Object Oriented Programming in Java

2023/24 H2 semester

Lecture 1

Summary

- These slides contain material about objects, classes, and object-oriented programming (OOP) in Java.

Contains:

- Key syntax of Java
  - Key principles of OOP in Java like interfaces, their implementation and inheritance
  - OOP patterns with examples in Java
- Lectures are demonstrated with examples in Java

| Part		Page
Intro	Why Java	3
Part I	Java syntax basics	8
Part II	Object oriented language syntax	203
Part III	Design Patterns	315

Why Java?
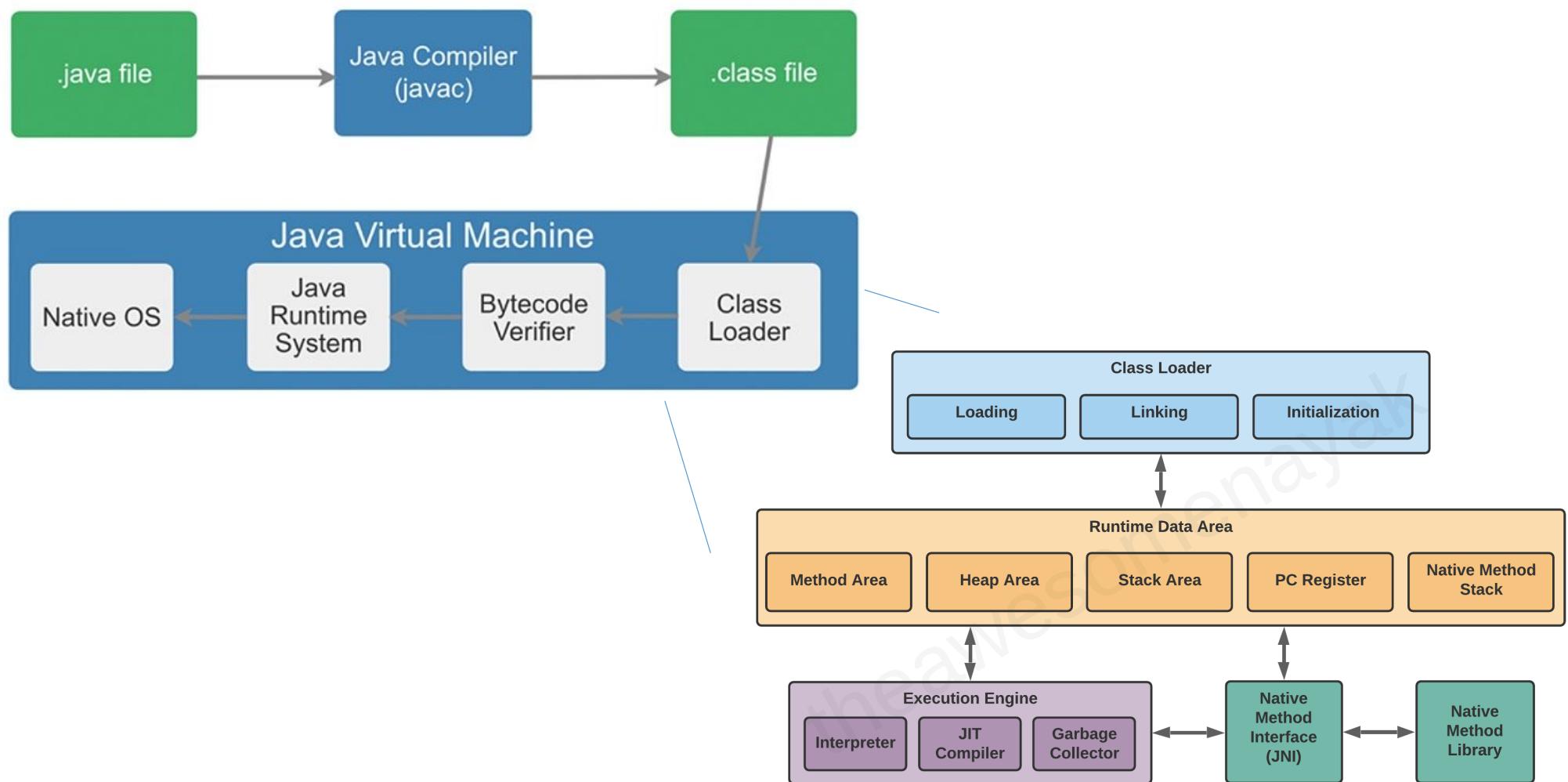
- TIOBE index
  https://www.tiobe.com/tiobe-index/ (2023)

Apr 2023	Apr 2022	Programming Language	Ratings	Change
1	1	Python	14.51%	+0.59%
2	2	C	14.41%	+1.71%
3	**3**	**Java**	**13.23%**	**+2.41%**
4	4	C++	12.96%	+4.68%
5	5	C#	8.21%	+1.39%
6	6	Visual Basic	4.40%	-1.00%
7	7	JavaScript	2.10%	-0.31%
8	9	SQL	1.68%	-0.61%
9	10	PHP	1.36%	-0.28%
10	13	Go	1.28%	+0.20%
11	12	Delphi/Object Pascal	1.23%	+0.05%
12	8	Assembly language	1.03%	-1.31%
13	16	Classic Visual Basic	0.92%	+0.09%
14	20	MATLAB	0.86%	+0.12%
15	24	Scratch	0.79%	+0.13%
16	11	R	0.76%	-0.79%
17	14	Swift	0.72%	-0.28%
18	15	Ruby	0.66%	-0.22%
19	28	Rust	0.63%	+0.18%
20	31	Fortran	0.59%	+0.24%

FEI STU

Intro

- **Jeff Goodell:** Would you explain, in simple terms, exactly what object-oriented software is?

- **Steve Jobs:** Objects are like people. They're living, breathing things that have knowledge inside them about how to do things and have memory inside them so they can remember things. And rather than interacting with them at a very low level, you interact with them at a very high level of abstraction, like we're doing right here.

- Here's an example: If I'm your laundry object, you can give me your dirty clothes and send me a message that says, "Can you get my clothes laundered, please." I happen to know where the best laundry place in San Francisco is. And I speak English, and I have dollars in my pockets. So I go out and hail a taxicab and tell the driver to take me to this place in San Francisco. I go get your clothes laundered, I jump back in the cab, I get back here. I give you your clean clothes and say, "Here are your clean clothes."

- You have no idea how I did that. You have no knowledge of the laundry place. Maybe you speak French, and you can't even hail a taxi. You can't pay for one, you don't have dollars in your pocket. Yet I knew how to do all of that. And you didn't have to know any of it. All that complexity was hidden inside of me, and we were able to interact at a very high level of abstraction. That's what objects are. They encapsulate complexity, and the interfaces to that complexity are high level.

How Java works

Part I – Java syntax basics

I. Java - Základné elementy

1. Výrazy (Expressions)

- Premenné
- Operácie a funkcie
- Syntaktický strom
- Vypočítanie hodnoty
- Pravidlá priority
- Syntaktické pravidlá

```java
public class Suma_1_az_n
{
  public static void main(String[] parametre)
  {
     int n = Integer.parseInt(parametre[0]);
    System.out.print(" Vysledok je: ");

    System.out.println( n * (n + 1) / 2 );
  }
}


Poznamka:
1 + 2 + 3 + 4 + 5 + .. + 100 = S
100 + 99 + 98 + …        + 1 = S
```

```java
public class Suma_1_az_n
{
  public static void main(String[] parametre)
  {
     int n = Integer.parseInt(parametre[0]);
    System.out.print(" Vysledok je: ");

    System.out.println( n * (n + 1) / 2 );
  }
}
```

Ďalšie príklady (výrazov)

- `n * (n + 1) / 2`
- `- sum + i`
- `a % b`   `// Zvyšok po delení a / b`

- `Math.min(a,b)` `// Minimum z hodnôt a a b v Jave`
- `Math.abs(a)`   `// Absolútna hodnota a v Jave`

- `Math.abs(Math.min(a % b, c)) // v Jave`
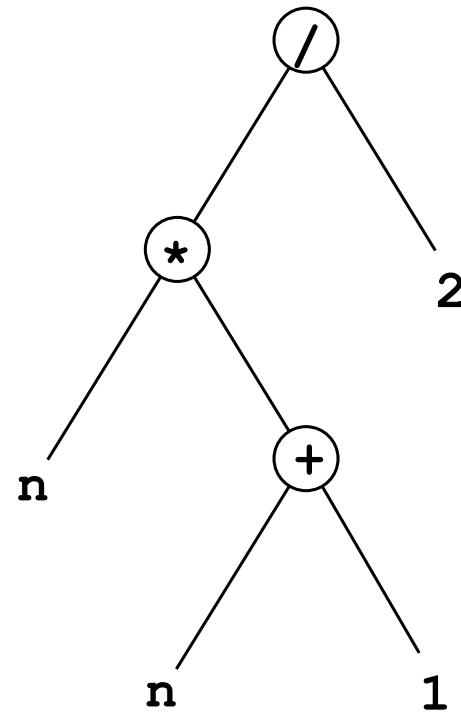
Výrazy

Sú vybudované z:

- Premenných: `n, a, sum`
  (musia byť deklarované: neskôr)

- Operátorov: $+, \quad -, \quad *, \quad /, \quad \%$

- Funkcií: `Math.abs(n)`

- Konštánt: `1, 2, 0, 4711, ...`

- Zátvoriek: `( )`
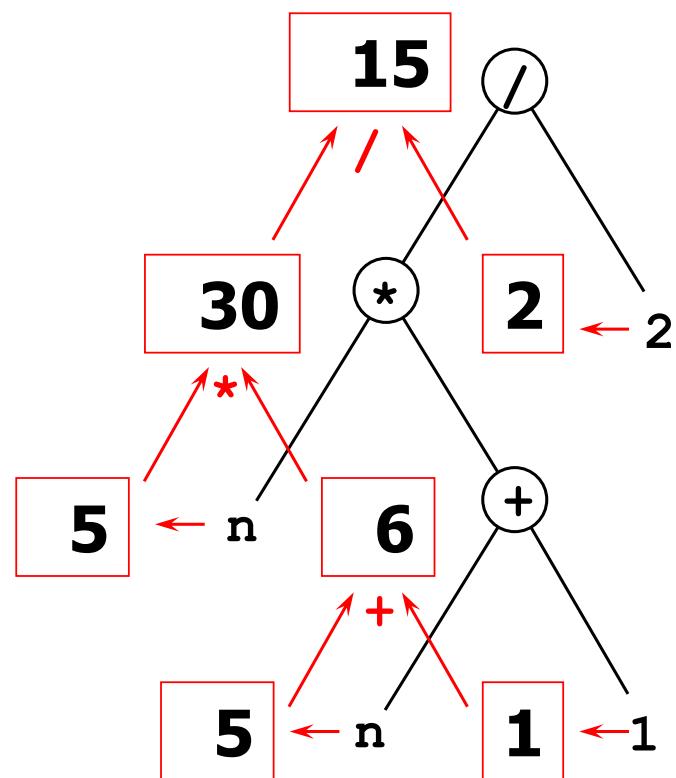
`n * (n + 1) / 2`
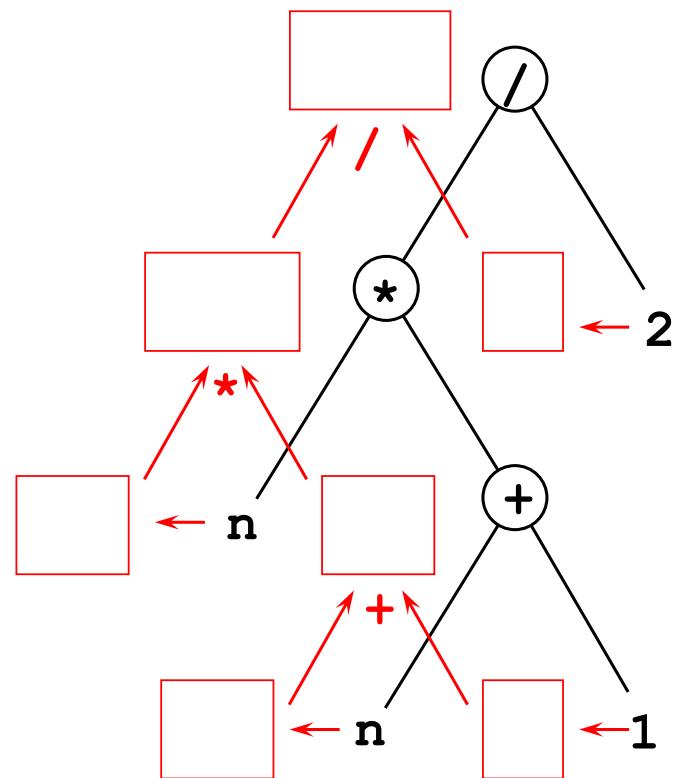
Ku každému výrazu
existuje **jednoznačný**
Syntaktický strom!

Vypočítanie hodnoty

Pre `n = 5`:

Každému výrazu je pomocou syntaktického stromu jednoznačne priradený formulár pre vyhodnotenie výrazu.
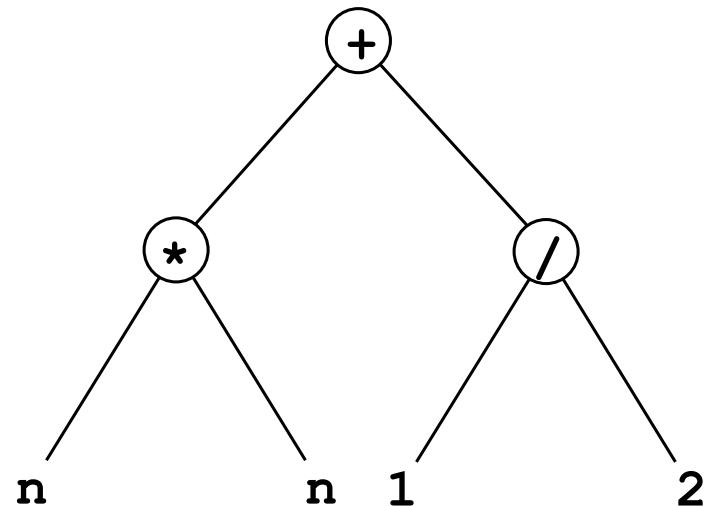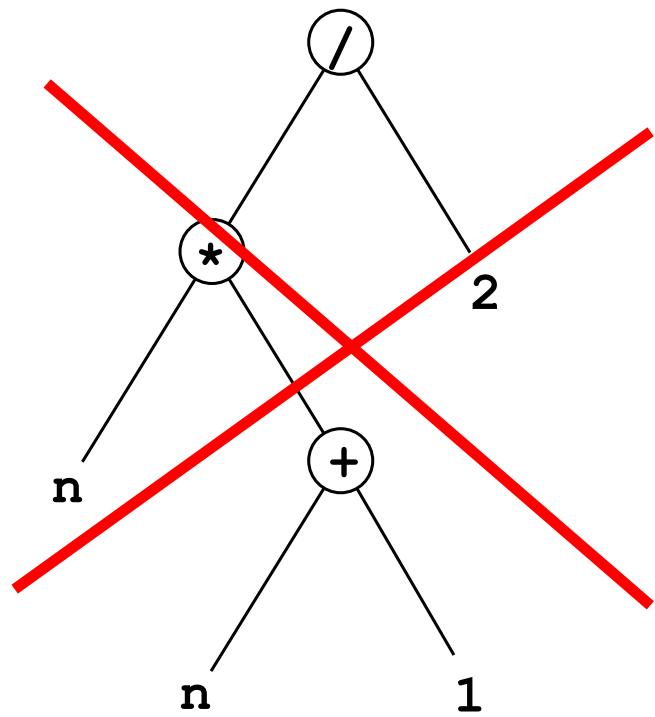
$$n * n + 1 / 2$$

$$n * n + 1 / 2$$

**Násobenie a delenie pred sčítaním**

$$- \ a \ + \ b$$

FEI STU
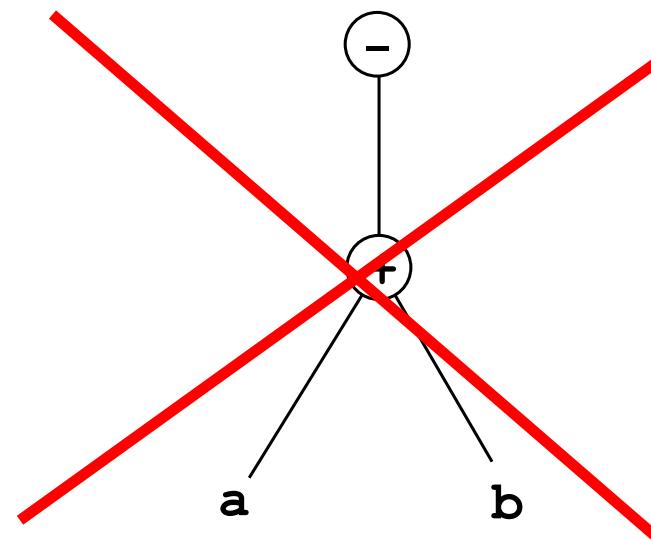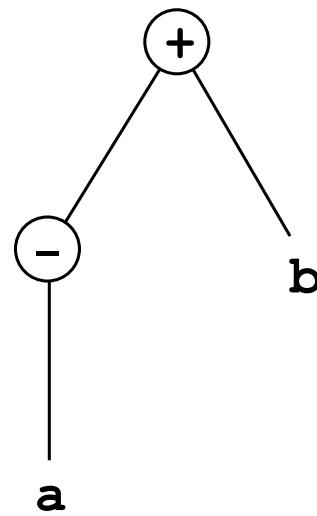
Ďalšie príklady
(unárne operácie pred binárnymi)
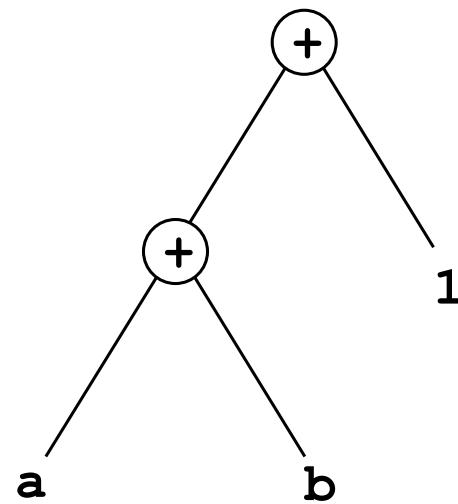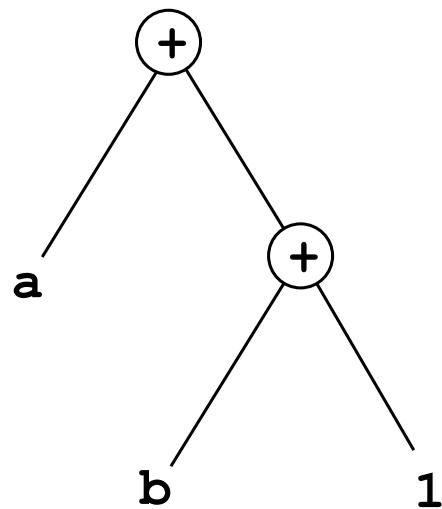
$$- \ a \ + \ b$$

unárne pred binárnymi
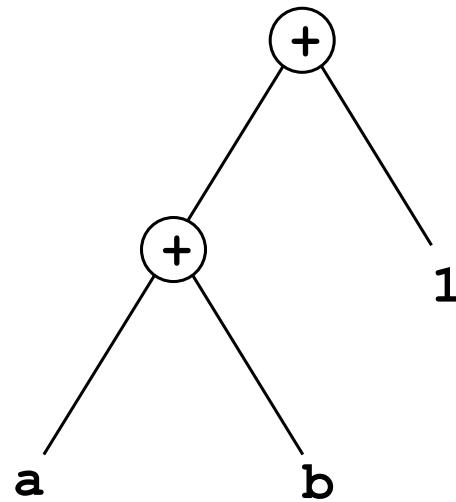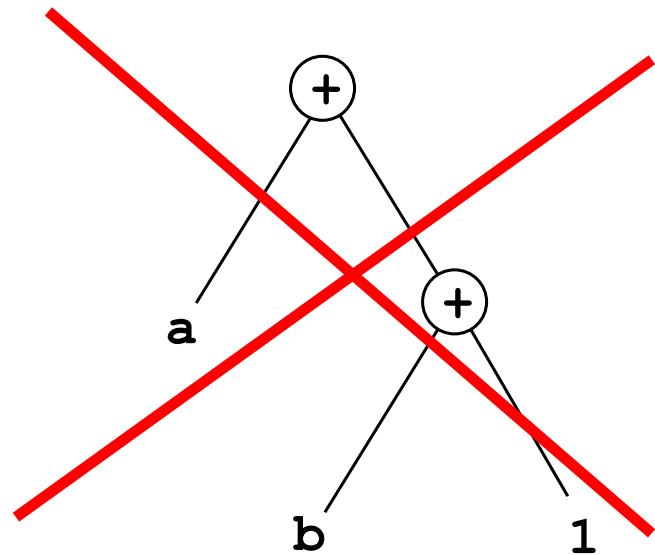
a + b + 1

a + b + 1
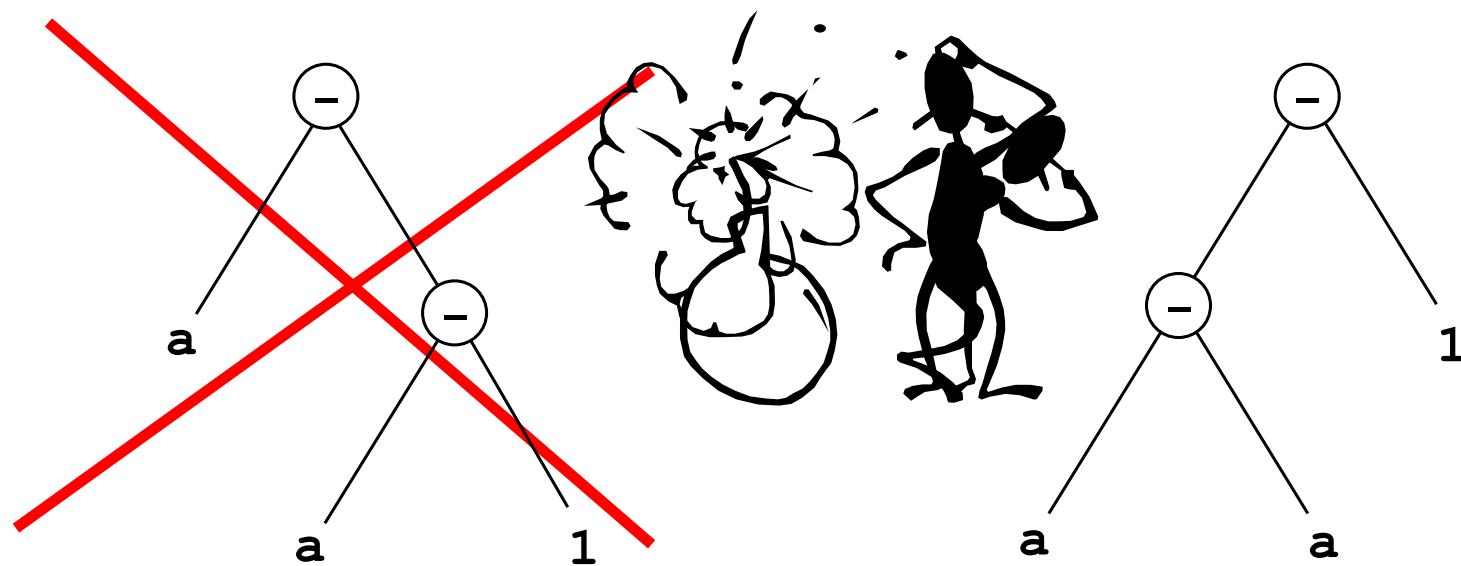
zľava doprava

$$a - a - 1$$

zľava doprava

```
n * (n + 1 / 2
- sum § j
)a + b(
Math.min(a;b)
(a + b) * c)
```

Ktoré výrazy sú korektné (prípustné)?

```
<Výraz> =

     <MultVýraz>      |
     <Výraz> "+" <MultVýraz>   |
     <Výraz> "-" <MultVýraz>


<MultVýraz> =

     <UnárnyVýraz>   |
     <MultVýraz> "*" <UnárnyVýraz> |
     <MultVýraz> "/" <UnárnyVýraz> |
     <MultVýraz> "%" <UnárnyVýraz>
```

Gramatika (pokračovanie)

```
<UnárnyVýraz> =
    <Identifikátor>                                            |
    <Konštanta>                                                |
    "(" <Výraz> ")"                                            |
    <Identifikátor> "(" <ZoznamArgumentov> ")"                 |
        "+" <UnárnyVýraz>                                      |
    "-" <UnárnyVýraz>


<ZoznamArgumentov> =
    <Výraz>                                                    |
    <ZoznamArgumentov> "," <Výraz>
```

opcionálne, t.j. môže byť vynechaný
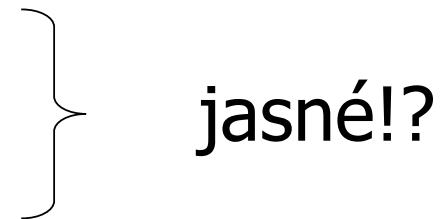
Gramatika (pokračovanie)

<Identifikátor> = ...

<Konštanta> = ...
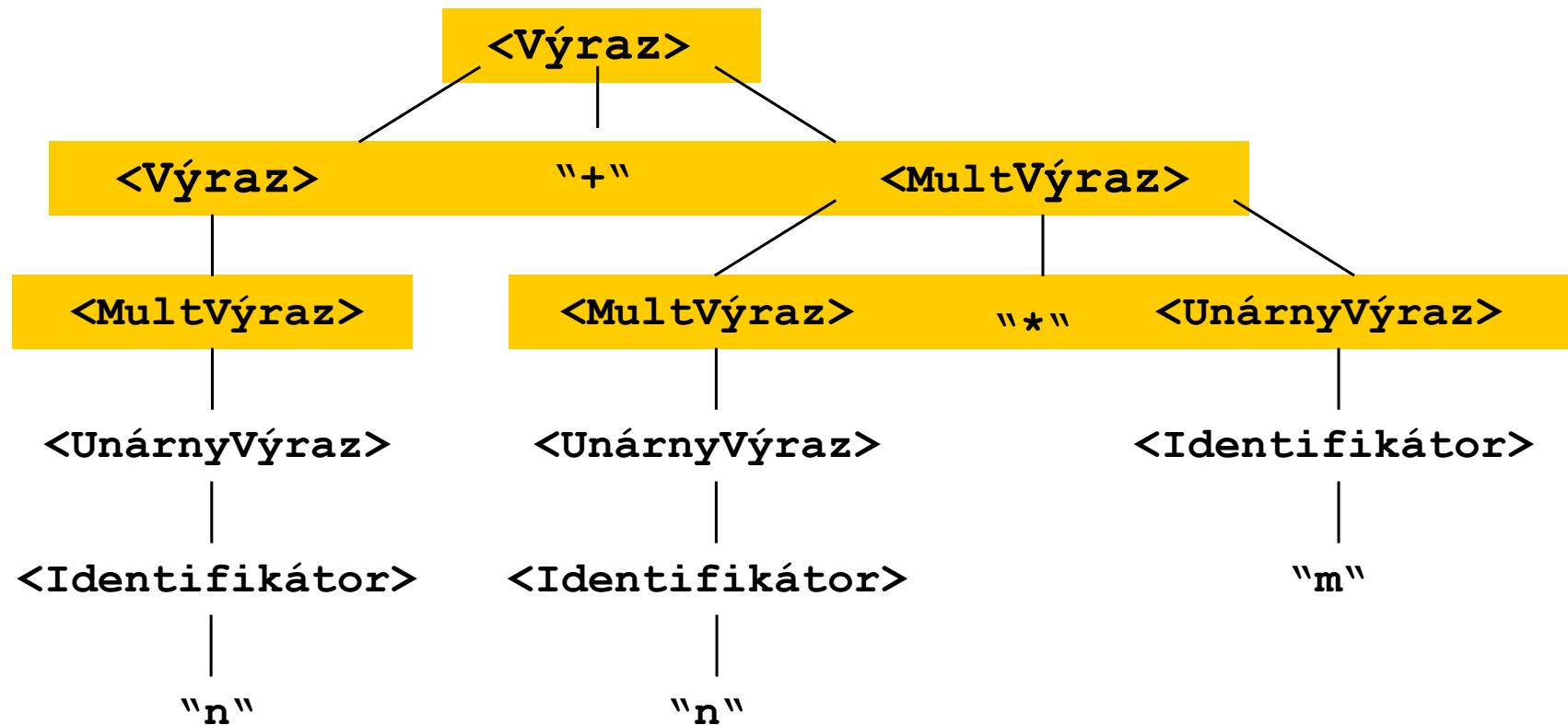
jasné!?

Táto notácia popisujúca syntax sa nazýva **EBNF** (Extended Backus-Naur-Form).
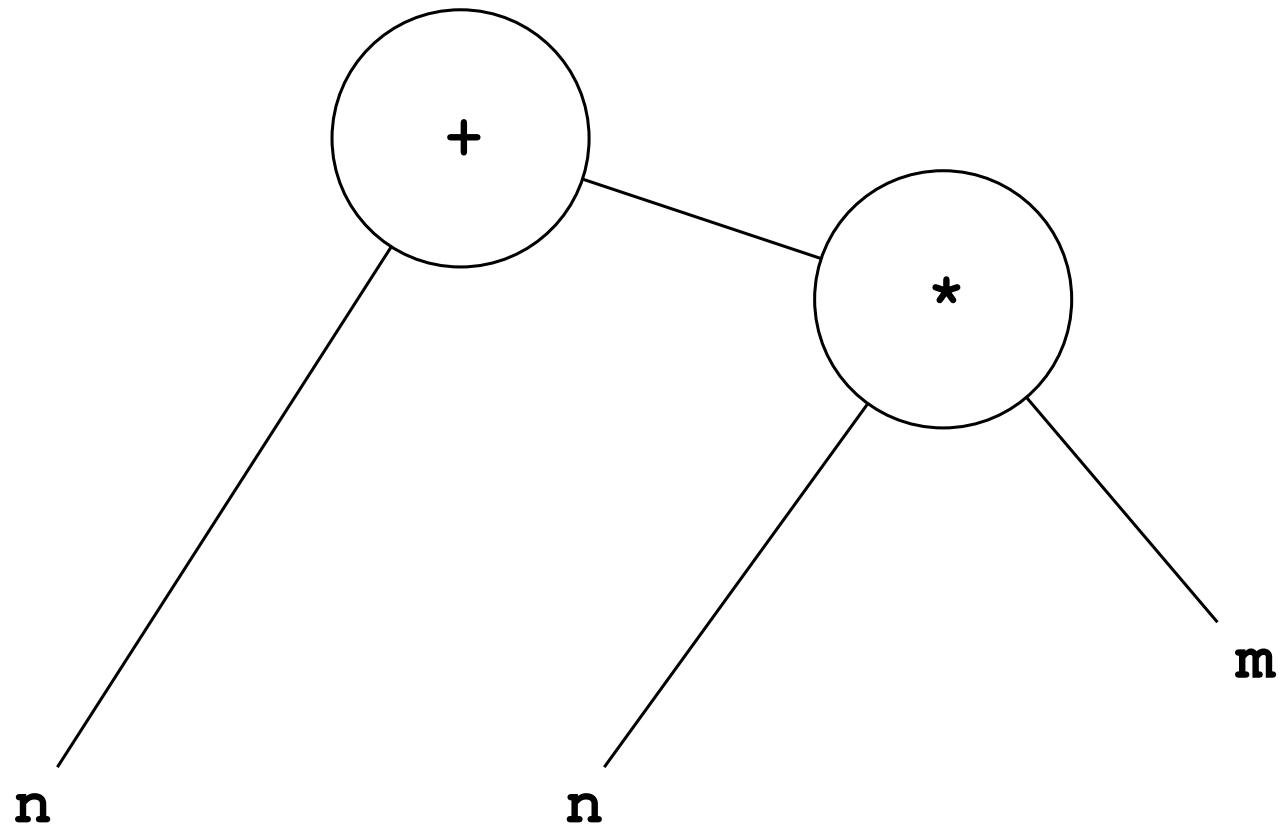
Syntax jazyka Java a C++ je definovaná veľmi podobnou notáciou

Odvodenie pre: `n + n * m`
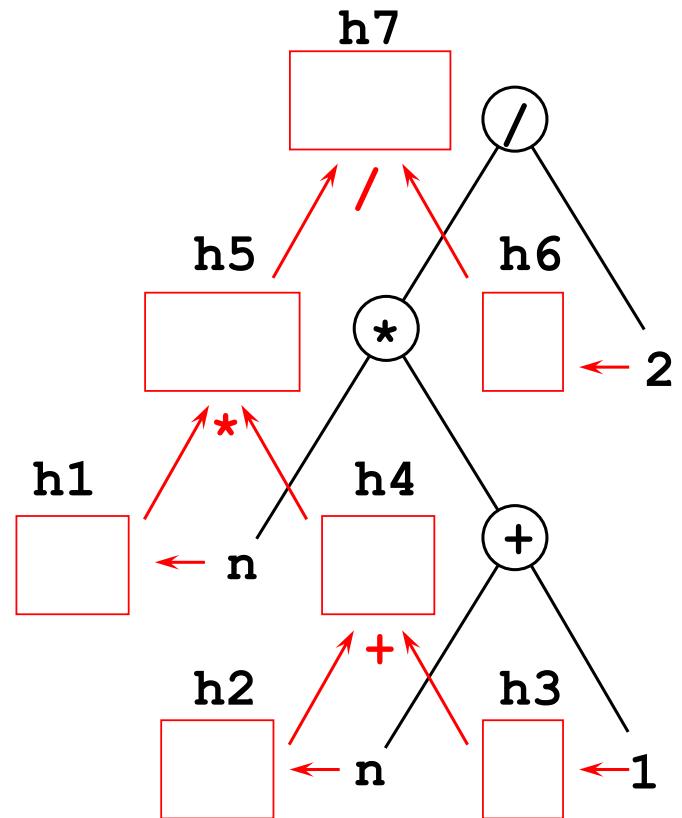
- Syntaktické pravidlá (v EBNF) definujú syntakticky prípustné výrazy.

- Odvodenie každého výrazu je jednoznačné.

- Výsledkom tohto odvodenia je syntaktický strom výrazu a jeho prostredníctvom vyhodnocovací formulár výrazu.

```
int h1 = n;
int h2 = n;
int h3 = 1;
int h4 = h2 + h3;
int h5 = h1 * h4;
int h6 = 2;
int h7 = h5 / h6;

return h7;
```

Lecture 2

2. Dátové typy

- Primitívne dátové typy
- Deklarácia premenných
- Priradenie
- Side effects

2.1. Primitívne dátové typy

- Doteraz: iba programy s celými číslami
- Celé čísla nie sú pre niektoré účely spracovania informácií vhodná a efektívna reprezentácia:
  - Funkcie nad reálnymi číslami
  - Spracovanie textu

Celé čísla

- **byte** −128 až 127
- **short** −32768 až 32767 (2 Byte) v Jave
- **int** -2147483648 až 2147483647 (4 Byte) v Jave
- **long** -9223372036854775808 až 9223372036854775807 (8 Byte) v Jave

- Pravé celé čísla (bez obmedzenia) neexistujú (ako primitívne dátové typy) !!!

- **`float`** ca. $-3*10^{38}$ až $3*10^{38}$ (4 Byte)
  ca. 6 bis 7 miest presnosť
  Príklady: 3.578671E+22F
                      0.0145456E-15F

- **`double`** ca. $-2*10^{308}$ až $2*10^{308}$ (8 Byte)
  15 miest presnosť
  Príklady: 1.234567897897E+220D
                      -3.956745656e-12

Znaky (Character)

- **char** Znaky v tzv. Kódovaní Unicode
  http://www.unicode.org/charts/web.html

- V Unicode su zobraziteľné „všetky znaky sveta" (2 Byte):
  `a`` , ... , `z` , `ä` , `ö` , `ü` , `ß` ,
  `A` , ... `Z` , ... , `0` , ... , `9` , `+` , `- ` ,
  `\t` , `\n` , `\r` , `\\` , `\`` , ...
  `\u2200` = $\forall$, `\u24B8` = ©

Znaky (Character)

- Znakové reťazce v Unicode! (Java)

- Znakové reťazce sú na rozdiel od znakov v dvojitých úvodzovkách:

```
"Toto je príklad pre jeden\n
znakový reťazec so zalomením riadku!"
```

Pozor!

- **'c'** je jeden znak!

- **"c"** je znakový reťazec, ktorý pozostáva z jedného znaku

- **"\u2200"** je tiež znakový reťazec, ktorý pozostáva z jedného znaku

- **"\\u2200"** je znakový reťazec, ktorý pozostáva zo 6 (!) znakov
  (\\ reprezentuje spätné lomítko Backslash)

Znakové reťazce nie sú prísne vzaté primitívny dátový typ! (vysvetlíme neskôr)

V Jave dátový typ String

Pravdivostné hodnoty

- **boolean (nie je v jazyku C)**

  hodnoty: **false** a **true**

- **V C pre pravdivostné hodnoty int – 0 je false a 1 je true**

Budeme používať

- **int**
- **double**
- **char**
- **boolean** a
- **String (nie je v jazyku C)**

> Iné programovacie jazyky poskytujú podobné primitívne dátové typy (viac alebo menej typov, iné názvy typov):
>
> ... **real**

- Pre iné dátové typy platia väčšinou analogické pravidlá

2.1. Deklarácie premenných

▮ Každá premenná má jednoznačne priradený dátový typ

▮ Typ premennej je pevne daný deklaráciou premennej:

```
int n;
int sum;
double x;
String name;
```

Deklarácie premenných

- Premenné môžu byť deklarované v texte programu na hociktorom mieste (ale iba jedenkrát, presnejšie vysvetlíme neskôr)
- Viaceré premenné môžu byť deklarované spolu:

```
int sum1, sum2, test;
double x, y, z;
```

EBNF: Deklarácie premenných

```
<Deklaration> = <Typ> <ZoznamIdentifikátorov> ";"

<Typ> =       "int" | "long" | "byte" | "short" |
        "double"  | "float" |
        "char"      |
        "boolean" |
        "String"

<ZoznamIdentifikátorov> =
 <Identifikátor>                           |

 < ZoznamIdentifikátorov > "," <Identifikátor>

<Identifikátor> = ...
```

- Identifikátory pozostávajú z reťazca **Písmen** a **Číslic** (v Unicode), pričom prvý znak musí byť písmeno
- Pojem písmeno je v tomto kontexte dosť široký: `ä`, `ö`, `ü`, `_`, `$`

```
NIE: '+', '-', '[', '%', '©', ...

Viac:    Character.isJavaIdentifierStart(c)
         Character.isJavaIdentifierPart(c)
```

Identifikátory – mená premenných

- Všetky písmená v mene sú dôležité
- Rozlišujú sa malé a veľké písmená:
    `Sum` a `sum` sú dva rozdielne identifikátory.

- Konvencia: Premenná začína s malým písmenom
- Alebo _temp

Typy Konštánt

Každá konštanta je jednoznačného typu:

- `0, -1, 4711, 32768, -1000000`
  sú typu `int` (aspoň pre nás, lebo short atď.)

- `0.1, -1.7359e-34, 0.0`
  sú typu `double`

- `'a', '©'`
  sú typu `char`

- `"Bla bla", "Test\n Test\n"`
  sú typu `String`

Typy výrazov

Každý výraz je jednoznačného typu:

```
int n, m;
double x, y;

1 * 2                      // je typu int
n  + m                     // je typu int
x    // je typu double
x / y                      // je typu double
x < y                      // je typu boolean
```

Typ operátorov

- Pre každý operátor (a pre každú funkciu) a operandy s daným dátovým typom je jednoznačne dané akého typu je výsledok!

- Príklady:

```
Math.min: int, int → int
Math.abs: int → int
  . + . : int int → int
  . < . : int int → boolean
```
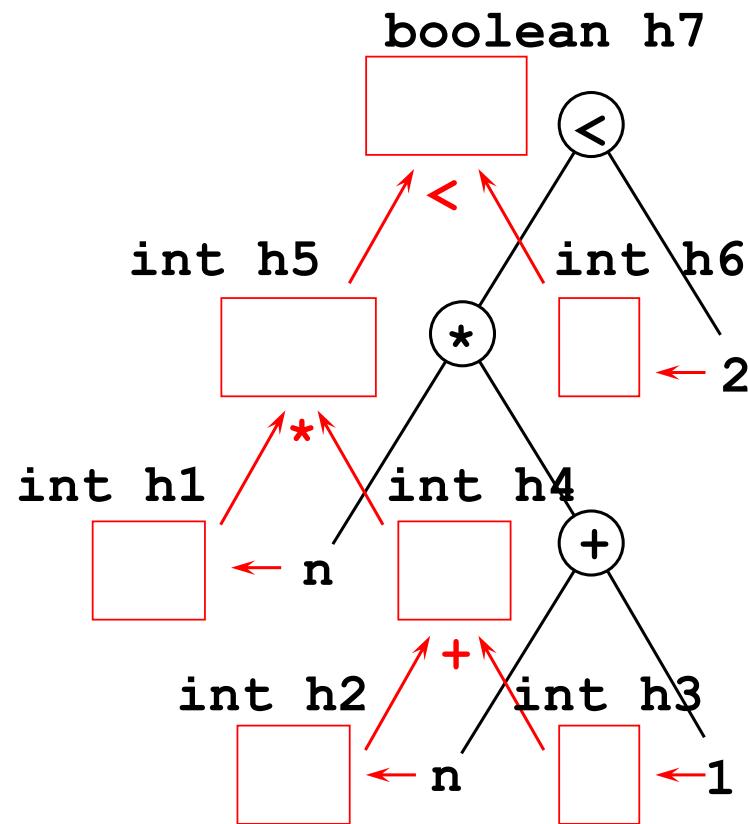
Typ operátorov

získame (induktívne) z Typu
konštánt,
premenných, a
operátorov
a funkcií

```
int n; ...
n * (n + 1) < 2
```

Polymorfizmus dátových typov

(polymorhph = „mnohoformný")

Operátor + je rezervovaný pre sčítanie dvoch celých čísel typu `int`!

Radi by sme však + použili pre sčítanie dvoch reálnych čísel typu `double`!

Dá sa to?

