

Správa paměti

Karel Richta a kol.

Katedra počítačů
Fakulta elektrotechnická
České vysoké učení technické v Praze

© Karel Richta, 2016

Objektové modelování, B36OMO
10/2016, Lekce 2

<https://cw.fel.cvut.cz/wiki/courses/XXb36omo/start>



Paměťový model

Paměť je rozdělena do tří (někdy čtyř) oblastí:

- **Kód** (code area, může být ROM)
 - **Statická data** (global memory)
 - **Zásobník** (stack)
 - **Halda** (heap)
-
- Tento model je zjednodušený!
 - Budeme uvažovat programy běžící v jednom vlákně.

Alokace paměti

- **Statická alokace** - paměť je přidělena prakticky v „době kompilace“.
 - Nelze použít pro dynamické datové struktury.
- **Alokace na zásobníku** - dynamická alokace na zásobníku v rámci současného rámce.
 - Po skončení volání je paměť automaticky de-alokována.
- **Alokace na haldě** - dynamická alokace na haldě, alokovanou paměť je třeba spravovat.
 - V jazyce Java to za nás dělá „Garbage Collector“.

Příklad

```
public class CoffeeMaker {  
    static int coffeeCount = 0;  
    Coffee makeCoffee() {  
        coffeeCount++;  
        Coffee c = new Coffee();  
        c.milk = true;  
        return c;  
    }  
    public static void main(String[] args) {  
        CoffeeMaker coffeeMaker;  
        coffeeMaker = new CoffeeMaker();  
        Coffee c = coffeeMaker.makeCoffee();  
    }  
}
```

statický
objekt

automatický
objekt na
zásobníku

dynamický
objekt na
haldě

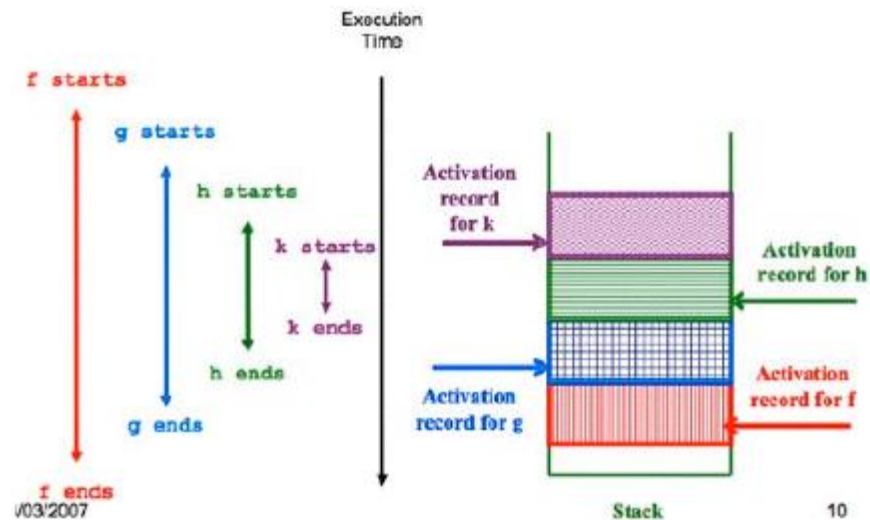
Správa zásobníku (automatická)

- Při volání metody je nutné zachytit vstup do kontextu metody.
- To znamená vytvořit lokální objekty metody, zaznamenat aktuální parametry, připravit místo pro výsledek metody.
- Poznamenat si, kam se po provedení metody vrátit.
- Při výstupu z metody je třeba tyto pomocné struktury zrušit a uvolnit, předat návratovou hodnotu a vrátit se do místa, kde byla metoda vyvolána.
- Vhodnou strukturou pro tyto účely je zásobník (LIFO).
- Na zásobník se při volání metody ukládají informace ve formě tzv. aktivačního záznamu.

Aktivační záznam (Activation Record)

- Když je metoda volána, tak veškeré informace nutné pro její běh jsou umístěny na zásobník.
- Tyto informace se nazývají aktivační záznam (Activation Record - AR).
- Když volání skončí (return), pak je AR odstraněn ze zásobníku.

```
void f(){  
    g();  
}  
  
void g(){  
    h();  
}  
  
void h(){  
    k();  
}
```



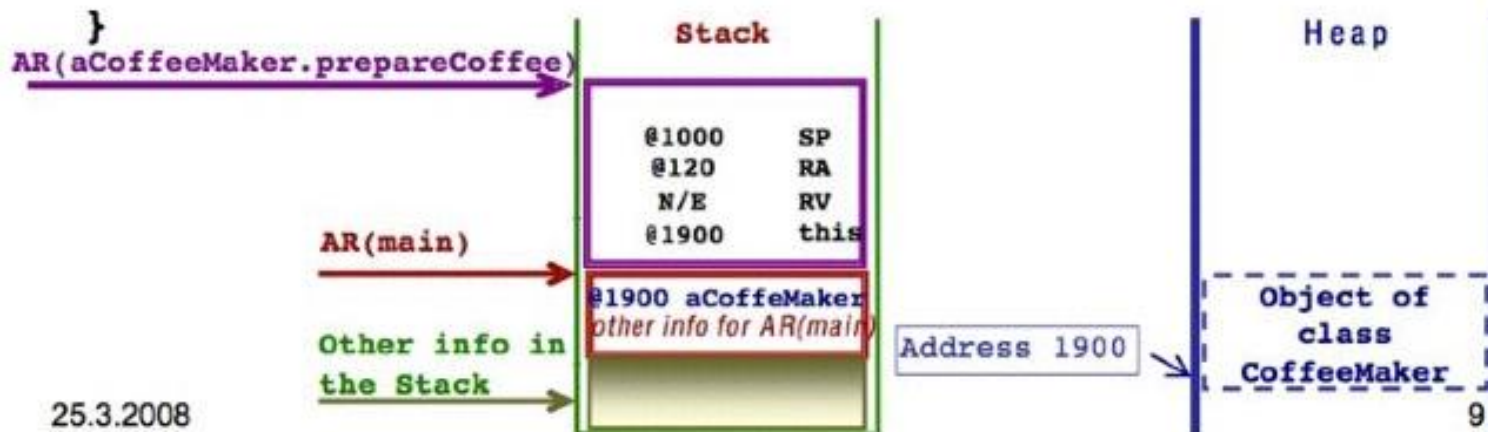
Zdroj: <http://ineed.coffee/wp-content/uploads/2011/04/object-oriented-memory-management-java-c++.pdf>

Aktivační záznam obsahuje

- Návratová hodnota (Co vracím?) **RV**
- Adresa návratu (Kam se vracím?) **RA**
- Ukazatel na zásobník (Odkud jsem přišel?) **SP**
- Hodnoty parametrů metody.
- Lokální proměnné.

