor dintr-un tip referință într-altul putem separa două cazuri. Dacă compilatorul poate decide în timpul compilării dacă conversia este corectă sau nu, o va decide. În cazul în care compilatorul nu poate decide pe loc, se va efectua o verificare a conversiei în timpul execuției. Dacă conversia se dovedeşte greşită, va apare o eroare de execuție şi programul va fi întrerupt.

Iată un exemplu de situație în care compilatorul nu poate decide dacă conversia este posibilă sau nu:

```
Minge mingeaMea;
?
MingeDeBaschet mingeaMeaDeBaschet;
// MingeDeBaschet este o clasă
// derivată din clasa Minge
mingeaMeaDeBaschet=(MingeDeBaschet)mingeaMea;
```

În acest caz, compilatorul nu poate fi sigur dacă referința memorată în variabila *mingeaMea* este de tip MingeDeBaschet sau nu pentru că variabilei de tip Minge i se pot atribui şi referințe către instanțe de tip Minge în general, care nu respectă întru totul definiția clasei MingeDeBaschet sau chiar referință către alte tipuri de minge derivate din clasa Minge, de exemplu MingeDePolo care implementează proprietăți şi operații diferite față de clasa MingeDeBaschet.

Iată şi un exemplu de conversie care poate fi decisă în timpul compilării:

```
Minge mingeaMea;
MingeDeBaschet mingeaMeaDeBaschet;
?
mingeaMea = ( Minge ) mingeaMeaDeBaschet;
```

În următorul exemplu însă, se poate decide în timpul compilării imposibilitatea conversiei:

```
MingeDeBaschet mingeaMeaDeBaschet;
MingeDePolo mingeaMeaDePolo;
```

?
mingeaMeaDePolo = (MingeDePolo) mingeaMeaDeBaschet;

În fine, tabelul următor arată conversiile de tip cast a căror corectitudine poate fi stabilită în timpul compilării. Conversia încearcă să transforme printr-un cast o referință de tip *T* într-o referință de tip *S*.

	T este o clasă care nu este finală	T este o clasă care este finală	T este o interfață	T = B[] este un tablou cu elemente de tipul B
S este o clasă care nu este finală	T trebuie să fie subclasă a lui S	T trebuie să fie o subclasă a lui S	Totdeauna corectă la compilare	S trebuie să fie Object
S este o clasă care este finală	S trebuie să fie subclasă a lui T	T trebuie să fie aceeaşi clasă ca şi S	S trebuie să implementeze interfața T	eroare la compilare
S este o interfață	Totdeauna corectă la compilare	T trebuie să implementeze interfața S	Totdeauna corectă la compilare	eroare la compilare
S = A[] este un tablou cu elemente de tipul A	T trebuie să fie Object	eroare la compilare	eroare la compilare	A sau B sunt acelaşi tip primitiv sau A este un tip referință şi B poate fi convertit cu un cast la A

Cazurile posibile la convertirea unei referințe de tip T într-o referință de tip S.