Polymorphismus

■ Dá: Keďže typy subvýrazov, ktoré sú operandami + , sú známe, môžu byť tieto typy použité na identifikáciu operátora + . Operátor + označuje rôzne operácie:

$$. + . : \texttt{int int} \rightarrow \texttt{int}$$

$$. + . : \texttt{double double} \rightarrow \texttt{double}$$

$$. + . : \texttt{String String} \rightarrow \texttt{String}$$

$$\texttt{"abc" + "def"} \; je \; \texttt{"abcdef"}$$

Ďalšie (implicitné) formy

```
. + . : int double → double
. + . : double int → double
. + . : int String → String
. + . : String int → String
. + . : double String → String
. + . : String double → String
. + . : String boolean → String
. + . : boolean String → String
```

Vždy implicitná konverzia na typ **double** resp. **string**

Ďalšie operátory

- To isté platí pre ďalšie aritmetické operátory: `-, *, /`

- `%` (Zvyšok po modulo delení) nie je definovaný pre reálne čísla

- Pre znakové reťazce nie sú definované operátory: `%, -, * a /`

`"abc" + 1` `je "abc1"`

`1 / 2` `je 0` (celočíselné delenie)

`1 / 2.0` `je 0.5` (reálne delenie)

`"abc" + false` `je "abcfalse"`

`"abc" + (1 + 2)` `je "abc3"`

Čo je výsledkom výrazu `"abc" + 1 + 2` ?

logické operácie:

- **!** Negácia (unárna)
- **&&** Konjunkcia „a súčasne" (binárna)
- **||** Disjunkcia „alebo" (binárna)

Porovnania:

- **==** Test na rovnosť (binárna)
- **<, <=, >, >=** menšie atď. , resp. väčšie (binárne)
- **!=** Test na nerovnosť (binárna)

iba pri číslach

Priorita operátorov

- unárne operátory   najvyššia priorita
- `* / %`
- `+ -`
- `< <= > >=`
- `== !=`
- `&&`
- `||`   najnižšia priorita

3. Priradenie
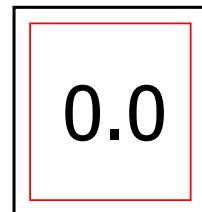
- Hodnota výrazu môže byť priradená premennej rovnakého typu:

```
int n, m;
  boolean b;

  ...
  m = n * (n + 1);
  b = m < n;
```
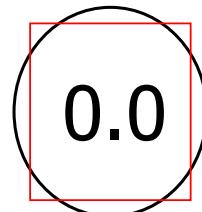
> Operátor priradenia = má najnižšiu prioritu!

```
double x;
x = 0.0;
```

0.0

```
int n;
n = 0.0;
```

0.0

Ale n = (double) 0.3 funguje (výsledok 0)

- Unárne operátory `(int),(float)` a `(double)` zmenia typ výrazu (type casting).

- Pozor:

  - Možná strata presnosti

  - Pri prekročení rozsahu daného typu (pri pretečení) žiadne chybové hlásenie v Jave a v C; čo môže viesť k zlému výsledku!

```
int n;
n = 0;
```

môžu byť zapísané spolu:

```
int n = 0;
```

V princípe musí byť **lokálnej** premennej priradená hodnota predtým ako je použitá v nejakom výraze, ktorý sa bude vyhodnocovať.

Globálne premenné sú inicializované automaticky.

4. Špeciality

- Skratky
- Operácie v postrannými efektami
- Skrátené vyhodnotenie logických výrazov

Skratky

- Pre premennú **i** typu int znamená

  - `i++;` a `++i;` skrátený zápis pre
    `i = i + 1;`

  - `i--;` a `--i;` skrátený zápis pre
    `i = i - 1;`

Skratky

- Pre aritmetický výraz `e` a premennú `x (int / double)` znamená
    - `x *= e;` skratku pre `x = x * e;`
    - `x /= e;` skratku pre `x = x / e;`
    - `x %= e;` skratku pre `x = x % e;`
    - `x += e;` skratku pre `x = x + e;`
    - `x -= e;` skratku pre `x = x - e;`

- Konštrukcie `i++`, `++i`, `i--` und `--i` sú výrazy typu `int`!

- `x = i++;` a `x = i--;`

  Premennej `x` je priradená hodnota `i` **pred** zvýšením, resp. znížením `i`.

- `x = ++i;` a `x = --i;`

  Premennej `x` je priradená hodnota `i` **po** zvýšení, resp. znížení `i`.

```
r = n++ * n--;
```
    má rovnaký efekt ako
```
r = n * (n + 1);
```
```
N = 5
5 * X
N <= 6
X = 6
R = 5*6
N <= 5
if (true) r = n* (n+1) else r = ++n;
```
```
r = n++ * n;
```
    má rovnaký efekt ako
```
r = n * (n + 1);
```
```
n = n + 1;
```
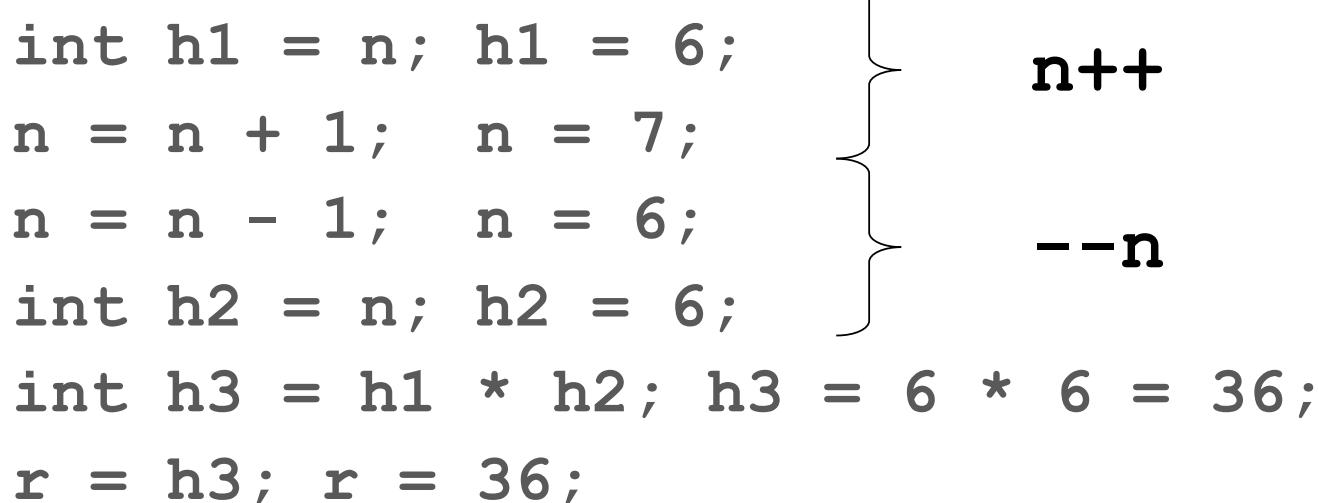
```
r = n++ * --n;
```

Vyhodnotenie v Jave cez elementárne operácie: (pre n = 6)

```
int h1 = n; h1 = 6;
n = n + 1;  n = 7;
n = n - 1;  n = 6;
int h2 = n; h2 = 6;
int h3 = h1 * h2; h3 = 6 * 6 = 36;
r = h3; r = 36;
```

n++

--n

Pozor! Poradie vyhodnocovania (priradenie temporálnym premenným h1 a h2) je dôležité!

- Je povolené (**ale nebudeme to (často:-) používať**) použiť priradenie priamo vo výraze!

- Príklad:

> Toto **nie je** porovnanie!

```
int r2;
int r1 = (r2 = n * (n + 1) ) / 2;
```

- = má najnižšiu prioritu z operátorov (okrem zátvorky) a vyhodnocovanie prebieha zľava doprava, preto **r2**

```
= n * (n + 1) ;
            r1 = r2 / 2;
```

Skrátené vyhodnotenie **&&**

- Keď je prvý operand logického súčinu (**&&**) vyhodnotený ako **false**, vyhodnotenie druhého operandu sa nevykoná a výsledok je **false**.

- Príklad (s **int n = 0**):

boolean b = (n != 0) && (5 / n >= 1)

Skrátené vyhodnotenie ||

- Keď je prvý operand logického súčtu (`||`) **true**, vyhodnotenie druhého operandu sa nevykoná a výsledok je **true**.

- Príklad (s `int n = 0`):

```
boolean b = (n = = 0) || (5 / n >= 1)
```

FEI STU

Komentare

5. Komentáre

- Komentáre sú pre dokumentáciu programu KĽÚČOVÉ

- Komentáre v skutočnosti nepatria k programu, sú ignorované počítačom

- Rôzne varianty komentárov

```
// po dvoch lomítkach bude
// text do konca riadku

// ignorovaný, napr.

int i; // tu deklarujeme

i = 0; // tu inicializujeme i
```

```
/* Dlhšie komentáre sa píšu medzi
kombináciu lomítko-hviezdička na začiatku
komentáru a hviezdička lomítko na konci
komentáru */
```

```
/**

    Komentáre s dvojitou hviezdičkou sú
automaticky použité na vytvorenie
dokumentácie. Taktiež je možné zadať
špeciálne informácie k programu, napr:

    @author The Author

*/
```

Príkazy

- Elementárne príkazy

- Postupnosti príkazov a bloky

- Podmienené príkazy

- Iteračné príkazy - cykly

6.1 Elementárne príkazy

- Deklarácie

- Priradenia

```
<Príkaz> =          <Deklarácia>          |
    <Priradenie>    |

    ...             // (neskôr viac)


<Deklarácia> =      ...             // (pozri stranu 16, Fólie3)


<Priradenie> =      <Identifikátor> "=„ <Výraz> ";" |

    ...             // (neskôr viac)
```

6.2 Bloky príkazov

- Programy sú v podstate postupnosti príkazov:

```
<PostupnosťPríkazov> =
    <Príkaz>   |
    <PostupnosťPríkazov> <Príkaz>
```

- Príkazy je možné spájať do blokov

```
...
int r;
{ int h1 = n;
  n = n + 1;
  i = i - 1;
  int h2 = i;
  int h3 = h1 * h2;
  int h4 = n;
  r = h3 / h4;
}
System.out.println("r = " +
```

Blok je v Jave rovnako ako v C označený zloženými zátvorkami;

Premenná, ktorá je deklarovaná v bloku, môže byť použitá iba v danom bloku, je platná iba v rámci bloku. Pred a za blokom je nedeklarovaná a teda neznáma.

Príklad: Blok

```
int i = 1;
{ int i = 0;
   ...
}
```

Premenná môže byť v bloku deklarovaná iba jedenkrát v bloku. V jazyku C môže byť deklarová v bloku aj v podbloku, pričom premenná v podbloku prekrýva premennú v nadbloku. V Jave nie je povolené deklarovať premennú v nadbloku aj podbloku.

```
<Príkaz> =...  |            // (ako doteraz)
  <Blok>          |
                  ...                    // (neskôr viac)


<Blok> = "{" [ <PostupnosťPríkazov> ] "}"
```

6.3 Riadiace štruktúry

6.3.1 Podmienené príkazy

Príklad 1:
Náš program `Suma` počíta pre prirodzené čísla zmysluplný výsledok!
Ak je zadané záporné číslo, mohol by program užívateľa informovať, že počíta sumu iba pre nezáporné čísla.

FEI STU
84

Príklad v Jave

```java
public class Suma2
{
  public static void main (String[] args)
  {
  int n = Integer.parseInt(args[0]);

    if (n >= 0)

    {
       int x = n * ( n + 1) / 2;
       System.out.println(„Suma: " + x);

    }

    else

            System.out.println("Prosim ziadne zaporne cisla!");

  }
}
```
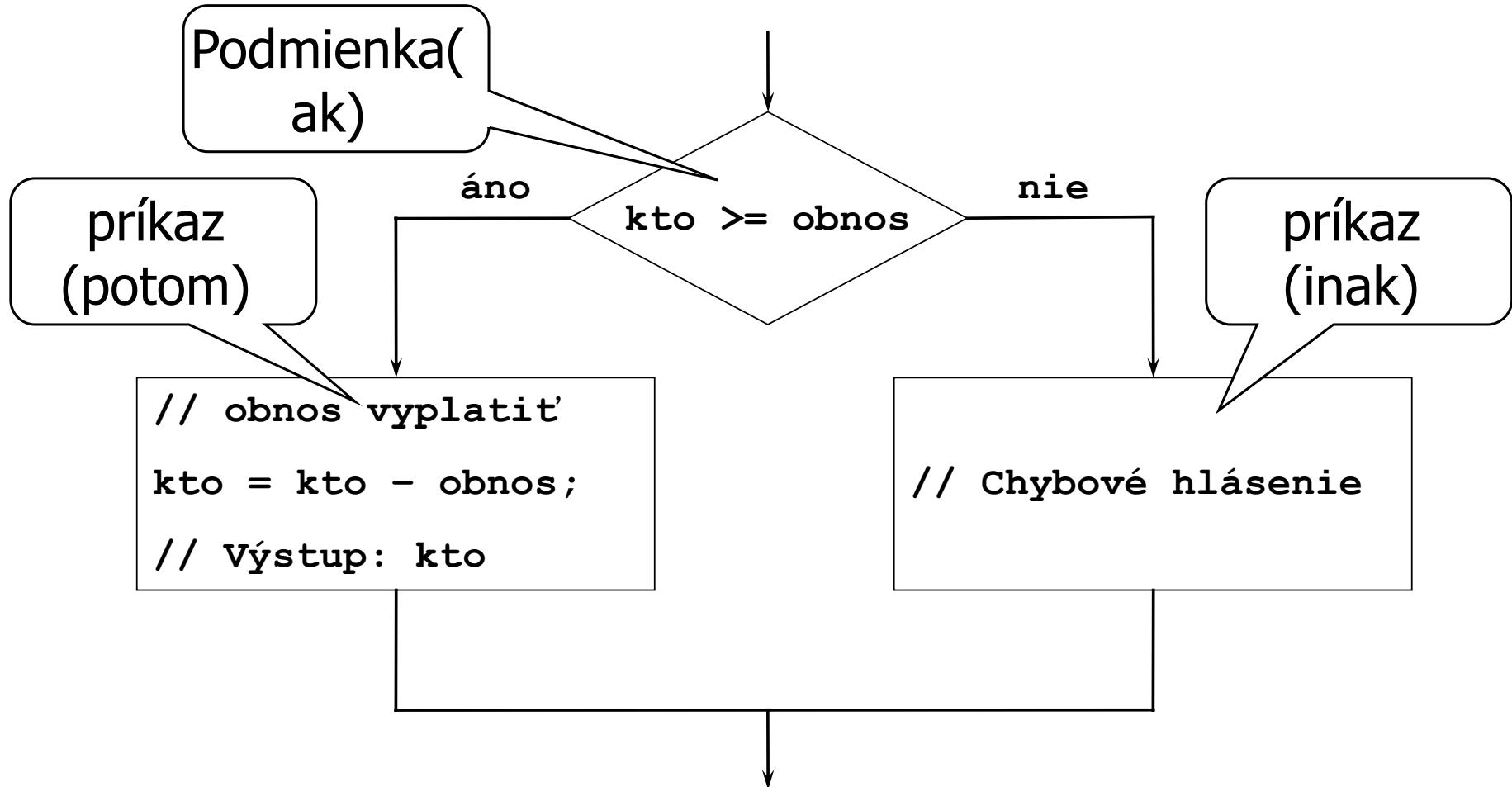
Príklad 2

Príklad 2:

Zákazník chce vybrať zo svojho konta so stavom konta uloženom v premennej `kto` hotovosť `obnos`.

Banka nepovoľuje možnosť ísť do mínusu!

```
...

if ( kto >= obnos )
{ // vyplatit(obnos); (Vyplatenie nie je naprogr.)
  kto = kto - obnos;
  System.out.println("Vas stav konta je: " + kto);
}
else
 System.out.println("Na konte nie je dostatok penazi.");


...
```

Príklad 2 (Pokračovanie)

Príklad 2 (pokračovanie):

Zákazník chce aby mu bol vyplatený aspoň zvyšok, ktorý sa na konte nachádza.

```java
if ( kto >= obnos )
{ // vyplatit(obnos); (Vyplatenie nie je naprogr.)
  kto = kto - obnos;
  System.out.println("Vas stav konta je: " + kto);
}
else if (kto > 0)
{ // vyplatit(kto);
  kto = 0;
  System.out.println("Vas stav konta je: " + kto);
}
else
{ System.out.println("Na konte nie je dostatok penazi.");
 }
```
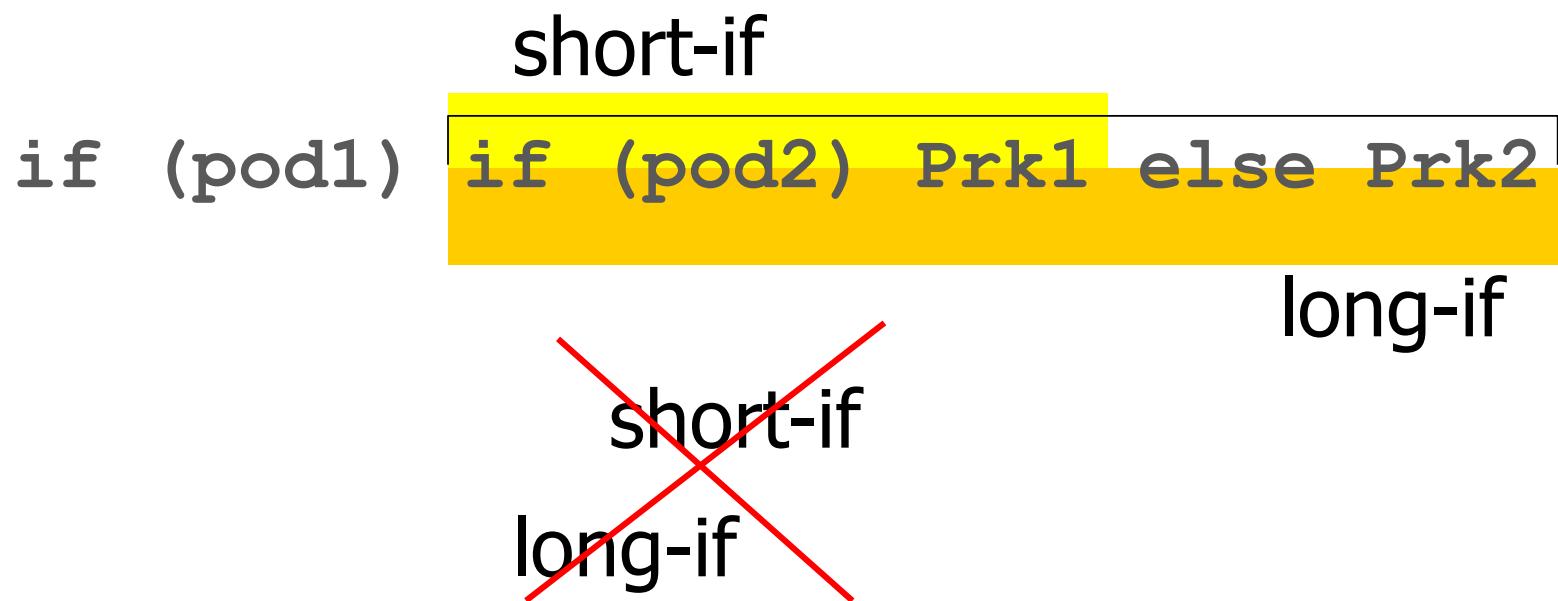
```
<Príkaz> =      ...                                              |
   <PodmienenyPrikaz>                                            |
   ...


<PodmienenyPrikaz> =
 "if" "(" <Vyraz> ")" <Prikaz>                                   |
 "if" "(" <Vyraz> ")" <Prikaz> "else" <Prikaz>                  |
```

Short-If: V inak-prípade je Príkaz ignorovaný!

Vyraz musí byť typu `boolean`!

Long-If: V inak-prípade bude vykonaný druhý príkaz!

Problém: Jednoznačnosť

Problém: Jednoznačnosť

- „Naša" EBNF pre podmienku neposkytuje vždy jednoznačné riešenie!

- Skutočná EBNF (pozri napr. dokumentáciu Javy), poskytuje vždy jednoznačné riešenie; táto EBNF je ale pomerne komplikovaná a ťažko čitateľná!

- Namiesto toho:

Pravidlo:

V dlhom If-príkaze (long-if) nikdy nie je ako prvý príkaz vnorený krátky If-príkaz (short-if)!

Lepšie: Zamedziť nedorozumeniu používaním zátvoriek (blokov):

```
if (pod1) { if (pod2) Pr1 else Pr2 }
if (pod1) { if (pod2) Pr1 } else Pr2
```

Cykly

6.3 Riadiace štruktúry (pokr.)

6.3.2 Iteračné príkazy (Cykly)

Príklad 1:

Chceme násobiť postupne všetky čísla od **1** po **n**, t.j. vypočítať faktoriál čísla **n**:

$$\texttt{n! = 1 * 2 * 3 * ... * n} = \prod_{i=1}^{n} i$$

Príklad

```
...

 int prod = 1;
  int i = 1;
  while (i <= n)
  { prod = prod * i;
     i = i + 1;
  }


...
```

```
...

int prod = 1;
 int i = 1;
 while (i <= n)
 { prod = prod * i;
   i = i + 1;
 }


...
```


Podmienka
(kým platí)

i <= n

nie

áno

prod = prod * i;
i = i + 1;
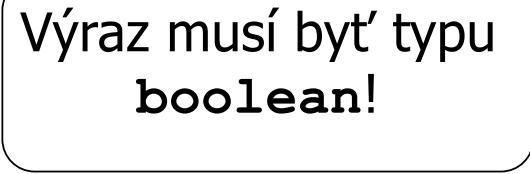
Príkaz

Príklad v Jave

```java
public class Faktorial
{
  public static void main (String[] args)
  {
    int n = Integer.parseInt(args[0]);
    int prod = 1;
    int i = 1;
        while (i <= n)
        { prod = prod * i;
        i = i + 1;
        }
  System.out.println("Faktorial cisla "+n+" je "+prod);
  }
}
```

```
<Príkaz> =      ...                          |
   <IteračnýPríkaz>                           |
   ...


<IteračnýPríkaz> =
 "while" "(" <Výraz> ")" <Príkaz>
```

Výraz musí byť typu **boolean**!

Kto garantuje, že podmienka iteračného cyklu sa stane niekedy nepravdivá (false) a tým sa ukončí cyklus?

Programátor, t.j. Vy!!!

Výpočet odmocniny $\sqrt{x}$ pomocou delenia intervalu!

Idea:

- Zvolíme interval [d, h], v ktorom sa $\sqrt{x}$ určite nachádza.

- Rozdelíme interval na dva intervaly [d,s] und [s,h];

- Zvolíme interval, v ktorom sa $\sqrt{x}$ určite nachádza a opakujeme procedúru, kým nedosiahneme požadovanú presnosť, t.j. kým interval nie je dostatočne malý.

Príklad: Výpočet odmocniny

```
// Vstup:  double x;  x >= 0
double d = 0;          // Spodná hranica počiatočného intervalu
double h = x + 1;      // Horná hranica počiatočného intervalu
double eps = 1.0e-10;  // želaná relatívna presnosť
double s = (d + h) / 2;

while ( (h – d) / s > eps)
{if (s * s >= x)
    h = s;
  else
    d = s;
  s = (d + h) / 2;
}
// Výstup: s
```

Pozorovanie

Iteračný príkaz tohto programu

(pre x >= 0) sa vykoná aspoň raz.

Bolo by lepšie podmienku testovať na konci cyklu.
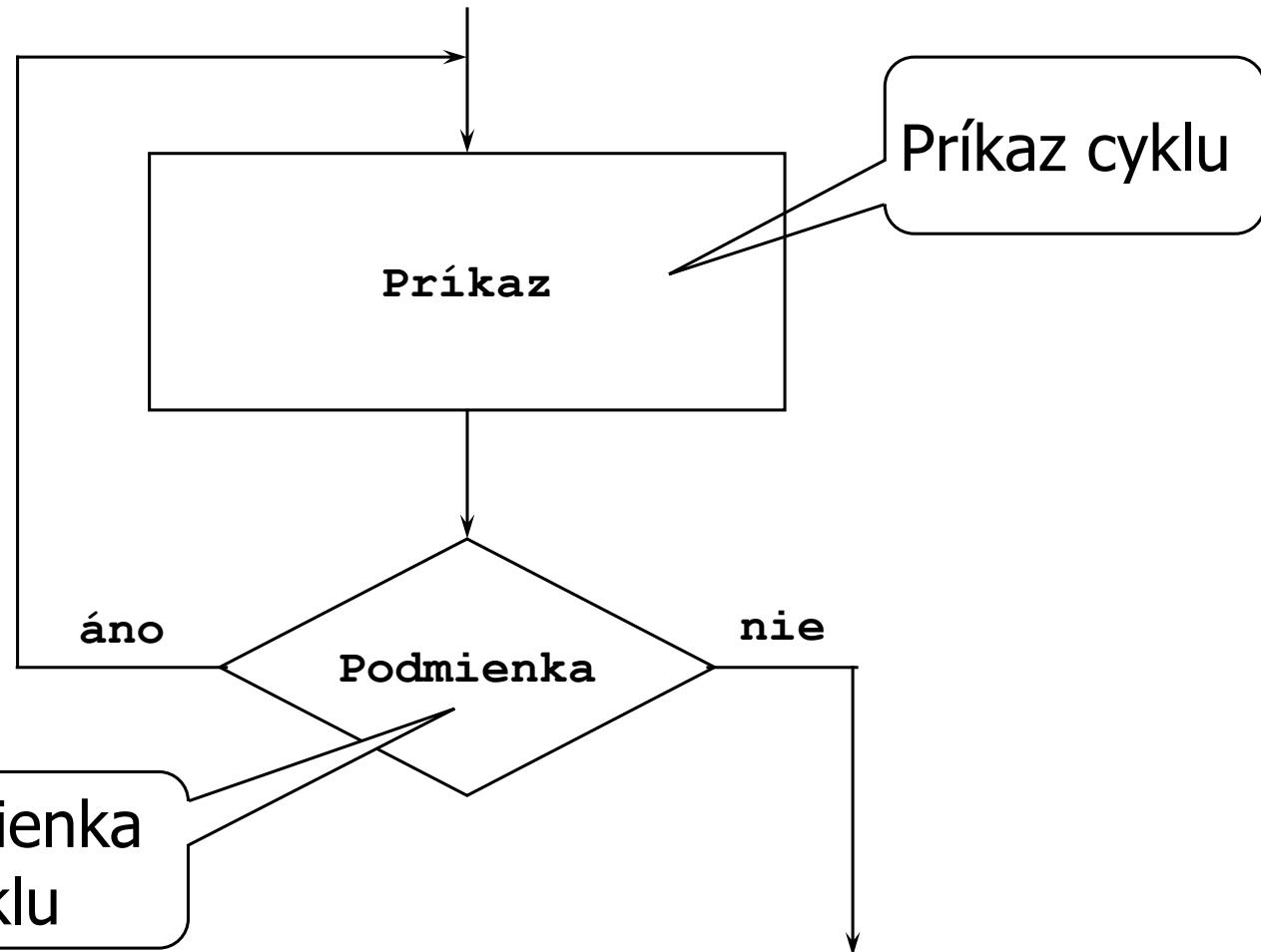
do-while cyklus

```
// Vstup:  double x;  x >= 0
double d = 0;          // Spodná hranica počiatočného intervalu
double h = x + 1;      // Horná hranica počiatočného intervalu
double eps = 1.0e-10;  // želaná relatívna presnosť
double s = (d + h) / 2;

do
{if (s * s >= x)
    h = s;
  else
    d = s;
  s = (d + h) / 2;
} while ( (h - d) / s > eps);
// Výstup: s
```
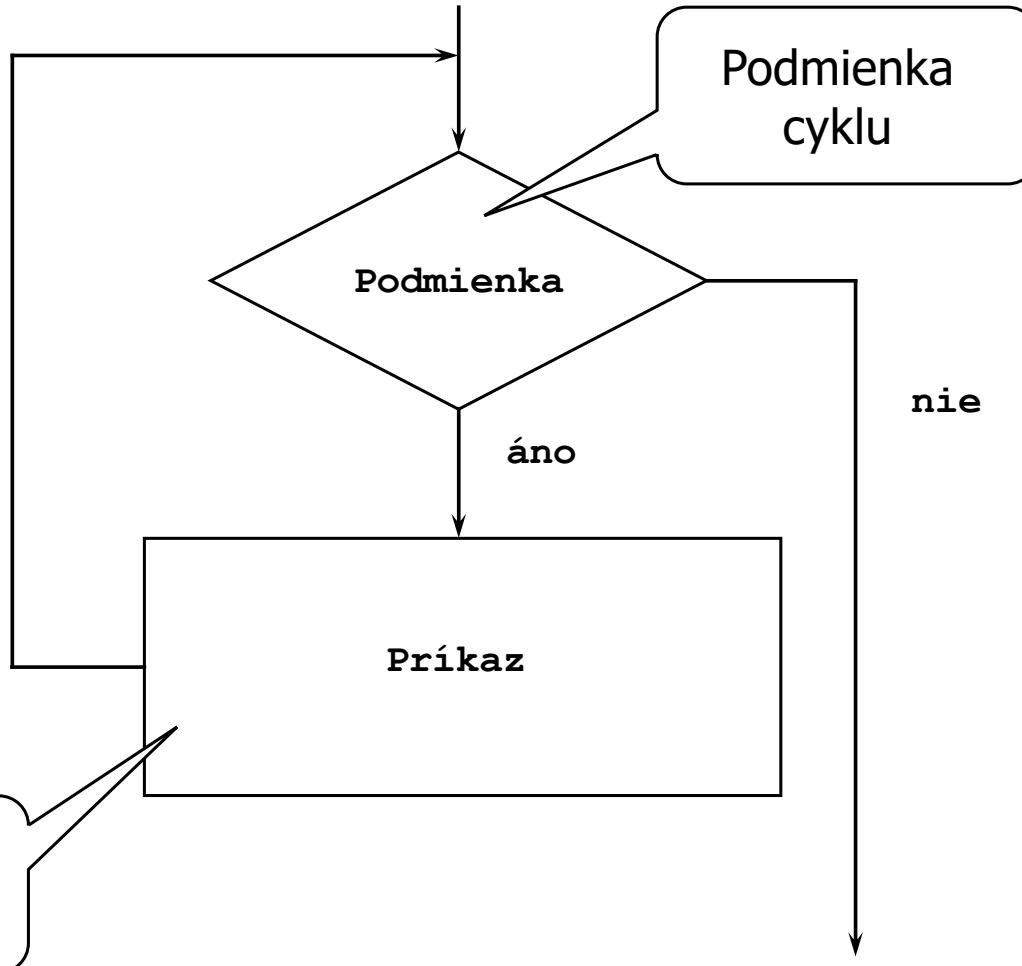
Vývojový diagram pre while-do cyklus

Príkaz cyklu nemusí byť nutne vykonaný.

Podmienka cyklu

Podmienka

nie

áno

Príkaz

Príkaz cyklu

Ďalšie možnosti cyklov (v iných jazykoch, nie Java a C)
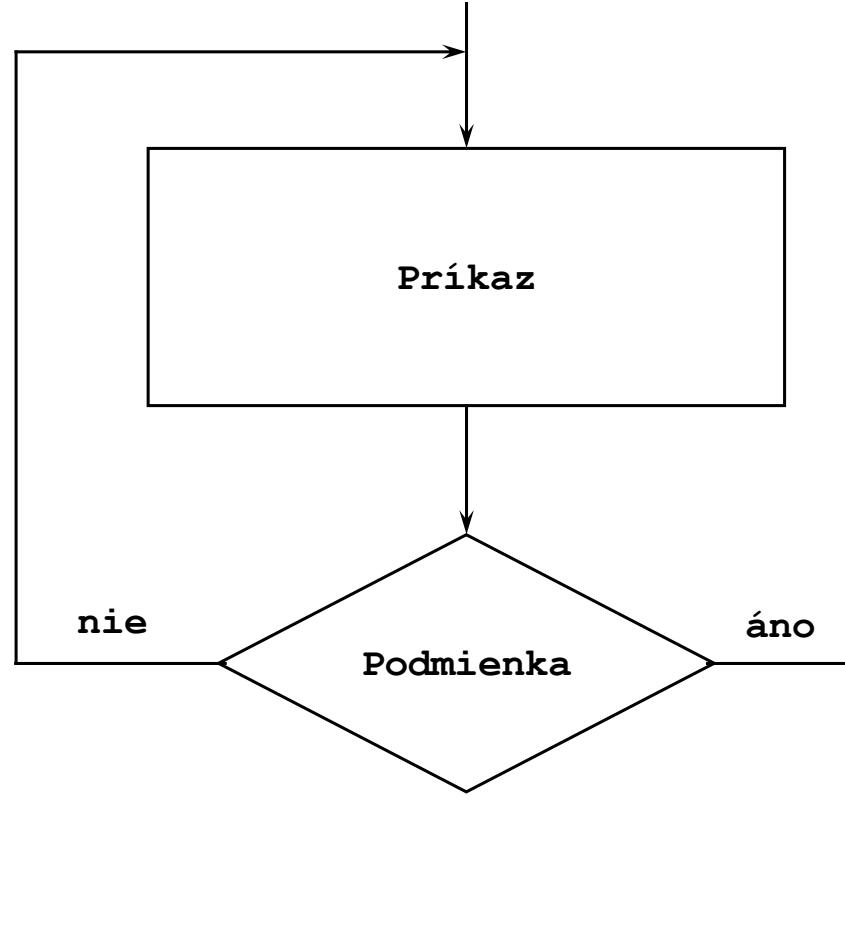
```
repeat
  Prikaz
until ( Podmienka );
```

Zodpovedá v Jave a v C:
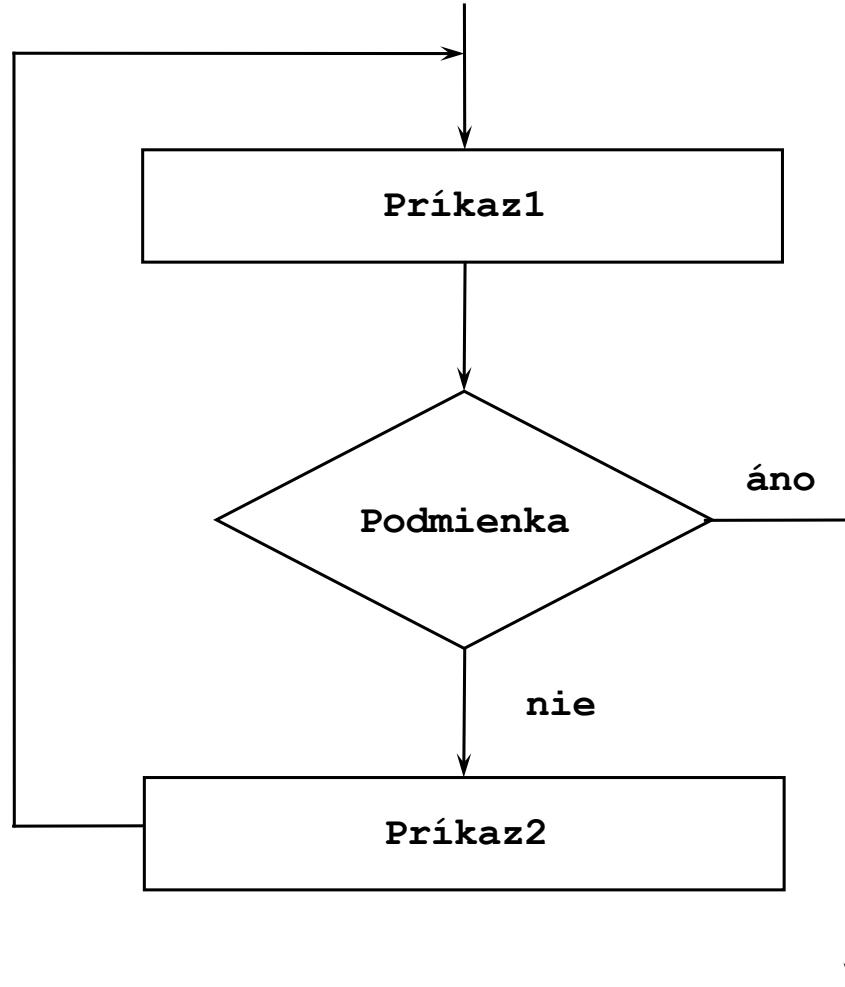
```
do
  Prikaz
while ( !Podmeinka );
```

Ďalšie možnosti cyklov (v iných jazykoch, nie Java a C)

```
repeat
{
   Prikaz1
until ( Podmienka );
   Prikaz2
}
```

Iteračné príkazy nad intervalom

▪ V mnohých prípadoch prechádza nejaká premenná (počítadlo) v cykle hodnoty vopred známeho rozsahu (napr. 1 ... n)
Príklady: `Suma, Faktorial`

▪ Pre takéto cykly existuje špeciálna konštrukcia: príkaz `for`

```
double product = 1.0;

   for (int i = 1; i <= n; i++)
   { product = product * i; }
```

```
Pre porovnanie:
int i = 1;
while (i <= n)
{ product = product * i;
   i = i + 1;}
```

Advantages of cycle `for`

- Jasné oddelenie jednotlivých častí cyklu (Initializácia, Podmienka, Iterácia) a príkazu cyklu

- Keď v príkaze cyklu iteračnej premennej-počítadlu a premenným použitým v podmienke nie je priradená žiadna hodnota, je možné priamo z hlavičky usúdiť, či cyklus terminuje.

Zoznam deklarácií alebo zoznam priradení

Zoznam priradení

```
for (Init; Pod; Iter)
      Príkaz
```

Ľubovoľný príkaz

Recommendations

- V cykle for použite iba jednu iteračnú premennú

- V časti „iterácia" cyklu for použite jednoduchú iteráciu iteračnej premennej (napr. **i++** alebo **i--**)

- V časti „podmienka" použite iba porovnanie (napr. **i < n** alebo **i >= 0**)

- Nepriraďujte iteračnej premennej a premennej vystupujúcej v podmienke žiadnu hodnotu v príkaze cyklu

Recommendations (continued)

- Používajte podmienku **i < …** resp. **i <= …** iba v spojení s **i++** alebo **i= i + c**

- Používajte podmienku **i > …** resp. **i >= …** iba v spojení s **i--** alebo **i= i - c**

- Nepoužívajte **i != …**

- Pre iteračné premenné používajte celočíselné premenné

Recommendations (continued)

- Používajte cyklus **for**, keď jeho použitie koncepčne zodpovedá cyklu nad intervalom (cyklu s počítadlom) – inak použite radšej cyklus **while (do while)**

■ V súlad s EBNF definíciou príkazov môžu byť všetky konštrukcie pre príkazy (`if-else` príkaz, cykly) vnorené (zložené)

■ Táto skutočnosť sa hojne využíva!