Příklad

```
public class CoffeeMaker {
    public void prepareCoffee() {
        /*Prepare the coffee with a standard amount of sugar*/
    }
    public void prepareCoffeeSweet(int sugarAm){
        /* Prepare the coffee with sugarAm(ount) of sugar */
    }
    void main(...) {
        CoffeeMaker aCoffeeMaker;
        aCoffeeMaker = new CoffeeMaker();
        aCoffeeMaker.prepareCoffee();
    }
}
```



Zdroj: <http://ined.coffee/wp-content/uploads/2011/04/object-oriented-memory-management-java-c++.pdf>

Příklad

```
public class CoffeeMaker {
    public void prepareCoffee() {
        /*Prepare the coffee with a standard amount of sugar*/
    }
    public void prepareCoffeeSweet(int sugarAm){
        /* Prepare the coffee with sugarAm(ount) of sugar */
    }
    void main(...) {
        CoffeeMaker aCoffeeMaker;
        aCoffeeMaker = new CoffeeMaker();
        aCoffeeMaker.prepareCoffee();
    }
}
```

static

statická data

stack

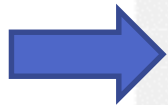
zásobník

heap

halda

Příklad

```
public class CoffeeMaker {
    public void prepareCoffee() {
        /*Prepare the coffee with a standard amount of sugar*/
    }
    public void prepareCoffeeSweet(int sugarAm){
        /* Prepare the coffee with sugarAm(ount) of sugar */
    }
    void main(...) {
        CoffeeMaker aCoffeeMaker;
        aCoffeeMaker = new CoffeeMaker();
        aCoffeeMaker.prepareCoffee();
    }
}
```



static

statická data

aCoffeeMaker = null

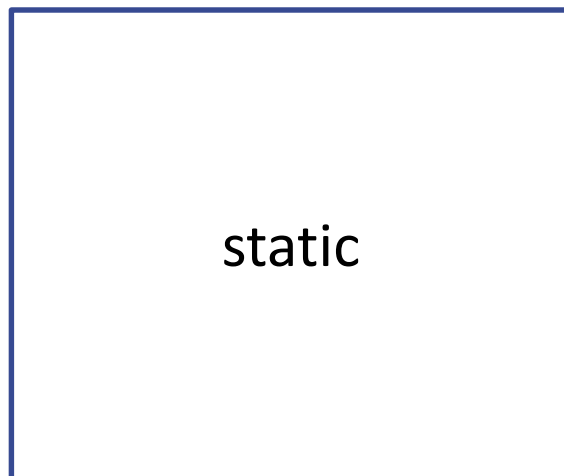
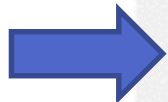
zásobník

heap

halda

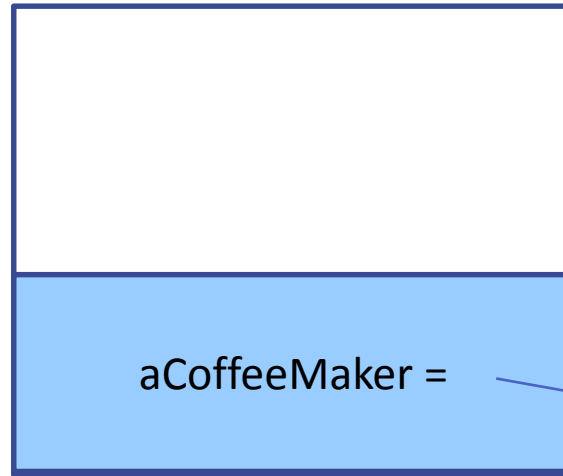
Příklad

```
public class CoffeeMaker {
    public void prepareCoffee() {
        /*Prepare the coffee with a standard amount of sugar*/
    }
    public void prepareCoffeeSweet(int sugarAm){
        /* Prepare the coffee with sugarAm(ount) of sugar */
    }
    void main(...) {
        CoffeeMaker aCoffeeMaker;
        aCoffeeMaker = new CoffeeMaker();
        aCoffeeMaker.prepareCoffee();
    }
}
```



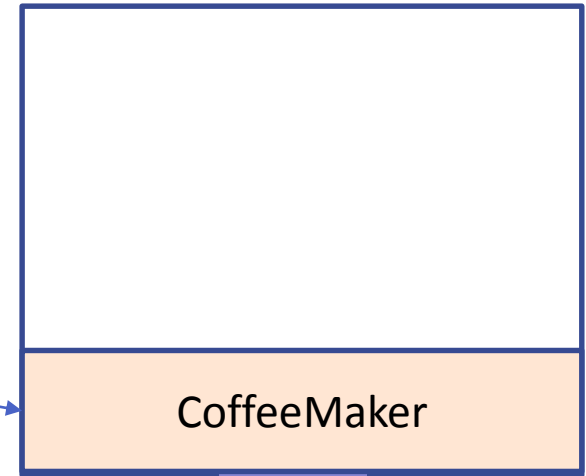
static

statická data



aCoffeeMaker =

zásobník

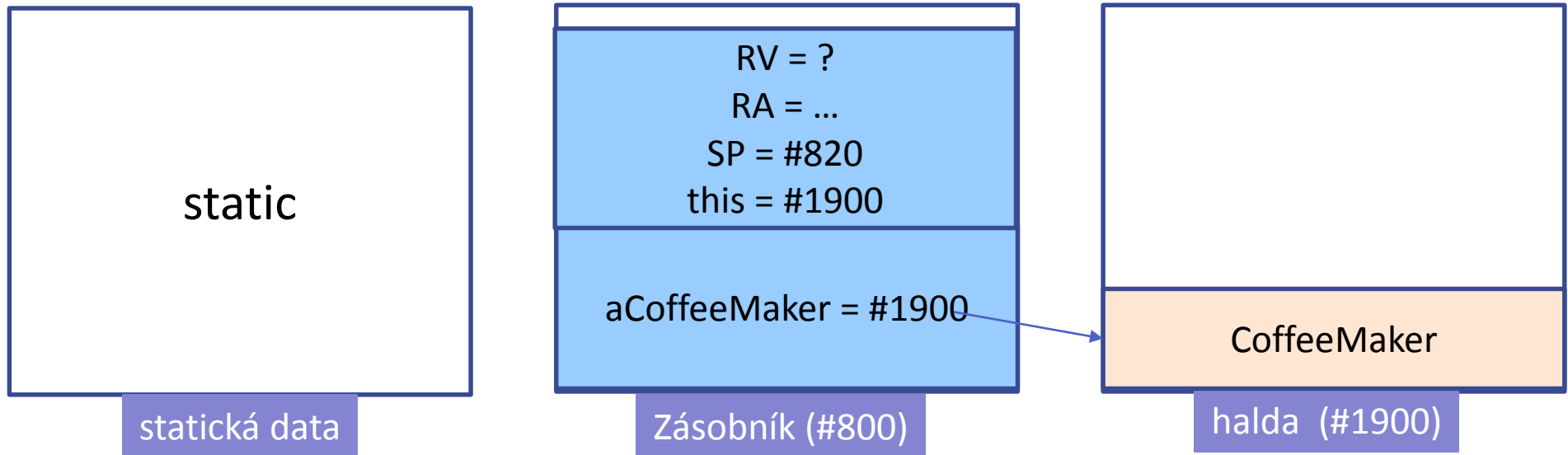


CoffeeMaker

halda

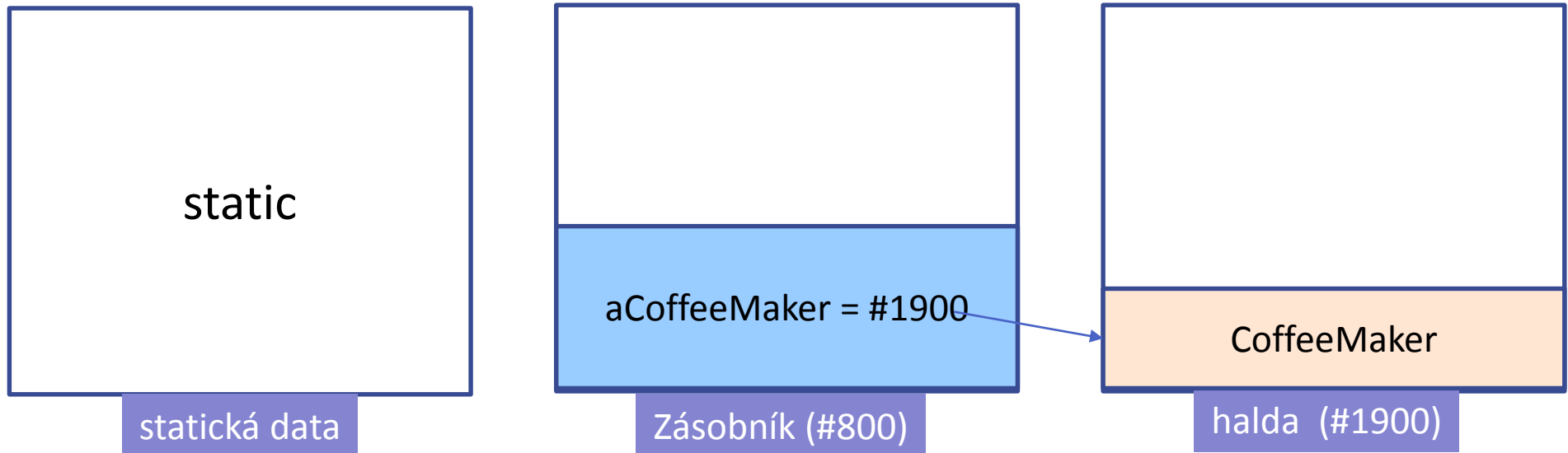
Příklad

```
public class CoffeeMaker {
    public void prepareCoffee() {
        /*Prepare the coffee with a standard amount of sugar*/
    }
    public void prepareCoffeeSweet(int sugarAm){
        /* Prepare the coffee with sugarAm(ount) of sugar */
    }
    void main(...) {
        CoffeeMaker aCoffeeMaker;
        aCoffeeMaker = new CoffeeMaker();
        aCoffeeMaker.prepareCoffee();
    }
}
```




Příklad

```
public class CoffeeMaker {
    public void prepareCoffee() {
        /*Prepare the coffee with a standard amount of sugar*/
    }
    public void prepareCoffeeSweet(int sugarAm){
        /* Prepare the coffee with sugarAm(ount) of sugar */
    }
    void main(...) {
        CoffeeMaker aCoffeeMaker;
        aCoffeeMaker = new CoffeeMaker();
        aCoffeeMaker.prepareCoffee();
    }
}
```



Příklad

```
public class CoffeeMaker {
    public void prepareCoffee() {
        /*Prepare the coffee with a standard amount of sugar*/
    }
    public void prepareCoffeeSweet(int sugarAm){
        /* Prepare the coffee with sugarAm(ount) of sugar */
    }
    void main(...) {
        CoffeeMaker aCoffeeMaker;
        aCoffeeMaker = new CoffeeMaker();
        aCoffeeMaker.prepareCoffee();
    }
}
```



static

statická data

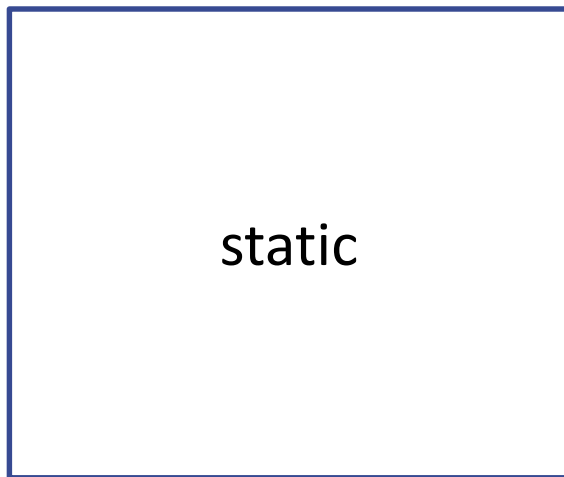
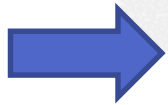
Zásobník (#800)

CoffeeMaker

halda (#1900)

Příklad (po úklidu GC)

```
public class CoffeeMaker {  
    public void prepareCoffee() {  
        /*Prepare the coffee with a standard amount of sugar*/  
    }  
    public void prepareCoffeeSweet(int sugarAm){  
        /* Prepare the coffee with sugarAm(ount) of sugar */  
    }  
    void main(...) {  
        CoffeeMaker aCoffeeMaker;  
        aCoffeeMaker = new CoffeeMaker();  
        aCoffeeMaker.prepareCoffee();  
    }  
}
```



static

statická data



Zásobník (#800)



halda (#1900)

SAR - pro jazyky s blokovou strukturou

- Návrátová hodnota (co vracím?) **RV**
- Adresa návratu (kam se vracím?) **RA**
- Ukazatel na zásobník (odkud jsem přišel?) **SP**
- Hodnoty parametrů metody.
- Lokální proměnné.
- Statický link (odkaz na nadřazený blok) **SL**

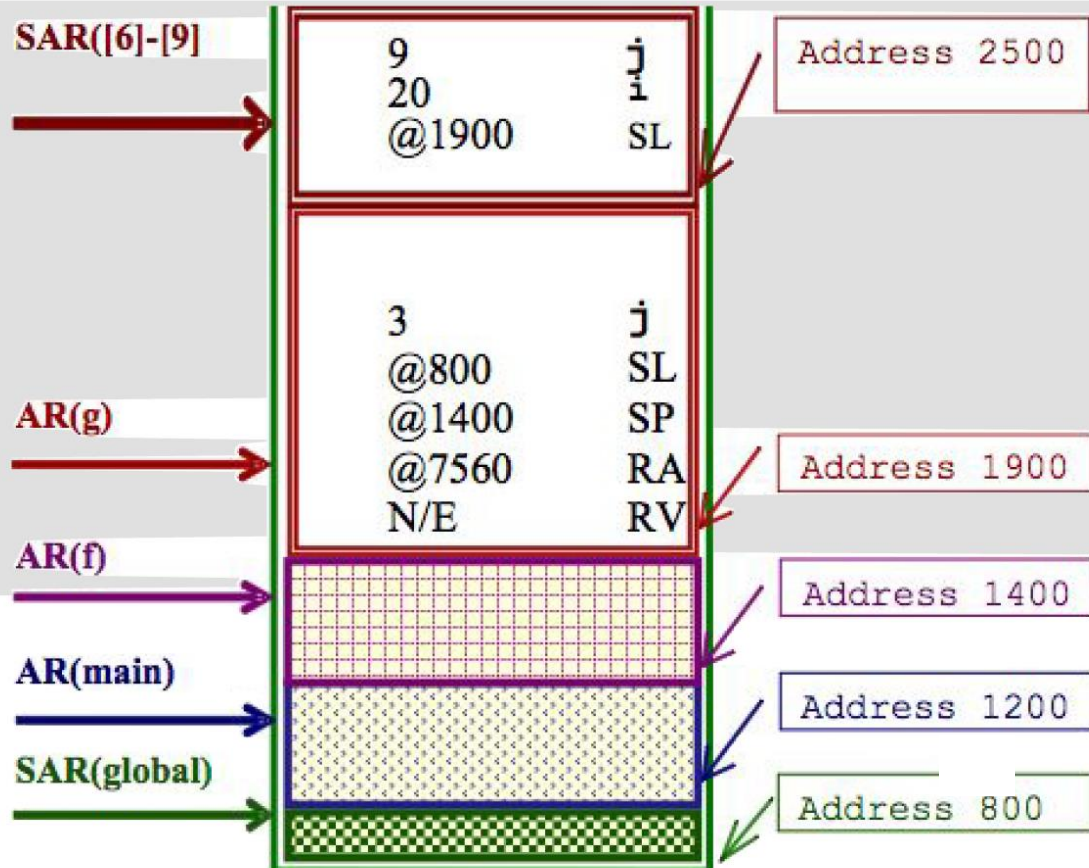
- **Pozn: SAR** – Scope Activation Record