Example in Java

```java
public class Trojuholnik_06
{
  public static void main (String[] args)
  { int vyska = Integer.parseInt(args[0]);

    for (int riadok = 1; riadok <= vyska; riadok++)
    { for (int stlpec = 1; stlpec <= riadok; stlpec++)
      {
        System.out.print("*");
      }
      System.out.println();
    }
  }
}
```

Example – output to console

```
>javac Trojuholnik.java
>java Trojuholnik 7
*
**
***
****
*****
******
*******
```
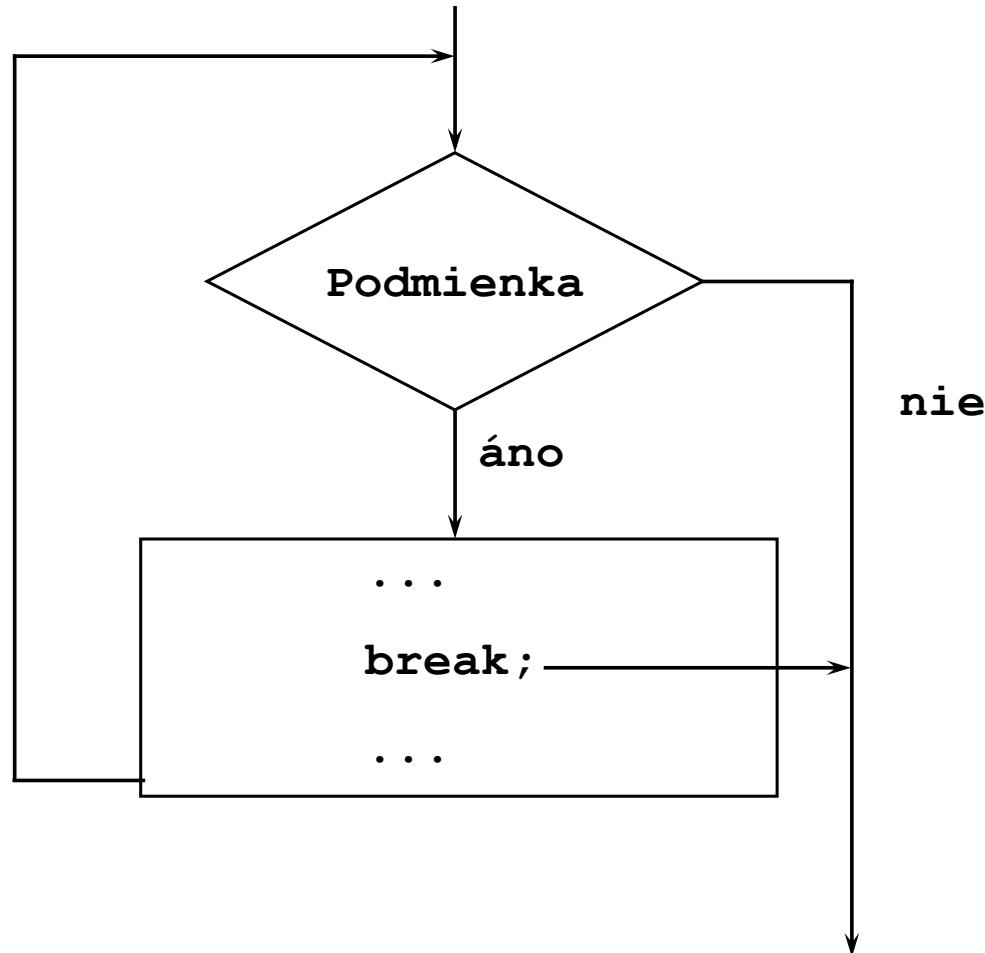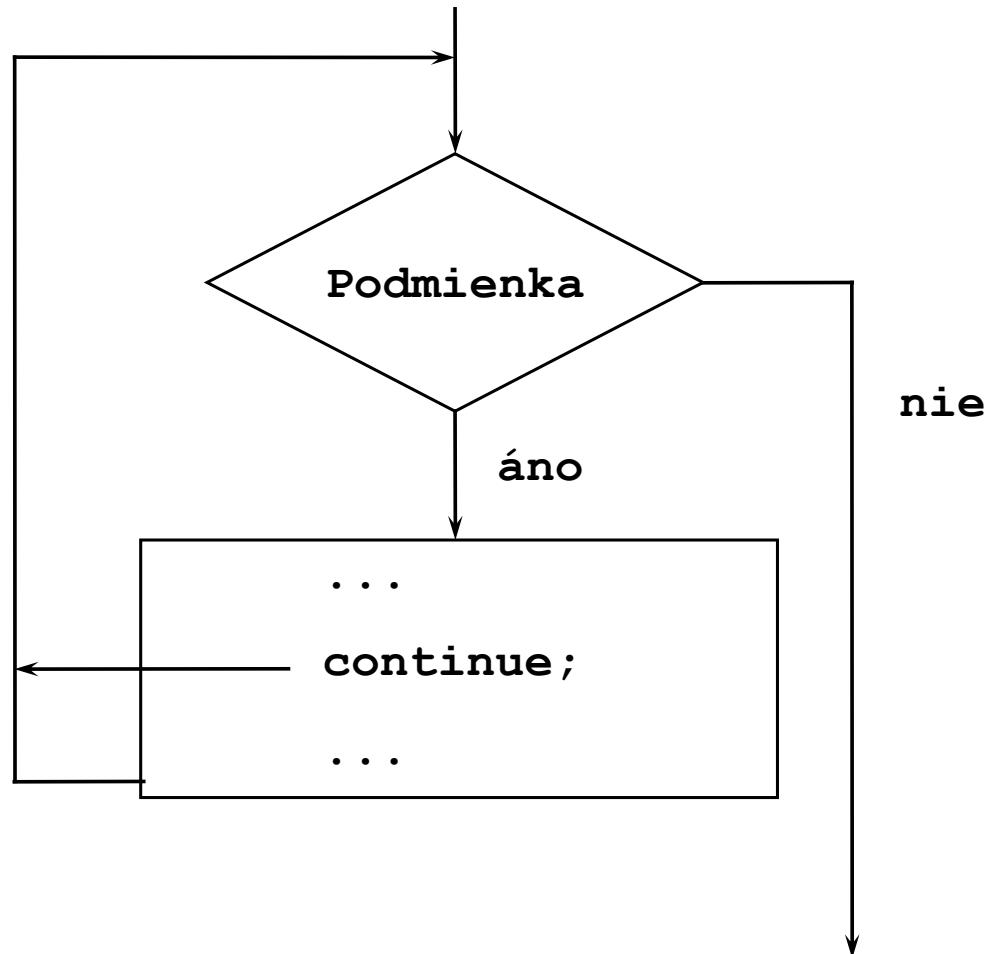
Predčasné ukončenie cyklu

- Aby bolo možné využiť všetky možnosti cyklov, je v Jave a v C k dispozícii príkaz `break;`
- príkaz `break;` zabezpečí okamžité ukončenie cyklu

Predčasné ukončenie JEDNEJ slučky cyklu príkazom break;

Predčasné ukončenie JEDNEJ slučky cyklu príkazom continue

```java
public class VynechajNegativne_for_06
{ public static void main (String[] args)
  { int n = args.length;  // Počet parametrov
  for (i = 0; i < n; i++)
    { double x = Double.parseDouble(args[i]);
      if ( x < 0.0 )
      { System.out.println(„Číslo "+ x + " je
            záporné: Preskoč");
        continue;}
      System.out.println( i + ": " + x );
      i = i + 1;
    }
  }
}
```

Ďalšie riadiace štruktúry

- **`break`** - a **`continue`** s návestím v Jave

- **`switch`** - v Jave (Výber z viacerých možností)

Príkaz **break** s návestím v Jave

- `a:`
- `while (pod1)`
- `{`
- `    while (pod2)`
- `    { ...`
- `        ... break;   // ukončí vnútorný cyklus`
- `        ... break a; // ukončí vnútorný aj vonkajší`
- `                // cyklus`
- `    }`
- `    ...  break;    // ukončí vonkajší cyklus`
- `}`

```
continue
analogicky!
```

Príkaz `switch`

- `// Vstup int cislo;`
- `String den = "";`

- `switch (cislo)`
- `{ case 1: den = "Pondelok"; break;`
- `    case 2: den = "Utorok";  break;`
- `    case 3: den = "Streda";  break;`
- `    case 4: den = "Stvrtok"; break;`
- `    case 5: den = "Piatok";  break;`
- `    case 6:`
- `    case 7: den = "Vikend";  break;`
- `    default: System.out.println("Nie je v tyzdni");`
- `}`

Dátové polia

7. Dátové štruktúry / Polia (Arrays)

V programovacích jazykoch:
  Postupnosť viacerých elementov (alebo objektov) rovnakého typu.

- Parametre príkazového riadku: **args**

  args[0], args[1], args[2], ...

- Lineárna algebra

  - Vektor **vektor**

    vektor[0], vektor[1], vektor[2]

  - Matica **matrix**

    matrix[i][j]

- Všeobecne: Indexované premenné:

$$x_i \qquad \texttt{x[i]}$$

```
double sum = 0.0;
for ( int i = 1; i <= n; i++)
{
  sum = sum + x[i];
}
```

$$\sum_{i=1}^{n} x_i$$

Príklady:

```
String[] args;   // Pole znakových reťazcov
String args[];   // to isté
double[] x, y;   // dve polia typu
    // double (Vektory)
double[][] matrix;  // Jedno dvojdimenzionálne
    // pole typu double (Matica)
int[][] k; // Dvojdimenzionálne pole typu
     // int
double[] vektor; // ďalší Vektor
```

Príklady:

```
String[] args;    // Pole znakových reťazcov
String args[];    // to isté
double[] x, y;    // dve polia typu
    // double (Vektory)
double[][] matrix;  // Jedno dvojdimenzionálne
    // pole typu double (Matica)
int[][] k; // Dvojdimenzionálne pole typu
     // int
double[] vektor; // ďalší Vektor
```

```
double[] x, y;

double[][] matrix;
```

Typ: **double**

Pole typu **double**

Name of the array

Meno poľa

Typ: **double**

Pole typu **double**

Pole typu Pole typu **double**

double[][] matrix
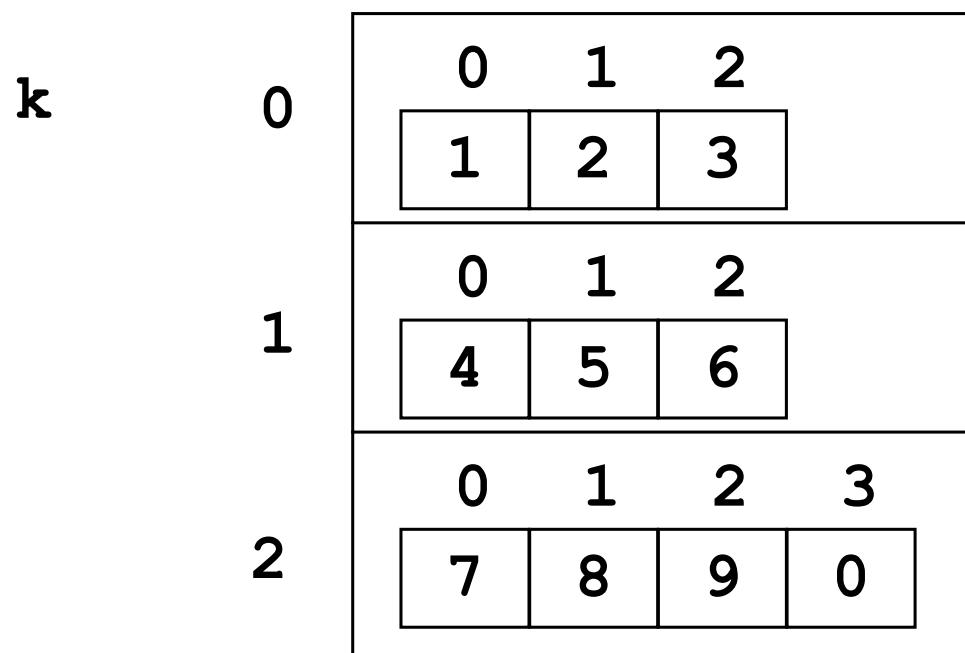
Príklady:

```
String[] mesiace = new String[]
  { "Jan","Feb","Mrc","Apr","Maj","Jun",
   "Jul","Aug","Sep","Okt","Nov","Dec" };
```

	0	1	2	3		11
mesiace	"Jan"	"Feb"	"Mrc"	"Apr"	...	"Dec"

```
int[][] k = new int[][]
   { { 1, 2, 3 }, {4, 5, 6}, {7, 8, 9, 0} };
```

- `double[] x = new double[100];`

- Pole so 100 `double` prvkami, pričom každý prvok má počiatočnú hodnotu `0.0`; prvky sú inicializované implicitne

  Pozor:

- Prvky majú index od 0 po 99!

- `double[][] matrix = new double[50][100];`

- Dvojdimenzionálne pole (Matica) s `double prvkami`; pole poyostáva s 50 –tich polí, každé po 100 prvkov typu `double`, pričom kaˇydý prvok má hodnotu `0.0`;

- Pozor:
`matrix[0][0], ..., matrix[49][99]`

- `double[][] matrix;`

- `...`

- `matrix = new double[50][100];`

- `...`

- `matrix = new double[][]`

- `   {{0.0, 0.7}, {0.25, 0.0}, {0.0} };`

- Premenným typu polí môže byť priradená hocikedy hodnota zodpovedajúceho typu, t.j. musí zodpovedať počet a typ dimenzií

Indexy v poliach začínajú 0!

- V Jave začína indexovanie prvkov poľa pevne a to indexom 0
- Prvky sú vždy oindexované zasebou idúcimi číslami

Implicitná inicializácia (Java)

- S pomocou **new** vytvoríme nové pole; ak hodnoty poľa nie sú pri inicializácii explcitne uvedené, ale je uvedený iba počet prvkov, nadobúdajú prvky nasledovné implicitné hodnoty
  - `0` (resp. `0.0`) pri všetkých číselných typoch
  - `false` pri type `boolean`
  - `'\u0000'` pri type `char`
  - `null` inak (bude neskôr vysvetlené)

7.3 Prístup k prvkom poľa

Prvok poľa sa správa presne ako premenná daného typu

▎ `matrix[0][1]` má typ `double`

▎ `matrix[0]` má typ `double[]`

▎ `vektor` má typ `double[]`

▎ `vektor[7]` má typ `double`

▌ Čítanie hodnoty vo výrazoch:
```
double erg = matrix[3][4] * vektor[2];
```

▌ Priradenie hodnoty prvku poľa:
```
matrix[3][4] = vektor[2] + erg;
matrix[2] = vektor;  //!!! Neskôr viac
```

Počítanie s indexom

- Index môže byť ľubovoľný výraz typu `int` (t.j. nielen konštanty ako doposiaľ)

Príklad:

```
double vysledok = 0.0;
for (int i = 0; i < vektor.length; i++)
{ vysledok = vysledok + vektor[i] * vektor[i]; }
System.out.println("l= " + Math.sqrt(vysledok));
```

```
// BubbleSort, Algoritmus na usporiadanie poľa.
public class BubbleSort
{
  public static void main (String[] args)
  {
    int[] a = new int[args.length];
    for ( int i = 0; i < args.length; i++ )
    { a[i] = Integer.parseInt(args[i]); }
```

```
boolean sorted;
do
{ sorted = true;
  for ( int i = 0; i < a.length - 1; i++ )
  { if ( a[i] > a[i+1] )
    { int h  = a[i];
      a[i]   = a[i+1];
      a[i+1] = h;
      sorted = false;
  } }
} while ( !sorted );
```

Pozor!

```
a[i] = a[i+1];

a[i+1] = a[i];
```

Nevymení prvky

```
a[i]  a a[i+1]
```
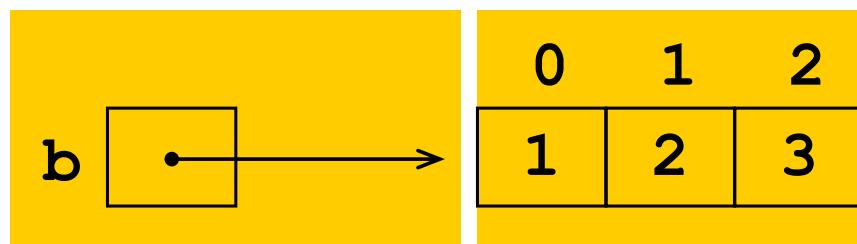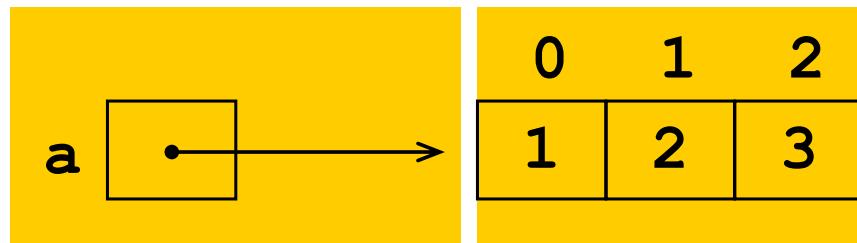
```
System.out.println(
   "Cisla vo od najmensieho po najvacsie:");
for ( int i = 0; i < a.length; i++ )
{ System.out.println(a[i]); }
}
}
```

7.4 Polia a pointery

- S **new** vytvoríme nové pole daného typu

- Pri priradení takto vytvoreného poľa na premennú sa vloží do premennej adresa vytvoreného poľa, teda premenná bude ukazovať na pole.

- Pri priradení poľa na do premennej teda pole nie je kopírované!
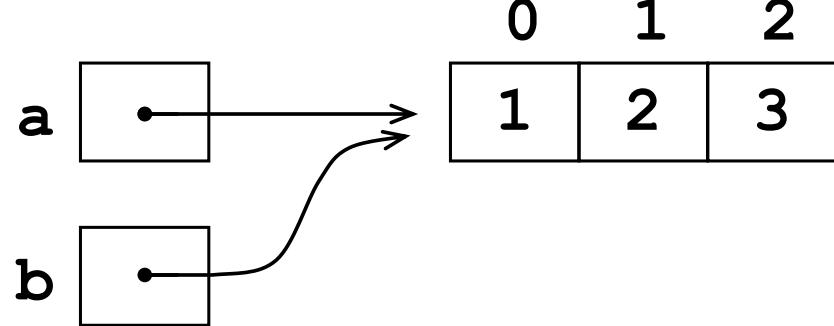
```
int[] a = new int[] {1, 2, 3};

int[] b = new int[] {1, 2, 3};
```

a → | 0 | 1 | 2 |
    | 1 | 2 | 3 |

b → | 0 | 1 | 2 |
    | 1 | 2 | 3 |

```
int[] a, b;

a = new int[] {1, 2, 3};

b = a;
```
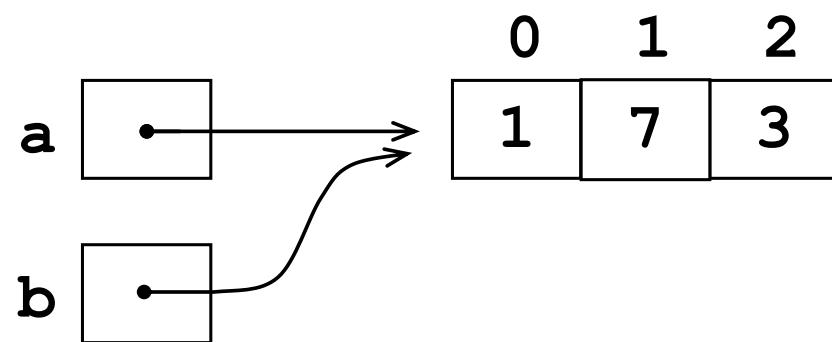
Pozor!

Pole nie je pri priradení kopírované!

```
a[1] = 7;

// Teraz platí b[1] = 7 !!!
```
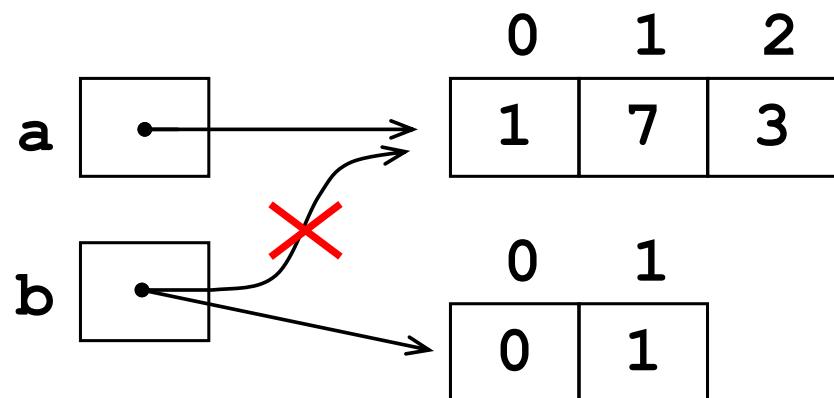
Pozor!

Priradenie hodnoty prvku a[1] zmenilo hodnotu prvku b[1], keďže obidve premenné a aj b ukazujú na to isté pole.

```
b = new int[] { 0, 1 };
```
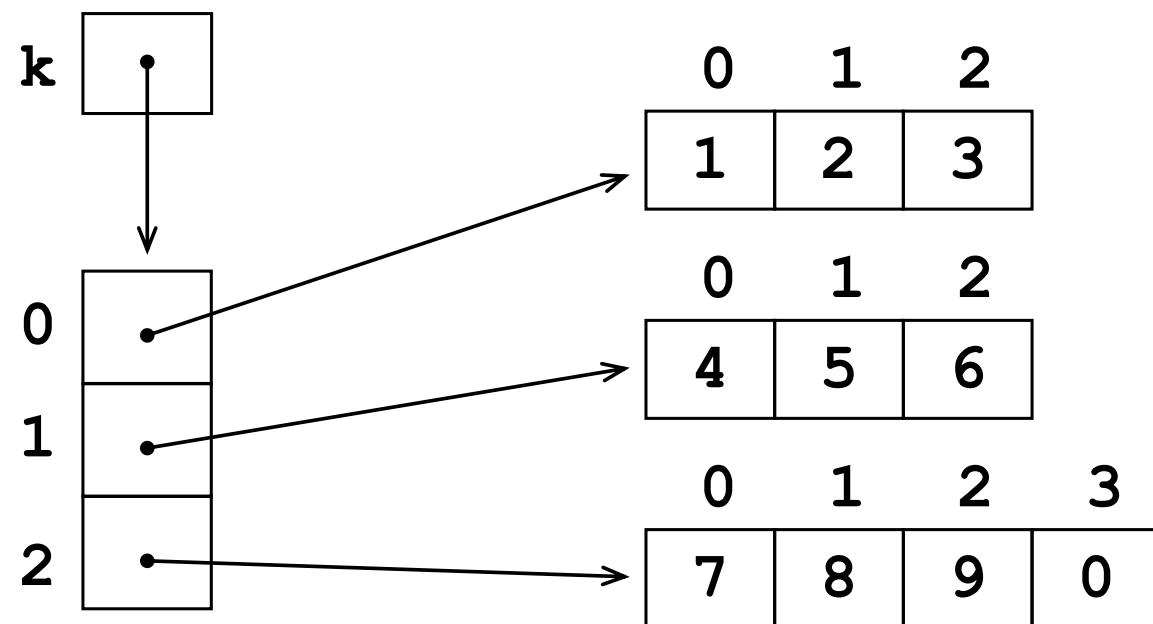
Pozor!

Pri priradení do premennej b sa samozrejme a nemení!

Premenná b bude totiž ukazovať na iné pole ako premenná a.
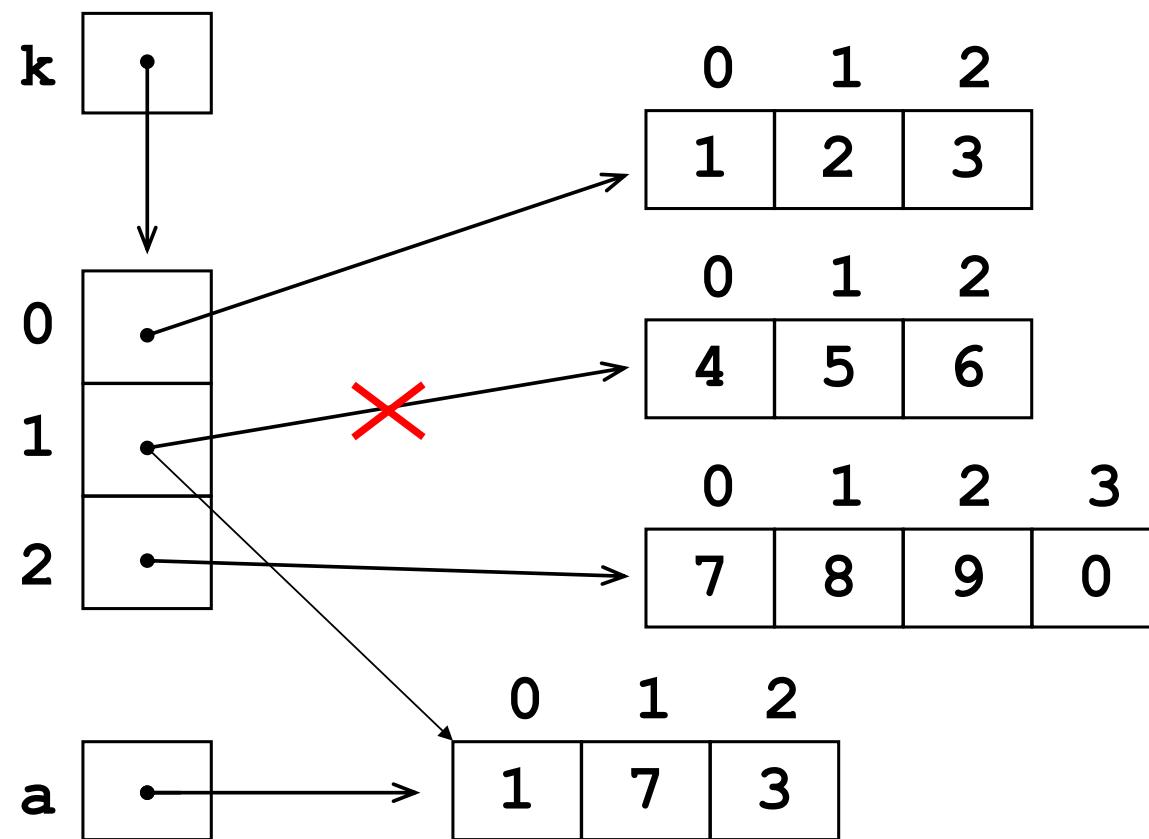
Príklad 3 Viacdimenzionálne polia

```
int[][] k = new int[][]
   { { 1, 2, 3 }, {4, 5, 6}, {7, 8, 9, 0} };
```

```
k[1] = a;
```

154

Funkcie a procedúry

8. Funkcie a procedúry

8.1 Motivácia

- Vo veľkých programoch sa často používa rovnaká časť programu viackrát
  (napr. výpočet odmocniny alebo usporiadanie poľa, ale aj výpis na obrazovku a pod.)
- **Funkcie** a **procedúry** umožňujú opakované použitie časti programu.

```
public class Binomialkoeficient
{
   static int fak(int n)
   { int vysledok = 1;
      for (int i = 1; i <= n; i++)
      { vysledok = vysledok * i; }

   return vysledok;

   }
```

Definícia funkcie

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

```
public static void main (String[] args){
    int n = Integer.parseInt(args[0]);
    int k = Integer.parseInt(args[1]);

    int vysledok = fak(n) / ( fak(k) * fak(n – k) );

    System.out.println(„Vysledok: " + vysledok);
  }
}
```

Volania funkcie

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Motivácia

- Vlastné funkcie môžeme používať rovnako ako preddefinované funkcie (napr. `Math.abs, Math.sqrt, Math.max`, ...).

- Použitím funkcií sa stáva program predľadnejším

Návratový typ
(ľubovoľný typ)

Meno funkcie

Zoznam parametrov:
typ a meno
(ľubovoľný počet)

Hlavička funkcie

```
fa...
{ int vysledok = 1;
    for (int i = 1; i <= n; i++)
    { vysledok = vysledok * i; }
    ...rn vysledok;
```

Lokálna premenná

Telo funkcie

Použitie parametrov

Ukončenie funkcie a vrátenie hodnoty

```
static int fak(int n)
```

- Definuje signatúru (Typy parametrov a typ návratovej hodnoty) funkcie. V našom príklade:
  ```
  fak: int -> int
  ```
- **Typ návratovej hodnoty** a **typy parametrov** môžu byť ľubovoľné typy (napr. aj polia a pointery)
- Funkcia môže mať ľubovoľný počet parametrov, napr. aj žiadny:
  ```
  f()
  ```

Telo funkcie

▎ Telo funkcie je blok príkazov, ktorý sa vykoná pri zavolaní funkcie.

▎ V tele funkcie je možné použiť parametre funkcie. Parametre sa používajú rovnako ako premenné.

▎ Pomocou príkazu `return` sa ukončí funkcia a hodnota výrazu za kľúčovým slovom `return` je vrátená naspäť ako výsledná hodnota funkcie.

- Výraz po kľúčovom slove **return** musí mať rovnaký typ ako návratový typ funkcie daný v hlavičke.

- Príkaz **return** nemusí byť na konci tela funkcie.

- V tele funkcie môže byť príkaz **return** použitý viackrát.

- Akonáhle sa vykoná príkaz **return** je funkcia ukončená.

8.3 Volanie funkcie (Použitie funkcie)

▌ Funkcia môže byť použitá vo výrazoch. Pri volaní musí byť každému parametru funkcie (**formálne parametre**) priradený výraz zodpovedajúceho typu. Jeho hodnota sa stáva **aktuálnym parametrom**.
Príklad: **fak(n-k)** volanie funkcie fak, kde formálny parameter n funkcie fak nadobudne hodnotu výrazu n-k (kde n a k sú premenné funkcie main)

▌ Zavolaná funkcia má vo výraze svoj návratový typ a hodnotu danú výrazom za príkazom return.
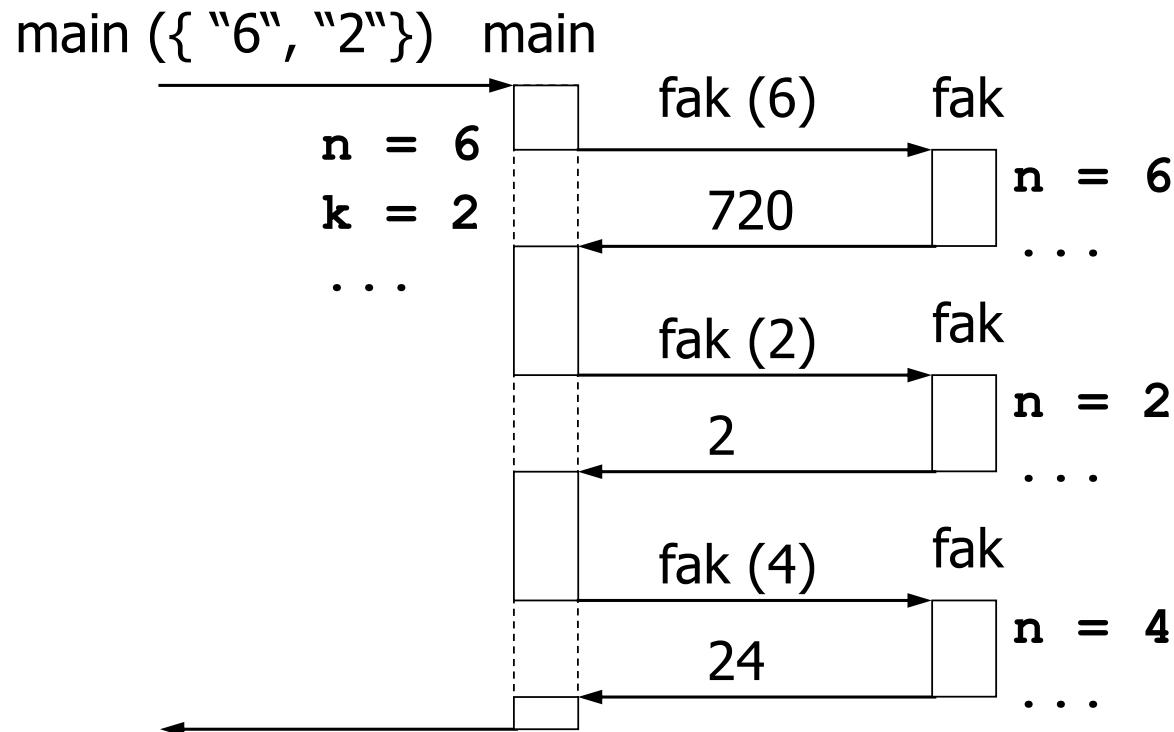
```
static int fak(int n)
 { int vysledok = 1;
    for (int i = 1; i <= n; i++)
    { vysledok = vysledok * i; }
    return vysledok; }

 public static void main (String[] args)
 { int n = Integer.parseInt(args[0]);
    int k = Integer.parseInt(args[1]);
    int vysledok = fak(n) / ( fak(k) * fak(n - k) );
    System.out.println(„Vysledok: " + vysledok); }
```

>Binomialkoeficient 6 2

8.4 Procedúry

- Procedúry sú funkcie, ktorých návratový typ je prázdny t.j. **void** (z angl. pre prázdny).

- Keďže procedúra nedáva žiadny výsledok, nemôže byť použitá vo výraze.

- Napriek tomu sú procedúry užitočné:
  ```
  System.out.println("Bla");
  ```

- Premenné typu pole sú pointery (ich hodnoty sú adresy polí)

- Funkcie

  - Hlavička funkcie

  - Telo funkcie

  - Volanie funkcie

  - Formálne vs. aktuálne parametre

- Procedúry (Návratový typ: `void`)

Bubble Sort (druhý pokus)
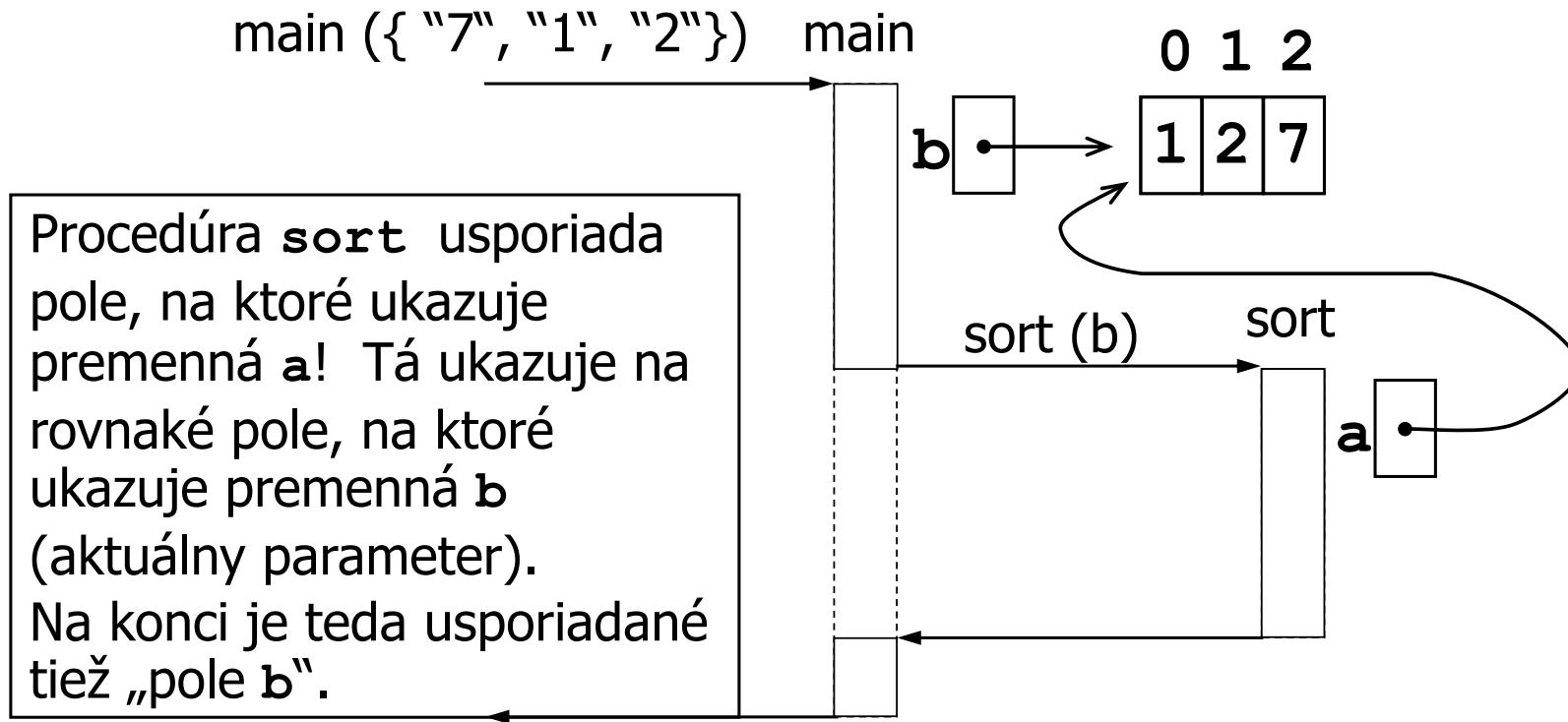
```java
public class BubbleSort
{ public static void sort (int[] a)
   { boolean sorted;
     do
     { sorted = true;
       for ( int i = 0; i < a.length - 1; i++ )
       { if ( a[i] > a[i+1] )
          { int h  = a[i];
            a[i]    = a[i+1];
            a[i+1] = h;
            sorted = false;
     } } }
     while ( !sorted );     }
```

Bubble Sort (druhý pokus)

- ```
  public static void main (String[] args)
  ```
- ```
  { int[] b = new int[args.length];
  ```
- ```
  for ( int i = 0; i < args.length; i++ )
  ```
- ```
  { b[i] = Integer.parseInt(args[i]); }
  ```

- ```
  sort(b);
  ```

- ```
  System.out.println(„Zoradené pole:");
  ```
- ```
  for ( int i = 0; i < b.length; i++ )
  ```
- ```
  { System.out.println(b[i]); }
  ```
- ```
  }
  ```
- ```
  }
  ```

- `>BubbleSort 7 1 2`

main ({ ”7”, ”1”, ”2”})   main

0 1 2

b → | 1 | 2 | 7 |

Procedúra **sort** usporiada pole, na ktoré ukazuje premenná **a**! Tá ukazuje na rovnaké pole, na ktoré ukazuje premenná **b** (aktuálny parameter).
Na konci je teda usporiadané tiež „pole **b**".

sort (b)   sort

a •

Polia ako parametre funkcií

▮ Parameter typu pole (prvkov nejakého typu) je vlastne adresa poľa.

▮ Do parametra typu pole je teda uložená adresa poľa; t.j. pole nie je skopírované (funkcia pracuje s originálnym poľom).

▮ Ak teda procedúra (alebo všeobecne funkcia) modifikuje prvky poľa, je modifikované pôvodné pole

Pozorovanie:

```
public static void sort (int[] a)
{ boolean sorted;
  do
  { sorted = true;
    for ( int i = 0; i < a.length - 1; i++ )
    { if ( a[i] > a[i+1] )
        { int h  = a[i];
          a[i]   = a[i+1];
          a[i+1] = h;
          sorted = false;
  } } }
  while ( !sorted );
}
```

Tu nemusí byť
return;

▮ Na konci tela procedúry nemusí byť príkaz `return`;
procedúra je na konci automaticky ukončená.
(V prípade funkcií s neprázdnym návratovým typom toto nie je
možné, keďže takéto funkcie musia vrátiť nejakú hodnotu)

▮ V procedúrach tiež môžeme použiť príkaz `return` . V
takom prípade za ním nenasleduje žiaden výraz.

8.5   Volanie hodnotou a volanie adresou

▮ Pri volaní funkcie je vždy kopírovaná do formálneho parametra hodnota. Ak je formálny dátový typ primitívny, tak takéto volanie nazývame volanie hodnotou (call by value);

▮ Pri neprimitívnych dátových typoch, napr. pri premenných typu pole, čo sú vlastne premenné, ktorých hodnotou je adresa objektu (napr. poľa) hovoríme o volaní adresou (call by name)

▮ Ak pri volaní hodnotou zmeníme formálny parameter, hodnota premennej použitej ako aktuálny parameter sa nezmení. Ak pri volaní adresou zmeníme hodnotu objektu, na ktorý ukazuje formálny parameter, zmeníme zároveň hodnotu objektu, na ktorý ukazuje aktuálny parameter. Je to logické, keďže sme odovzdali adresu a teda formálny aj aktuálny parameter ukazujú na to isté.

Volanie adresou a postranné efekty

- Pomocou volania adresou môže funkcia alebo procedúra „vrátiť" nové hodnoty (t.j. zmeniť hodnoty) (pozri. Bubble Sort)!

- Pomocou volania hodnotou to nie je možné.

Polymorphismus

- Je tiež možné definovať funkcie s rovnakým menom a s rozdielnym počtom a typom parametrov.
- Ktorá funkcia má byť zavolaná je dané typom a počtom aktuálnych parametrov.

Globálne premenné

8.6 Globálne premenné

- Doposiaľ: Deklarácia premenných iba vo funkciách
  - **lokálne premenné** resp.
  - **parametre funkcií**

- Teraz: Deklarácia premenných mimo funkcie:
  globálne premenné

```
public class Binomialkoeficient
{ public static int pocitadlo = 0;

  public static int fak(int n)
  { int vysledok = 1;
    for (int i = 1; i <= n; i++)
    { vysledok = vysledok * i; }

    pocitadlo = pocitadlo + 1;
    return vysledok;
  }
```

Príklad: globálne premenné (part 2)

- `public static void main (String[] args)`
- `{ int n = Integer.parseInt(args[0]);`
- `    int k = Integer.parseInt(args[1]);`

- `    int vysledok = fak(n) / ( fak(k) * fak(n - k) );`

- `    System.out.println("Vysledok: " + vysledok);`

- `    System.out.print("Funkcia fak bola ");`
- `    System.out.println(pociatadlo + " krat zavolana");`
- `    }`
- `}`

- Globálna premenná môže byť použitá v každej funkcii programu.

- Jej zmena v jednej funkcii je viditeľná v ostatných funkciách.

- Pomocou globálnych premenných si môžu funkcie jednoducho odovzdávať hodnoty.

- Pri deklarácii je globálnej premennej priradená automaticky preddefinovaná hodnota.

„Protipríklad":

```java
public class BubbleSort
{ public static int[] a;

  public static void sort()    // Ziadny parameter
  { boolean sorted;            // Usporiada VZDY
    do { sorted = true;        // pole a!
        for ( int i = 0; i < a.length - 1; i++ )
        { if ( a[i] > a[i+1] )
          { int h  = a[i];
            a[i]   = a[i+1];
            a[i+1] = h;
            sorted = false; } }
      } while ( !sorted ); }
```

Globálne premenné:
Odporúčania

- Vrátenie hodnoty pomocou globálnej premennej sa neodporúča.

- Zmysluplné použitie globálnej premennej je napr.:

  - Počítadlo

  - Debug-Modus (ďalšia Fólia)
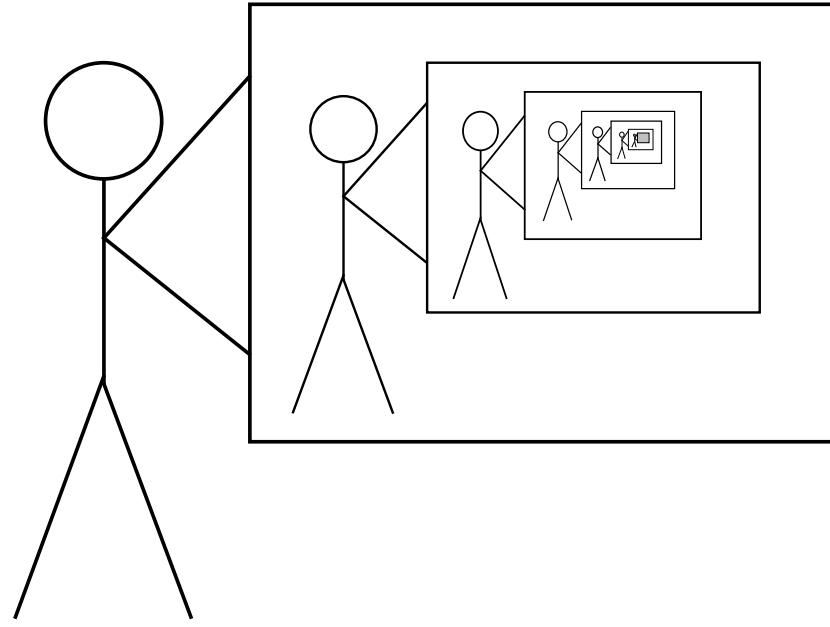
```java
public class Binomialkoeficient
{ public static boolean debug = true; false;

  public static int fak(int n)
  { int vysledok = 1;
    if (debug)
      System.out.println("Pred: n = " + n);
    for (int i = 1; i <= n; i++)
    { if (debug)
        System.out.println("Vnutri: i = " + i);
      vysledok = vysledok * i; }
```

Platnosť

- Vo funkcii môže byť deklarovaná lokálna premenná s rovnakým názvom ako globálna premenná.
V takomto prípade globálna premenná
zatienená je lokálnou premennou.

Rekurzia v Jave

Motivácia

- Môže byť v tele funkcie použité volanie tej istej funkcie?
- Áno, je to možné!
- Áno, je to dokonca zmysluplné!