Příklad SAR

```
void g(){
    int j = 3;
    {
        int i = 20;
        int j = 9;
    }
}

void f(){
    g();
}

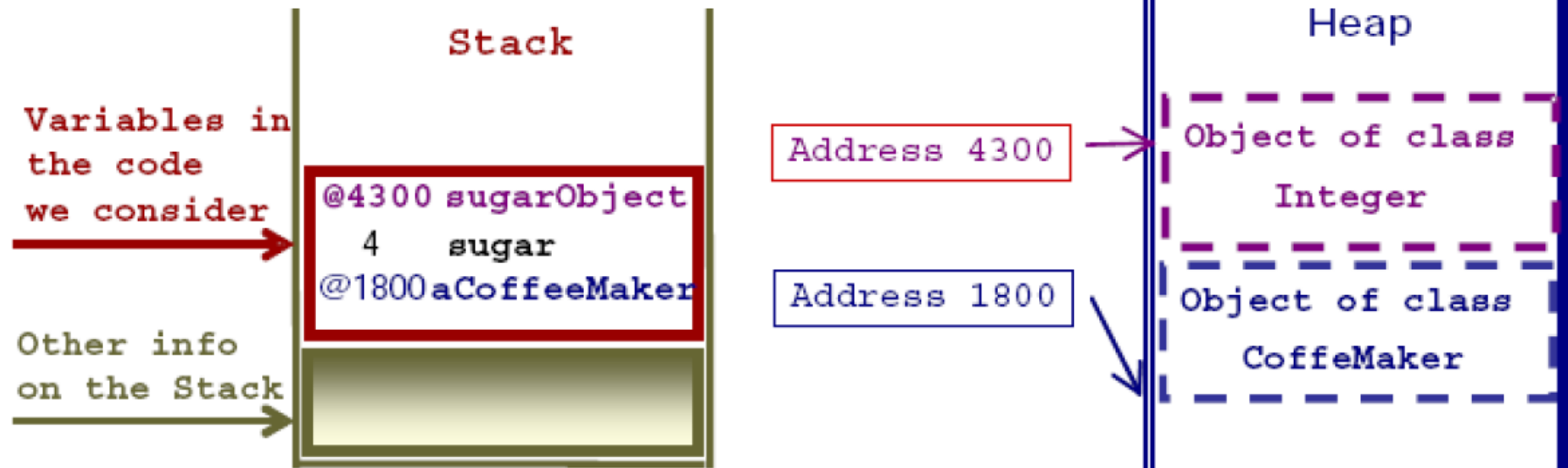
void main(){
    f();
}
```



Zdroj: <http://ineed.coffee/wp-content/uploads/2011/04/object-oriented-memory-management-java-c++.pdf>

Jiný příklad

```
public class CoffeeMaker {
    public CoffeeMaker(){}
}
..
..
CoffeeMaker aCoffeeMaker;
aCoffeeMaker = new CoffeeMaker();
int sugar = 4;
Integer sugarObject = new Integer(3);
```



Formalismus pro zachycení obsahu paměti

- Umožňuje přehledně zaznamenat stav programu.
- Lze ho použít pro krokování programu „na papír“.

Hodnoty:

- Logické: true, false
- Čísla: 0, 1, -1, 2, -2, ...
- Znaky: 'a', 'A', 'b', 'B', ...
- Řetězce: “Super řetězec”, ...
- Adresy: null, #1, #2, #3, ...

Formalismus pro zachycení obsahu paměti (pokr.)

Proměnné:

- Jméno = hodnota
 - např. contents = 5, nebo next = #3
- Hodnota proměnné musí odpovídat jejímu typu.

Objekty:

- Třída(attr1 = hodn1, attr2 = hodn2, ...)
 - např. Node(contents = 1, next = #2)
- Jména atributů musí korespondovat s danou třídou, na jejich pořadí nezáleží. Pokud atributy neznáme, můžeme je vypustit:
 - Node(...)

Formalismus pro zachycení obsahu paměti (pokr.)

Halda:

- Mapování z adres na objekty.
 - Používáme šipkovou notaci (symbol není podstatný).
 - #1 -> Node(contents = 2, next = #2),
 - #2 -> Node(contents = 3, next = #2)
- Na jedné adrese nemohou být dva objekty.

Statické proměnné:

- Třída1.proměnná1 = hodnota1, Třída2.proměnná2 = hodnota2
 - Např. Node.globalhead = #2, Car.totalCount = 10211432
- Statické proměnné musí korespondovat s definicemi použitých tříd.

Formalismus pro zachycení obsahu paměti (pokr.)

Zásobník:

Rámec = Stack Frame = Activation Record

- Zásobník se skládá z rámců, každý rámec odpovídá jedné zavolané metodě.
- Uvnitř rámce jsou vypsány existující lokální proměnné a jejich hodnoty.
- Rámce zpravidla píšeme nad sebe a oddělujeme vodorovnou čarou.
- Nesmíme zapomenout na this!

Příklad:

a = 3, b = 5, this = #1

x = #1, a = "abecede"

Příklad

```
class Node {
    final int v;
    Node next;
    Node(int v, Node next) {
        this.v = v; this.next = next;
    }
    boolean f(int t) {
        // stav B
        return v == t || (next != null && next.f(t));
    }
    public static void main(String[] args) {
        Node n = new Node(2, new Node(3, new Node(1, null)));
        new Node(1, n);
        // stav A
        if (n.f(1)) { n.next = null; }
        // stav C
    }
}
```

Příklad

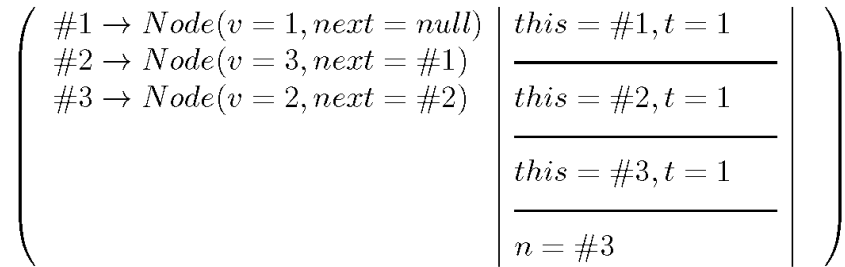
```
class Node {
    final int v;
    Node next;
    Node(int v, Node next) {
        this.v = v; this.next = next;
    }
    boolean f(int t) {
        // stav B
        return v == t || (next != null && next.f(t));
    }
    public static void main(String[] args) {
        Node n = new Node(2, new Node(3, new Node(1, null)));
        new Node(1, n);
        // stav A
        if (n.f(1)) { n.next = null; }
        // stav C
    }
}
```

stav A:

$$\left(\begin{array}{l} \#1 \rightarrow \text{Node}(v = 1, \text{next} = \text{null}) \\ \#2 \rightarrow \text{Node}(v = 3, \text{next} = \#1) \\ \#3 \rightarrow \text{Node}(v = 2, \text{next} = \#2) \end{array} \middle| n = \#3 \right)$$

stav B: (poslední)

```
class Node {
    final int v;
    Node next;
    Node(int v, Node next) {
        this.v = v; this.next = next;
    }
    boolean f(int t) {
        // stav B
        return v == t || (next != null && next.f(t));
    }
    public static void main(String[] args) {
        Node n = new Node(2, new Node(3, new Node(1, null)));
        new Node(1, n);
        // stav A
        if (n.f(1)) { n.next = null; }
        // stav C
    }
}
```



Příklad

```
class Node {
    final int v;
    Node next;
    Node(int v, Node next) {
        this.v = v; this.next = next;
    }
    boolean f(int t) {
        // stav B
        return v == t || (next != null && next.f(t));
    }
    public static void main(String[] args) {
        Node n = new Node(2, new Node(3, new Node(1, null)));
        new Node(1, n);
        // stav A
        if (n.f(1)) { n.next = null; }
        // stav C      stav C:
    }
}
```

(#3 → Node(v = 2, next = null) | n = #3 |)

Příklad

```
class Coffee {
    public boolean milk;
    public Coffee() {}
}

public class CoffeeMaker {
    static int coffeeCount = 0;
    Coffee makeCoffee() {
        coffeeCount++;
        Coffee c = new Coffee();
        c.milk = true;
        return c;
    }
    public static void main(String[] args) {
        CoffeeMaker coffeeMaker;
        coffeeMaker = new CoffeeMaker();
        Coffee c = coffeeMaker.makeCoffee();
    }
}
```

Úklid paměti - Garbage Collection

- Manuální správa paměti může vést k chybám:
 - úniky paměti (memory leak),
 - nesprávné ukazatele - ukazují na paměť, která již není alokována.
- GC umožňuje programátorovi se více soustředit na samotný program (ovšem za cenu určité režie).
- Cíl: uvolnit objekty, které nebudou již více používány.
- Dosažitelnost: tranzitivní uzávěr ukazatelů začínající v kořenové množině (root set) (všechny lokální a globální proměnné).

Metody GC

Algoritmy:

- Reference Counting.
- Mark and Sweep, Mark and Compact, Mark and Copy.

Přístupy:

- Hybrid Collecting - kombinace předchozích.
- Generational Collectors (Java) - zavedení generací podle stáří objektů.

Reference Counting

- Každý objekt obsahuje čítač.
- Přiřazení objektu do proměnné → snížení stavu čítače starého objektu a zvýšení stavu čítače objektu nově přiřazeného (pokud existují).
- Když čítač u objektu dosáhne 0, objekt se uvolní (jeho paměť připadne na free list).
- Nevýhody: velká přidaná zátěž, neporadí si s cyklickými odkazy.

Příklad

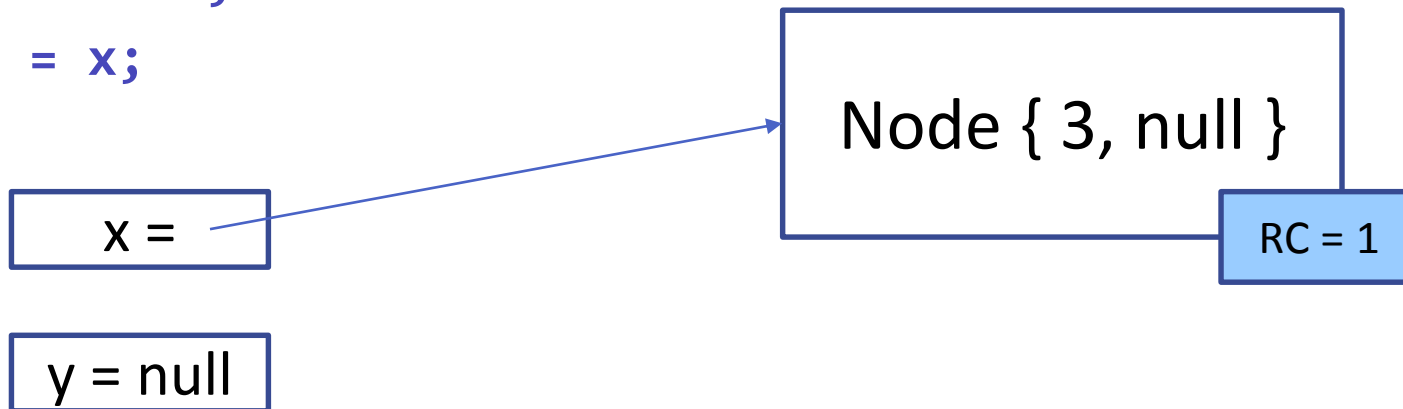
```
Node x, y;  
x = new Node (3, null);  
y = x;  
x = null;  
y = x;
```

x = null

y = null

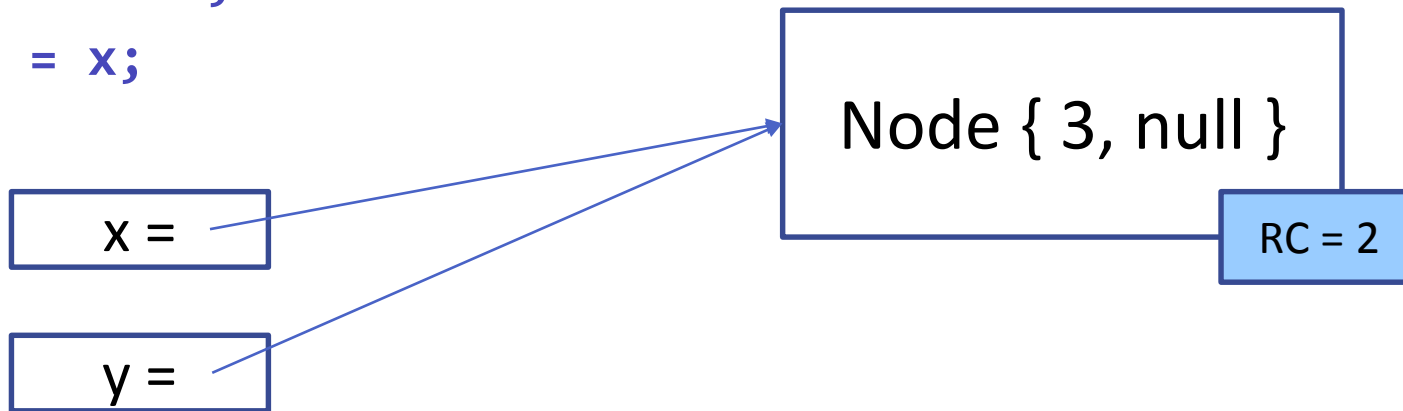
Příklad

```
Node x, y;  
x = new Node (3, null);  
y = x;  
x = null;  
y = x;
```



Příklad

```
Node x, y;  
x = new Node (3, null);  
y = x;  
x = null;  
y = x;
```

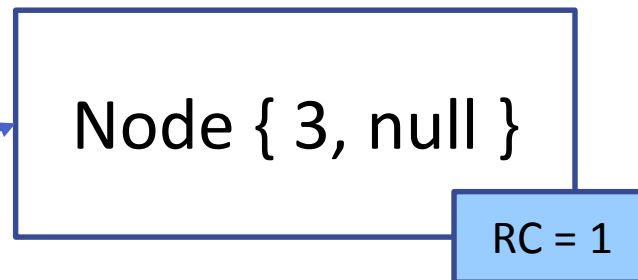


Příklad

```
Node x, y;  
x = new Node (3, null);  
y = x;  
x = null;  
y = x;
```

x = null

y =

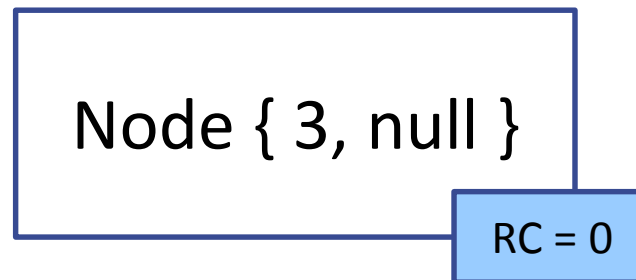


Příklad

```
Node x, y;  
x = new Node (3, null);  
y = x;  
x = null;  
y = x;
```