FEI STU
189
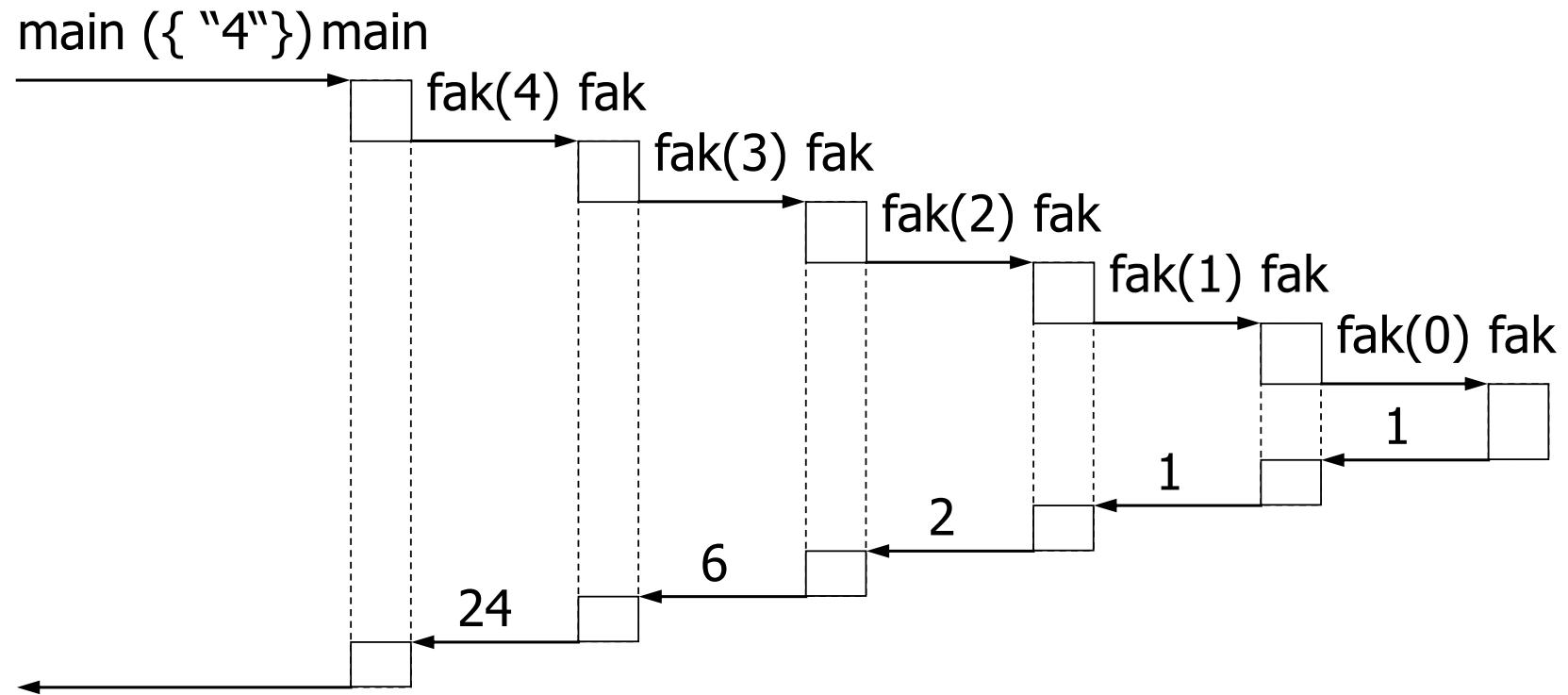
Príklad: Faktoriál

```java
public class Faktorial
{
  public static int fak(int n)
  { if (n == 0)
    { return 1; }
    else
    { return n * fak(n-1); }
  }

  public static void main (String[] args)
  { int n = Integer.parseInt(args[0]);
    System.out.println(„Faktorial je: " + fak(n));
  }
}
```

- `>Faktorial 4`

Princíp rekurzie

- Previesť problém na riešenie rovnakého problému s jednoduchšími dátami:

`fak(n)` -> **`fak(n-1)`** .

- Kombinovať riešenie problémov pre jednoduchšie dáta na získanie riešenia problému s pôvodnými dátami:

`fak(n) = n * fak(n-1)` .

Vykonanie rekurzívnej funkcie

- Pri volaní funkcie sa ukladajú hodnoty lokálnych premenných a miesto, z ktorého bola funkcia zavolaná.

- Po ukončení volanej funkcie pokračuje program vo vykonávaní volajúcej funkcie s uloženými hodnotami príkazom nasledujúcim za volajúcim miestom ďalej.

FEI STU

193

Stack

- Všetky hodnoty sú uložené s Stacku...

fak(0): n = 0
fak(1): n = 1, h = .
fak(2): n = 2, h = .
fak(3): n = 3, h = .
fak(4): n = 4, h = .

Stack
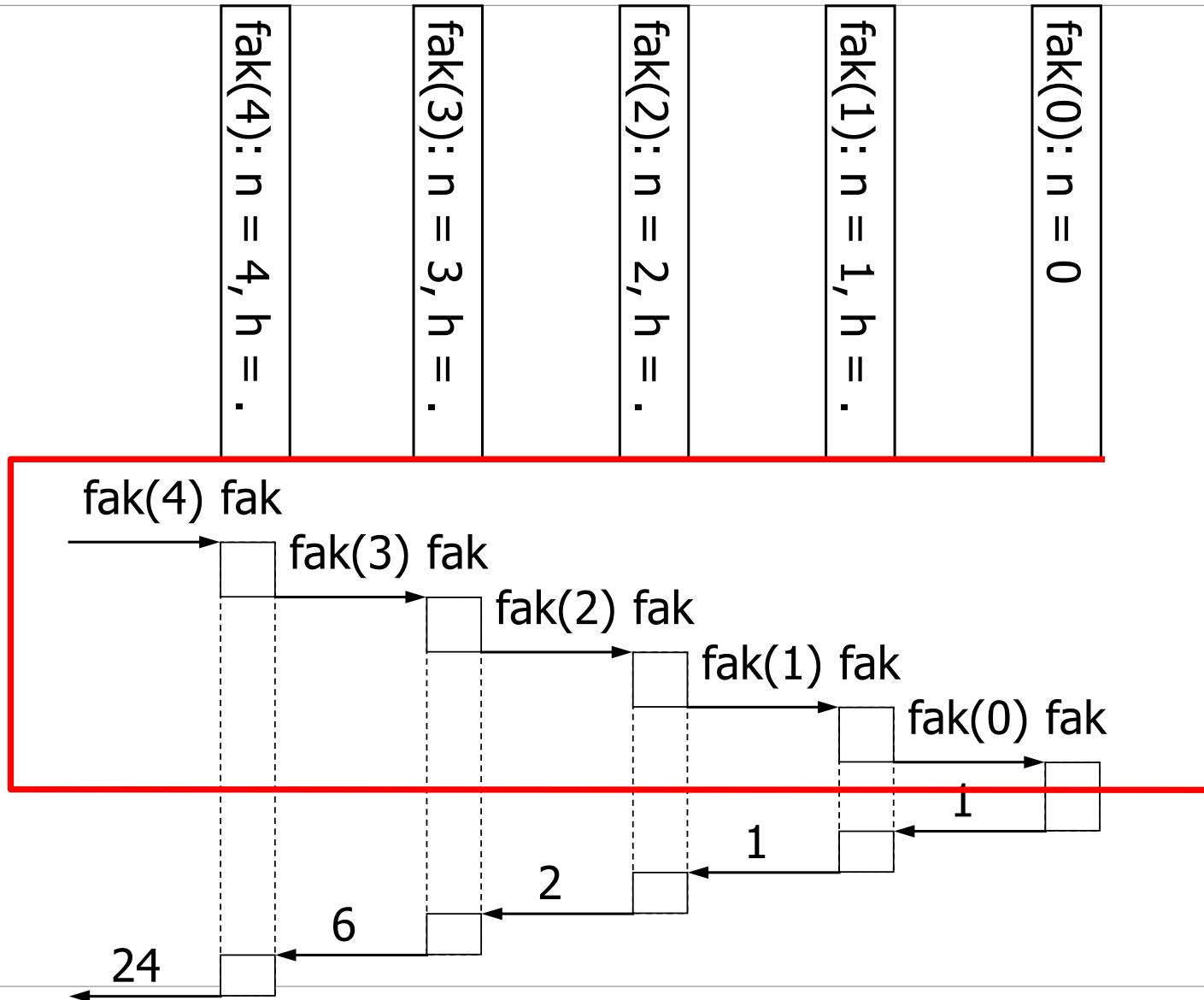
▪ A po ukončení zavolanej funkcie je starý stav obnovený...

```
return 1;

return n*h;

return n*h;

return n*h;

return n*h;
```

fak(4): n = 4, h = 6

24

Stack a volania

fak(4): n = 4, h = .
fak(3): n = 3, h = .
fak(2): n = 2, h = .
fak(1): n = 1, h = .
fak(0): n = 0

fak(4) fak
fak(3) fak
fak(2) fak
fak(1) fak
fak(0) fak

1

1

2

6

24

Funkcia sa nazýva rekurzívna, ak sama priamo alebo nepriamo (prostredníctvom volania iných funckií) je volaná vo svojej definícii.

▎Často je možné rekurzívnymi funkciami fomulovať elegantné riešenie problému!

▎Zvyčajne je rekurzívna funkcia menej efektívna ako zodpovedajúci `while`-cyklus

Nepriame volanie:

```
public static boolean even(int n)
{ // predpoklad: n >= 0
  if (n == 0)       return true;
  else if (n == 1) return false;
  else              return odd(n-1);
}


public static boolean odd(int n)
{ // predpoklad: n >= 0
  if (n == 0)       return false;
  else if (n == 1) return true;
  else              return even(n-1);
}
```

Terminovanie

- Musíme dávať pozor, aby rekurzívne volanie niekedy problém vyriešilo.
- V definícii rekurzívnej funkcie musí byť preto aspoň jedna alternatíva, ktorá rieši problém priamo, teda bez rekurzívneho volania.

Indukcia (Princíp):

1. **Počiatočný krok (PK)**:
   Dokáž správnosť pre n = 0

2. **Indukčný predpoklad (IP)**:
   Predpokladajme správnosť pre n – 1 >= 0

3. **Indukčný krok (IK)**:
   Pod podmienkou ak platí IP dokážeme, že výrok platí pre n

Tým je dokázaná správnosť pre všetky n >= 0.

Príklad: `fak(n) = n!`

1. **Počiatočný krok:**
    Zjavne platí `fak(0)` vracia 1 = `0!`

2. **Indukčný predpoklad:**
    Predpokladajme, že `fak(n-1) pre` n-1 >= 0 vracia hodnotu `(n-1)!`

3. **Indukčný krok:**
    Volanie `fak(n)` pre n > 0 zavolá
    `fak(n-1).` Podľa (IP) platí, že volanie `fak(n-1)` vráti naspäť hodnotu `(n-1)!`,
    teda hodnotu `h = (n-1)!` Volanie `fak(n)` vracia `return n * h`, t.j. hodnotu
    `n * (n-1)!` ; a teda hodnotu `n!`

FEI STU
201

▌ Procedurálne programovanie

    ▌ Výrazy: Syntaktický strom, Vyhodnotenie

    ▌ Dátové typy: Premenné, Polymorphismus

    ▌ Priradenie

    ▌ Riadiace štruktúry: Bloky, Alternatívy, Cykly

    ▌ Dátové štruktúry: Polia

    ▌ Adresové premenné (Pointery, Smerníky, Ukazovatele)

    ▌ Funkcie a procedúry (Rekurzia)

Part II – Basics of Object oriented programming syntax

Primitives vs. objects; value and reference semantics

A `swap` method?

- Does the following `swap` method work? Why or why not?

```java
public static void main(String[] args) {
    int a = 7;
    int b = 35;

    // swap a with b?
    swap(a, b);

    System.out.println(a + " " + b);
}

public static void swap(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
}
```

Value semantics
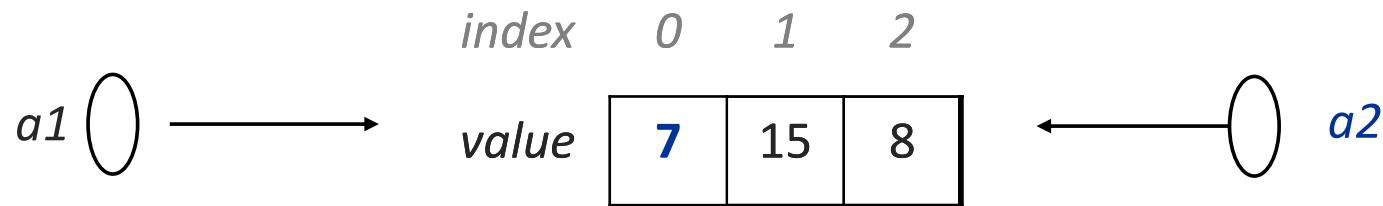
- **value semantics**: Behavior where values are copied when assigned, passed as parameters, or returned.

  - All primitive types in Java use value semantics.
  - When one variable is assigned to another, its value is copied.
  - Modifying the value of one variable does not affect others.

```
int x = 5;
int y = x;      // x = 5, y = 5
y = 17;         // x = 5, y = 17
x = 8;          // x = 8, y = 17
```

Reference semantics (objects)

- **reference semantics**: Behavior where variables actually store the address of an object in memory.

  - When one variable is assigned to another, the object is *not* copied; both variables refer to the *same object*.
  - Modifying the value of one variable *will* affect others.
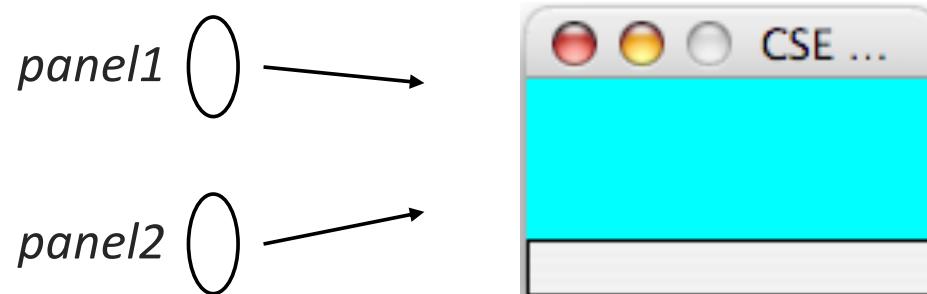
```
int[] a1 = {4, 15, 8};
int[] a2 = a1;          // refer to same array as a1
a2[0] = 7;
System.out.println(Arrays.toString(a1)); // [7, 15, 8]
```

References and objects

- Arrays and objects use reference semantics. Why?
  - *efficiency.* Copying large objects slows down a program.
  - *sharing.* It's useful to share an object's data among methods.

```
DrawingPanel panel1 = new DrawingPanel(80, 50);
DrawingPanel panel2 = panel1;   // same window
panel2.setBackground(Color.CYAN);
```
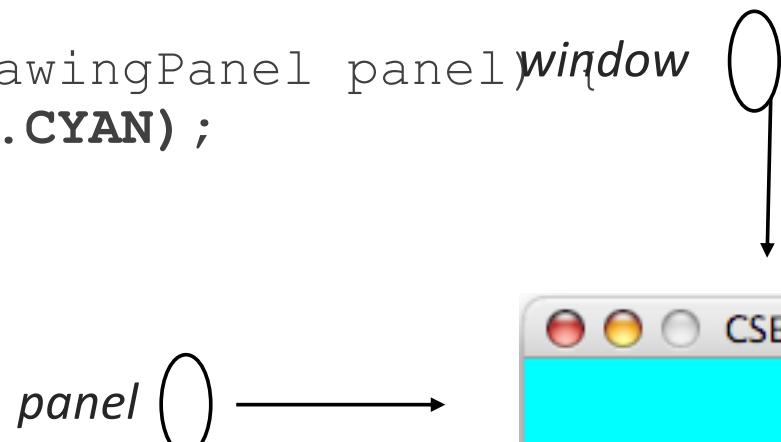
Objects as parameters

- When an object is passed as a parameter, the object is *not* copied. The parameter refers to the same object.
  - If the parameter is modified, it *will* affect the original object.

```
public static void main(String[] args) {
    DrawingPanel window = new DrawingPanel(80, 50);
    window.setBackground(Color.YELLOW);
    example(window);
}

public static void example(DrawingPanel panel) {
    panel.setBackground(Color.CYAN);
    ...
}
```

window

panel

Arrays as parameters
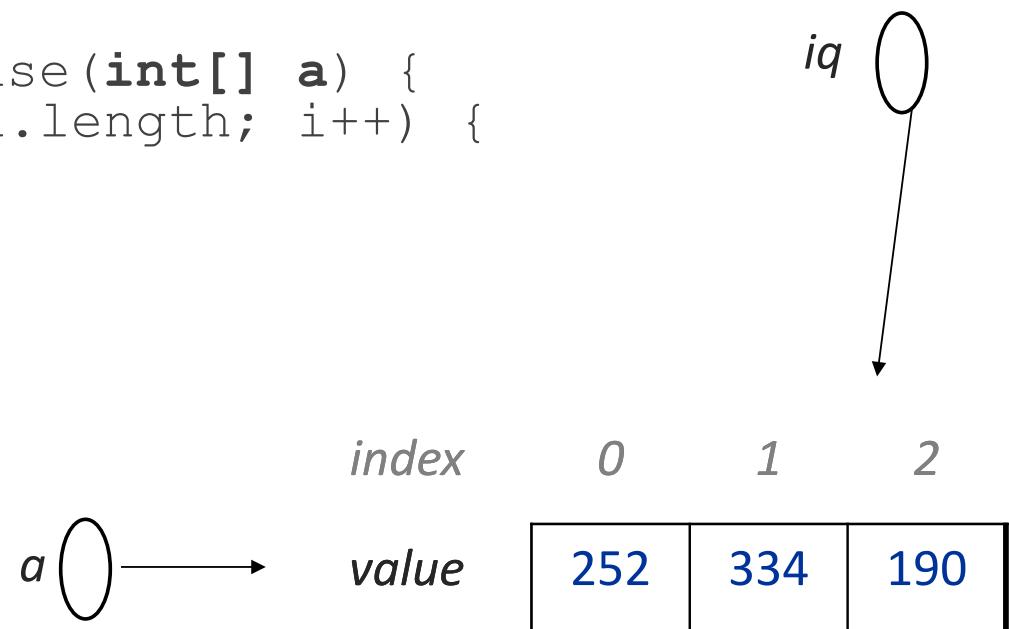
- Arrays are also passed as parameters by reference.
  - Changes made in the method are also seen by the caller.

```java
public static void main(String[] args) {
    int[] iq = {126, 167, 95};
    increase(iq);
    System.out.println(Arrays.toString(iq));
}
public static void increase(int[] a) {
    for (int i = 0; i < a.length; i++) {
        a[i] = a[i] * 2;
    }
}
```

- Output:

  [252, 334, 190]

iq

index	0	1	2
a → value	252	334	190
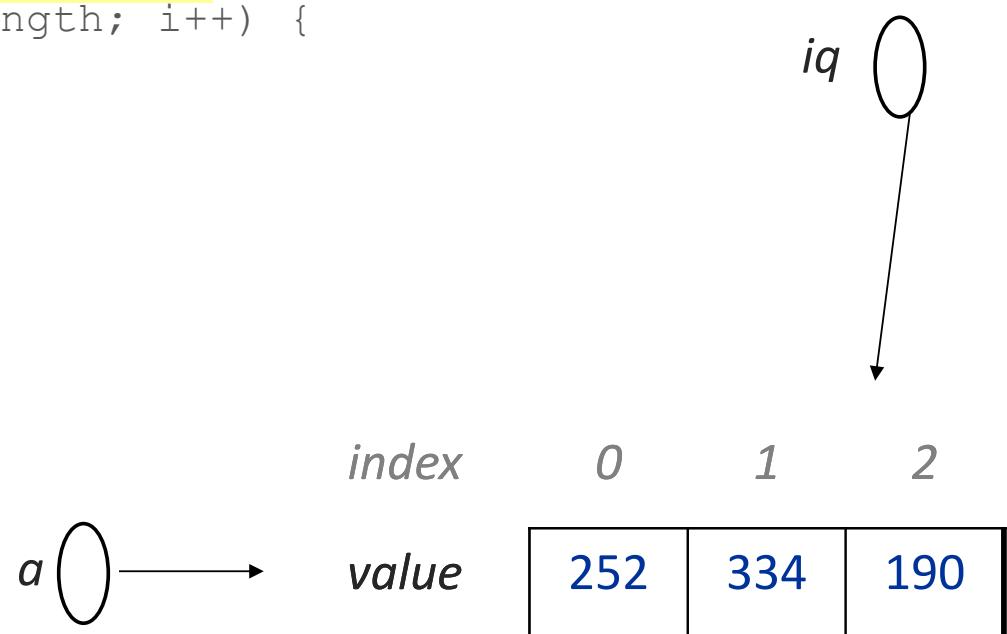
Arrays pass by reference

- Arrays are also passed as parameters by reference.
  - Changes made in the method are also seen by the caller.

```java
public static void main(String[] args) {
    int[] iq = {126, 167, 95};
    increase(iq);
    System.out.println(Arrays.toString(iq));
}

public static void increase(int[] a) {
    for (int i = 0; i < a.length; i++) {
        a[i] = a[i] * 2;
    }
}
```

- Output:

  `[252, 334, 190]`

iq

index	0	1	2
value	252	334	190

a

Classes and Objects

Objects

- **object:** An entity that encapsulates data and behavior.
  - *data*:   variables inside the object
  - *behavior*:   methods inside the object
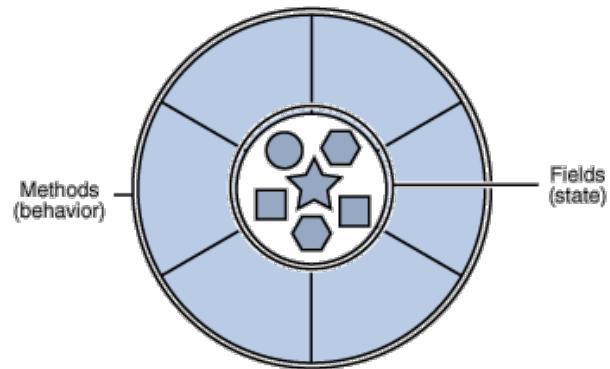
    - You interact with the methods;
      the data is hidden in the object.

- Constructing (creating) an object:

  Type objectName `= new` **Type(parameters)**`;`

- Calling an object's method:
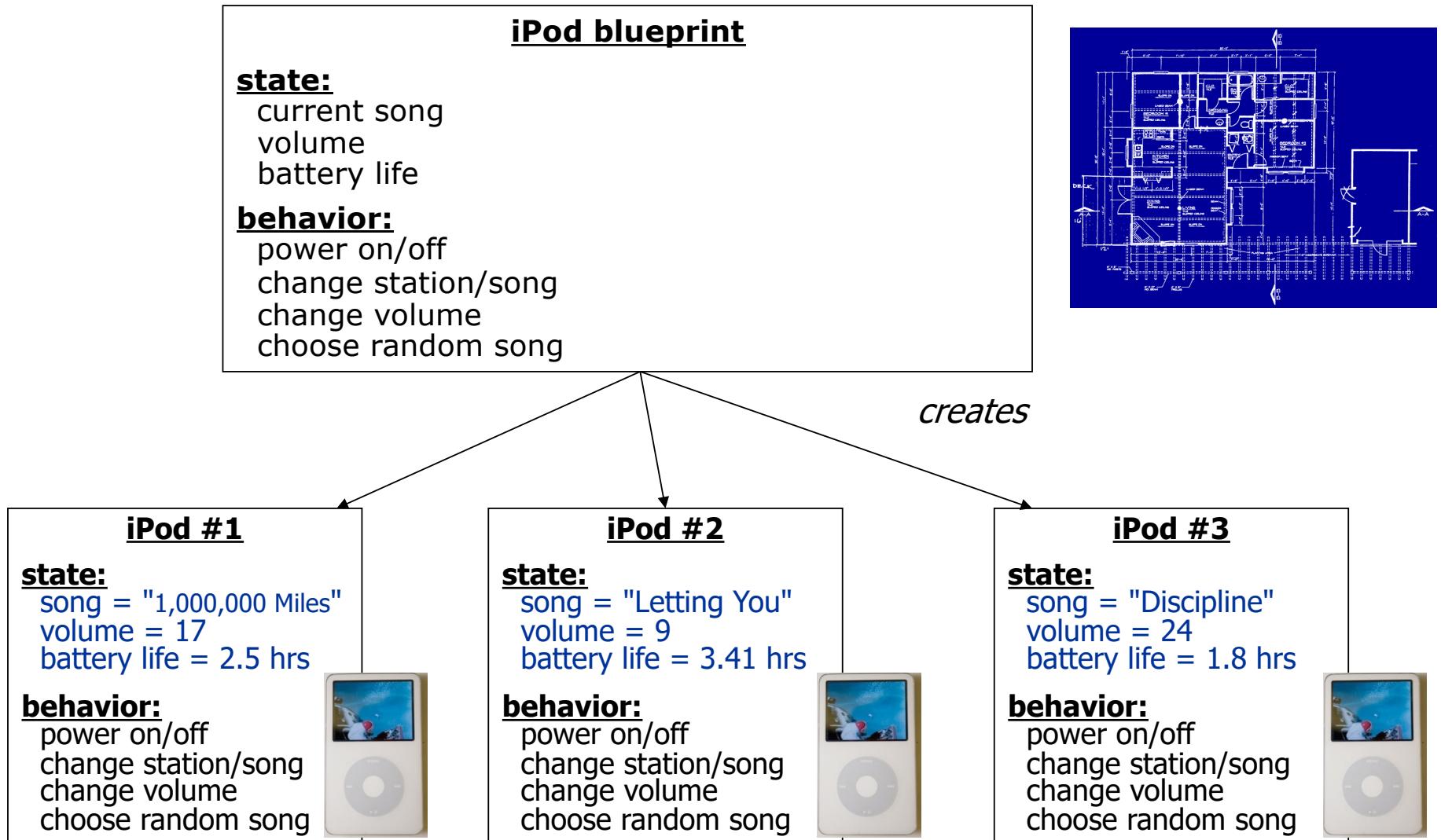
  objectName`.`**methodName(parameters)**`;`

Methods (behavior) — Fields (state)

Classes

- **class**: A program entity that represents either:
  1.   A program / module, or
  2.   A template for a new type of objects.

- **object-oriented programming (OOP)**: Programs that perform their behavior as interactions between objects.

  - **abstraction**: Separation between concepts and details.
    Objects and classes provide abstraction in programming.

Blueprint analogy

iPod blueprint

state:
  current song
  volume
  battery life

behavior:
  power on/off
  change station/song
  change volume
  choose random song

creates

iPod #1

state:
  song = "1,000,000 Miles"
  volume = 17
  battery life = 2.5 hrs

behavior:
  power on/off
  change station/song
  change volume
  choose random song

iPod #2

state:
  song = "Letting You"
  volume = 9
  battery life = 3.41 hrs

behavior:
  power on/off
  change station/song
  change volume
  choose random song

iPod #3

state:
  song = "Discipline"
  volume = 24
  battery life = 1.8 hrs

behavior:
  power on/off
  change station/song
  change volume
  choose random song

Point objects

```java
import java.awt.*;
...
Point p1 = new Point(5, -2);
Point p2 = new Point();       // origin (0, 0)
```
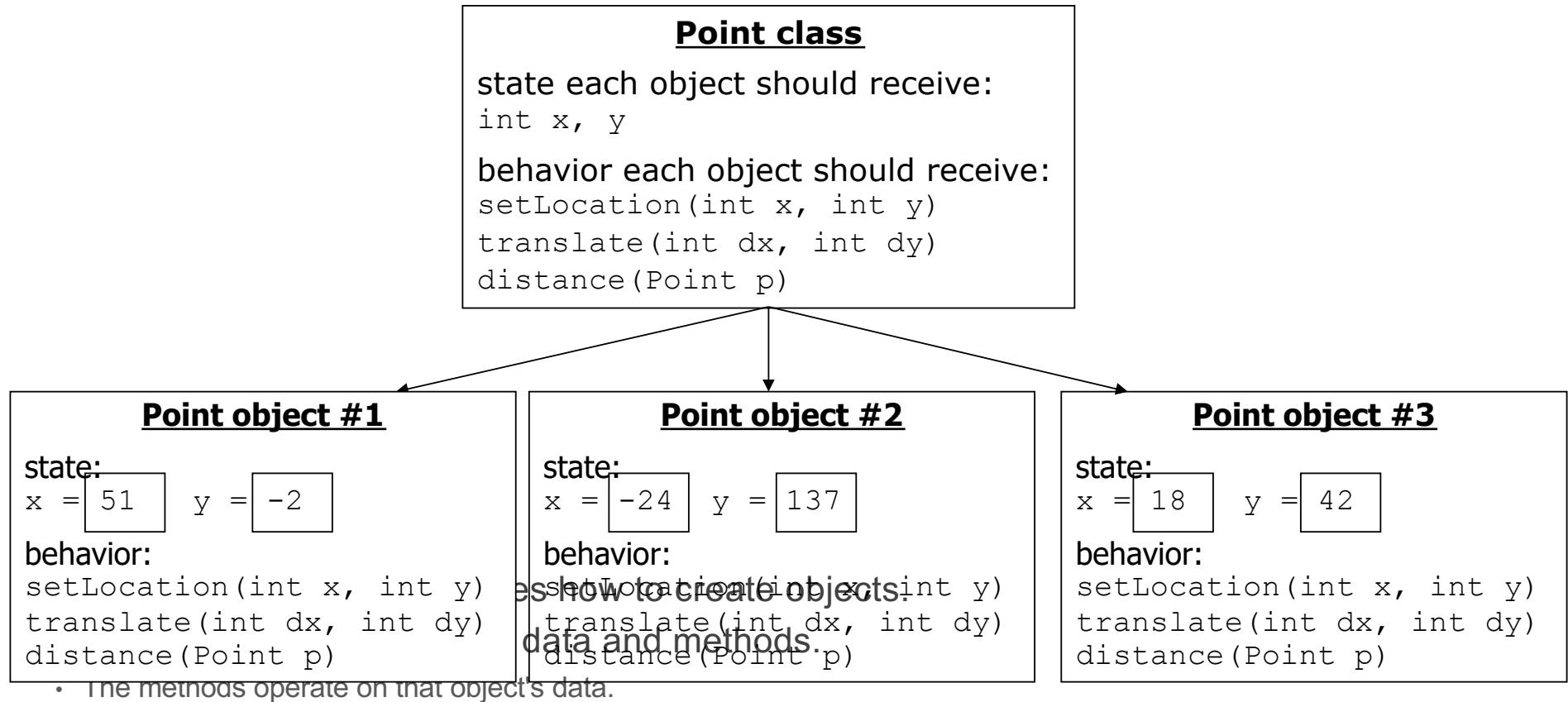
- Data:

- Methods:

Name	Description
x	the point's x-coordinate
y	the point's y-coordinate
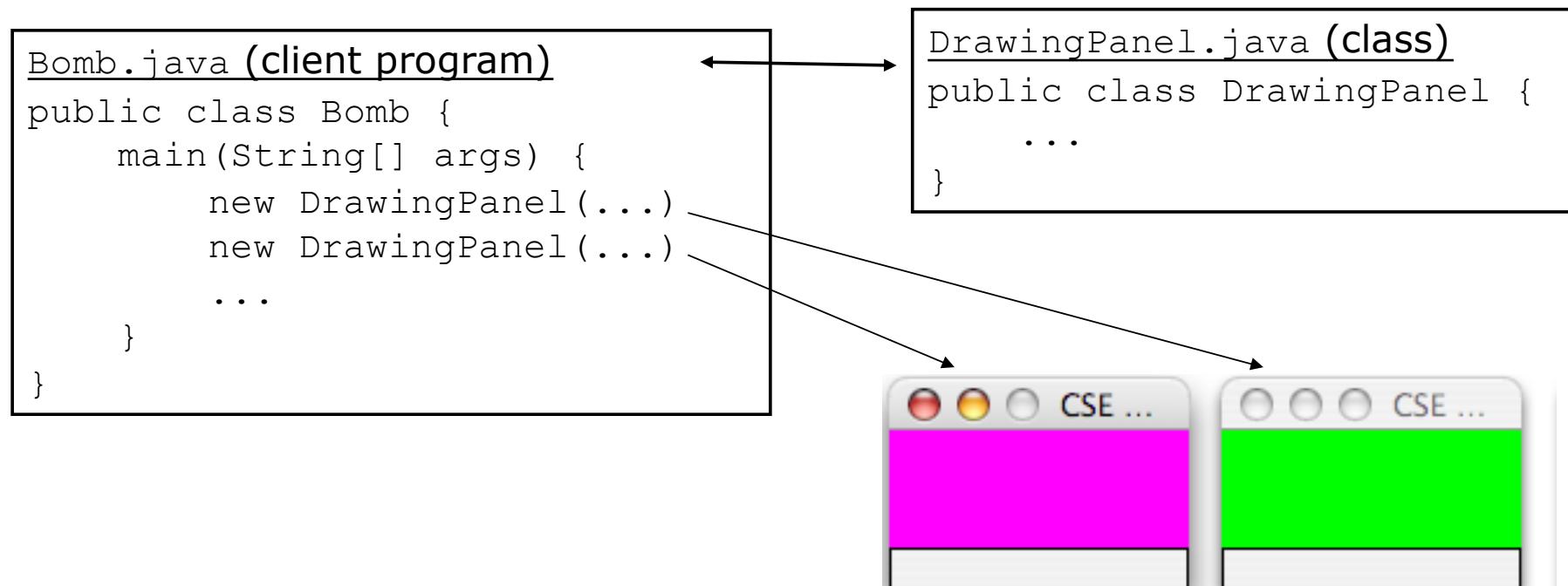
Name	Description
setLocation(**x, y**)	sets the point's x and y to the given values
translate(**dx, dy**)	adjusts the point's x and y by the given amounts
distance(**p**)	how far away the point is from point *p*

`Point` class as blueprint

Point class

state each object should receive:
`int x, y`
 	
behavior each object should receive:
`setLocation(int x, int y)`
`translate(int dx, int dy)`
`distance(Point p)`

Point object #1	Point object #2	Point object #3

state:
x = 51   y = -2
 	
behavior:
`setLocation(int x, int y)`
`translate(int dx, int dy)`
`distance(Point p)`
	state:
	x = -24 y = 137
	
	behavior:
	`setLocation(int x, int y)`
	`translate(int dx, int dy)`
	`distance(Point p)`
		state:
		x = 18 y = 42
		
		behavior:
		`setLocation(int x, int y)`
		`translate(int dx, int dy)`
		`distance(Point p)`

...es how to create objects:
data and methods.

- The methods operate on that object's data.

Clients of objects

- **client program**: A program that uses objects.
  - Example: `Bomb` is a client of `DrawingPanel` and `Graphics`.

Fields

- **field**: A variable inside an object that is part of its state.
  - Each object has *its own copy* of each field.

- Declaration syntax:
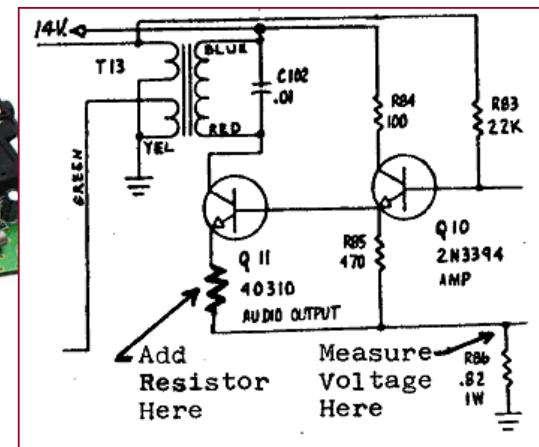
  ```
  private type name;
  ```

  - Example:

  ```
  public class Point {
      private int x;
      private int y;

      ...
  }
  ```
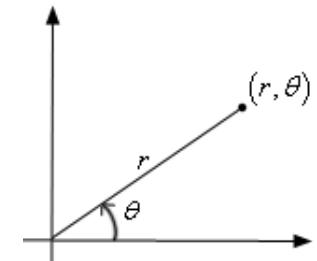
- **encapsulation**: Hiding implementation details from clients.

  - Encapsulation enforces *abstraction*.
    - separates external view (behavior) from internal view (state)
    - protects the integrity of an object's data

Benefits of encapsulation

- Abstraction between object and clients

- Protects object from unwanted access
  - Example: Can't fraudulently increase an `Account`'s balance.

- Can change the class implementation later
  - Example: `Point` could be rewritten in polar coordinates (r, θ) with the same methods.

- Can constrain objects' state (**invariants**)
  - Example: Only allow `Account`s with non-negative balance.
  - Example: Only allow `Date`s with a month from 1-12.

Instance methods

- **instance method** (or **object method**): Exists inside each object of a class and gives behavior to each object.

```
public type name(parameters) {
    statements;
}
```

- same syntax as static methods, but without `static` keyword

Example:

```
public void tranlate(int dx, int dy) {
    x += dx;
    y += dy;
}
```

The implicit parameter

- **implicit parameter**:
  The object on which an instance method is being called.

  – If we have a `Point` object `p1` and call `p1.translate(5, 3);`
  the object referred to by `p1` is the implicit parameter.

  – If we have a `Point` object `p2` and call `p2.translate(4, 1);`
  the object referred to by `p2` is the implicit parameter.

  – The instance method can refer to that object's fields.
    - We say that it executes in the *context* of a particular object.
    - `translate` can refer to the `x` and `y` of the object it was called on.

FEI STU
223

Categories of methods

- **accessor**:   A method that lets clients examine object state.
  - Examples: `distance, distanceFromOrigin`
  - often has a non-`void` return type

- **mutator**:   A method that modifies an object's state.
  - Examples: `setLocation, translate`

- **helper**: Assists some other method in performing its task.
  - often declared as private so outside clients cannot call it

The `toString` method

tells Java how to convert an object into a `String` for printing

```
public String toString() {
    code that returns a String representing this object;

}
```

- Method name, return, and parameters must match *exactly*.

- Example:

```
// Returns a String representing this Point.
public String toString() {
    return "(" + x + ", " + y + ")";
}
```

Constructors

- **constructor**: Initializes the state of new objects.

```
public type(parameters) {
    statements;
}
```

– runs when the client uses the `new` keyword

– no return type is specified; implicitly "returns" the new object

```
public class Point {
    private int x;
    private int y;

    public Point(int initialX, int initialY) {
        x = initialX;
        y = initialY;
    }
```

Multiple constructors

- A class can have multiple constructors.
  - Each one must accept a unique set of parameters.

- *Example:* A `Point` constructor with no parameters that initializes the point to (0, 0).
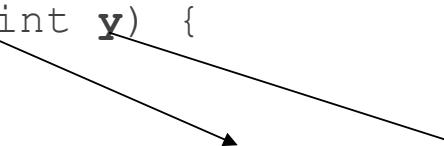
```
// Constructs a new point at (0, 0).
public Point() {
    x = 0;
    y = 0;
}
```

The keyword `this`

- **`this`** : Refers to the implicit parameter inside your class.

  (a variable that stores the object on which a method is called)

  - Refer to a field: `this`**.field**

  - Call a method: `this`**.method**(**parameters**)**;**

  - One constructor `this`(**parameters**)**;**
    can call another:

Calling another constructor

```java
public class Point {
    private int x;
    private int y;

    public Point() {
        this(0, 0);
    }

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    ...
}
```

- Avoids redundancy between constructors
- Only a constructor (not a method) can call another constructor
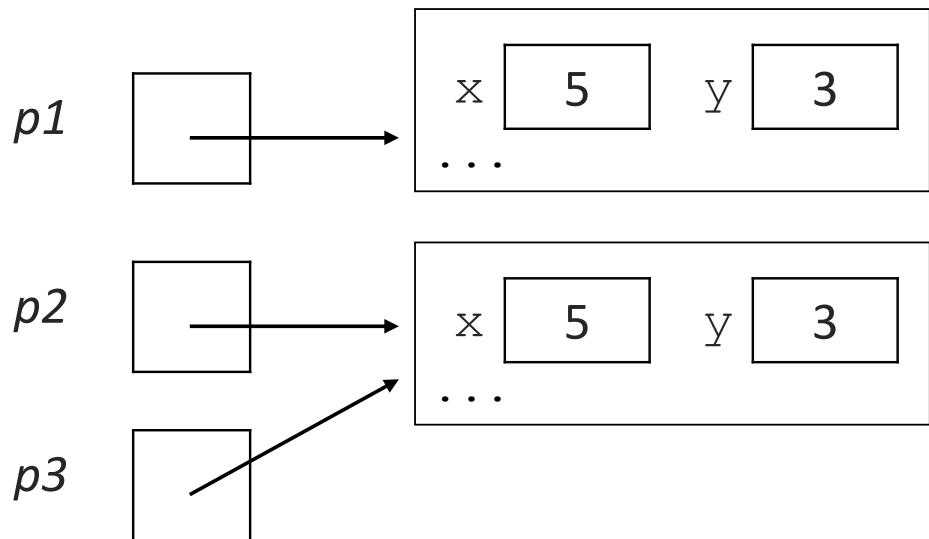
Comparing objects for equality and ordering

Comparing objects

- The `==` operator does not work well with objects.

  `==` compares references to objects, not their state.

  It only produces `true` when you compare an object to itself.

```
Point p1 = new Point(5, 3);
Point p2 = new Point(5, 3);
Point p3 = p2;
```

```
// p1 == p2 is false;
// p1 == p3 is false;
// p2 == p3 is true
```

The `equals` method

- The `equals` method compares the state of objects.

```
if (str1.equals(str2)) {
    System.out.println("the strings are equal");
}
```

- But if you write a class, its `equals` method behaves like `==`

```
if (p1.equals(p2)) {    // false :-(
    System.out.println("equal");
}
```

- This is the default behavior we receive from class `Object`.

- Java doesn't understand how to compare new classes by default.

- The standard way for a Java class to define a comparison function for its objects is to define a `compareTo` method.

    - Example: in the `String` class, there is a method:

    `public int compareTo(String other)`

- A call of **A**.`compareTo`(**B**) will return:

    a value <   0   if **A** comes "before" **B** in the ordering,

    a value >   0   if **A** comes "after" **B** in the ordering,

    or   0   if **A** and **B** are considered "equal" in the ordering.

Using `compareTo`

- `compareTo` can be used as a test in an `if` statement.

```
String a = "alice";
String b = "bob";
if (a.compareTo(b) < 0) {   // true

    ...

}
```

Primitives	Objects
`if (a < b) { ...`	`if (a.compareTo(b) < 0) { ...`
`if (a <= b) { ...`	`if (a.compareTo(b) <= 0) { ...`
`if (a == b) { ...`	`if (a.compareTo(b) == 0) { ...`
`if (a != b) { ...`	`if (a.compareTo(b) != 0) { ...`
`if (a >= b) { ...`	`if (a.compareTo(b) >= 0) { ...`
`if (a > b) { ...`	`if (a.compareTo(b) > 0) { ...`

compareTo and collections

- You can use an array or list of strings with Java's included binary search method because it calls `compareTo` internally.

```
String[] a = {"al", "bob", "cari", "dan", "mike"};
int index = Arrays.binarySearch(a, "dan");   // 3
```

- Java's `TreeSet`/`Map` use `compareTo` internally for ordering.

```
Set<String> set = new TreeSet<String>();
for (String s : a) {
    set.add(s);
}
System.out.println(s);
// [al, bob, cari, dan, mike]
```

Comparable (10.2)

```
public interface Comparable<E> {
    public int compareTo(E other);
}
```

- A class can implement the `Comparable` interface to define a natural ordering function for its objects.

- A call to your `compareTo` method should return:

  a value <   0   if `this` object comes "before" the `other` object,

  a value >   0   if `this` object comes "after" the `other` object,

  or   0   if `this` object is considered "equal" to the `other`.

- If you want multiple orderings, use a `Comparator` instead (see Ch. 13.1)

FEI STU
236

Comparable template

```
public class name implements Comparable<name> {

    ...

    public int compareTo(name other) {
        ...
    }
}
```

Comparable example

```java
public class Point implements Comparable<Point> {
    private int x;
    private int y;
    …

    // sort by x and break ties by y
    public int compareTo(Point other) {
        if (x < other.x) {
            return -1;
        } else if (x > other.x) {
            return 1;
        } else if (y < other.y) {
            return -1;    // same x, smaller y
        } else if (y > other.y) {
            return 1;    // same x, larger y
        } else {
            return 0;    // same x and same y
        }
    }
}
```

- *subtraction trick* - Subtracting related numeric values produces the right result for what you want `compareTo` to return:

```
// sort by x and break ties by y
public int compareTo(Point other) {
    if (x != other.x) {
        return x - other.x;   // different x
    } else {
        return y - other.y;   // same x; compare y
    }
}
```

- The idea:
  - if `x > other.x,`   then `x - other.x > 0`
  - if `x < other.x,`   then `x - other.x < 0`
  - if `x == other.x,`   then `x - other.x == 0`

    - NOTE: This trick doesn't work for `doubles` (but see `Math.signum`)

compareTo tricks 2

- *delegation trick* - If your object's fields are comparable (such as strings), use their `compareTo` results to help you:

```java
// sort by employee name, e.g. "Jim" < "Susan"
public int compareTo(Employee other) {
    return name.compareTo(other.getName());
}
```
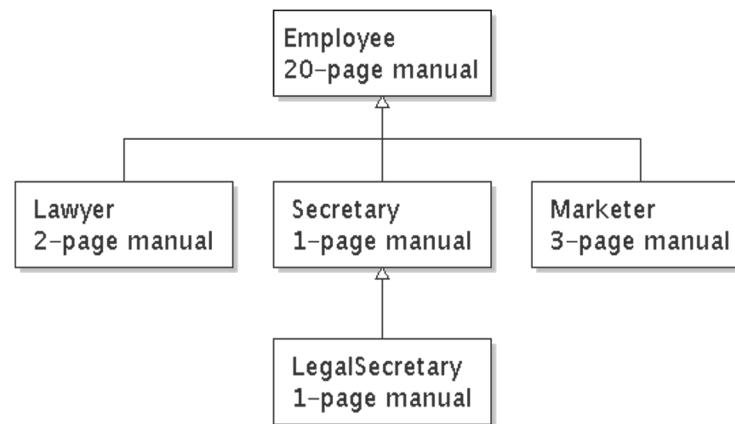
- *toString trick* - If your object's `toString` representation is related to the ordering, use that to help you:

```java
// sort by date, e.g. "09/19" > "04/01"
public int compareTo(Date other) {
    return toString().compareTo(other.toString());
}
```

Inheritance

Inheritance

- **inheritance**: Forming new classes based on existing ones.
  - a way to share/**reuse code** between two or more classes

  - **superclass**: Parent class being extended.
  - **subclass**: Child class that inherits behavior from superclass.
    - gets a copy of every field and method from superclass

  - **is-a relationship**: Each object of the subclass also "is a(n)" object of the superclass and can be treated as one.

Inheritance syntax

```
public class name extends superclass {
```

- Example:

```
public class Lawyer extends Employee {

    ...

}
```

- By extending `Employee`, each `Lawyer` object now:
  - receives a copy of each method from `Employee` automatically
  - can be treated as an `Employee` by client code

- Lawyer can also replace ("override") behavior from Employee.

Overriding Methods

- **override**: To write a new version of a method in a subclass that replaces the superclass's version.

  - No special syntax required to override a superclass method.
    Just write a new version of it in the subclass.

```
public class Lawyer extends Employee {
    // overrides getVacationForm in Employee class
    public String getVacationForm() {
        return "pink";
    }
    ...
}
```

The `super` keyword

- A subclass can call its parent's method/constructor:

```
super.method(parameters)      // method
super(parameters);            // constructor
```

```java
public class Lawyer extends Employee {
    public Lawyer(String name) {
        super(name);
    }

    // give Lawyers a $5K raise (better)
    public double getSalary() {
        double baseSalary = super.getSalary();
        return baseSalary + 5000.00;
    }
}
```

Subclasses and fields

```
public class Employee {
    private double salary;
    ...
}


public class Lawyer extends Employee {
    ...
    public void giveRaise(double amount) {
        salary += amount;    // error; salary is private
    }
}
```

- Inherited private fields/methods cannot be directly accessed by subclasses. *(The subclass has the field, but it can't touch it.)*

  - How can we allow a subclass to access/modify these fields?

Protected fields/methods

```
protected type name;        // field

protected type name(type name, ..., type name) {
    statement(s);           // method
}
```

- a **protected field** or **method** can be seen/called only by:
  - the class itself, and its subclasses
  - also by other classes in the same "package" (discussed later)
  - useful for allowing selective access to inner class implementation

```
public class Employee {
    protected double salary;
    ...
}
```

Inheritance and constructors

- If we add a constructor to the `Employee` class, our subclasses do not compile. The error:

```
Lawyer.java:2: cannot find symbol
symbol  : constructor Employee()
location: class Employee
public class Lawyer extends Employee {
       ^
```

- The short explanation: Once we write a constructor (that requires parameters) in the superclass, we must now write constructors for our employee subclasses as well.

Inheritance and constructors

- Constructors are not inherited.
  - Subclasses don't inherit the `Employee(int)` constructor.

  - Subclasses receive a default constructor that contains:

```
public Lawyer() {
    super();        // calls Employee() constructor
}
```

- But our `Employee(int)` replaces the default `Employee()`.
  - The subclasses' default constructors are now trying to call a non-existent default `Employee` constructor.

Calling superclass constructor

super(**parameters**);

- Example:
```
public class Lawyer extends Employee {
    public Lawyer(int years) {
        super(years);  // calls Employee c'tor
    }
    ...
}
```

- The super call must be the first statement in the constructor.

Polymorphism

Polymorphism

- **polymorphism**: Ability for the same code to be used with different types of objects and behave differently with each.

  - `System.out.println` can print any type of object.
    - Each one displays in its own way on the console.

  - `CritterMain` can interact with any type of critter.
    - Each one moves, fights, etc. in its own way.

- A variable of type *T* can hold an object of any subclass of *T*.

  ```
  Employee ed = new Lawyer();
  ```

  - You can call any methods from the `Employee` class on `ed`.

- When a method is called on `ed`, it behaves as a `Lawyer`.

  ```
  System.out.println(ed.getSalary());       // 50000.0
  System.out.println(ed.getVacationForm()); // pink
  ```

Polymorphic parameters

- You can pass any subtype of a parameter's type.

```java
public static void main(String[] args) {
    Lawyer lisa = new Lawyer();
    Secretary steve = new Secretary();
    printInfo(lisa);
    printInfo(steve);
}

public static void printInfo(Employee e) {
    System.out.println("pay  : " + e.getSalary());
    System.out.println("vdays: " + e.getVacationDays());
    System.out.println("vform: " + e.getVacationForm());
    System.out.println();
}
```

OUTPUT:

```
pay  : 50000.0    pay  : 50000.0
vdays: 15         vdays: 10
vform: pink       vform: yellow
```

- Arrays of superclass types can store any subtype as elements.

```
public static void main(String[] args) {
    Employee[] e = {new Lawyer(),   new Secretary(),
                    new Marketer(), new LegalSecretary()};

    for (int i = 0; i < e.length; i++) {
        System.out.println("pay  : " + e[i].getSalary());
        System.out.println("vdays: " + i].getVacationDays());
        System.out.println();
    }
}
```

Output:

```
pay  : 50000.0        pay  : 60000.0
vdays: 15             vdays: 10

pay  : 50000.0        pay  : 55000.0
vdays: 10             vdays: 10
```

Casting references

- A variable can only call that type's methods, not a subtype's.

  ```
  Employee ed = new Lawyer();
  int hours = ed.getHours();   // ok; in Employee
  ed.sue();                    // compiler error
  ```

  - The compiler's reasoning is, variable `ed` could store any kind of employee, and not all kinds know how to `sue` .

- To use `Lawyer` methods on `ed`, we can type-cast it.

  ```
  Lawyer theRealEd = (Lawyer) ed;
  theRealEd.sue();                // ok

  ((Lawyer) ed).sue();            // shorter version
  ```

More about casting

- The code crashes if you cast an object too far down the tree.

```
Employee eric = new Secretary();
((Secretary) eric).takeDictation("hi");    // ok
((LegalSecretary) eric).fileLegalBriefs(); // error
//    (Secretary doesn't know how to file briefs)
```

- You can cast only up and down the tree, not sideways.

```
Lawyer linda = new Lawyer();
((Secretary) linda).takeDictation("hi");   // error
```
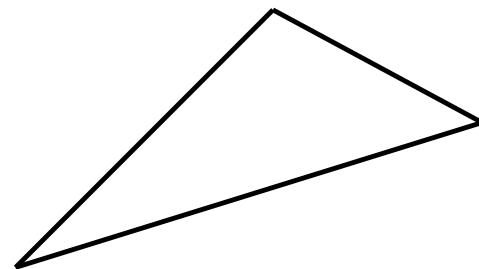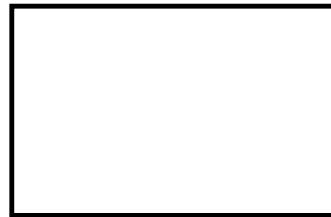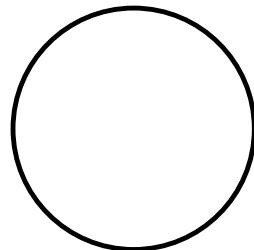
- Casting doesn't actually change the object's behavior.
  It just gets the code to compile/run.

```
((Employee) linda).getVacationForm()       // pink
```

Interfaces

Shapes example

- Consider the task of writing classes to represent 2D shapes such as `Circle`, `Rectangle`, and `Triangle`.

- Certain operations are common to all shapes:
  - perimeter: distance around the outside of the shape
  - area: amount of 2D space occupied by the shape

  - Every shape has these, but each computes them differently.

FEI STU
259
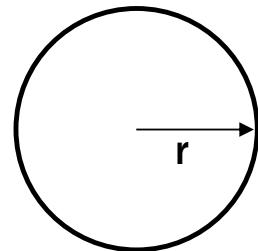
Shape area and perimeter

- Circle (as defined by radius r):

  area $= \pi r^2$

  perimeter $= 2 \pi r$

- Rectangle (as defined by width w and height h):

  area $= w h$

  perimeter $= 2w + 2h$
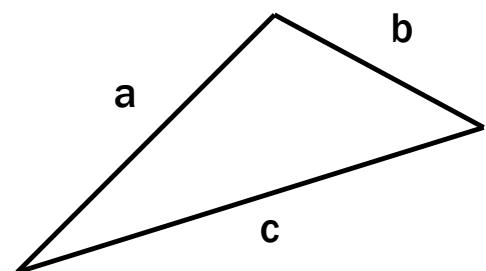
- Triangle (as defined by side lengths a, b, and c)

  area $= \sqrt{(s(s-a)(s-b)(s-c))}$

  where $s = \frac{1}{2}(a+b+c)$

  perimeter $= a + b + c$

Common behavior

- Suppose we have 3 classes `Circle`, `Rectangle`, `Triangle`.
  - Each has the methods `perimeter` and `area`.

- We'd like our client code to be able to treat different kinds of shapes in the same way:
  - Write a method that prints any shape's area and perimeter.
  - Create an array to hold a mixture of the various shape objects.
  - Write a method that could return a rectangle, a circle, a triangle, or any other kind of shape.
  - Make a `DrawingPanel` display many shapes on screen.

Interfaces

- **interface**: A list of methods that a class can promise to implement.

  - Inheritance gives you an is-a relationship *and* code sharing.
    - A `Lawyer` can be treated as an `Employee` and inherits its code.

  - Interfaces give you an is-a relationship *without* code sharing.
    - A `Rectangle` object can be treated as a `Shape` but inherits no code.

  - Analogous to non-programming idea of roles or certifications:
    - "I'm certified as a CPA accountant.
      This assures you I know how to do taxes, audits, and consulting."
    - "I'm 'certified' as a Shape, because I implement the Shape interface.
      This assures you I know how to compute my area and perimeter."

Interface syntax

```
public interface name {
    public type name(type name, ..., type name);
    public type name(type name, ..., type name);
    ...
    public type name(type name, ..., type name);
}
```

Example:
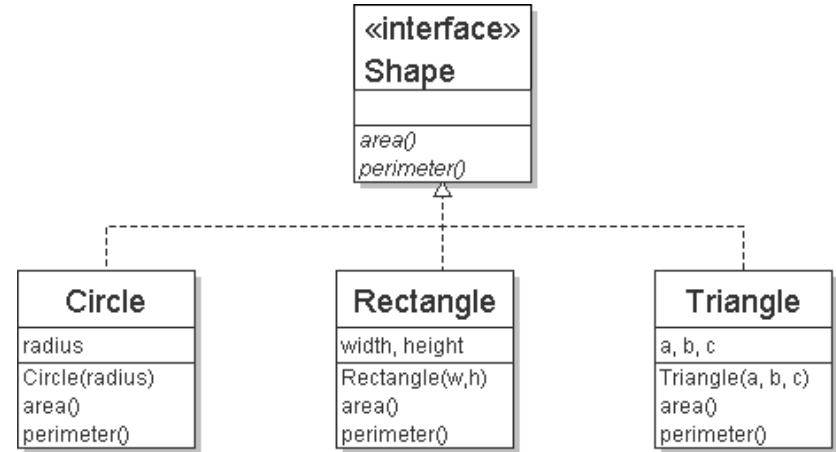
```
public interface Vehicle {
    public int getSpeed();
    public void setDirection(int direction);
}
```

Shape interface

```
// Describes features common to all shapes.
public interface Shape {
    public double area();
    public double perimeter();
}
```

- Saved as `Shape.java`

```
«interface»
Shape

area()
perimeter()
```

```
Circle
radius
Circle(radius)
area()
perimeter()
```

```
Rectangle
width, height
Rectangle(w,h)
area()
perimeter()
```

```
Triangle
a, b, c
Triangle(a, b, c)
area()
perimeter()
```

- **abstract method**: A header without an implementation.
  - The actual bodies are not specified, because we want to allow each class to implement the behavior in its own way.

Implementing an interface

```
public class name implements interface {
    ...
}
```

- A class can declare that it "implements" an interface.
  - The class promises to contain each method in that interface.
    (Otherwise it will fail to compile.)

  - Example:
    ```
    public class Bicycle implements Vehicle {
        ...
    }
    ```

FEI STU
265

Interface requirements

```java
public class Banana implements Shape {

    // haha, no methods! pwned

}
```

- If we write a class that claims to be a `Shape` but doesn't implement `area` and `perimeter` methods, it will not compile.

```
Banana.java:1: Banana is not abstract and does not override abstract
method area() in Shape
public class Banana implements Shape {
          ^
```

Interfaces + polymorphism

- Interfaces benefit the *client code* author the most.

  - they allow **polymorphism**
    (the same code can work with different types of objects)

```java
public static void printInfo(Shape s) {
    System.out.println("The shape: " + s);
    System.out.println("area : " + s.area());
    System.out.println("perim: " + s.perimeter());
    System.out.println();
}
...
Circle circ = new Circle(12.0);
Triangle tri = new Triangle(5, 12, 13);
printInfo(circ);
printInfo(tri);
```

Abstract Classes

List classes example

- Suppose we have implemented the following two list classes:

  - `ArrayList`
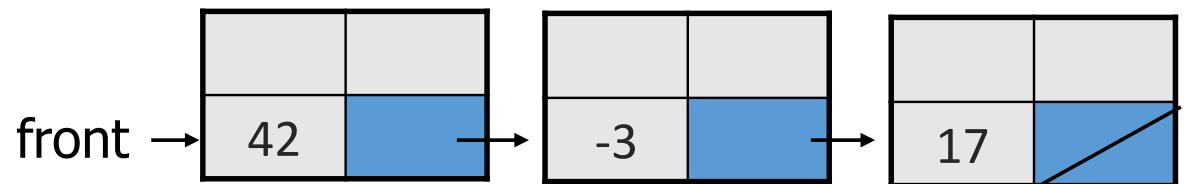
  - `LinkedList`

	42	-3	17

- We have a `List` interface to indicate that both implement a List ADT.
- Problem:
  - Some of their methods are implemented the same way (redundancy).

Common code

- Notice that some of the methods are implemented the same way in both the array and linked list classes.

  - add(**value**)
  - contains
  - isEmpty

- Should we change our interface to a class? Why / why not?
  - How can we capture this common behavior?

Abstract classes (9.6)

- **abstract class**: A hybrid between an interface and a class.
  - defines a superclass type that can contain method declarations (like an interface) and/or method bodies (like a class)
  - like interfaces, abstract classes that cannot be instantiated (cannot use `new` to create any objects of their type)

- What goes in an abstract class?
  - implementation of common state and behavior that will be inherited by subclasses (parent class role)
  - declare generic behaviors that subclasses implement (interface role)

Abstract class syntax

```
// declaring an abstract class
public abstract class name {

    ...

    // declaring an abstract method
    // (any subclass must implement it)
    public abstract type name(parameters);


}
```

- A class can be `abstract` even if it has no abstract methods
- You can create variables (but not objects) of the abstract type

Abstract and interfaces

- Normal classes that claim to implement an interface must implement all methods of that interface:

```
public class Empty implements List {}  // error
```

- Abstract classes can claim to implement an interface without writing its methods; subclasses must implement the methods.

```
public abstract class Empty implements List {} // ok

public class Child extends Empty {}            // error
```

An abstract list class

```java
// Superclass with common code for a list of integers.
public abstract class AbstractList implements List {
    public void add(int value) {
        add(size(), value);
    }

    public boolean contains(int value) {
        return indexOf(value) >= 0;
    }

    public boolean isEmpty() {
        return size() == 0;
    }
}


public class ArrayList extends AbstractList { …

public class LinkedList extends AbstractList { …
```

Abstract class vs. interface

- Why do both interfaces and abstract classes exist in Java?
  - An abstract class can do everything an interface can do and more.
  - So why would someone ever use an interface?

- Answer: Java has single inheritance.
  - can extend only one superclass
  - can implement many interfaces

  - Having interfaces allows a class to be part of a hierarchy (polymorphism) without using up its inheritance relationship.
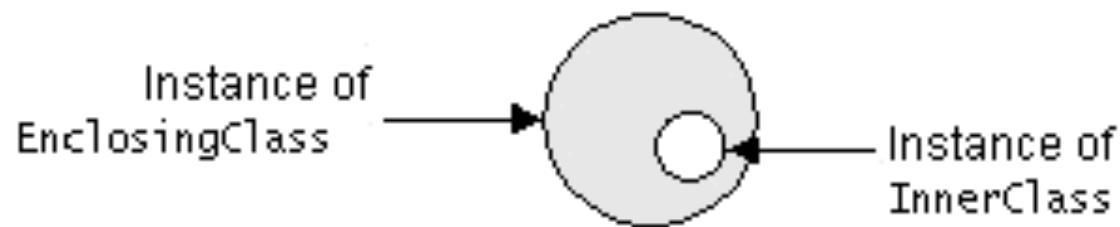
public class Pokus extends Shape implements interface 1, interface 2, … {

{

Inner Classes

Inner classes

- **inner class**: A class defined inside of another class.
  - can be created as `static` or non-static
  - we will focus on standard non-static ("nested") inner classes

- usefulness:
  - inner classes are hidden from other classes (encapsulated)
  - inner objects can access/modify the fields of the outer object

Instance of EnclosingClass → ⟶ Instance of InnerClass

Inner class syntax

```
// outer (enclosing) class
public class name {
    ...

    // inner (nested) class
    private class name {
        ...
    }
}
```

- Only this file can see the inner class or make objects of it.
- Each inner object is associated with the outer object that created it, so it can access/modify that outer object's methods/fields.
  - If necessary, can refer to outer object as **OuterClassName**`.this`

Example: Array list iterator

```java
public class ArrayList extends AbstractList {
    ...

    // not perfect; doesn't forbid multiple removes in a row
    private class ArrayIterator implements Iterator<Integer> {
        private int index;    // current position in list

        public ArrayIterator() {
            index = 0;
        }

        public boolean hasNext() {
            return index < size();
        }

        public Integer next() {
            index++;
             return get(index - 1);
          }

        public void remove() {
            ArrayList.this.remove(index - 1);
            index--;
        }
    }
}
```
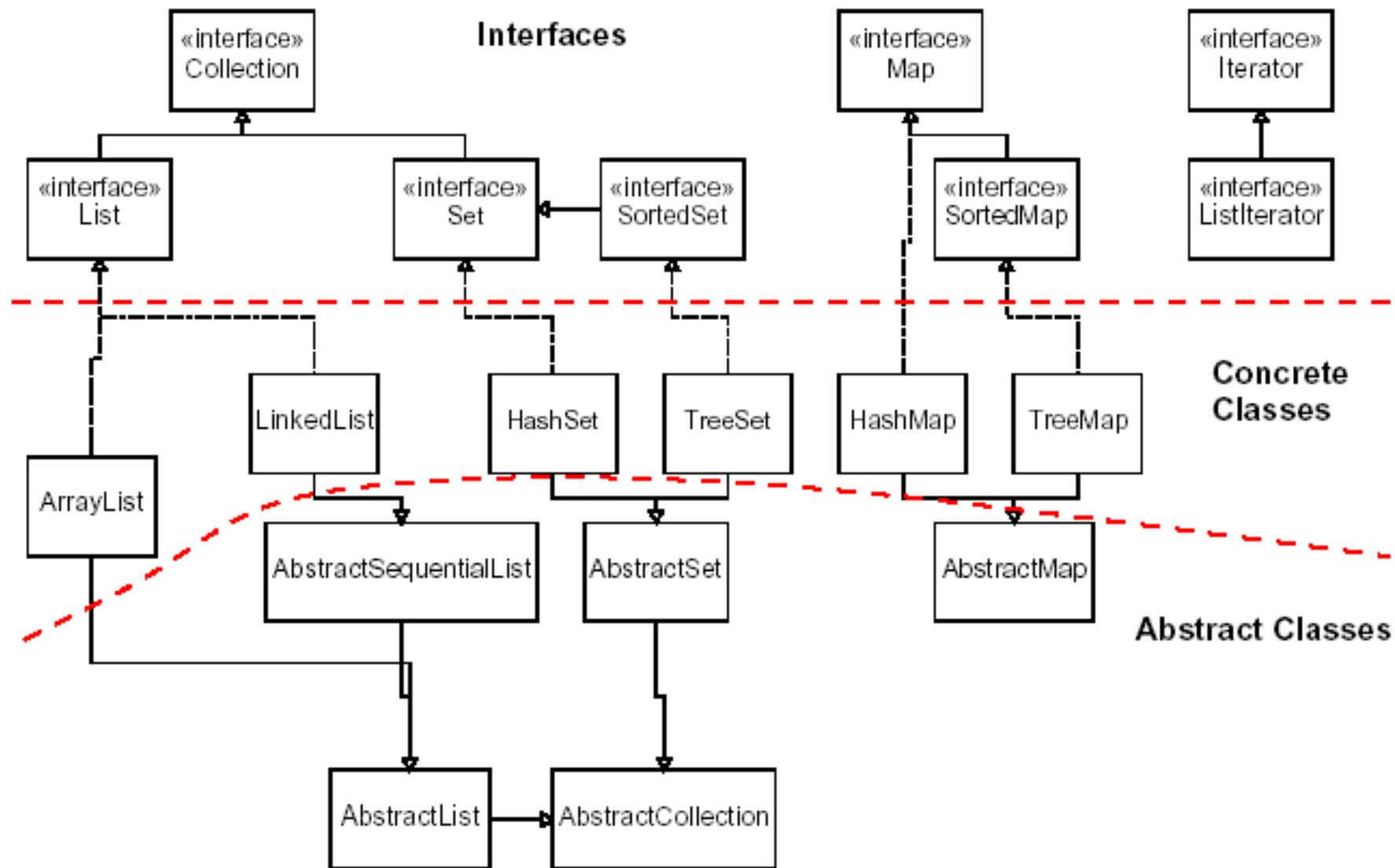
Collections

Collections

- **collection**: an object that stores data; a.k.a. "data structure"
  - the objects stored are called **elements**
  - some collections maintain an ordering; some allow duplicates
  - typical operations: *add*, *remove*, *clear*, *contains* (search), *size*

  - examples found in the Java class libraries:
    - `ArrayList, LinkedList, HashMap, TreeSet, PriorityQueue`

  - all collections are in the `java.util` package
    ```
    import java.util.*;
    ```
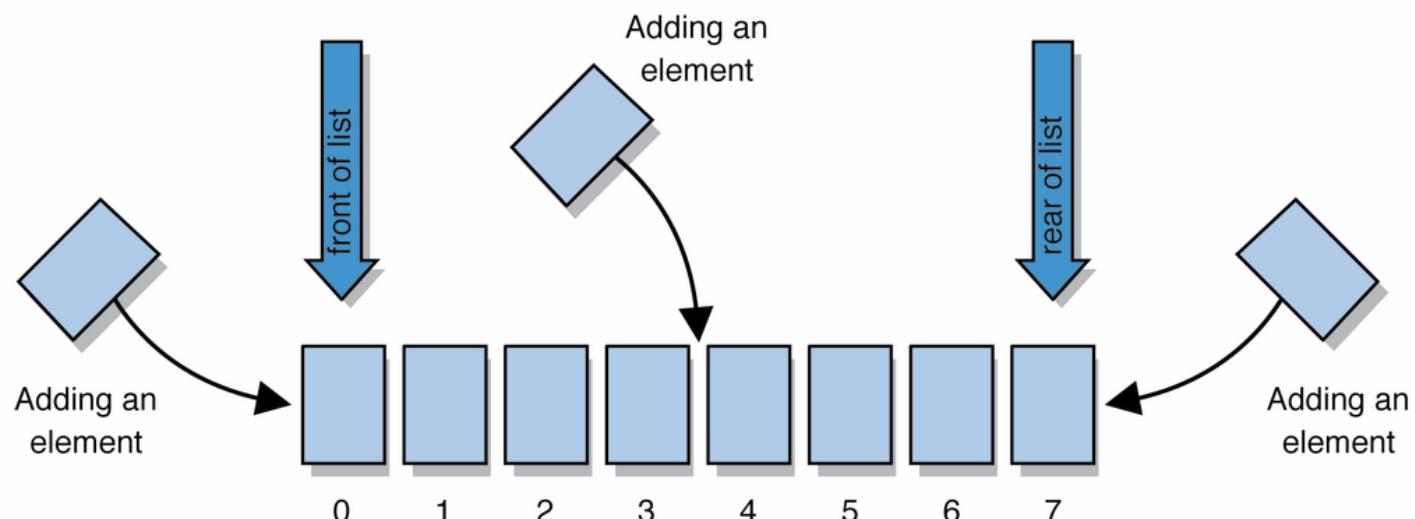
Java collection framework

Lists

- **list**: a collection storing an ordered sequence of elements
  - each element is accessible by a 0-based **index**
  - a list has a **size** (number of elements that have been added)
  - elements can be added to the front, back, or elsewhere
  - in Java, a list can be represented as an **ArrayList** object

Idea of a list

- Rather than creating an array of boxes, create an object that represents a "list" of items. (initially an empty list.)

  ```
  []
  ```

- You can add items to the list.
  - The default behavior is to add to the end of the list.

    ```
    [hello, ABC, goodbye, okay]
    ```

- The list object keeps track of the element values that have been added to it, their order, indexes, and its total size.
  - Think of an "array list" as an automatically resizing array object.
  - Internally, the list is implemented using an array and a size field.

`add(`**`value`**`)`	appends value at end of list
`add(`**`index, value`**`)`	inserts given value just before the given index, shifting subsequent values to the right
`clear()`	removes all elements of the list
`indexOf(`**`value`**`)`	returns first index where given value is found in list (-1 if not found)
`get(`**`index`**`)`	returns the value at given index
`remove(`**`index`**`)`	removes/returns value at given index, shifting subsequent values to the left
`set(`**`index, value`**`)`	replaces value at given index with given value
`size()`	returns the number of elements in list
`toString()`	returns a string representation of the list such as `"[3, 42, -7, 15]"`

`addAll(`**`list`**`)` `addAll(`**`index,`** **`list`**`)`	adds all elements from the given list to this list (at the end of the list, or inserts them at the given index)
`contains(`**`value`**`)`	returns true if given value is found somewhere in this list
`containsAll(`**`list`**`)`	returns true if this list contains every element from given list
`equals(`**`list`**`)`	returns true if given other list contains the same elements
`iterator()` `listIterator()`	returns an object used to examine the contents of the list
`lastIndexOf(`**`value`** `)`	returns last index value is found in list (-1 if not found)
`remove(`**`value`**`)`	finds and removes the given value from this list
`removeAll(`**`list`**`)`	removes any elements found in the given list from this list
`retainAll(`**`list`**`)`	removes any elements *not* found in given list from this list
`subList(`**`from, to`**`)`	returns the sub-portion of the list between indexes **from** (inclusive) and **to** (exclusive)
`toArray()`	returns the elements in this list as an array

Type Parameters (Generics)
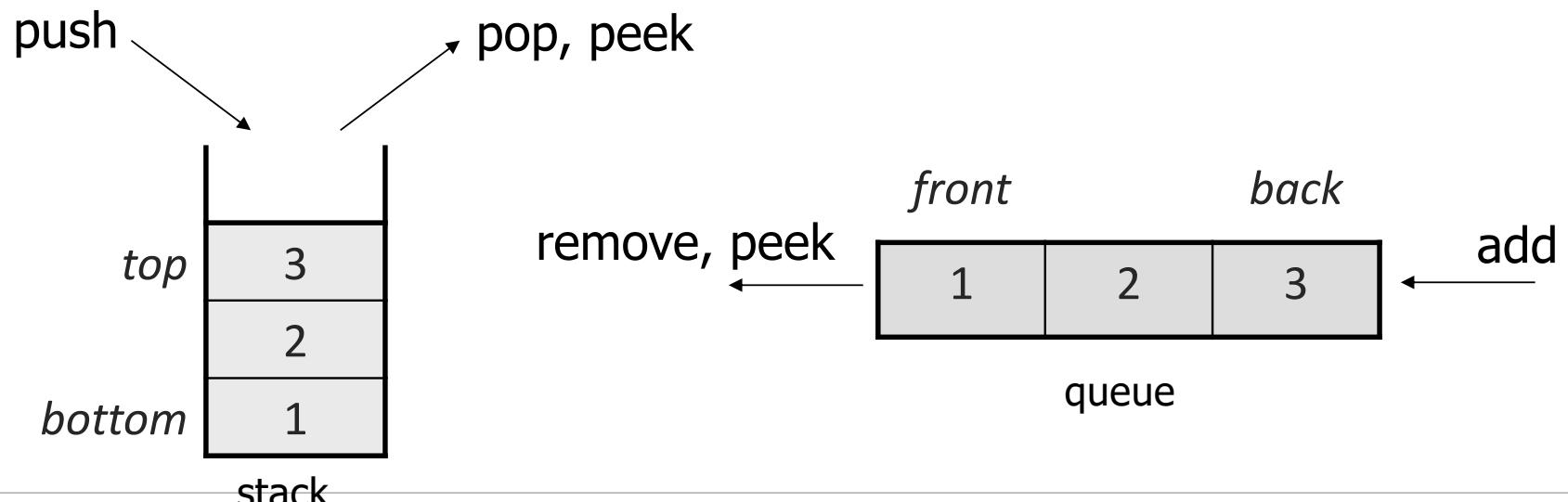
```
List<Type> name = new ArrayList<Type>();
```

- When constructing an `ArrayList`, you must specify the type of elements it will contain between `<` and `>`.
  - This is called a *type parameter* or a *generic* class.
  - Allows the same `ArrayList` class to store lists of different types.

```
List<String> names = new ArrayList<String>();
names.add("Marty Stepp");
names.add("Stuart Reges");
```

FEI STU
287

Stacks and queues

- Sometimes it is good to have a collection that is less powerful, but is optimized to perform certain operations very quickly.

- Two specialty collections:
  - **stack**: Retrieves elements in the reverse of the order they were added.
  - **queue**: Retrieves elements in the same order they were added.

push → top 3, 2, bottom 1 — pop, peek

stack

remove, peek ← *front* 1, 2, 3 *back* ← add

queue
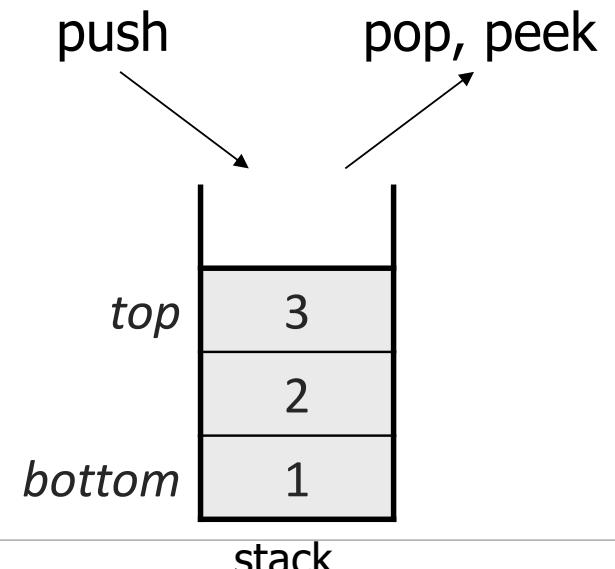
Stacks

- **stack**: A collection based on the principle of adding elements and retrieving them in the opposite order.
  - Last-In, First-Out ("LIFO")
  - The elements are stored in order of insertion, but we do not think of them as having indexes.
  - The client can only add/remove/examine the last element added (the "top").

- basic stack operations:
  - **push**: Add an element to the top.
  - **pop**: Remove the top element.
  - **peek**: Examine the top element.

push   pop, peek

top	3
	2
bottom	1

stack

`Stack<`**E**`>()`	constructs a new stack with elements of type **E**
`push(`**value**`)`	places given value on top of stack
`pop()`	removes top value from stack and returns it; throws `EmptyStackException` if stack is empty
`peek()`	returns top value from stack without removing it; throws `EmptyStackException` if stack is empty
`size()`	returns number of elements in stack
`isEmpty()`	returns `true` if stack has no elements

```
Stack<Integer> s = new Stack<Integer>();
s.push(42);
s.push(-3);
s.push(17);                     // bottom [42, -3, 17] top

System.out.println(s.pop()); // 17
```

• `Stack` has other methods, but you should not use them.
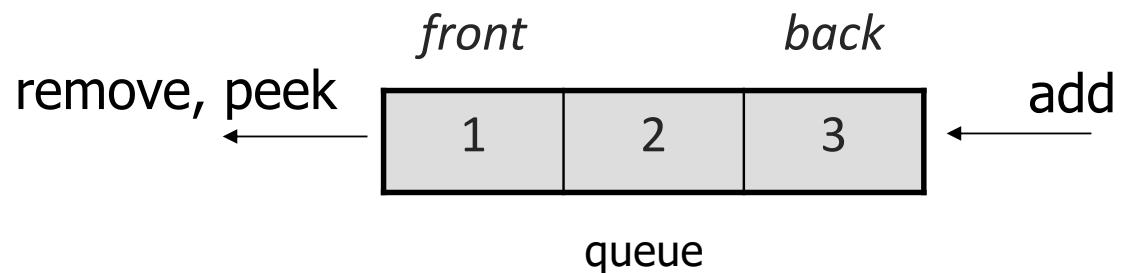
Queues

- **queue**: Retrieves elements in the order they were added.
  - First-In, First-Out ("FIFO")
  - Elements are stored in order of insertion but don't have indexes.
  - Client can only add to the end of the queue, and can only examine/remove the front of the queue.

- basic queue operations:
  - **add** (enqueue): Add an element to the back.
  - **remove** (dequeue): Remove the front element.
  - **peek**: Examine the front element.

add(**value**)	places given value at back of queue
remove()	removes value from front of queue and returns it; throws a NoSuchElementException if queue is empty
peek()	returns front value from queue without removing it; returns null if queue is empty
size()	returns number of elements in queue
isEmpty()	returns true if queue has no elements

```
Queue<Integer> q = new LinkedList<Integer>();
q.add(42);
q.add(-3);
q.add(17);          // front [42, -3, 17] back

System.out.println(q.remove());   // 42
```

- **IMPORTANT**: When constructing a queue you must use a new LinkedList object instead of a new Queue object.

Queue idioms

- As with stacks, must pull contents out of queue to view them.

```
// process (and destroy) an entire queue
while (!q.isEmpty()) {
    do something with q.remove();
}
```

- another idiom: Examining each element exactly once.

```
int size = q.size();
for (int i = 0; i < size; i++) {
    do something with q.remove();
    (including possibly re-adding it to the queue)
}
```

Abstract data types (ADTs)

- **abstract data type (ADT)**: A specification of a collection of data and the operations that can be performed on it.
  - Describes *what* a collection does, not *how* it does it

- We don't know exactly how a stack or queue is implemented, and we don't need to.
  - We just need to understand the idea of the collection and what operations it can perform.

    (Stacks are usually implemented with arrays; queues are often implemented using another structure called a linked list.)

ADTs as interfaces (11.1)

- **abstract data type (ADT)**: A specification of a collection of data and the operations that can be performed on it.
  - Describes *what* a collection does, not *how* it does it.

- Java's collection framework uses interfaces to describe ADTs:
  - `Collection`, `Deque`, `List`, `Map`, `Queue`, `Set`

- An ADT can be implemented in multiple ways by classes:
  - `ArrayList` and `LinkedList`   implement `List`
  - `HashSet` and `TreeSet`   implement `Set`
  - `LinkedList`, `ArrayDeque`, etc.   implement `Queue`

    - They messed up on `Stack`; there's no `Stack` interface, just a class.

Using ADT interfaces

When using Java's built-in collection classes:

- It is considered good practice to always declare collection variables using the corresponding ADT interface type:

```
List<String> list = new ArrayList<String>();
```

- Methods that accept a collection as a parameter should also declare the parameter using the ADT interface type:
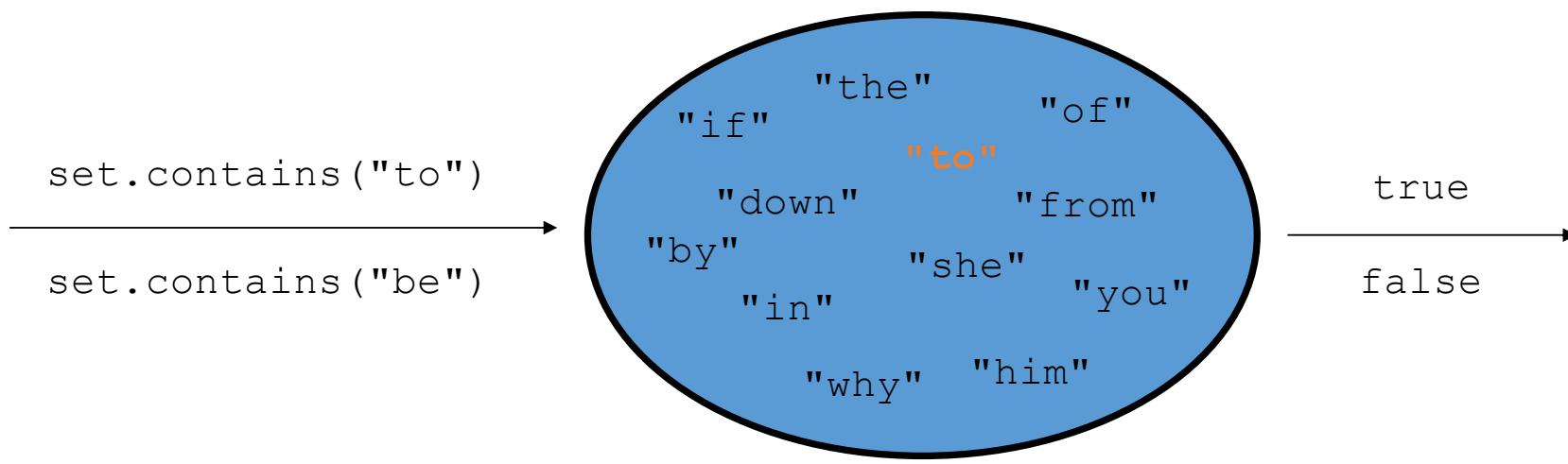
```
public void stutter(List<String> list) {

    ...

}
```

Why use ADTs?

- Why would we want more than one kind of list, queue, etc.?

- Answer: Each implementation is more efficient at certain tasks.
  - `ArrayList` is faster for adding/removing at the end;
    `LinkedList` is faster for adding/removing at the front/middle.
    Etc.

  - You choose the optimal implementation for your task, and if the rest of your code is written to use the ADT interfaces, it will work.