x = null

y = null

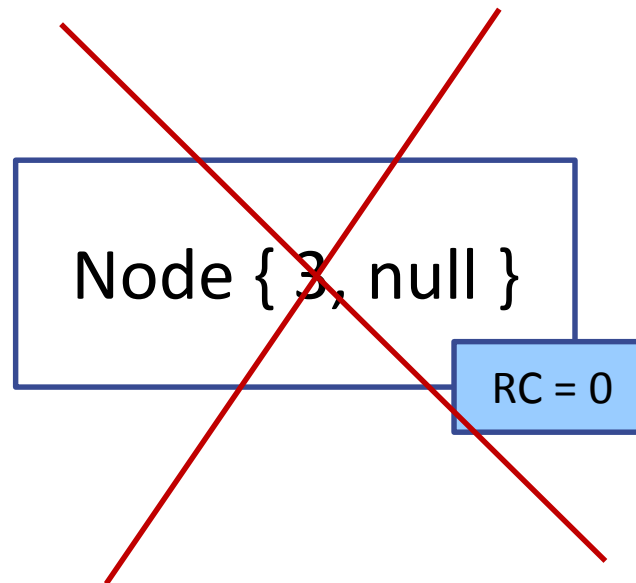


Příklad

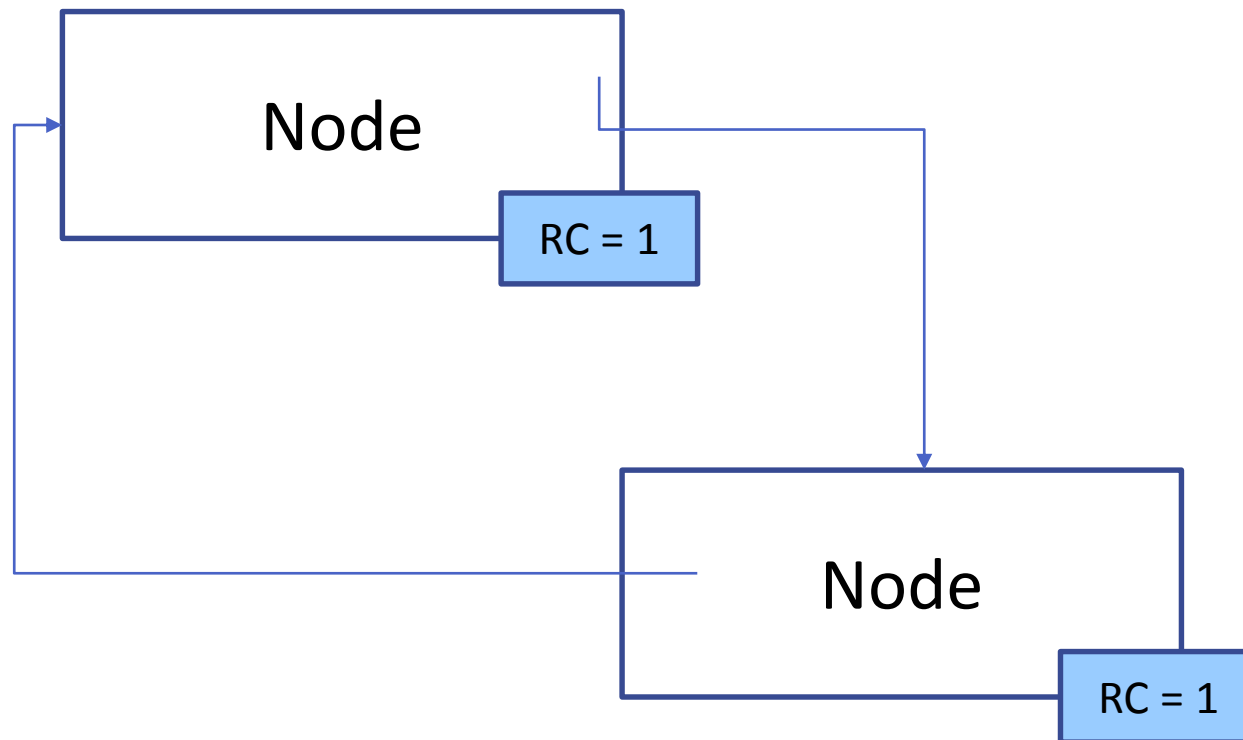
```
Node x, y;  
x = new Node (3, null);  
y = x;  
x = null;  
y = x;
```