Sets

- **set**: A collection of unique values (no duplicates allowed) that can perform the following operations efficiently:
  - add, remove, search (contains)

  - We don't think of a set as having indexes; we just add things to the set in general and don't worry about order

```
set.contains("to")
```

```
set.contains("be")
```

"the"

"if"   "of"

"to"

"down"   "from"

"by"

"she"

"in"   "you"

"why" "him"

true

false

set

`Set` implementation

- in Java, sets are represented by `Set` interface in `java.util`

- `Set` is implemented by `HashSet` and `TreeSet` classes

  - `HashSet`: implemented using a "hash table" array;
    very fast: **O(1)** for all operations
    elements are stored in unpredictable order

  - `TreeSet`: implemented using a "binary search tree";
    pretty fast: **O(log N)** for all operations
    elements are stored in sorted order

  - `LinkedHashSet`: **O(1)** but stores in order of insertion

Set methods
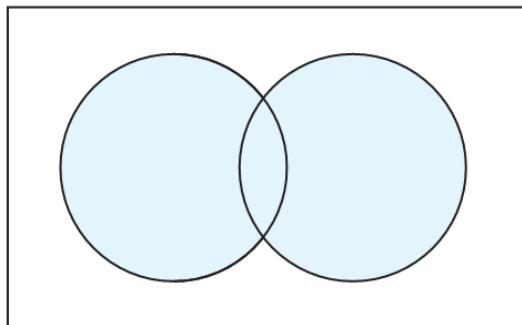
```
List<String> list = new ArrayList<String>();
...
Set<Integer> set = new TreeSet<Integer>();     // empty
Set<String> set2 = new HashSet<String>(list);
```

• can construct an empty set, or one based on a given collection

add(**value**)	adds the given value to the set
contains(**value**)	returns `true` if the given value is found in this set
remove(**value**)	removes the given value from the set
clear()	removes all elements of the set
size()	returns the number of elements in list
isEmpty()	returns `true` if the set's size is 0
toString()	returns a string such as `"[3, 42, -7, 15]"`
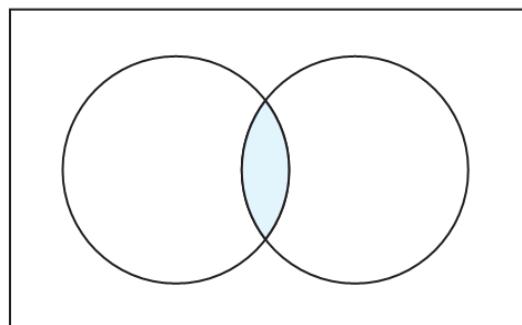
Set operations

A ∪ B   Union
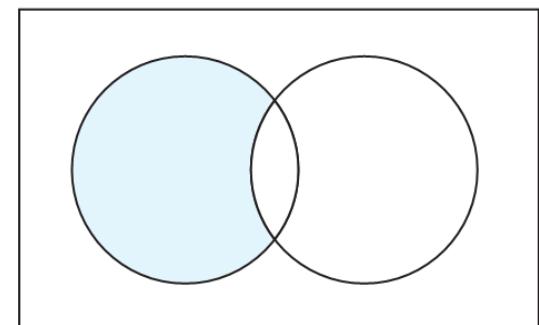
A ∩ B   Intersection

A - B   Difference

addAll

retainAll

removeAll

addAll(**collection**)	adds all elements from the given collection to this set
containsAll(**coll**)	returns `true` if this set contains every element from given set
equals(**set**)	returns `true` if given other set contains the same elements
iterator()	returns an object used to examine set's contents *(seen later)*
removeAll(**coll**)	removes all elements in the given collection from this set
retainAll(**coll**)	removes elements *not* found in given collection from this set
toArray()	returns an array of the elements in this set

Sets and ordering

- `HashSet` : elements are stored in an unpredictable order

```
Set<String> names = new HashSet<String>();
names.add("Jake");
names.add("Robert");
names.add("Marisa");
names.add("Kasey");
System.out.println(names);
// [Kasey, Robert, Jake, Marisa]
```

- `TreeSet` : elements are stored in their "natural" sorted order

```
Set<String> names = new TreeSet<String>();
...
// [Jake, Kasey, Marisa, Robert]
```

- `LinkedHashSet` : elements stored in order of insertion

```
Set<String> names = new LinkedHashSet<String>();
...
// [Jake, Robert, Marisa, Kasey]
```

The "for each" loop (7.1)

```
for (type name : collection) {
    statements;
}
```

- Provides a clean syntax for looping over the elements of a `Set`, `List`, array, or other collection

```
Set<Double> grades = new HashSet<Double>();
...

for (double grade : grades) {
    System.out.println("Student's grade: " + grade);
}
```
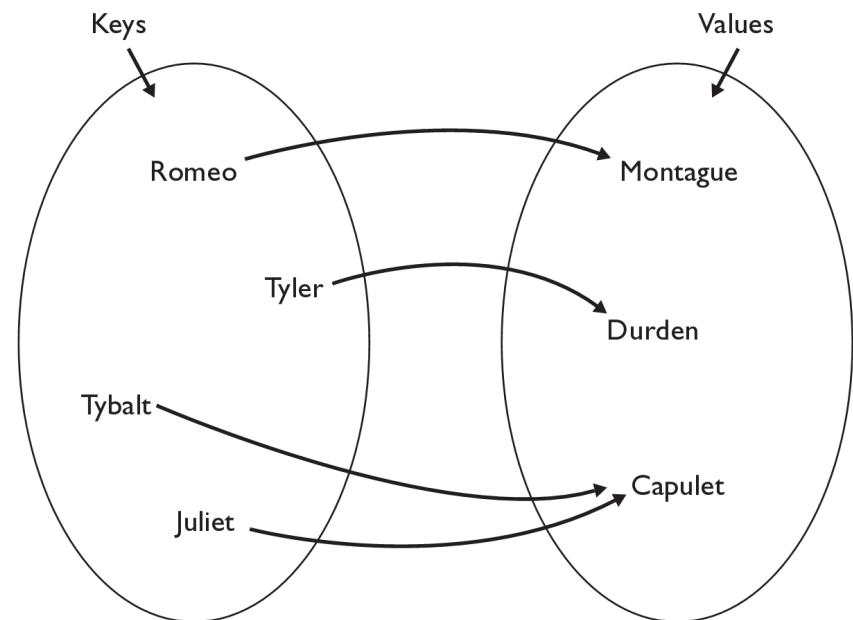
- needed because sets have no indexes; can't `get` element `i`

- **map**: Holds a set of unique *keys* and a collection of *values*, where each key is associated with one value.
  - a.k.a. "dictionary", "associative array", "hash"

- basic map operations:
  - **put**(*key, value*): Adds a mapping from a key to a value.

  - **get**(*key*): Retrieves the value mapped to the key.

  - **remove**(*key*): Removes the given key and its mapped value.

Keys

Values

Romeo

Montague

Tyler

Durden

Tybalt

Capulet

Juliet

myMap.get("Juliet") returns "Capulet"

Map concepts
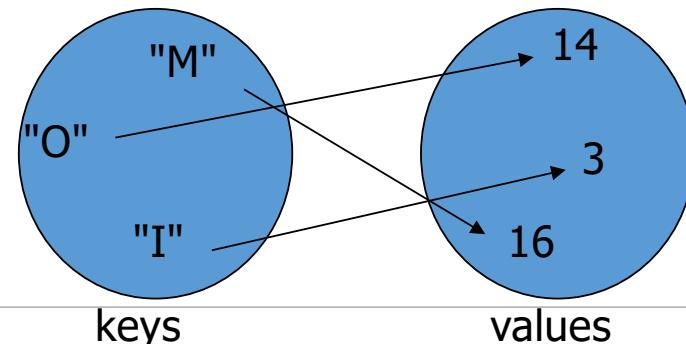
- a map can be thought of as generalization of a tallying array
  - the "index" (key) doesn't have to be an `int`

- recall previous tallying examples from CSE 142
  - count digits: `22092310907`

index 0 1 2 3 4 5 6 7 8 9

value	3	1	3	0	0	0	0	1	0	2

```
// (M)cCain, (O)bama, (I)ndependent
```
  - count votes: `"MOOOOOOMMMMMOOOOOOMOMMIMOMMIMOMMIO"`

key   "M"   "O"   "I"

value	16	14	3

keys   values

`Map` implementation

- in Java, maps are represented by `Map` interface in `java.util`

- `Map` is implemented by the `HashMap` and `TreeMap` classes

  - `HashMap`: implemented using an array called a "hash table"; extremely fast: **O(1)** ; keys are stored in unpredictable order

  - `TreeMap`: implemented as a linked "binary tree" structure; very fast: **O(log N)** ; keys are stored in sorted order

  - A map requires 2 type parameters: one for keys, one for values.

```java
// maps from String keys to Integer values
Map<String, Integer> votes = new HashMap<String, Integer>();
votes.put("Obama", Integer(0));
votes.put("McCain", Integer(0));
a = votes.get("Obama");
votes.put("Obama", a+1);
```

FEI STU
306

Map methods

`put(`**`key,  value`**`)`	adds a mapping from the given key to the given value; if the key already exists, replaces its value with the given one
`get(`**`key`**`)`	returns the value mapped to the given key (`null` if not found)
`containsKey(`**`key`**`)`	returns `true` if the map contains a mapping for the given key
`remove(`**`key`**`)`	removes any existing mapping for the given key
`clear()`	removes all key/value pairs from the map
`size()`	returns the number of key/value pairs in the map
`isEmpty()`	returns `true` if the map's size is 0
`toString()`	returns a string such as `"{a=90, d=60, c=70}"`

`keySet()`	returns a set of all keys in the map
`values()`	returns a collection of all values in the map
`putAll(`**`map`**`)`	adds all key/value pairs from the given map to this map
`equals(`**`map`**`)`	returns `true` if given map has the same mappings as this one
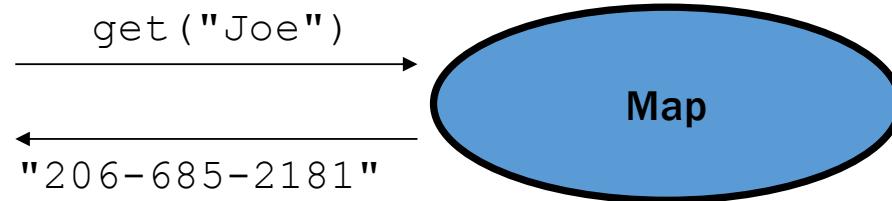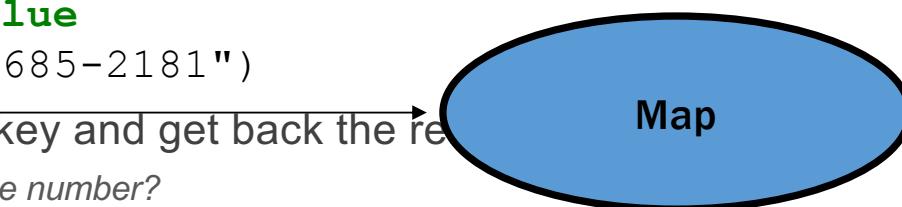
Using maps

- A map allows you to get from one half of a pair to the other.
  - Remembers one piece of information about every index (key).

```
//    key        value
put("Joe", "206-685-2181")
```
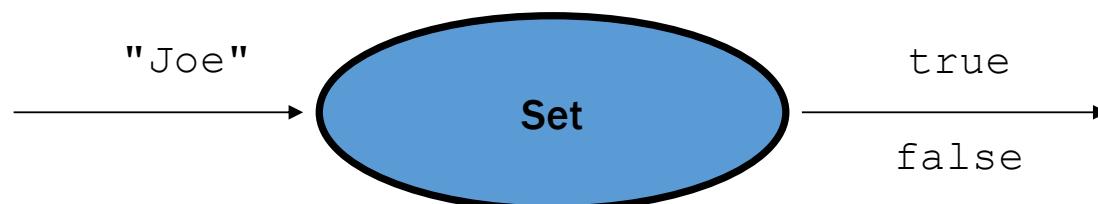
- Later, we can supply only the key and get back the re[...]

Allows us to ask: What is Joe's phone number?

Map

```
get("Joe")
```

```
"206-685-2181"
```

Map
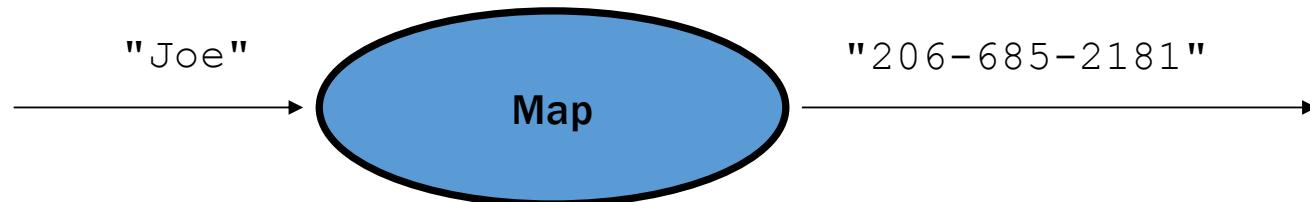
Maps vs. sets

- A set is like a map from elements to `boolean` values.
  - *Set: Is Joe found in the set? (true/false)*

"Joe" → **Set** → true / false

- *Map: What is Joe's phone number?*

"Joe" → **Map** → "206-685-2181"

keySet and values

- `keySet` method returns a `Set` of all keys in the map
  - can loop over the keys in a foreach loop
  - can get each key's associated value by calling `get` on the map

```
Map<String, Integer> ages = new TreeMap<String, Integer>();
ages.put("Joe", 19);
ages.put("Geneva", 2);   // ages.keySet() returns Set<String>
ages.put("Vicki", 57);
for (String name : ages.keySet()) {        // Geneva -> 2
    int age = ages.get(name);              // Joe -> 19
    System.out.println(name + " -> " + age);  // Vicki -> 57
}
```

- `values` method returns a collection of all values in the map
  - can loop over the values in a foreach loop
  - no easy way to get from a value to its associated key(s)

Priority queue ADT

- **priority queue**: a collection of ordered elements that provides fast access to the minimum (or maximum) element
- usually implemented using a tree structure called a *heap*

- priority queue operations:
  - `add`   adds in order;   O(log *N*) worst
  - `peek`   returns **minimum** value;   O(1)   always
  - `remove` removes/returns **minimum** value;   O(log *N*) worst
  - `isEmpty,`
    `clear,`
    `size,`
    `iterator`   O(1)   always

`public class PriorityQueue<`**`E`**`> implements Queue<`**`E`**`>`

Method/Constructor	Description	Runtime
`PriorityQueue<`**`E`**`>()`	constructs new empty queue	O(1)
`add(`**`E`** `value)`	adds value in sorted order	O(log N)
`clear()`	removes all elements	O(1)
`iterator()`	returns iterator over elements	O(1)
`peek()`	returns minimum element	O(1)
`remove()`	removes/returns min element	O(log N)

`Queue<String> pq = new `**`PriorityQueue`**`<String>();`

`pq.add("Stuart");`

`pq.add("Marty");`

`...`

Priority queue ordering

- For a priority queue to work, elements must have an ordering
  - in Java, this means implementing the `Comparable` interface

- Reminder:

```
public class Foo implements Comparable<Foo> {

    ...

    public int compareTo(Foo other) {
        // Return positive, zero, or negative number
    }

}
```

Part III – Design Patterns - Object Oriented Programming in real life

Design patterns

- **Creational Patterns**

- **Structural Patterns**

- **Behavioral Patterns**

Patterns

Creational patterns

- **Factory method**
- **Abstract factory**
- **Builder**
- **Prototype**
- **Singleton (8)**
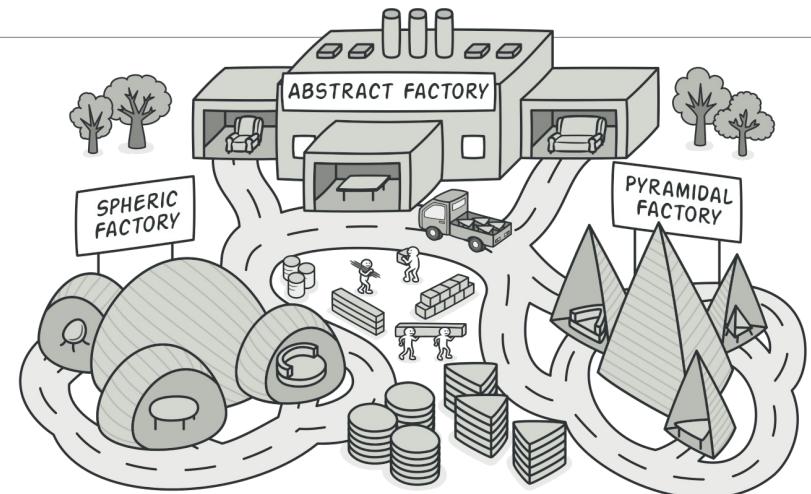
Structural patterns

- **Adapter**
- **Bridge (12)**
- **Composite**
- **Decorator**
- **Facade**
- **Flyweight (12)**
- **Proxy**

Creational patterns:
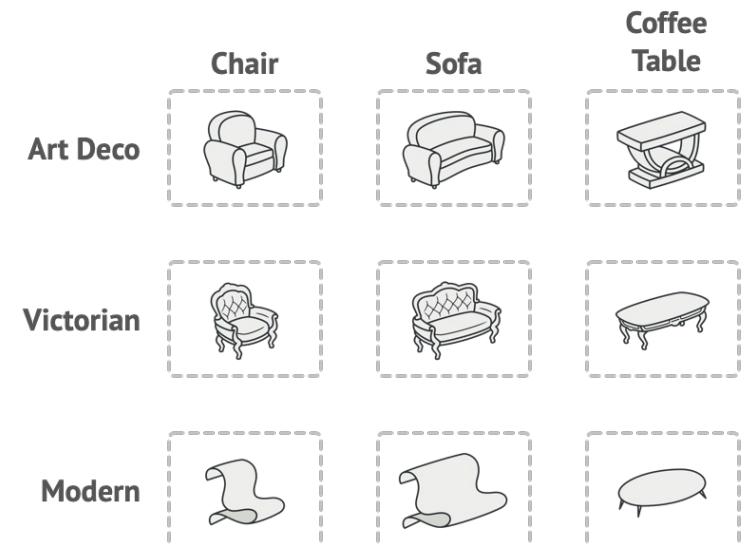
- **Chain of responsibility** (12)
- Command
- **Iterator**
- Mediator
- Memento
- Observer
- **State**
- Strategy
- Template method
- Visitor

FEI STU
316

Creational patterns: Abstract factory

Problem
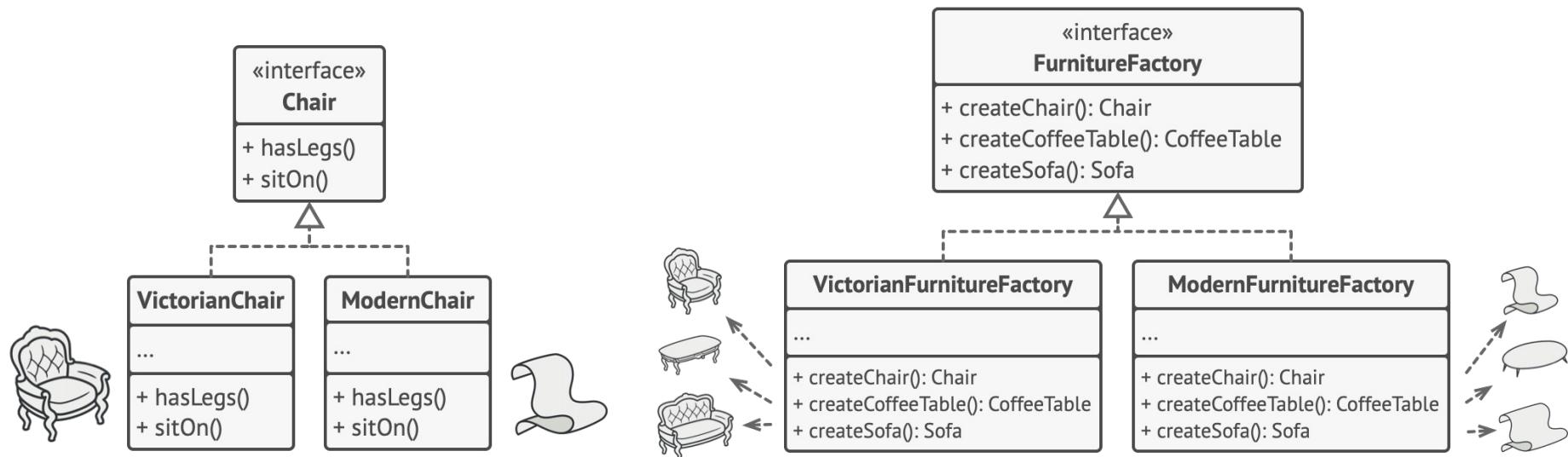
- Several variants of object of the same type

- Need a way to create individual furniture objects so that they match other objects of the same family.

- Do not change code when new type is added

Creational patterns: Abstract factory (continued)
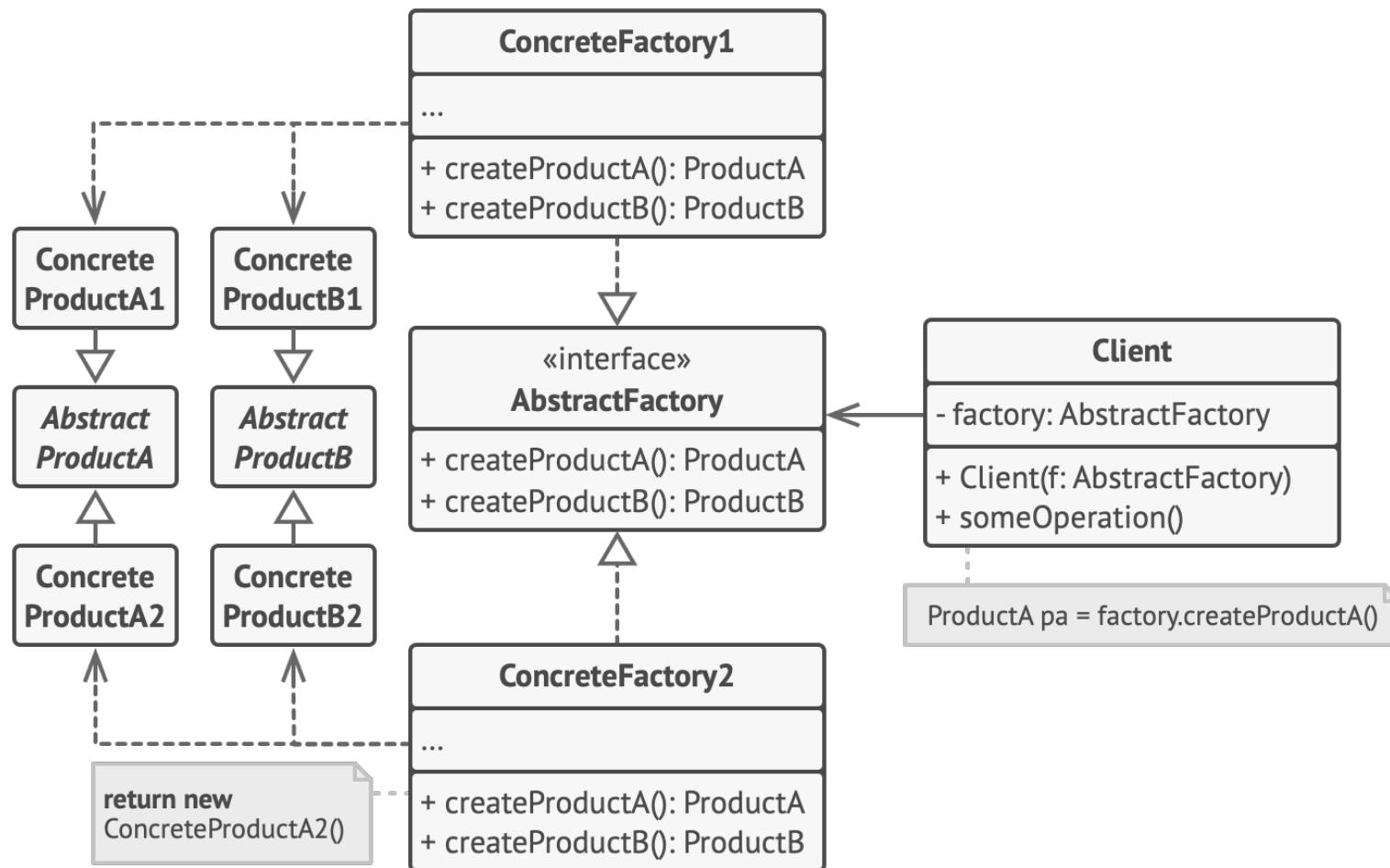
Solution

- Abstract Factory pattern suggests is to explicitly declare interfaces for each distinct product of the product family (e.g., chair, sofa or coffee table)

- Declare the *Abstract Factory*—an interface with a list of creation methods for all products that

  are part of the product family (for example，createChair, createSofa and createCoffeeTable). These methods must return **abstract** product types represented by the interfaces we extracted previously: Chair, Sofa, CoffeeTable and so on
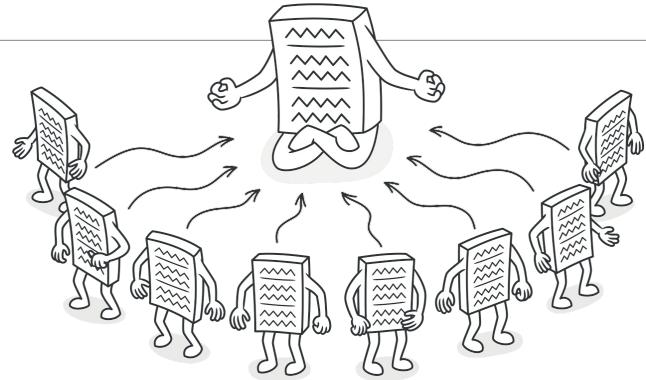
Creational patterns: Abstract factory (continued)

Usually, the application creates a concrete factory object at the initialization stage. Just before that, the app must select the factory type depending on the configuration or the environment settings.

Problem

- **Ensure that a class has just a single instance**. Why would anyone want to control how many instances a class has? The most common reason for this is to control access to some shared resource—for example, a database or a file.

- **Provide a global access point to that instance**. Remember those global variables that you (all right, me) used to store some essential objects? While they're very handy, they're also very unsafe since any code can potentially overwrite the contents of those variables and crash the app.

Note: This behaviour is impossible to implement with a regular constructor since a constructor call **must** always return a new object by design.

Creational patterns: Singleton (continued)

Solution

All implementations of the Singleton have these two steps in common:
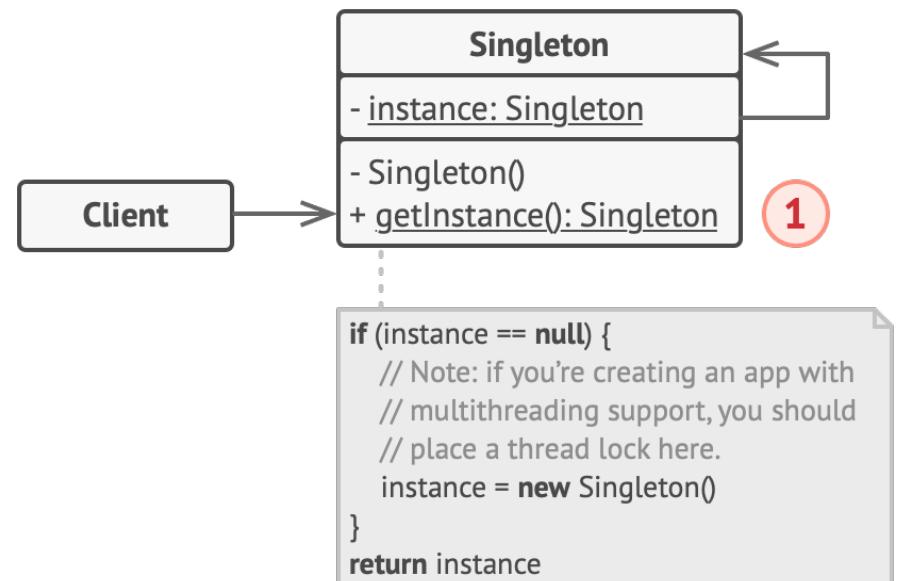
• Make the default constructor private, to prevent other objects from using the new operator with the Singleton class.

• Create a static creation method that acts as a constructor. Under the hood, this method calls the private constructor to create an object and saves it in a static field. All following calls to this method return the cached object.

If your code has access to the Singleton class, then it's able to call the Singleton's static method. So whenever that method is called, the same object is always returned.

Structure

The **Singleton** class declares the static method getInstance that returns the same instance of its own class.

The Singleton's constructor should be hidden from the client code. Calling the getInstance method should be the only way of getting the Singleton object.

Singleton
- instance: Singleton
- Singleton()   + getInstance(): Singleton

Client

(1)

```
if (instance == null) {
    // Note: if you're creating an app with
    // multithreading support, you should
    // place a thread lock here.
    instance = new Singleton()
}
return instance
```
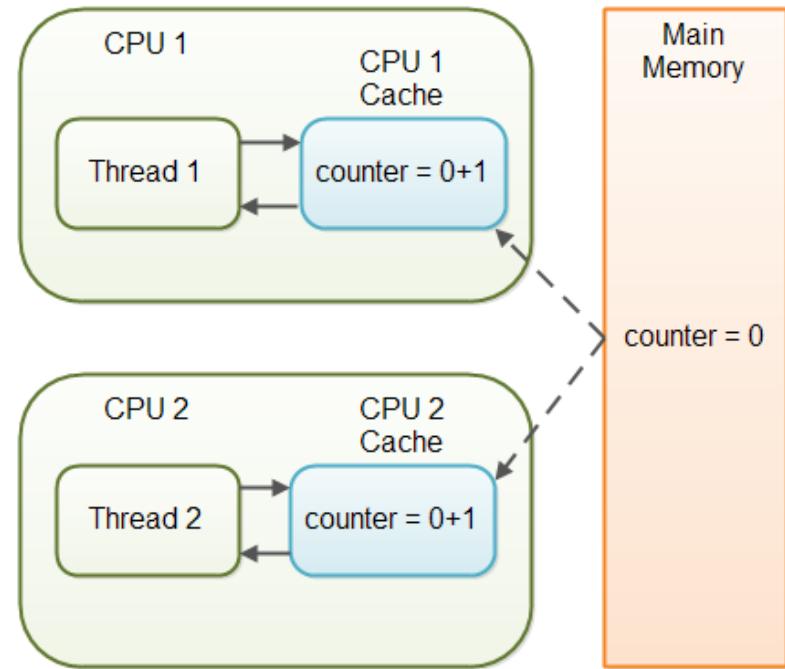
Creational patterns: Singleton (continued)
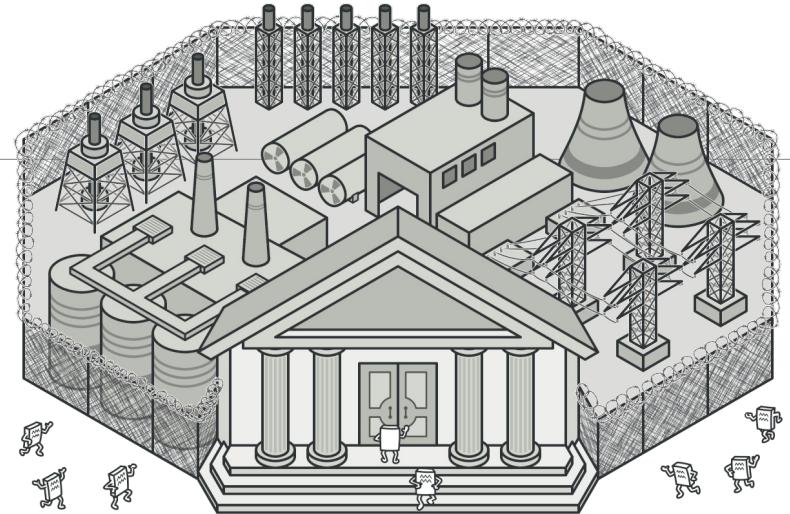
Java language notes:

Volatile keyword is **used to modify the value of a variable by different threads**. It is also used to make classes thread safe. It means that multiple threads can use a method and instance of the classes at the same time without any problem. The volatile keyword can be used either with primitive type or objects.

A *Java synchronized block* marks a method or a block of code as *synchronized*. A synchronized block in Java can only be executed a single thread at a time (depending on how you use it). Java synchronized blocks can thus be used to avoid race conditions.
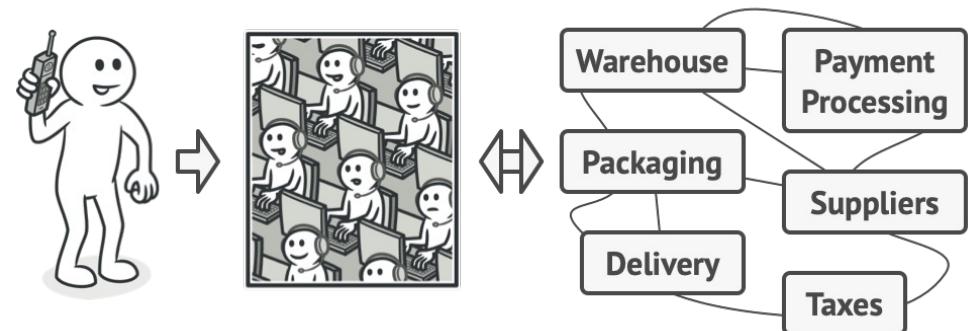
Structural patterns: Facade

Problem

- You must make your code work with a broad set of objects that belong to a sophisticated library or framework. Ordinarily, you'd need to initialize all of those objects, keep track of dependencies, execute methods in the correct order, and so on.

- As a result, the business logic of your classes would become tightly coupled to the implementation details of 3rd-party classes, making it hard to comprehend and maintain.
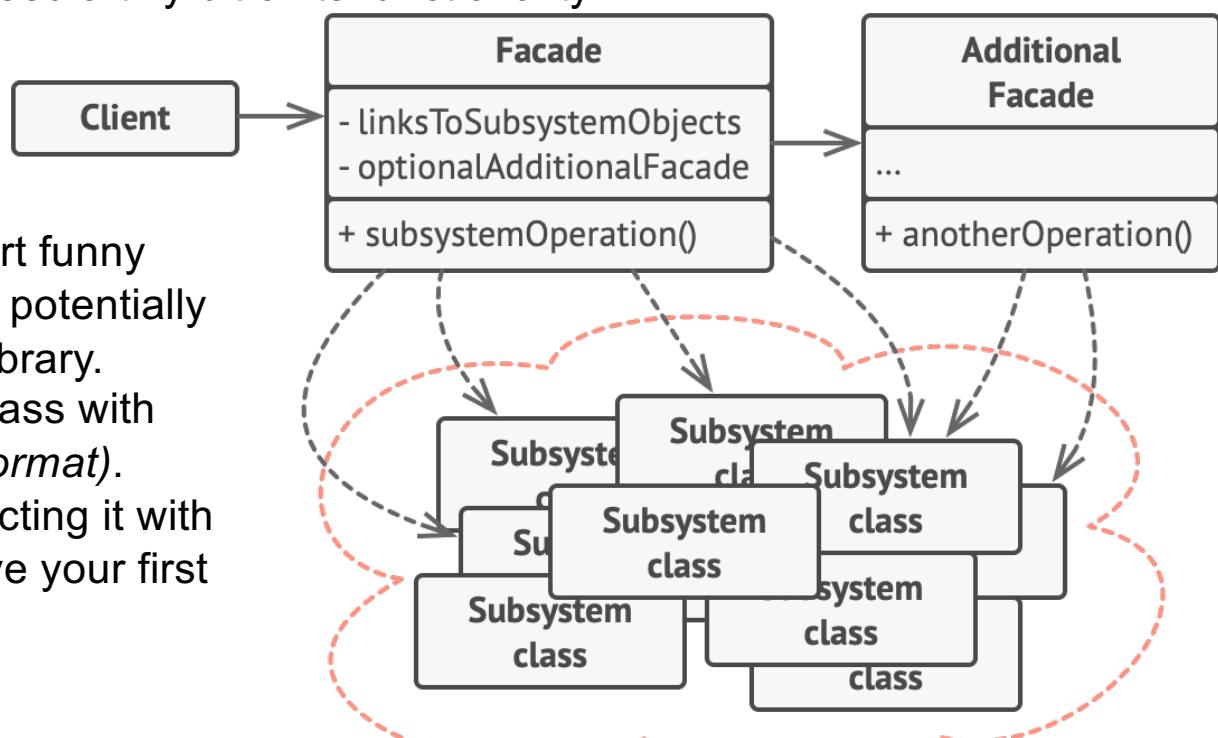
Structural patterns: Facade (continued)

Solution

A facade is a class that provides a simple interface to a complex subsystem which contains lots of moving parts. A facade might provide limited functionality in comparison to working with the subsystem directly. However, it includes only those features that clients really care about.

Having a facade is handy when you need to integrate your app with a sophisticated library that has dozens of features, but you just need a tiny bit of its functionality.
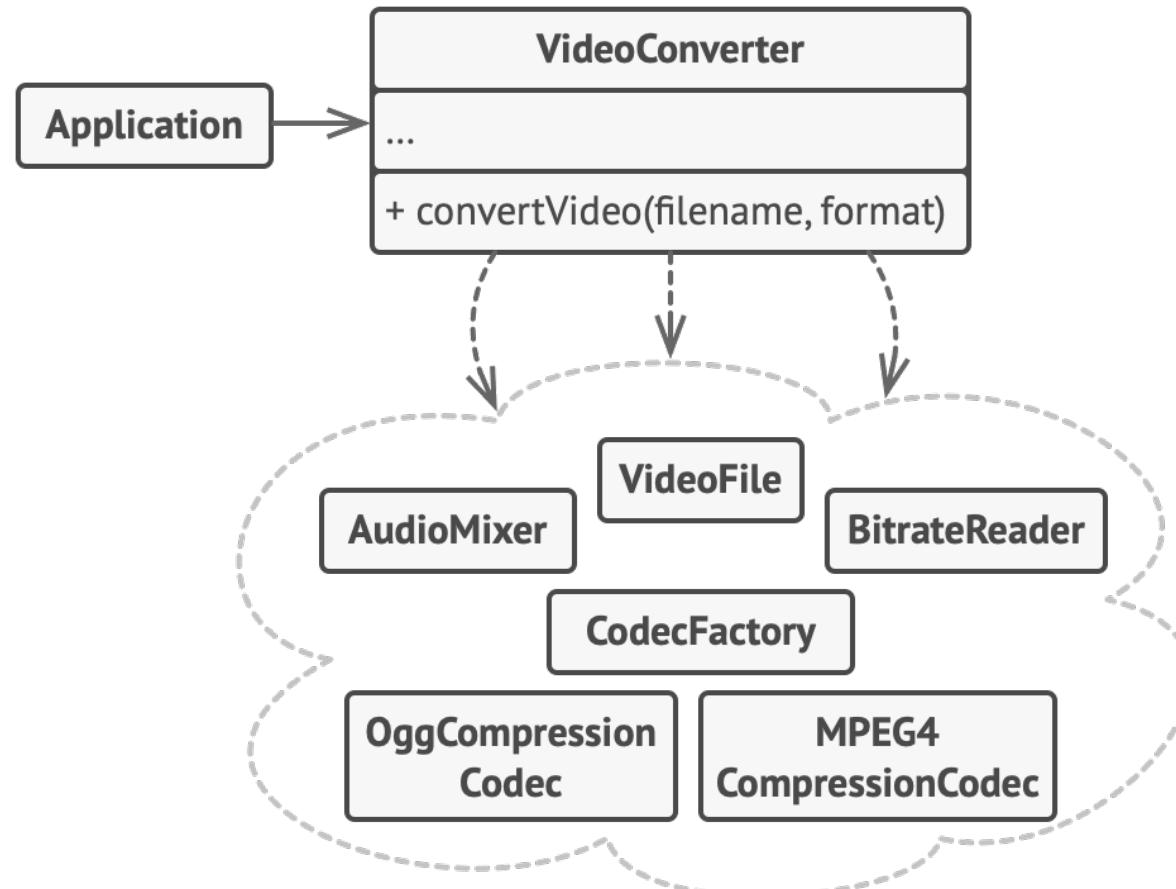
For instance, an app that uploads short funny videos with cats to social media could potentially use a professional video conversion library. However, all that it really needs is a class with the single method *encode(filename, format)*. After creating such a class and connecting it with the video conversion library, you'll have your first facade.

Facade
- linksToSubsystemObjects - optionalAdditionalFacade
+ subsystemOperation()

Additional Facade
...
+ anotherOperation()

Client → Facade → Additional Facade

Subsystem class (multiple)

Example
Complex video processing

Behavioral patterns: State
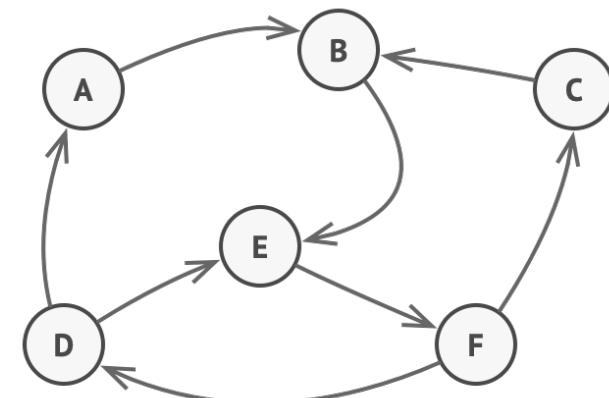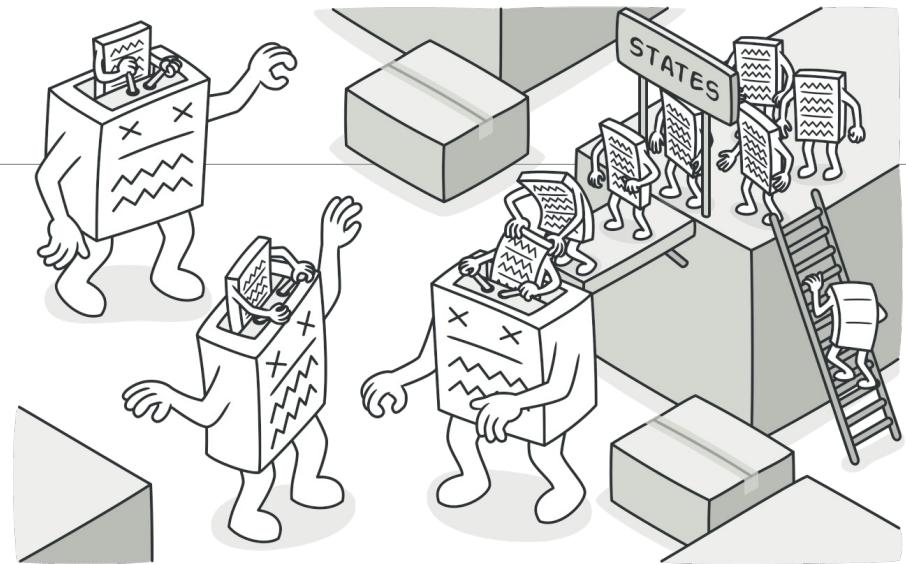
Problem

At any given moment, there's a *finite* number of *states* which a program can be in. Within any unique state, the program behaves differently, and the program can be switched from one state to another instantaneously. However, depending on a current state, the program may or may not switch to certain other states. These switching rules, called *transitions*, are also finite and predetermined.