x = null

y = null



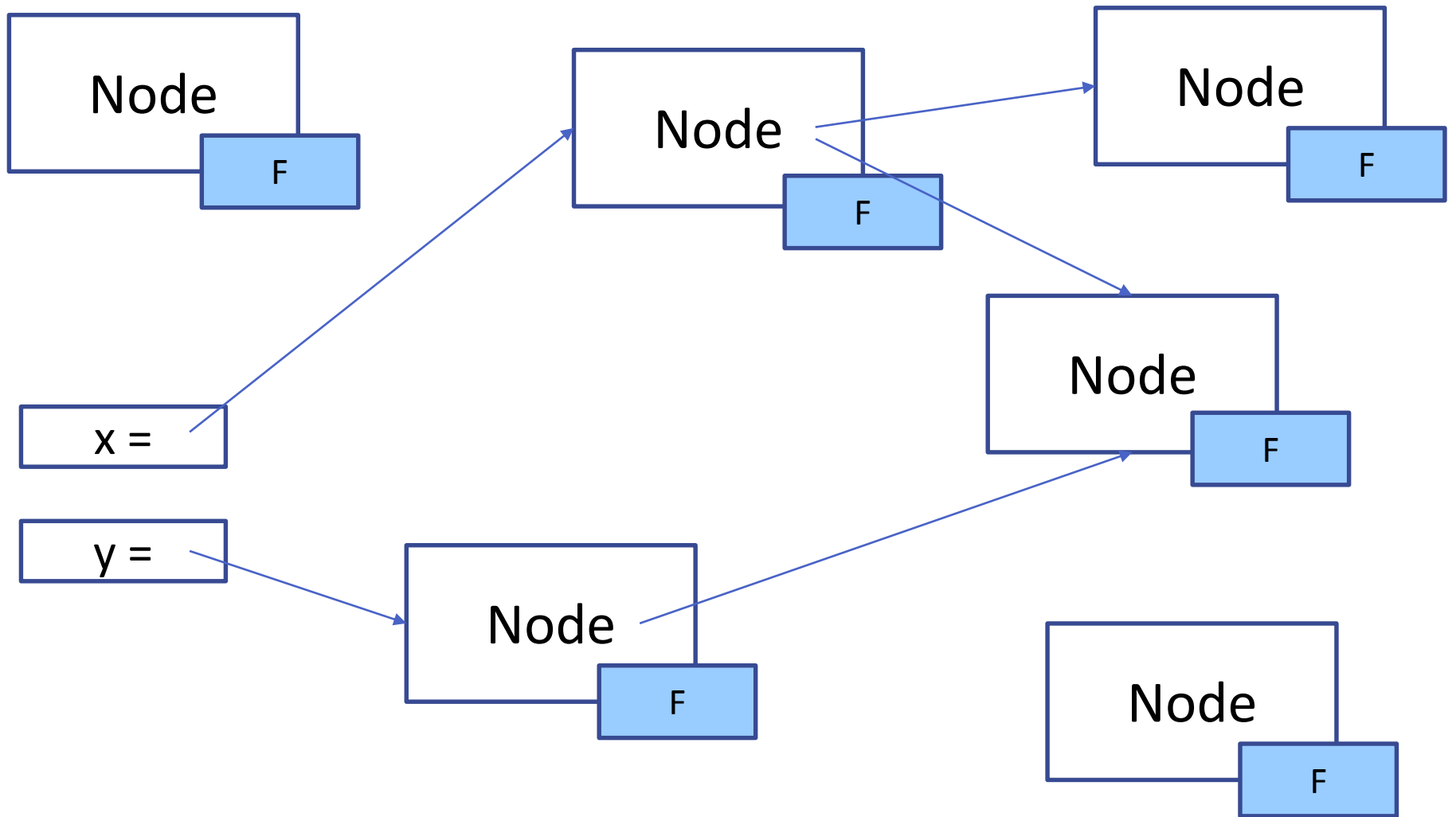
Problém – cyklické odkazy



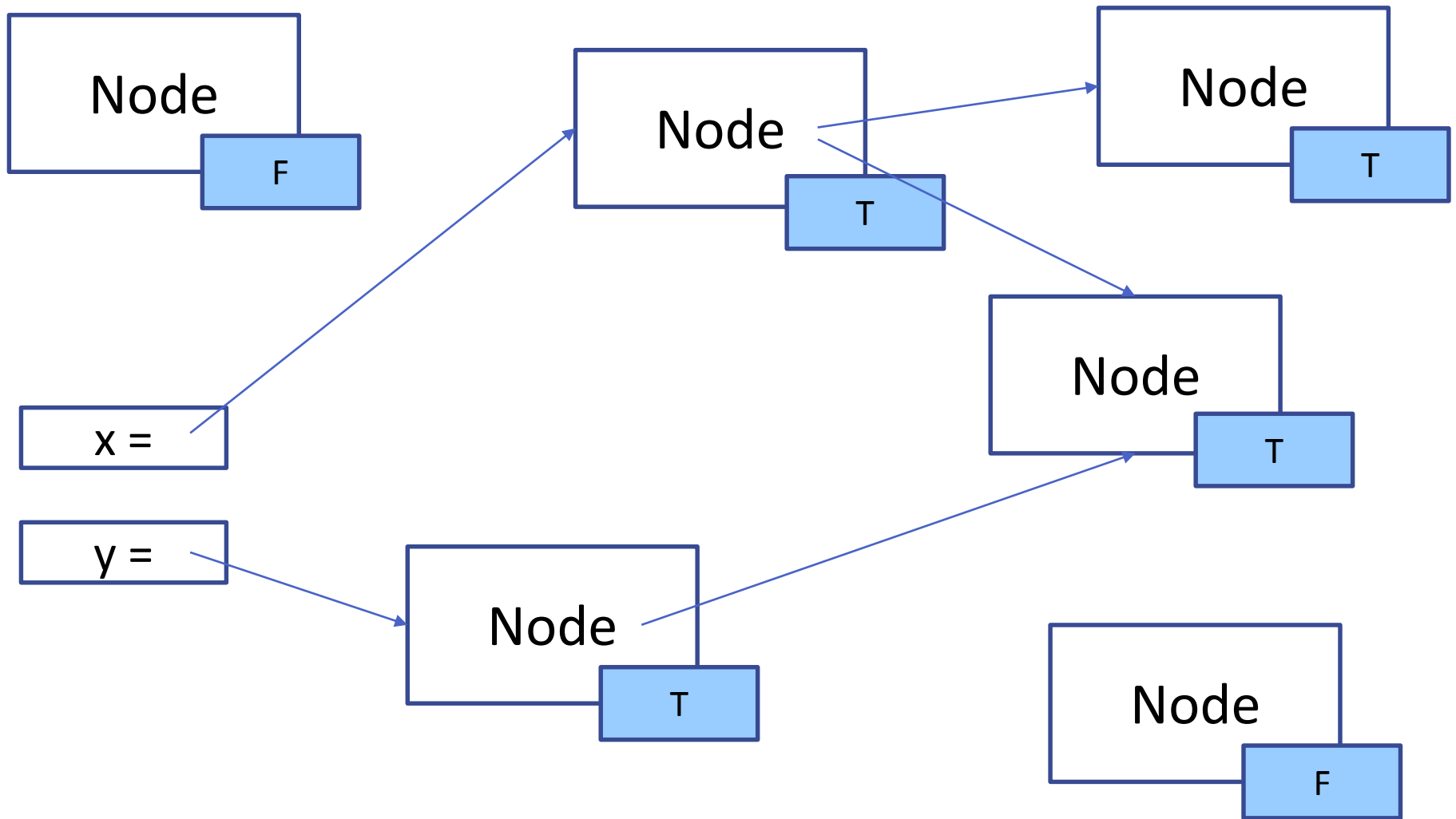
Mark and Sweep

- Každý objekt obsahuje značku (mark).
- Jednou za čas dojde ke konstrukci transitivního uzávěru z kořenové množiny a všechny dosažitelné objekty jsou označeny.
- Neoznačené objekty jsou uvolněny (sweep) (přidání jejich paměti na free-list).
- Zbylé objekty jsou opět odznačeny.