You can also apply this approach to objects. Imagine that we have a Document class. A document can be in one of three states: Draft, Moderation and Published. The publish method of the document works a little bit differently in each state:

- In Draft, it moves the document to moderation.
- In Moderation, it makes the document public, but only if the current user is an administrator.
- In Published, it doesn't do anything at all.
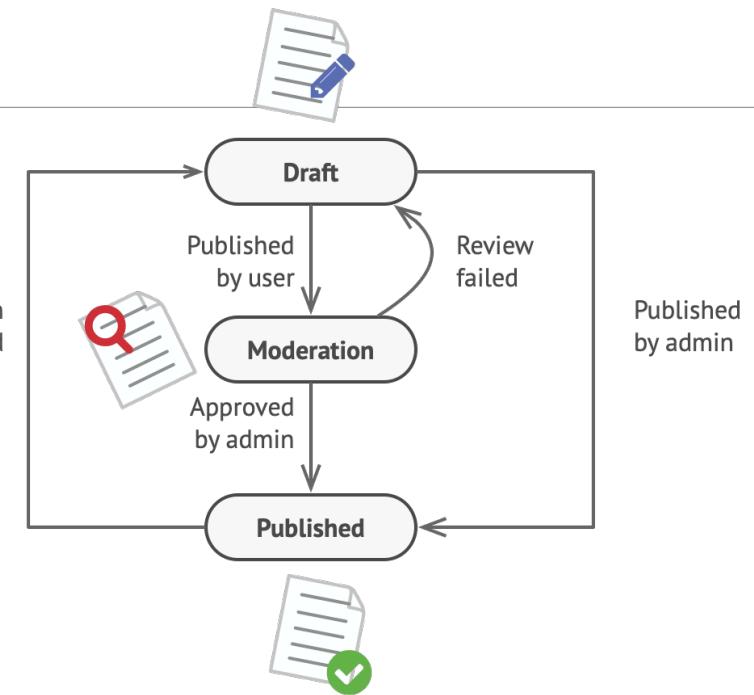
Behavioral patterns: State (continued)

The biggest weakness of a state machine based on conditionals reveals itself once we start adding more and more states and state-dependent behaviors to the *Document* class. Most methods will contain monstrous conditionals that pick the proper behavior of a method according to the current state. Code maintenance becomes complex because any change to the transition logic may require changing state conditionals in every method.

The problem tends to get bigger as a project evolves. It's quite difficult to predict all possible states and transitions at the design stage. Hence, a lean state machine built with a limited set of conditionals can grow into a bloated mess over time.
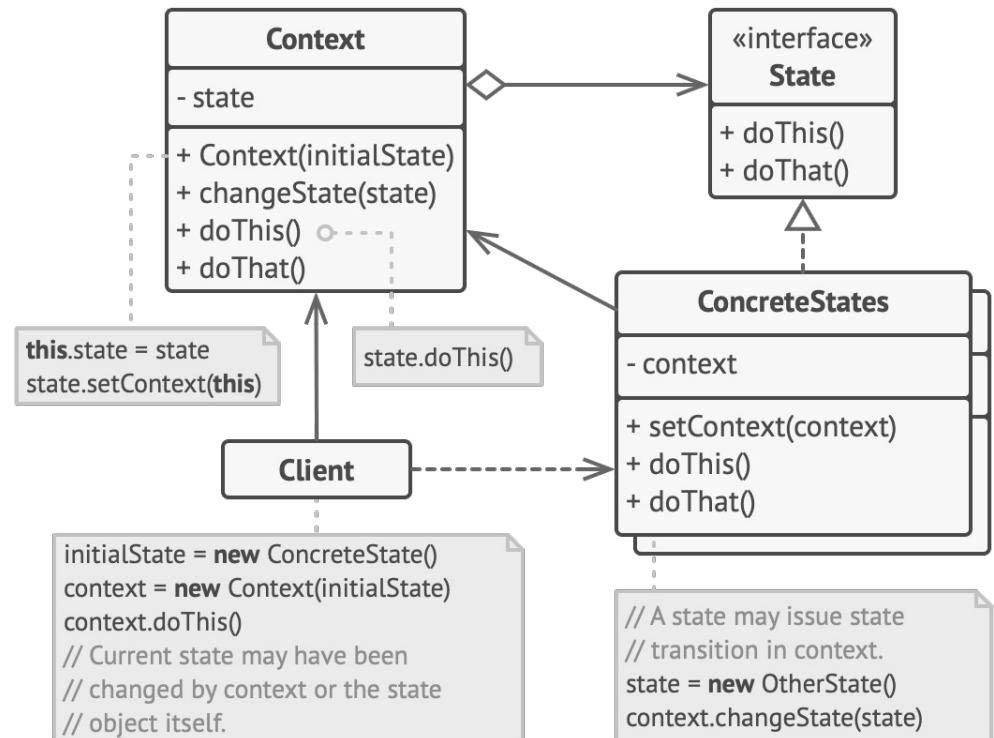
Solution

- The State pattern suggests that you create new classes for all possible states of an object and extract all state-specific behaviors into these classes.

- Instead of implementing all behaviors on its own, the original object, called *context*, stores a reference to one of the state objects that represents its current state, and delegates all the state-related work to that object.
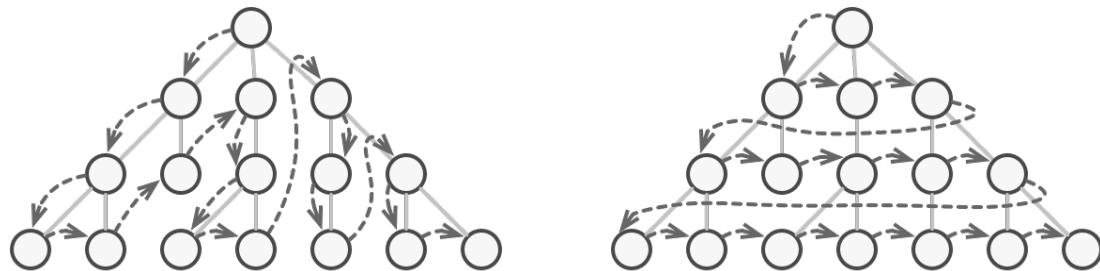
Behavioral patterns: State (continued)

- The buttons and switches in your smartphone behave differently depending on the current state of the device:

- When the phone is unlocked, pressing buttons leads to executing various functions.

- When the phone is locked, pressing any button leads to the unlock screen.

- When the phone's charge is low, pressing any button shows the charging screen.
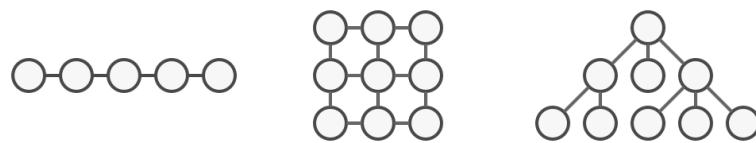
Problem

- Collections are one of the most used data types in programming. Nonetheless, a collection is just a container for a group of objects.

- Most collections store their elements in simple lists. However, some of them are based on stacks, trees, graphs and other complex data structures.

- But no matter how a collection is structured, it must provide some way of accessing its elements so that other code can use these elements. There should be a way to go through each element of the collection without accessing the same elements over and over.
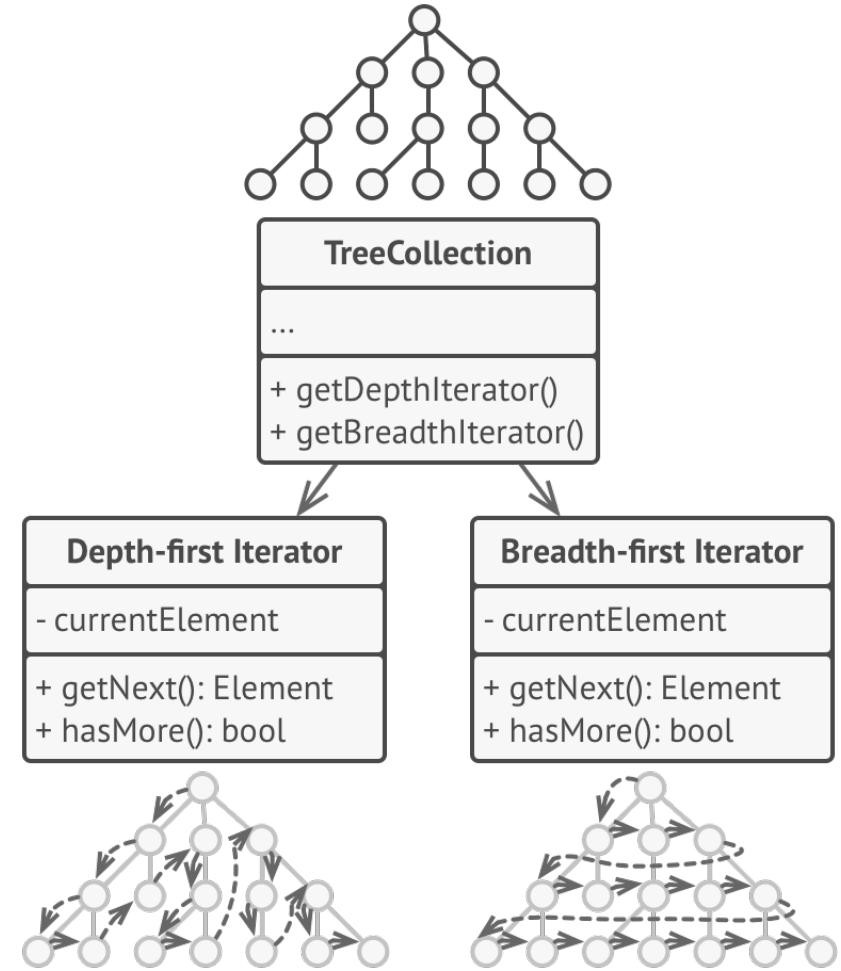
Behavioral patterns: Iterator (continued)
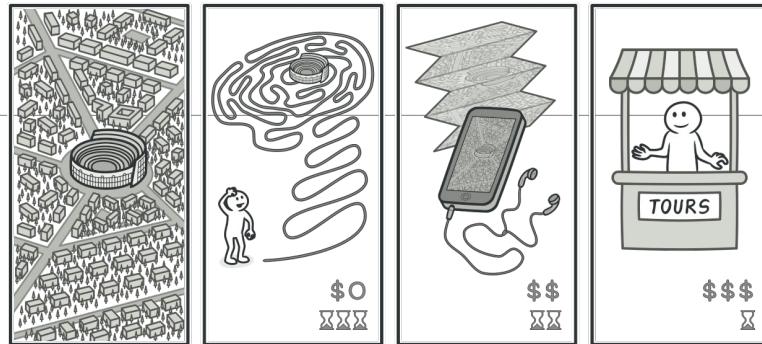
Solution

The main idea of the Iterator pattern is to extract the traversal behavior of a collection into a separate object called an *iterator*.

- In addition to implementing the algorithm itself, an **iterator object encapsulates all of the traversal details**, such as the current position and how many elements are left till the end. Because of this, several iterators can go through the same collection at the same time, independently of each other.

- Usually, iterators provide one primary method for fetching elements of the collection. The client can keep running this method until it doesn't return anything, which means that the iterator has traversed all of the elements.

- All iterators must implement the same interface. This makes the client code compatible with any collection type or any traversal algorithm as long as there's a proper iterator. If you need a special way to traverse a collection, you just create a new iterator class, without having to change the collection or the client.
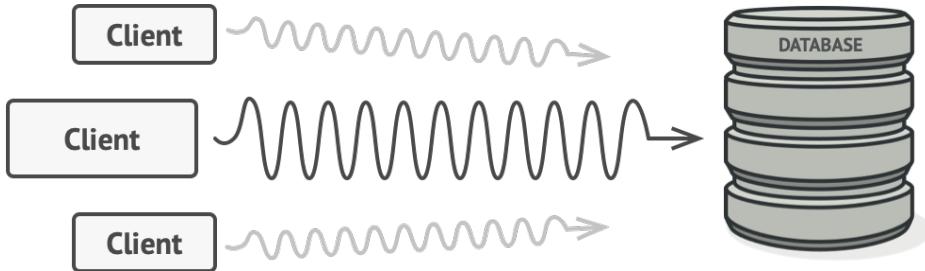
Behavioral patterns: Iterator (continued)

1. The **Iterator** interface declares the operations required for traversing a collection: fetching the next element, retrieving the current position, restarting iteration, etc.

2. **Concrete Iterators** implement specific algorithms for traversing a collection. The iterator object should track the traversal progress on its own - traverse the same collection independently of each other.

3. The **Collection** interface declares one or multiple methods for getting iterators compatible with the collection.

4. **Concrete Collections** return new instances of a particular concrete iterator class each time the client requests one. It's just that these details aren't crucial to the actual pattern.

5. The **Client** works with both collections and iterators via their interfaces. This way the client isn't coupled to concrete classes, allowing you to use various collections and iterators with the same client code.

6. Typically, clients don't create iterators on their own, but instead get them from collections. Yet, in certain cases, the client can create one directly; for example, when the client defines its own special iterator.

Structural patterns: Proxy

Proxy is a structural design pattern that lets you provide a substitute or placeholder for another object. A proxy controls access to the original object, allowing you to perform something either before or after the request gets through to the original object.

Problem

- You have a massive object that consumes a vast amount of system resources. You need it from time to time, but not always.

- You could implement lazy initialization: create this object only when it's actually needed. All of the object's clients would need to execute some deferred initialization code. Unfortunately, this would probably cause a lot of code duplication.

- You could **put this code directly into our object's class, but that isn't possible**. For instance, the class may be part of a closed 3rd-party library.

Solution

The Proxy pattern suggests that you create a new proxy class with the same interface as an original service object. Then you update your app so that it passes the proxy object to all of the original object's clients. Upon receiving a request from a client, the proxy creates a real service object and delegates all the work to it.

Benefit: If you need to execute something either before or after the primary logic of the class, the proxy lets you do this without changing that class. Since the proxy implements the same interface as the original class, it can be passed to any client that expects a real service object.

Example
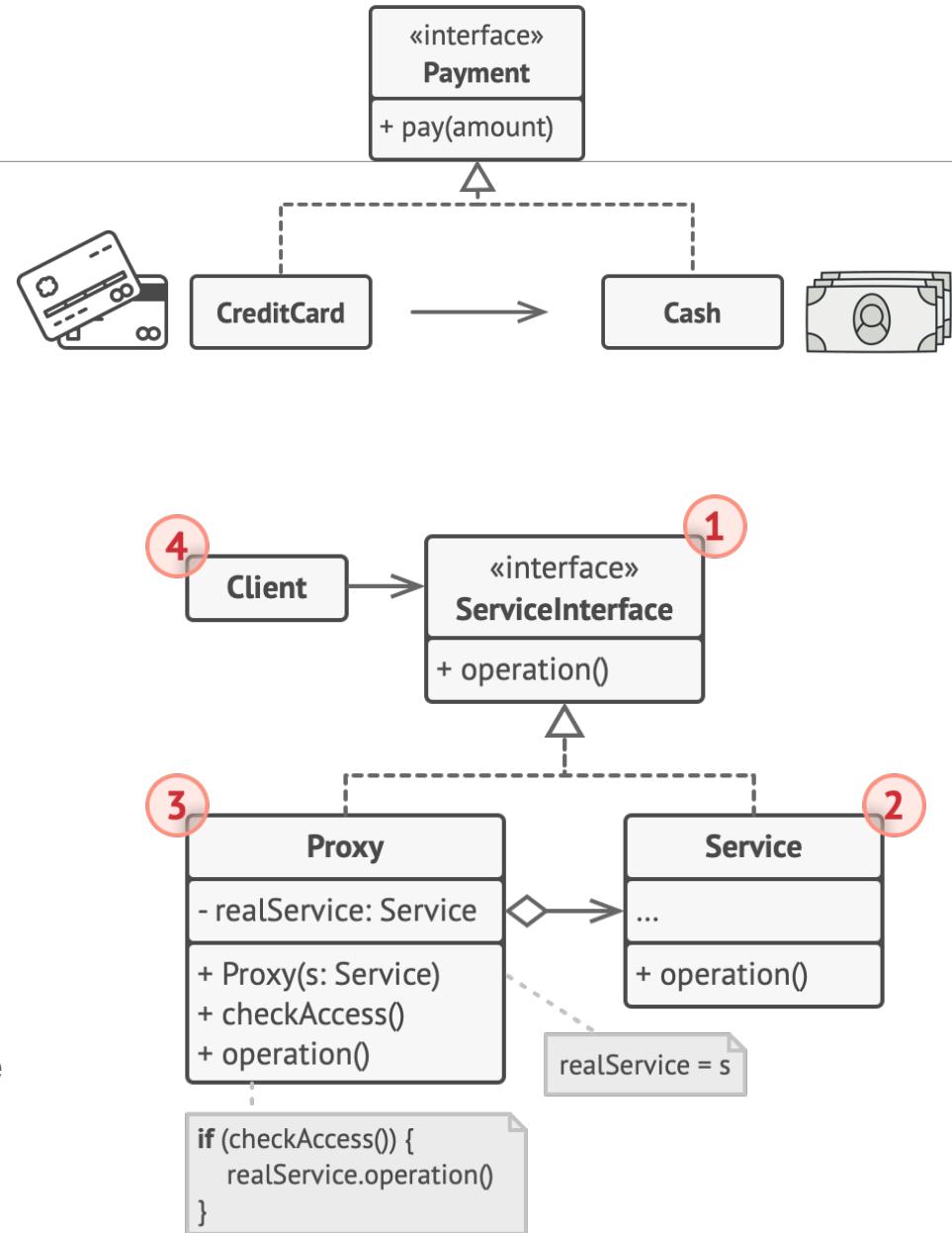
A credit card is a proxy for a bank account, which is a proxy for a bundle of cash. Both implement the same interface: they can be used for making a payment. A consumer feels great because there's no need to carry loads of cash around. A shop owner is also happy since the income from a transaction gets added electronically to the shop's bank account without the risk of losing the deposit or getting robbed on the way to the bank.

Structural patterns: Proxy (continued)

Structure

1. The **Service Interface** declares the interface of the Service. The proxy must follow this interface to be able to disguise itself as a service object.

2. The **Service** is a class that provides some useful business logic.

3. The **Proxy** class has a reference field that points to a service object. After the proxy finishes its processing (e.g., lazy initialization, logging, access control, caching, etc.), it passes the request to the service object.

4. Usually, proxies manage the full lifecycle of their service objects.

5. The **Client** should work with both services and proxies via the same interface. This way you can pass a proxy into any code that expects a service object.

«interface»
Payment

+ pay(amount)

CreditCard → Cash

Client → «interface» **ServiceInterface**
+ operation()

Proxy
- realService: Service
+ Proxy(s: Service)
+ checkAccess()
+ operation()

Service
...
+ operation()

realService = s

if (checkAccess()) {
    realService.operation()
}
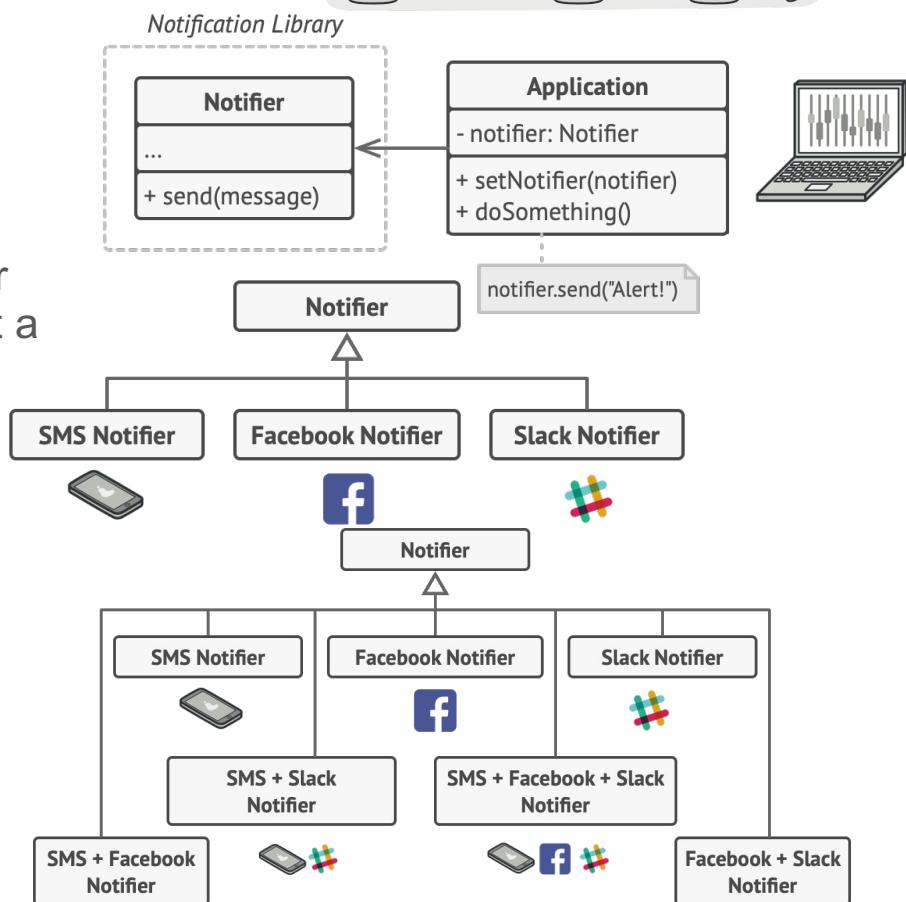
Structural patterns: Decorator

Decorator is a structural design pattern that lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.

Problem

- Imagine that you're working on a notification library which lets other programs notify their users about important events.

- The initial version of the library was based on the *Notifier* class that had only a few fields, a constructor and a single send method. The method could accept a message argument from a client and send the message to a list of emails that were passed to the notifier via its constructor.

- Then need more types of notifiers

- Need a combination – not only statically via inheritance, but also dynamically during run time

Structural patterns: Decorator (continued)
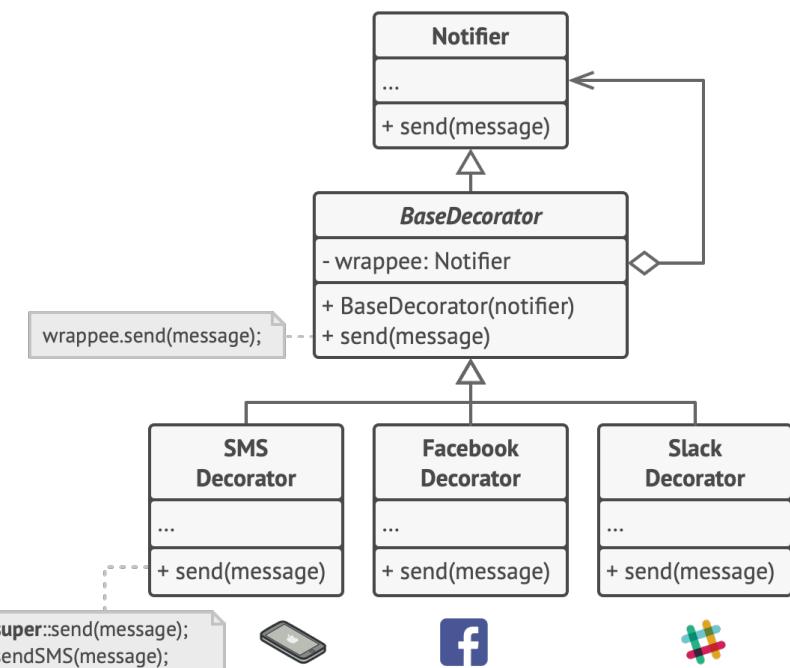
Solution

To overcome these caveats is by using Aggregation or Composition instead of Inheritance. One object has a reference to another and delegates it some work, whereas with inheritance, the object itself is able to do that work, inheriting the behavior from its superclass.

We substitute the linked "helper" object with another, changing the behavior of the container at runtime. An object can use the behavior of various classes, having references to multiple objects and delegating them all kinds of work. Aggregation/composition is the key principle behind many design patterns, including Decorator.

A *wrapper* is an object that can be linked with some *target* object. The wrapper contains the same set of methods as the target and delegates to it all requests it receives. However, the wrapper may alter the result by doing something either before or after it passes the request to the target.

Structural patterns: Decorator (continued)

Structure
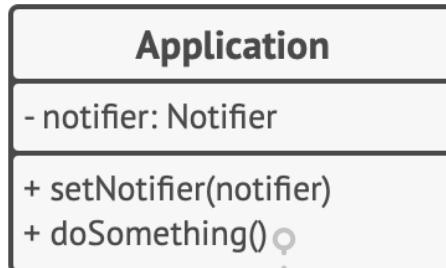
- The last decorator in the stack would be the object that the client works with. Since all decorators implement the same interface as the base notifier, the rest of the client code won't care whether it works with the "pure" notifier object or the decorated one.

- We could apply the same approach to other behaviours such as formatting messages or composing the recipient list. The client can decorate the object with any custom decorators, as long as they follow the same interface as the others.

Real-life

Wearing clothes is an example of using decorators. When you're cold, you wrap yourself in a sweater. If you're still cold with a sweater, you can wear a jacket on top. If it's raining, you can put on a raincoat. All of these garments "extend" your basic behavior but aren't part of you, and you can easily take off any piece of clothing whenever you don't need it.

```
stack = new Notifier()
if (facebookEnabled)
    stack = new FacebookDecorator(stack)
if (slackEnabled)
    stack = new SlackDecorator(stack)

app.setNotifier(stack)
```

Application

- notifier: Notifier
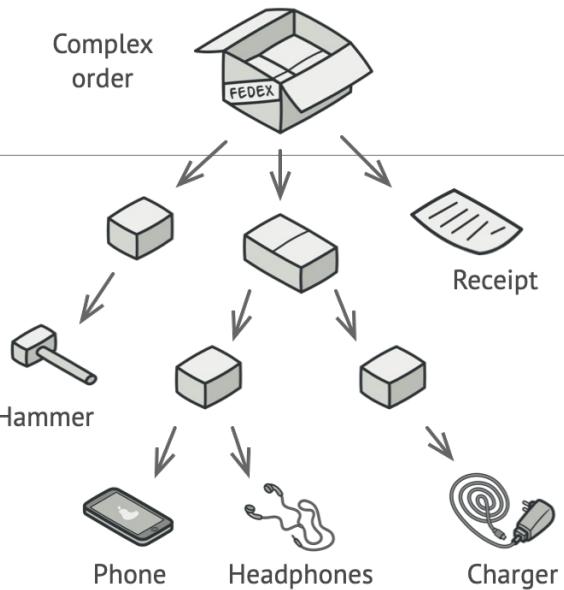
+ setNotifier(notifier)
+ doSomething()

```
notifier.send("Alert!")
// Email → Facebook → Slack
```

Structural patterns: Composite

Composite is a structural design pattern that lets you compose objects into tree structures and then work with these structures as if they were individual objects.

Problem

- Using the Composite pattern makes sense only when the core model of your app can be represented as a tree.
- For example, imagine that you have two types of objects: Products and Boxes. A Box can contain several Products as well as a number of smaller Boxes. These little Boxes can also hold some Products or even smaller Boxes, and so on.
- Say you decide to create an ordering system that uses these classes. Orders could contain simple products without any wrapping, as well as boxes stuffed with products...and other boxes. How would you determine the total price of such an order?

Complication

- Direct approach: unwrap all the boxes, go over all the products and then calculate the total. That would be doable in the real world; but in a program, it's not as simple as running a loop. You have to know the classes of Products and Boxes you're going through, the nesting level of the boxes and other nasty details beforehand.

Solution

The Composite pattern suggests that you work with Products and Boxes through a common interface which declares a method for calculating the total price.

Approach:

- For a product, it'd simply return the product's price.
- For a box, it'd go over each item the box contains, ask its price and then return a total for this box. If one of these items were a smaller box, that box would also start going over its contents and so on, until the prices of all inner components were calculated.
- A box could even add some extra cost to the final price, such as packaging cost.
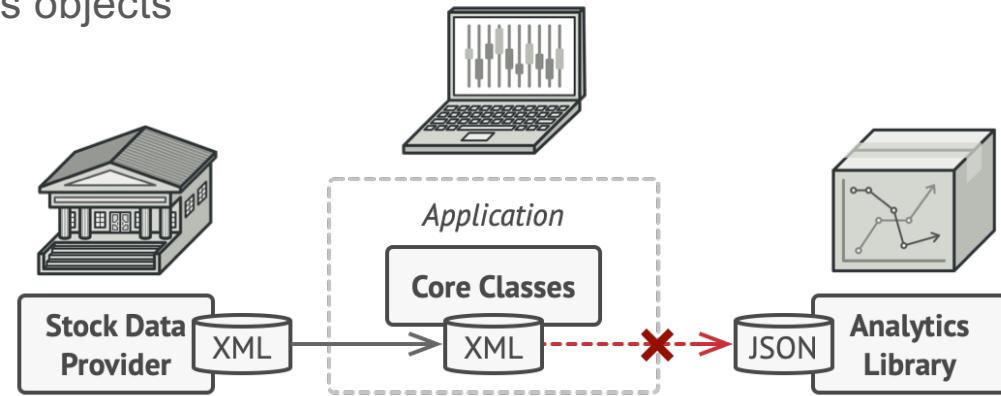
The greatest benefit of this approach is that you don't need to care about the concrete classes of objects that compose the tree. You don't need to know whether an object is a simple product or a sophisticated box. You can treat them all the same via the common interface. When you call a method, the objects themselves pass the request down the tree.

Real world alternative – army units:

Structural patterns: Adapter

Adapter is a structural design pattern that allows objects with incompatible interfaces to collaborate.

Problem

- Creating a stock market monitoring app. The app downloads the stock data from multiple sources in XML format and then displays nice-looking charts and diagrams for the user.

- At some point, you decide to improve the app by integrating a smart 3rd-party analytics library. But there's a catch: the analytics library only works with data in JSON format.
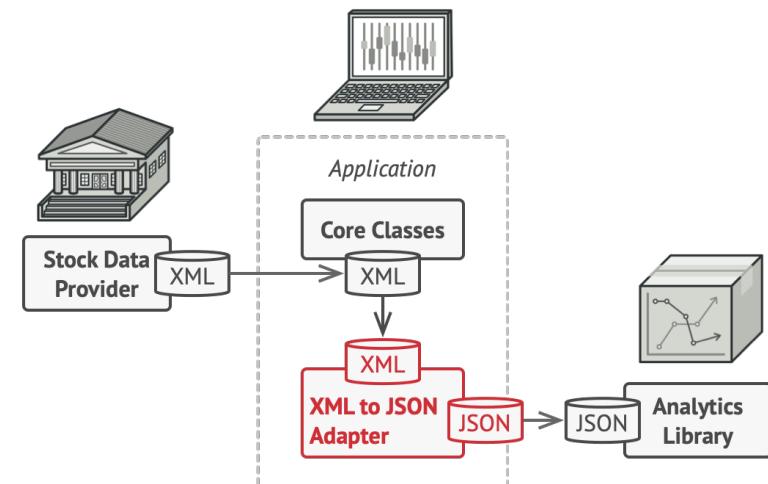
Typical complications:

- You could change the library to work with XML. However, this might break some existing code that relies on the library. And worse, you might not have access to the library's source code in the first place, making this approach impossible.

Solution

• You can create an *adapter*. This is a special object that converts the interface of one object so that another object can understand it.

• An adapter wraps one of the objects to hide the complexity of conversion happening behind the scenes. The wrapped object isn't even aware of the adapter. For example, you can wrap an object that operates in meters and kilometers with an adapter that converts all of the data to imperial units such as feet and miles.

• Adapters can not only convert data into various formats but can also help objects with different interfaces collaborate. Here's how it works:

1. The adapter gets an interface, compatible with one of the existing objects.

2. Using this interface, the existing object can safely call the adapter's methods.

3. Upon receiving a call, the adapter passes the request to the second object, but in a format and order that the second object expects.

• Sometimes it's even possible to create a two-way adapter that can convert the calls in both directions.
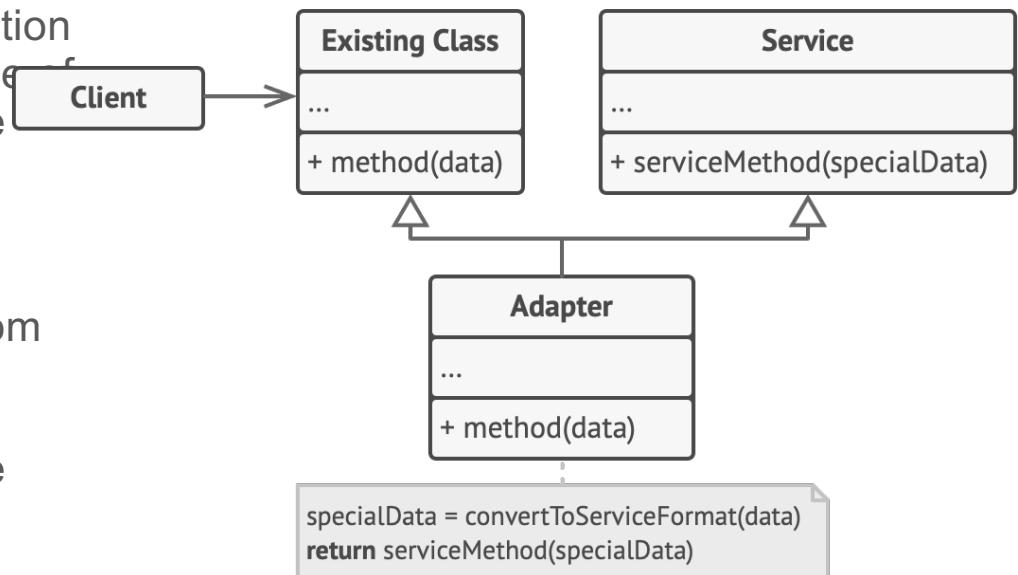
Structural patterns: Adapter (continued)

A *wrapper* is an object that can be linked with some *target* object. The wrapper contains the same set of methods as the target and delegates to it all requests it receives. However, the wrapper may alter the result by doing something either before or after it passes the request to the target.

Client → «interface» Client Interface
+ method(data)

Adapter
- adaptee: Service
+ method(data)

Service
...
+ serviceMethod(specialData)

specialData = convertToServiceFormat(data)
return adaptee.serviceMethod(specialData)

Structure

- This implementation uses the object composition principle: the adapter implements the interface of one object and wraps the other one. It can be implemented in all popular programming languages.

- Note: Another implementation can use inheritance: the adapter inherits interfaces from both objects at the same time. Note that this approach can only be implemented in programming languages that support multiple inheritance, such as C++.

Client → Existing Class
...
+ method(data)

Service
...
+ serviceMethod(specialData)

Adapter
...
+ method(data)

specialData = convertToServiceFormat(data)
return serviceMethod(specialData)
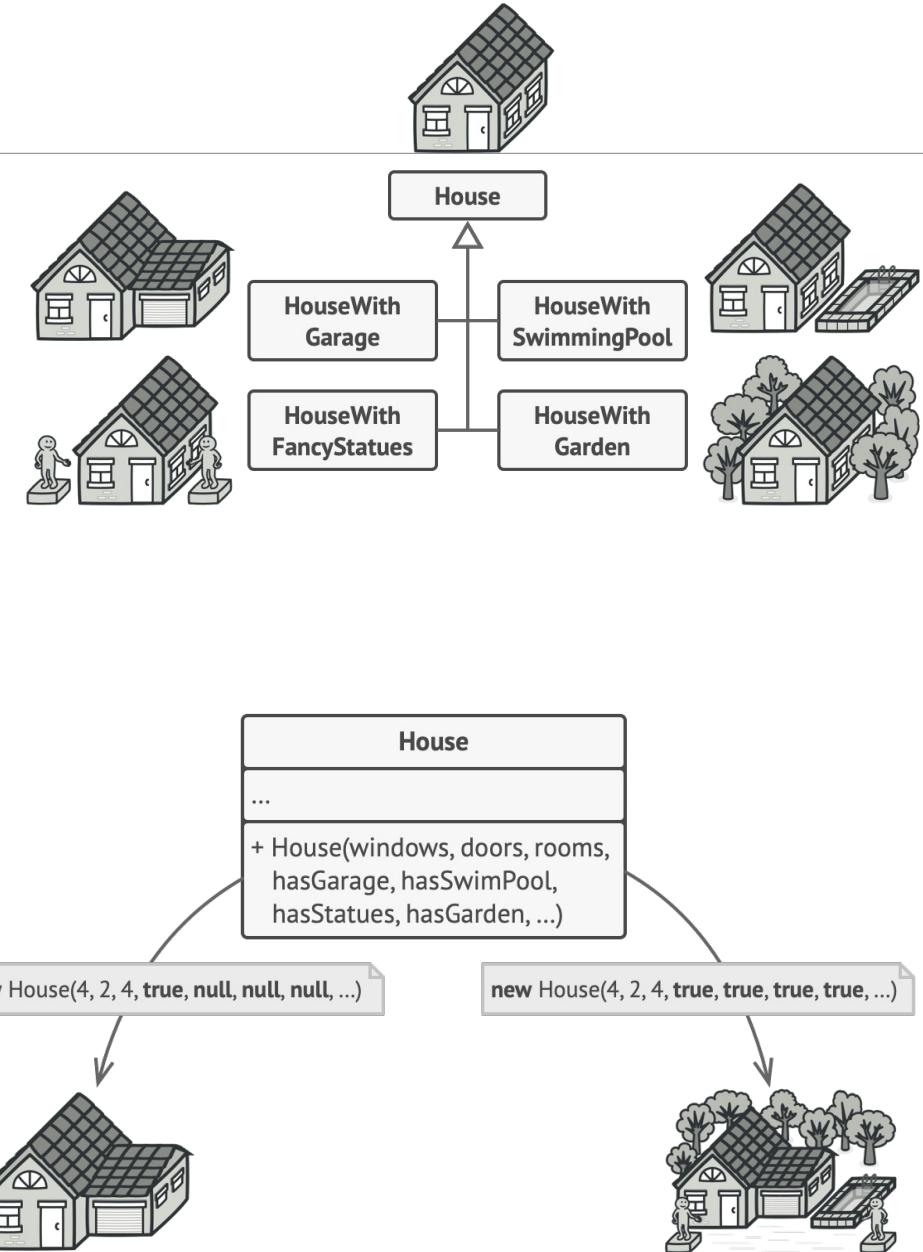
Creational patterns: Builder

Builder is a creational design pattern that lets you construct complex objects step by step. The pattern allows to produce different types and representations of an object using the same construction code.

Problem
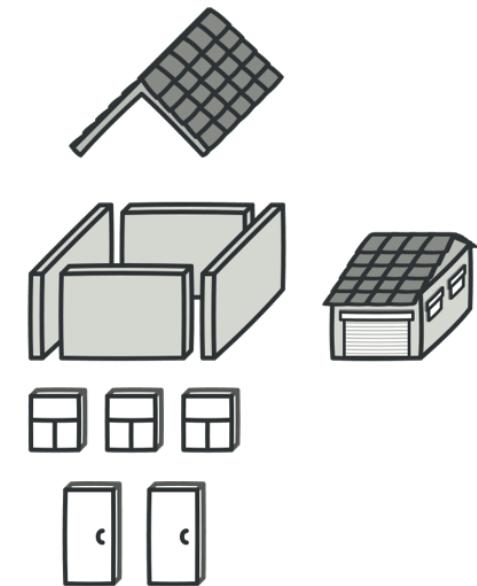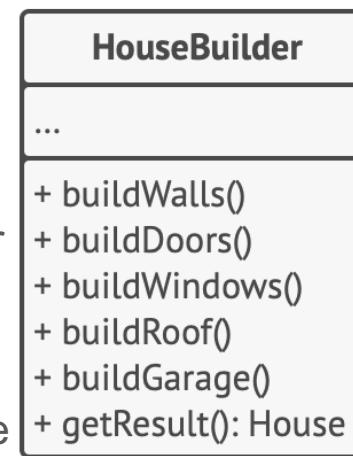
- A complex object that requires laborious, step-by-step initialization of many fields and nested objects.

- Such initialization code is usually buried inside a monstrous constructor with lots of parameters.

- The simplest solution is to extend the base House class and create a set of subclasses to cover all combinations of the parameters. But eventually you'll end up with a considerable number of subclasses. Any new parameter, such as a roof style, will require growing this hierarchy even more.

Solution

- The Builder pattern suggests that you extract the object construction code out of its own class and move it to separate objects called *builders*.

- You don't need to call all of the steps. You can call only those steps that are necessary for producing a particular configuration of an object.

- Some of the construction steps might require different implementation when you need to build various representations of the product. For example, walls of a cabin may be built of wood, but the castle walls must be built with stone.

- In this case, you can create several different builder classes that implement the same set of building steps, but in a different manner. Then you can use these builders in the construction process (i.e., an ordered set of calls to the building steps) to produce different kinds of objects.

- You can go further and extract a series of calls to the builder steps you use to construct a product into a separate class called ***director***. The director class defines the order in which to execute the building steps, while the builder provides the implementation for those steps.

HouseBuilder
...
+ buildWalls()
+ buildDoors()
+ buildWindows()
+ buildRoof()
+ buildGarage()
+ getResult(): House
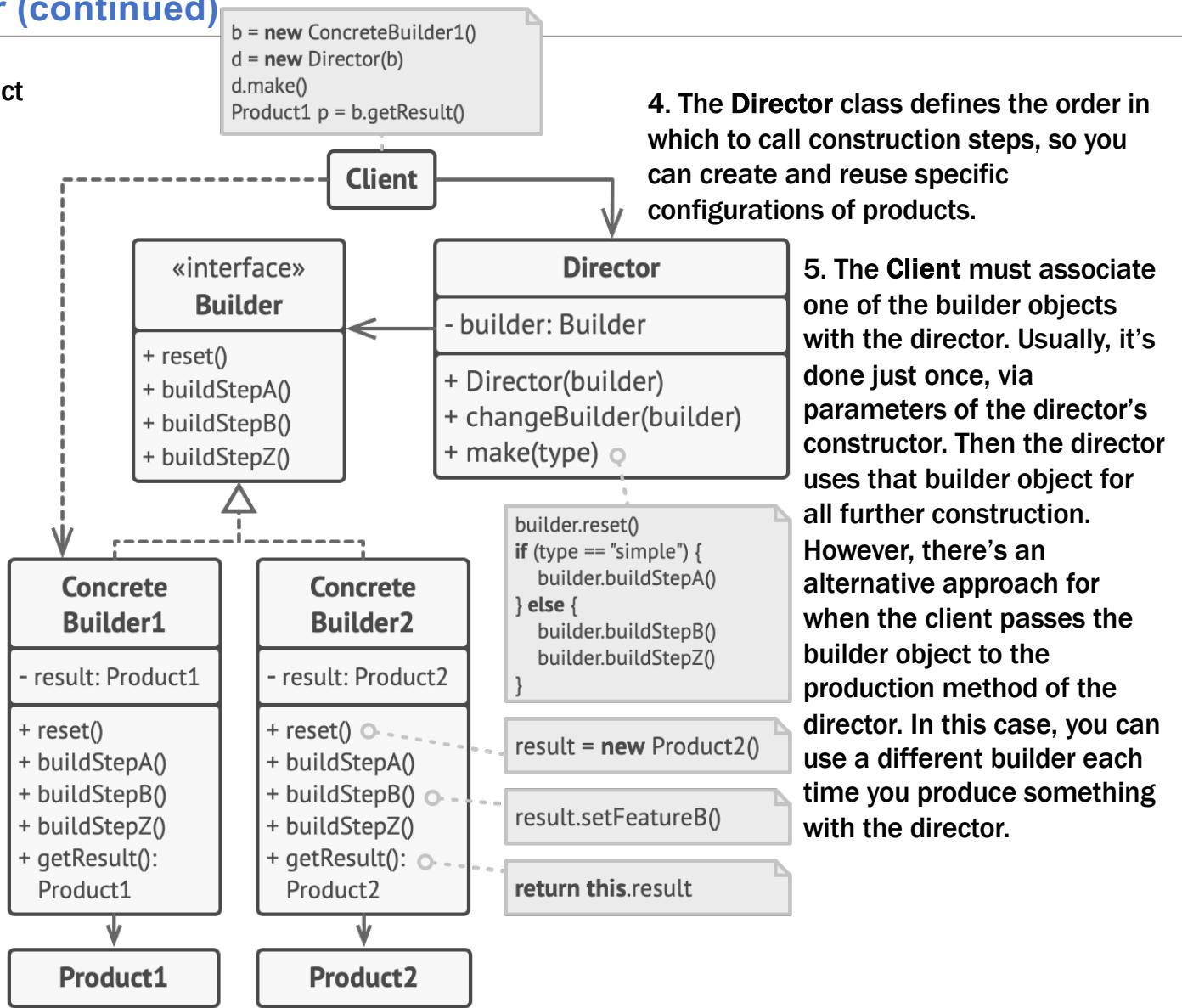
Creational patterns: Builder (continued)

```
b = new ConcreteBuilder1()
d = new Director(b)
d.make()
Product1 p = b.getResult()
```

1. The Builder interface declares product construction steps that are common to all types of builders.

2. Concrete Builders provide different implementations of the construction steps. Concrete builders may produce products that don't follow the common interface.