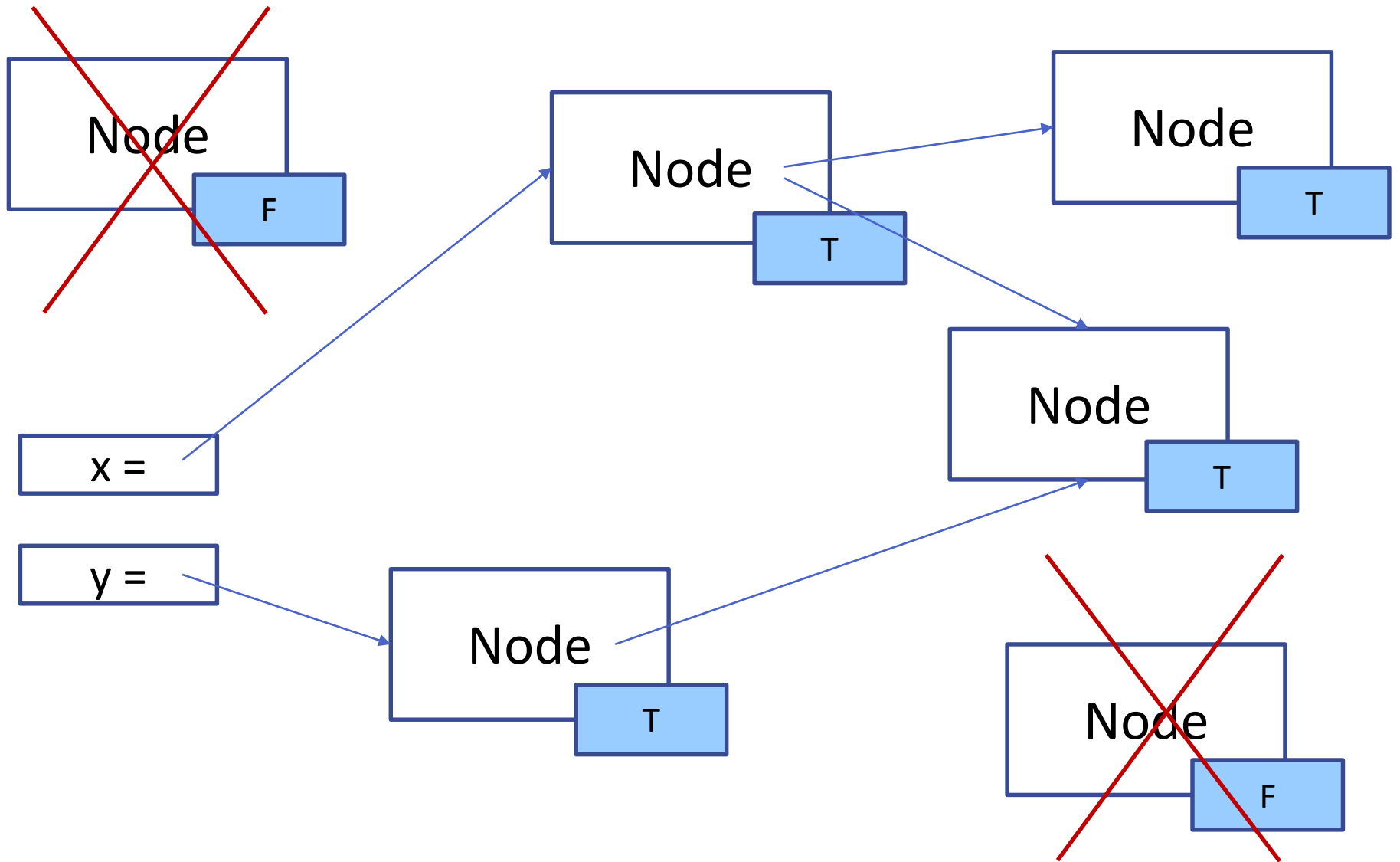
Příklad



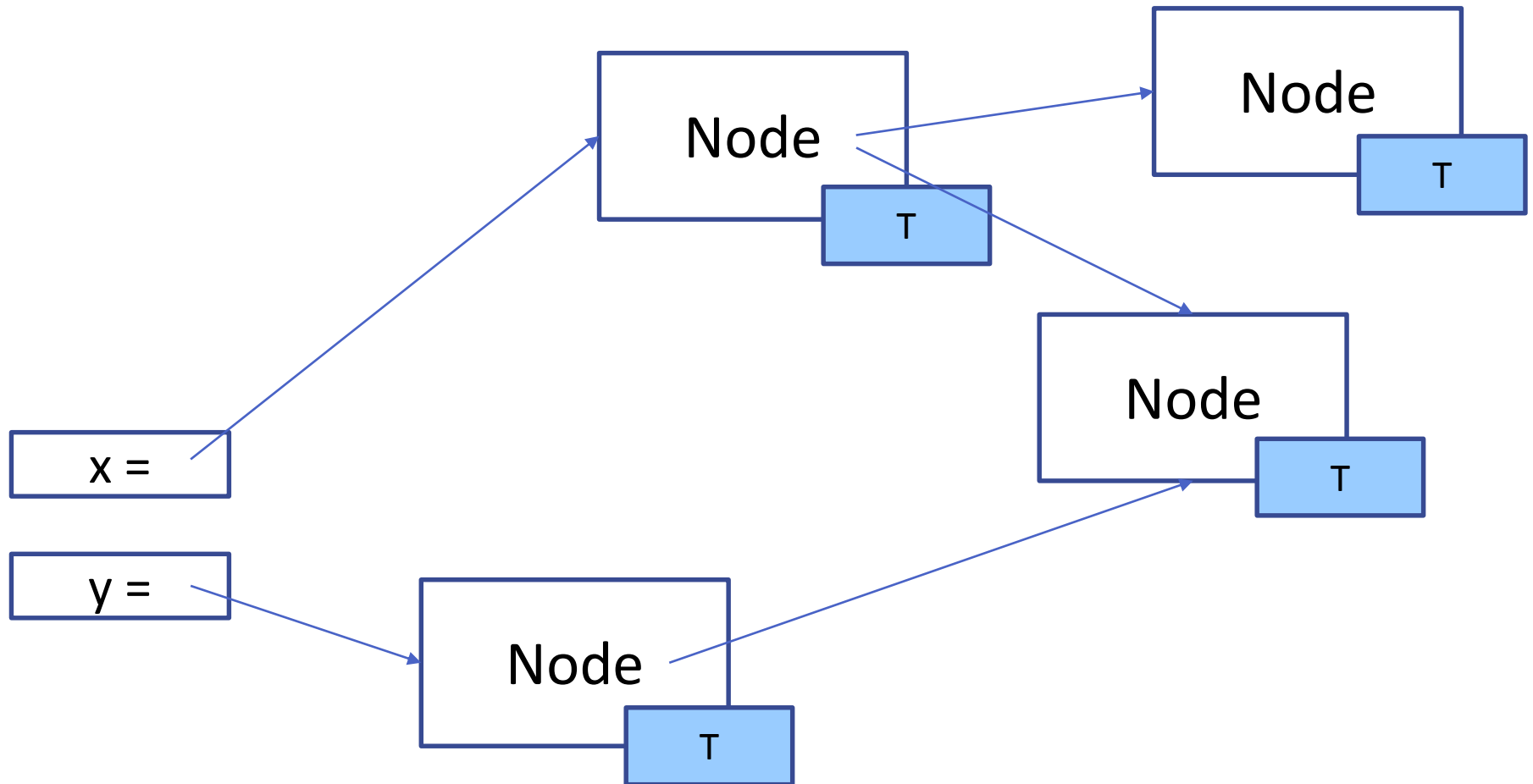
Příklad



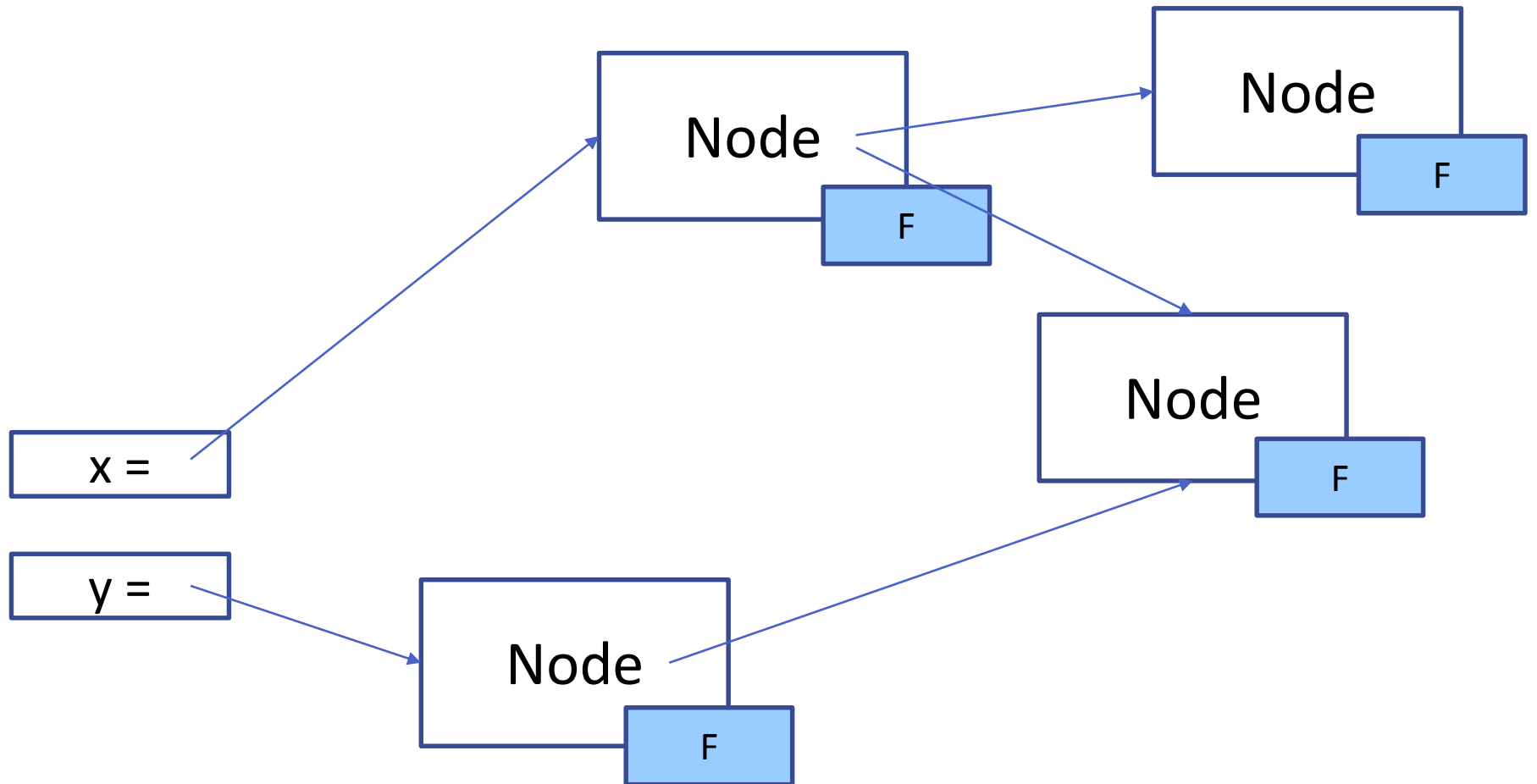
Příklad



Příklad



Příklad



Generační GC

- Využívá statistický předpoklad, že mladší objekty „umírají“ dříve.
- Zavádí generace objektů.
- Různé generace udržuje v různých paměťových prostorech.
- Když se tento prostor zaplní, zkoumá, zda jsou objekty v něm „živé“ – zda je na ně odkaz ze starší generace objektů.
- Proces úklidu se tím zrychluje, neboť se neprobírají vždy všechny objekty.

The End

<http://ineed.coffee/wp-content/uploads/2011/04/object-oriented-memory-management-java-c++.pdf>