3. Products are resulting objects. Products constructed by different builders don't have to belong to the same class hierarchy or interface.
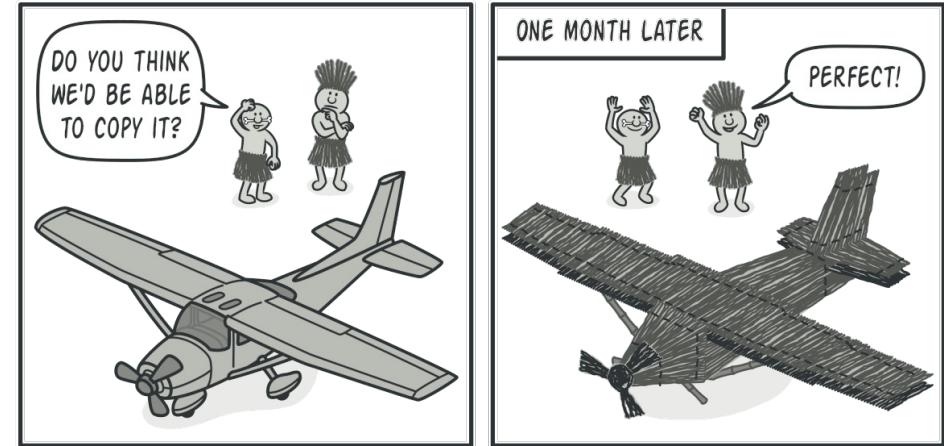
4. The Director class defines the order in which to call construction steps, so you can create and reuse specific configurations of products.

5. The Client must associate one of the builder objects with the director. Usually, it's done just once, via parameters of the director's constructor. Then the director uses that builder object for all further construction. However, there's an alternative approach for when the client passes the builder object to the production method of the director. In this case, you can use a different builder each time you produce something with the director.

Client

«interface»
Builder

+ reset()
+ buildStepA()
+ buildStepB()
+ buildStepZ()

Director

- builder: Builder

+ Director(builder)
+ changeBuilder(builder)
+ make(type)

```
builder.reset()
if (type == "simple") {
    builder.buildStepA()
} else {
    builder.buildStepB()
    builder.buildStepZ()
}
```

Concrete
Builder1

- result: Product1

+ reset()
+ buildStepA()
+ buildStepB()
+ buildStepZ()
+ getResult():
  Product1

Concrete
Builder2

- result: Product2

+ reset()
+ buildStepA()
+ buildStepB()
+ buildStepZ()
+ getResult():
  Product2

`result = new Product2()`

`result.setFeatureB()`

return this.result

Product1

Product2

Creational patterns: Prototype

Prototype is a creational design pattern that lets you copy existing objects without making your code dependent on their classes.
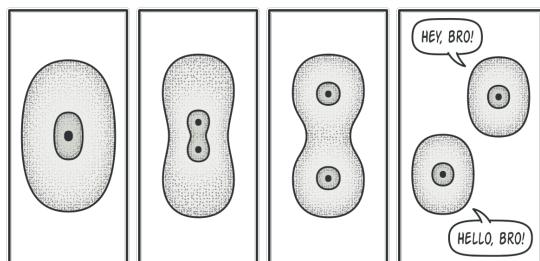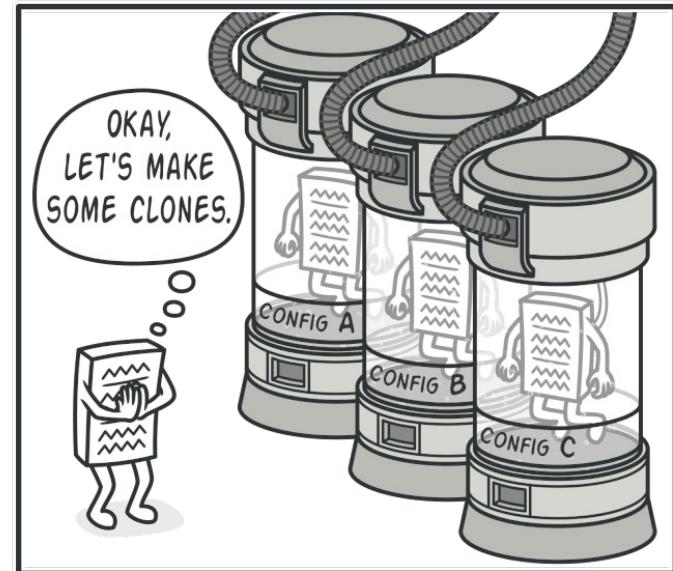
Problem

- You have an object, and you want to create an exact copy of it. How would you do it? First, you have to create a new object of the same class. Then you have to go through all the fields of the original object and copy their values over to the new object.

- BUT - not all objects can be copied that way because some of the object's fields may be private and not visible from outside of the object itself.

- There's one more problem with the direct approach. Since you have to know the object's class to create a duplicate, your code becomes dependent on that class. If the extra dependency doesn't scare you, there's another catch. Sometimes you only know the interface that the object follows, but not its concrete class, when, for example, a parameter in a method accepts any objects that follow some interface.

Creational patterns: Prototype (continued)

- The Prototype pattern delegates the cloning process to the actual objects that are being cloned. The pattern declares a common interface for all objects that support cloning. This interface lets you clone an object without coupling your code to the class of that object. Usually, such an interface contains just a single clone method.

- The implementation of the clone method is very similar in all classes. The method creates an object of the current class and carries over all of the field values of the old object into the new one. You can even copy private fields because most programming languages let objects access private fields of other objects that belong to the same class.

- An object that supports cloning is called a prototype. When your objects have dozens of fields and hundreds of possible configurations, cloning them might serve as an alternative to subclassing.
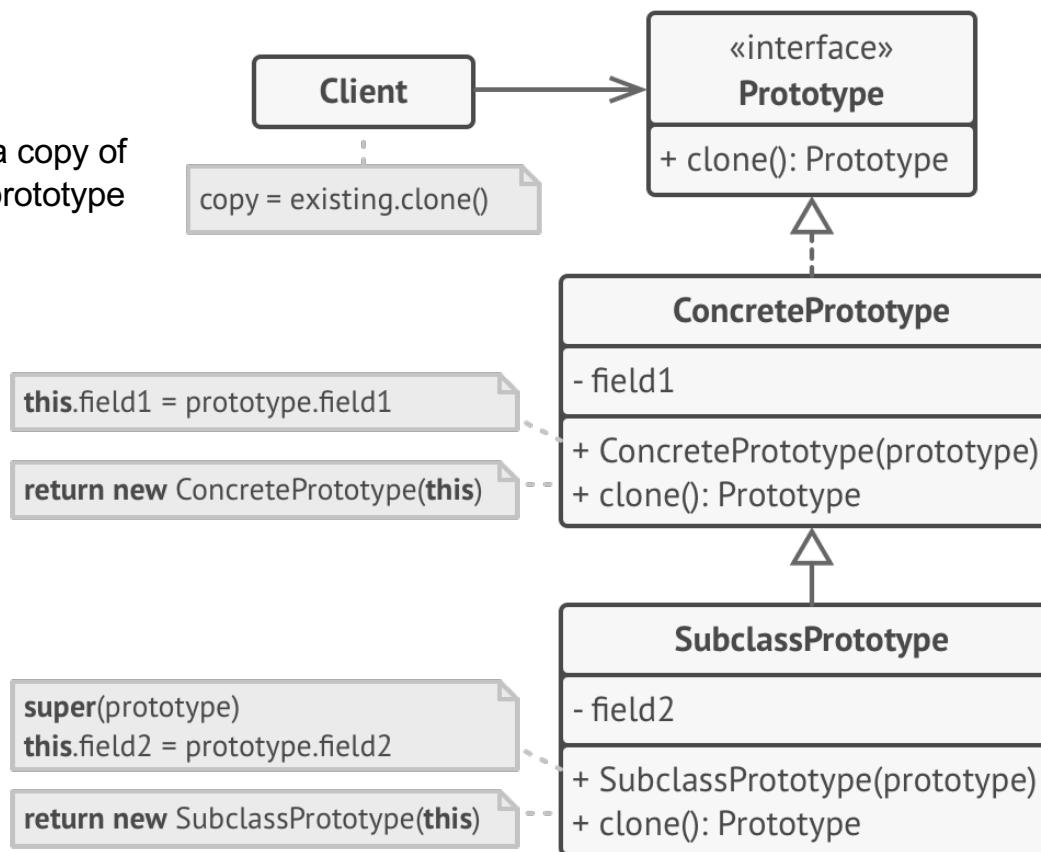
Real world analogy to the pattern is the process of mitotic cell division (biology, remember?). After mitotic division, a pair of identical cells is formed. The original cell acts as a prototype and takes an active role in creating the copy.

Use the Prototype pattern when your code shouldn't depend on the concrete classes of objects that you need to copy.

1. The Prototype interface declares the cloning methods. In most cases, it's a single clone method.
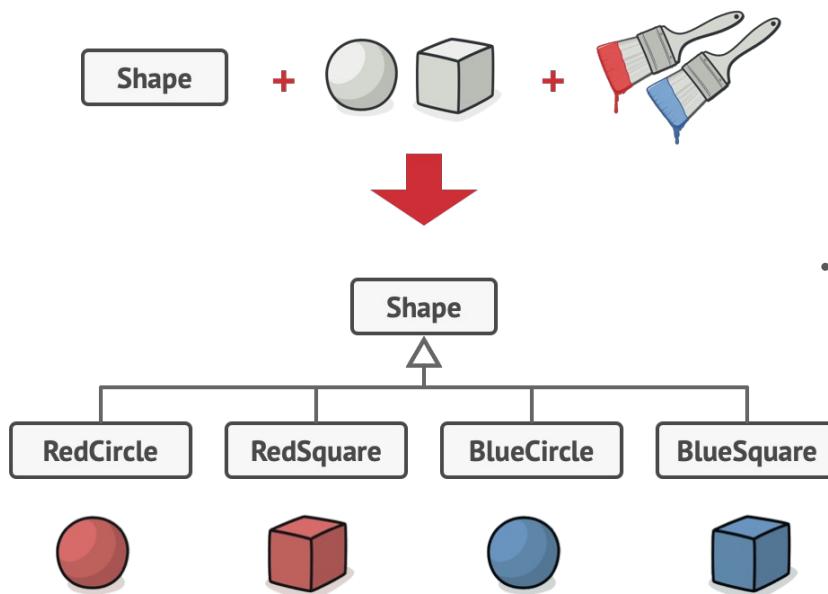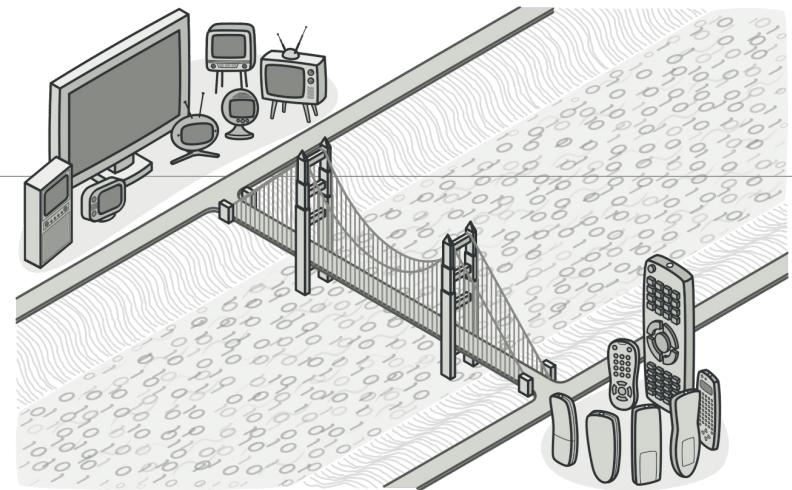
2. The **Client** can produce a copy of any object that follows the prototype interface.

copy = existing.clone()

Client → «interface» **Prototype**
+ clone(): Prototype

2. he **Concrete Prototype** class implements the cloning method. In addition to copying the original object's data to the clone, this method may also handle some edge cases of the cloning process related to cloning linked objects, untangling recursive dependencies, etc.

ConcretePrototype

- field1

+ ConcretePrototype(prototype)
+ clone(): Prototype

this.field1 = prototype.field1

return new ConcretePrototype(**this**)

SubclassPrototype

- field2

+ SubclassPrototype(prototype)
+ clone(): Prototype

super(prototype)
this.field2 = prototype.field2

return new SubclassPrototype(**this**)

Structural patterns: Bridge

Bridge is a structural design pattern that lets you split a large class or a set of closely related classes into two separate hierarchies—abstraction and implementation—which can be developed independently of each other.
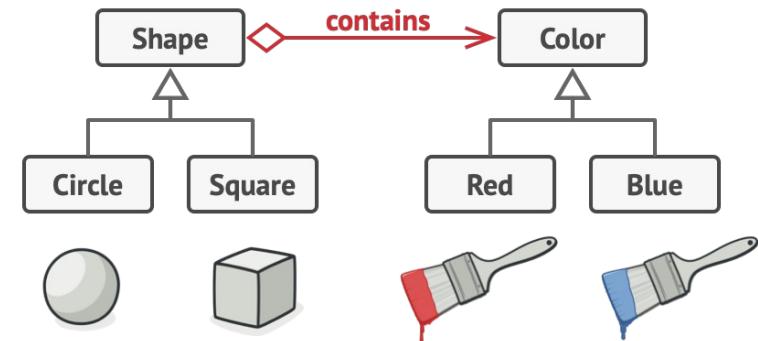
Problem

- Say you have a geometric Shape class with a pair of subclasses: Circle and Square. You want to extend this class hierarchy to incorporate colors, so you plan to create Red and Blue shape subclasses. However, since you already have two subclasses, you'll need to create four class combinations such as BlueCircle and RedSquare.

- **Adding new shape types and colors to the hierarchy will grow it exponentially.** For example, to add a triangle shape you'd need to introduce two subclasses, one for each color. And after that, adding a new color would require creating three subclasses, one for each shape type. The further we go, the worse it becomes.

Structural patterns: Bridge (continued)

Solution

- This problem occurs because we're trying to extend the shape classes in two independent dimensions: by form and by color. That's a very common issue with class inheritance.

- The Bridge pattern attempts to solve this problem by switching from inheritance to the object composition. What this means is that you extract one of the dimensions into a separate class hierarchy, so that the original classes will reference an object of the new hierarchy, instead of having all of its state and behaviors within one class.

- Following this approach, we can extract the color-related code into its own class with two subclasses: Red and Blue. The Shape class then gets a reference field pointing to one of the color objects. Now the shape can delegate any color-related work to the linked color object. That reference will act as a bridge between the Shape and Color classes. From now on, adding new colors won't require changing the shape hierarchy, and vice versa.
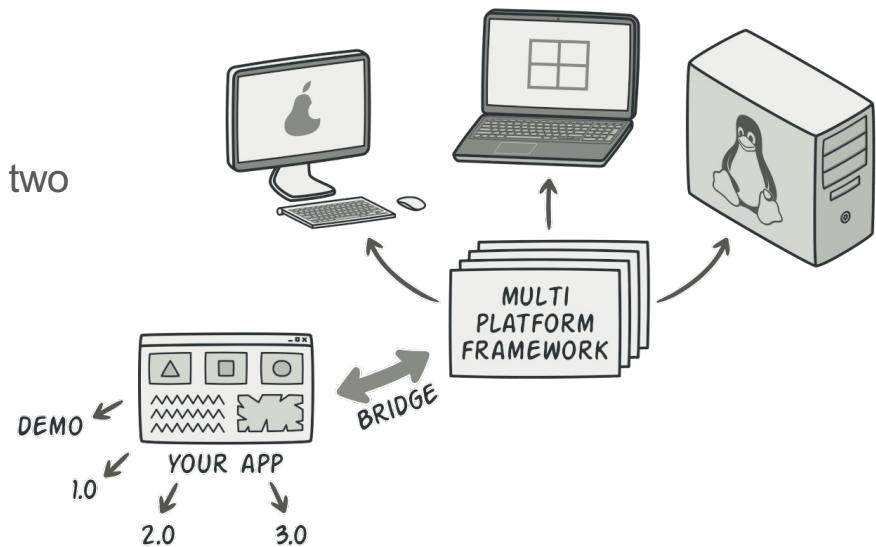
Structural patterns: Bridge (continued)

Abstraction and *Implementation* are alternative names of the Bridge definition.
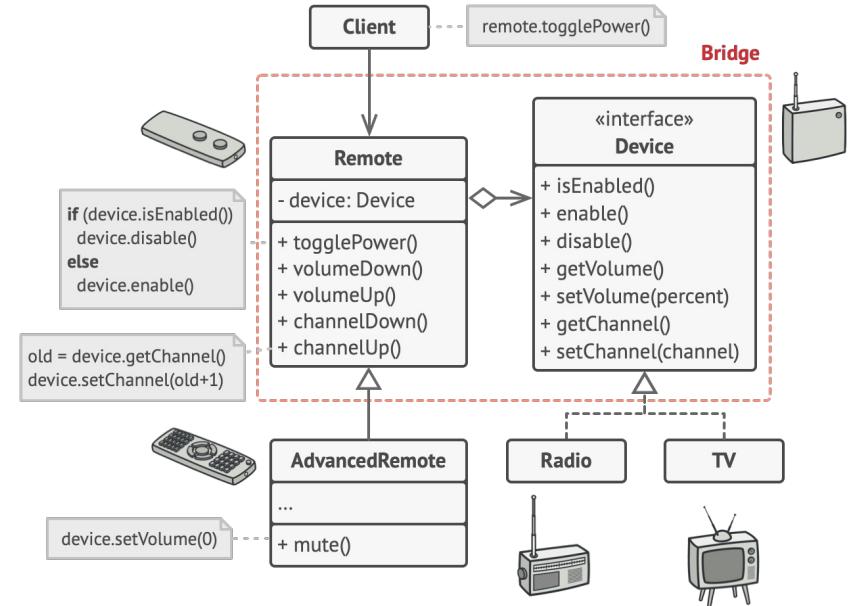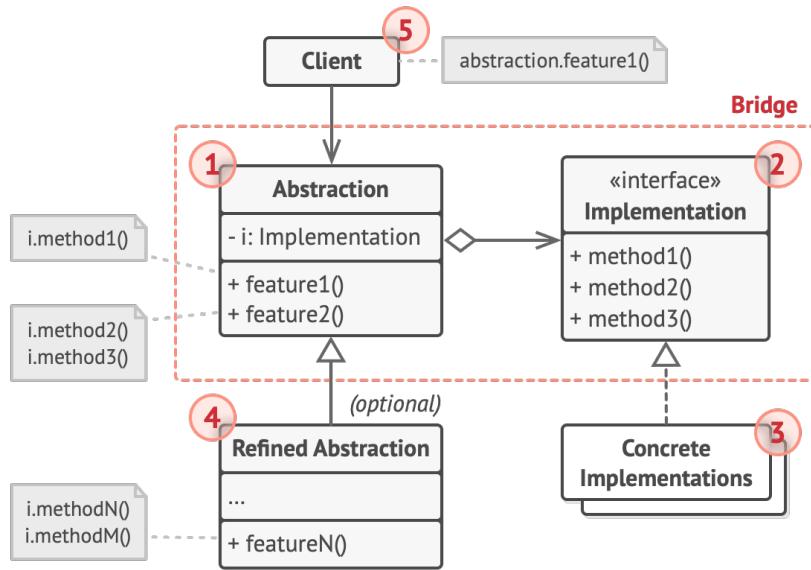
In real applications, the abstraction can be represented by a graphical user interface (GUI), and the implementation could be the underlying operating system code (API) which the GUI layer calls in response to user interactions.

- Generally speaking, you can extend such an app in two independent directions:
- Have several different GUIs (for instance, tailored for regular customers or admins).
- Support several different APIs (for example, to be able to launch the app under Windows, Linux, and macOS).
- In a worst-case scenario, this app might look like a giant spaghetti bowl, where hundreds of conditionals connect different types of GUI with various APIs all over the code.

- Bridge pattern suggests that we divide the classes into two hierarchies:
- Abstraction: the GUI layer of the app.
- Implementation: the operating systems' APIs.
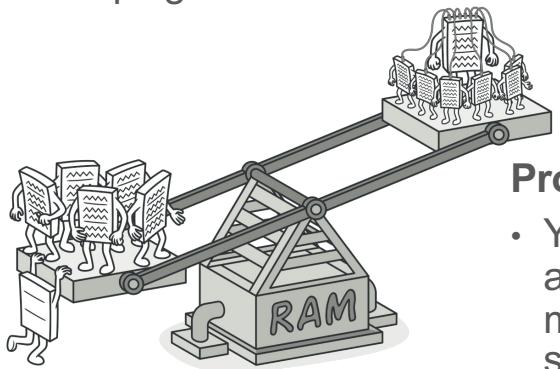
Structural patterns: Bridge (continued)

1. The **Abstraction** provides high-level control logic. It relies on the implementation object to do the actual low-level work.

2. The **Implementation** declares the interface that's common for all concrete implementations. An abstraction can only communicate with an implementation object via methods that are declared here.

3. The abstraction may list the same methods as the implementation, but usually the abstraction declares some complex behaviors that rely on a wide variety of primitive operations declared by the implementation.

4. **Concrete Implementations** contain platform-specific code.

5. **Refined Abstractions** provide variants of control logic. Like their parent, they work with different implementations via the general implementation interface.

6. Usually, the **Client** is only interested in working with the abstraction. However, it's the client's job to link the abstraction object with one of the implementation objects.
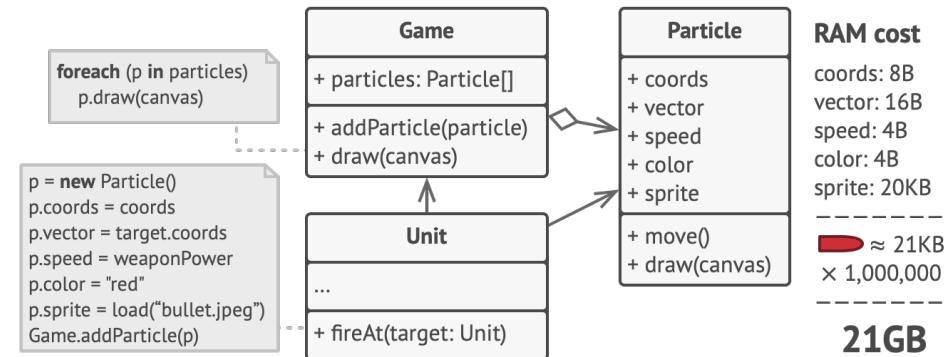
Structural patterns: Flyweight

Flyweight is a structural design pattern that lets you fit more objects into the available amount of RAM by sharing common parts of state between multiple objects instead of keeping all of the data in each object.


```
foreach (p in particles)
    p.draw(canvas)
```

```
p = new Particle()
p.coords = coords
p.vector = target.coords
p.speed = weaponPower
p.color = "red"
p.sprite = load("bullet.jpeg")
Game.addParticle(p)
```

Game
+ particles: Particle[]
+ addParticle(particle)
+ draw(canvas)

Unit
...
+ fireAt(target: Unit)

Particle
+ coords
+ vector
+ speed
+ color
+ sprite
+ move()
+ draw(canvas)

RAM cost
coords: 8B
vector: 16B
speed: 4B
color: 4B
sprite: 20KB

≈ 21KB
× 1,000,000

21GB

Problem

- You decided to create a simple video game: players would be moving around a map and shooting each other. You chose to implement a realistic particle system and make it a distinctive feature of the game. Vast quantities of bullets, missiles, and shrapnel from explosions should fly all over the map and deliver a thrilling experience to the player.

- Upon its completion, you built the game and sent it to your friend for a test drive. Although the game was running flawlessly on your machine, your friend wasn't able to play for long. On his computer, the game kept crashing after a few minutes of gameplay. After spending several hours digging through debug logs, you discovered that the game crashed because of an insufficient amount of RAM. It turned out that your friend's rig was much less powerful than your own computer, and that's why the problem emerged so quickly on his machine.

- The actual problem was related to your particle system. Each particle, such as a bullet, a missile or a piece of shrapnel was represented by a separate object containing plenty of data. At some point, when the carnage on a player's screen reached its climax, newly created particles no longer fit into the remaining RAM, so the program crashed.

Structural patterns: Flyweight (continued)

Solution

- On closer inspection of the Particle class, you may notice that the color and sprite fields consume a lot more memory than other fields. What's worse is that these two fields store almost identical data across all particles. For example, all bullets have the same color and sprite.

- Other parts of a particle's state, such as coordinates, movement vector and speed, are unique to each particle. After all, the values of these fields change over time. This data represents the always changing context in which the particle exists, while the color and sprite remain constant for each particle.

- This constant data of an object is usually called the *intrinsic state*. It lives within the object; other objects can only read it, not change it. The rest of the object's state, often altered "from the outside" by other objects, is called the *extrinsic state*.

- The Flyweight pattern suggests that you stop storing the extrinsic state inside the object. Instead, you should pass this state to specific methods which rely on it. Only the intrinsic state stays within the object, letting you reuse it in different contexts. As a result, you'd need fewer of these objects since they only differ in the intrinsic state, which has much fewer variations than the extrinsic.

Unique (extrinsic) state (mutable)

Repeating (intrinsic) state (immutable)

Particle

+ coords
+ vector
+ speed
+ color
+ sprite

+ move()
+ draw(canvas)

MovingParticle

- particle
- coords
- vector
- speed

+ MovingParticle(...)
+ move()
+ draw(canvas)

Lots

Few

Particle

- color
- sprite

+ Particle(color, sprite)
+ move(coords, vector, speed)
+ draw(coords, canvas)

particle.move(coords, vector, speed)

particle.draw(coords, canvas)

Game

+ mps: MovingParticles[]
- particles: Particle[]

+ addParticle(coords, vector, speed, color, sprite)
+ draw(canvas)

Particle

- color
- sprite

+ Particle(color, sprite)
+ move(coords, vector, speed)
+ draw(coords, canvas)

Unit

...

+ fireAt(target: Unit)

MovingParticle

- particle
- coords
- vector
- speed

+ MovingParticle(...)
+ move()
+ draw(canvas)

Game.addParticle(coords, target.coords, weaponPower, "red", "bullet.jpeg")

RAM cost

color: 4B
sprite: 20KB
――――――
≈ 21KB

coords: 8B
vector: 16B
speed: 4B
particle: 4B
――――――
≈ 32B

× 1

× 1,000,000

**32MB**

Structural patterns: Flyweight (continued)

Where does the extrinsic state move to? Some class should still store it, right? In most cases, it gets moved to the container object, which aggregates objects before we apply the pattern.

In our case, that's the main Game object that stores all particles in the particles field. To move the extrinsic state into this class, you need to create several array fields for storing coordinates, vectors, and speed of each individual particle. But that's not all. You need another array for storing references to a specific flyweight that represents a particle. These arrays must be in sync so that you can access all data of a particle using the same index.

- A more elegant solution is to create a separate context class that would store the extrinsic state along with reference to the flyweight object. This approach would require having just a single array in the container class.

Game

- particles: Particle[]

+ addParticle(particle)
+ draw(canvas)

Game

+ coords
+ vectors
+ speeds
+ references
+ particles: Particle[]

+ addParticle(particle)
+ draw(canvas)

Game

- mps: MovingParticles[]
- particles: Particle[]

+ addParticle(coords, vector, speed, color, sprite): Particle
+ draw(canvas)

foreach (p **in** mps)
p.draw(canvas)

1. Go over the particles array and try to find an existing particle with the given color and sprite. If there's none then create a new one.

2. Create a moving particle with given data and the particle object from the first step.

Structural patterns: Flyweight (continued)

1. The Flyweight pattern is merely an optimization.

2. The **Flyweight** class contains the portion of the original object's state that can be shared between multiple objects. The same flyweight object can be used in many different contexts. The state stored inside a flyweight is called *intrinsic.* The state passed to the flyweight's methods is called *extrinsic.*

3. The **Context** class contains the extrinsic state, unique across all original objects. When a context is paired with one of the flyweight objects, it represents the full state of the original object.

4. Usually, the behavior of the original object remains in the flyweight class. In this case, whoever calls a flyweight's method must also pass appropriate bits of the extrinsic state into the method's parameters. On the other hand, the behavior can be moved to the context class, which would use the linked flyweight merely as a data object.

5. The **Client** calculates or stores the extrinsic state of flyweights. From the client's perspective, a flyweight is a template object which can be configured at runtime by passing some contextual data into parameters of its methods.

6. The **Flyweight Factory** manages a pool of existing flyweights. With the factory, clients don't create flyweights directly. Instead, they call the factory, passing it bits of the intrinsic state of the desired flyweight. The factory looks over previously created flyweights and either returns an existing one that matches search criteria or creates a new one if nothing is found.

Chain of Responsibility is a behavioral design pattern that lets you pass requests along a chain of handlers. Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain.


Problem
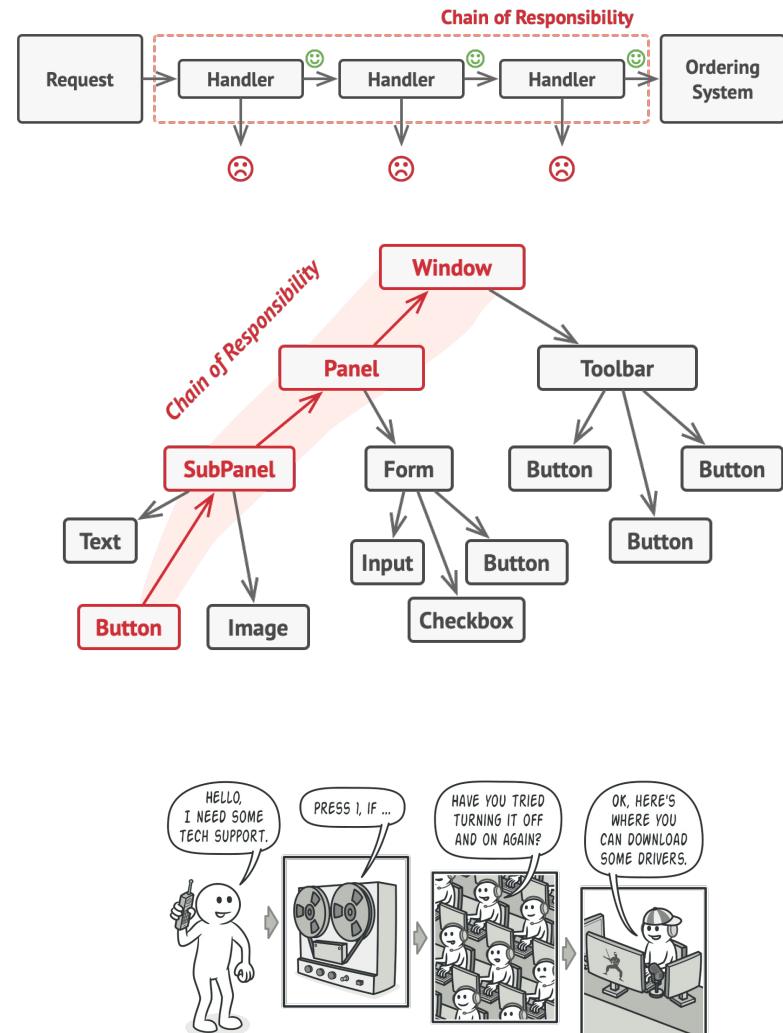
- Example of an online ordering system. You want to restrict access to the system so only authenticated users can create orders. Also, users who have administrative permissions must have full access to all orders.

- These checks must be performed sequentially. The application can attempt to authenticate a user to the system whenever it receives a request that contains the user's credentials. However, if those credentials aren't correct and authentication fails, there's no reason to proceed with any other checks.

- The code of the checks, which had already looked like a mess, became more and more bloated as you added each new feature. Changing one check sometimes affected the others. When you tried to reuse the checks to protect other components of the system, you had to duplicate some of the code since those components required some of the checks, but not all of them.

- The system became very hard to comprehend and expensive to maintain. You struggled with the code for a while, until one day you decided to refactor the whole thing.
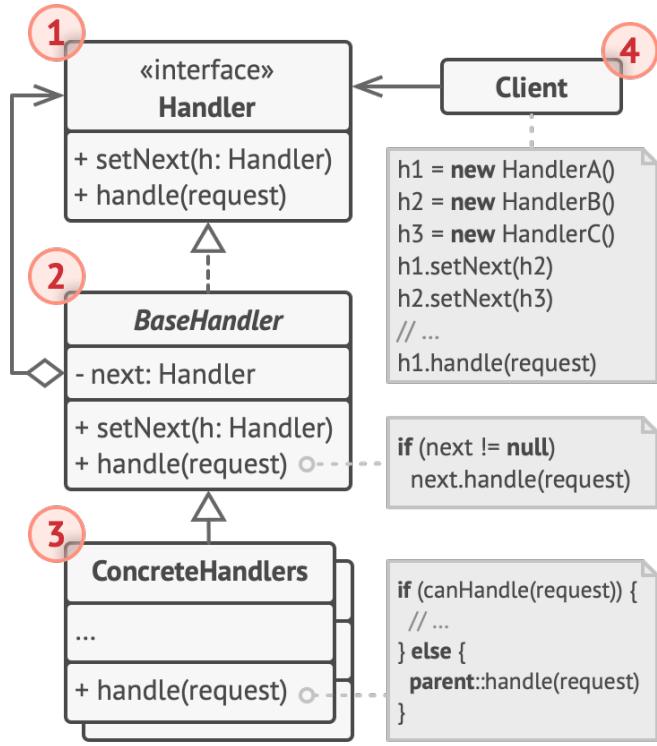
Solution

- Like many other behavioral design patterns, the **Chain of Responsibility** relies on transforming particular behaviors into stand-alone objects called *handlers*. In our case, each check should be extracted to its own class with a single method that performs the check. The request, along with its data, is passed to this method as an argument.

- The pattern suggests that you link these handlers into a chain. Each linked handler has a field for storing a reference to the next handler in the chain. In addition to processing a request, handlers pass the request further along the chain. The request travels along the chain until all handlers have had a chance to process it.

- Here's the best part: a handler can decide not to pass the request further down the chain and effectively stop any further processing.

- In our example with ordering systems, a handler performs the processing and then decides whether to pass the request further down the chain. Assuming the request contains the right data, all the handlers can execute their primary behavior, whether it's authentication checks or caching.

Real world analogy

Behavioral patterns: Chain on command (continued)


```
1  «interface»
   Handler
   + setNext(h: Handler)
   + handle(request)

2  BaseHandler
   - next: Handler
   + setNext(h: Handler)
   + handle(request)

3  ConcreteHandlers
   ...
   + handle(request)
```

```
4  Client

h1 = new HandlerA()
h2 = new HandlerB()
h3 = new HandlerC()
h1.setNext(h2)
h2.setNext(h3)
// ...
h1.handle(request)
```

```
if (next != null)
  next.handle(request)
```

```
if (canHandle(request)) {
  // ...
} else {
  parent::handle(request)
}
```
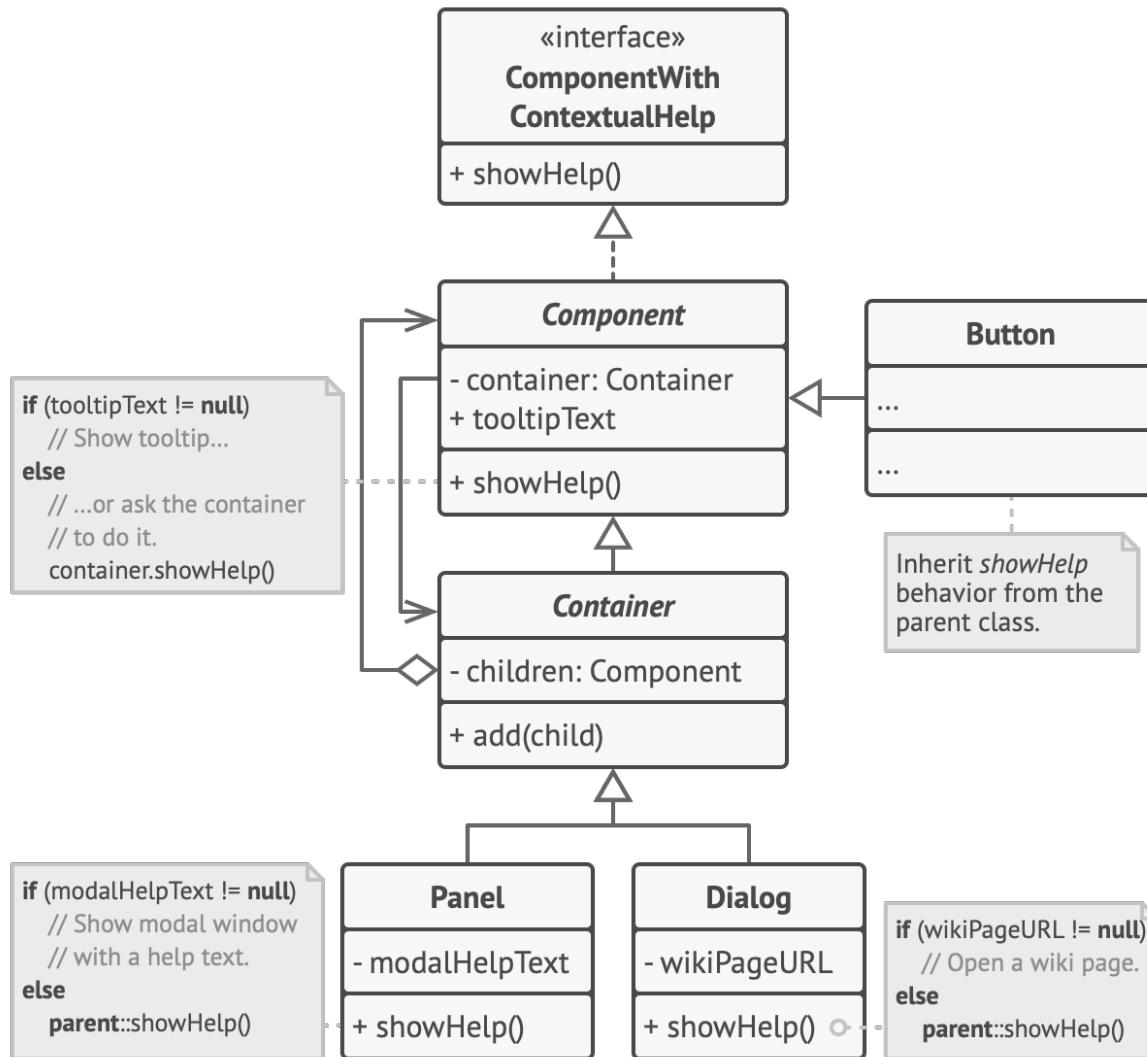
1. The **Handler** declares the interface, common for all concrete handlers. It usually contains just a single method for handling requests, but sometimes it may also have another method for setting the next handler on the chain.

2. The **Base Handler** is an optional class where you can put the boilerplate code that's common to all handler classes.

3. Usually, this class defines a field for storing a reference to the next handler. The clients can build a chain by passing a handler to the constructor or setter of the previous handler. The class may also implement the default handling behavior: it can pass execution to the next handler after checking for its existence.

4. **Concrete Handlers** contain the actual code for processing requests. Upon receiving a request, each handler must decide whether to process it and, additionally, whether to pass it along the chain.

5. Handlers are usually self-contained and immutable, accepting all necessary data just once via the constructor.

6. The **Client** may compose chains just once or compose them dynamically, depending on the application's logic. Note that a request can be sent to any handler in the chain—it doesn't have to be the first one.

Behavioral patterns: Chain on command (continued)

In this example, the Chain of Responsibility pattern is responsible for displaying contextual help information for active GUI elements.

«interface»
ComponentWith ContextualHelp

+ showHelp()

Component

- container: Container
+ tooltipText

+ showHelp()

Button

...

...

if (tooltipText != **null**)
   // Show tooltip...
else
   // ...or ask the container
   // to do it.
   container.showHelp()

Inherit *showHelp* behavior from the parent class.

Container

- children: Component

+ add(child)

if (modalHelpText != **null**)
   // Show modal window
   // with a help text.
else
   parent::showHelp()

Panel

- modalHelpText

+ showHelp()

Dialog

- wikiPageURL

+ showHelp()

if (wikiPageURL != **null**)
   // Open a wiki page.
else
   parent::showHelp()

Sources & other reading

- **Design Patterns: Elements of Reusable Object-Oriented Software** by Erich Gamma (Author), Richard Helm (Author), Ralph Johnson (Author), John Vlissides (Author)
- Design Patterns Design Patterns in Java (refactoring.guru)
- CSE 331 23wi (washington.edu)

Contact:

RNDr. Július Šiška, PhD.

E: julius.siska at gmail.com