

IDT, Přednáška 2

Libor Váša

Katedra informatiky a výpočetní techniky, Západočeská univerzita v Plzni

19. 2. 2024

Jak funguje počítač

Jak funguje počítač

- je hloupý

Jak funguje počítač

- je hloupý
- vykonává instrukce nad daty

Jak funguje počítač

- je hloupý
- vykonává instrukce nad daty
- data = registry, paměť

- řada binárních stavů (8, 16, 32, 64, ...) - bitů
- význam datům je přiřazený pouze **konvencí**

- řada binárních stavů (8, 16, 32, 64, ...) - bitů
- význam datům je přiřazený pouze **konvencí**
- v paměti
 - dlouhá řada "buněk"
 - dají se pouze číst/zapisovat

- řada binárních stavů (8, 16, 32, 64, ...) - bitů
- význam datům je přiřazený pouze **konvencí**
- v paměti
 - dlouhá řada "buněk"
 - dají se pouze číst/zapisovat
- v registrech
 - přímo v procesoru
 - dají se zpracovávat

- přiřazuje význam
- určuje možné operace

- MOV, INC, JMP, CMP, ...
- procesor umí jen nějaké
- některé procesory umí rafinovanější ("multimediální"- MMX, "vektorové", ...)

- MOV, INC, JMP, CMP, ...
- procesor umí jen nějaké
- některé procesory umí rafinovanější ("multimediální"- MMX, "vektorové", ...)
- programovat v instrukcích je poměrně obtížné

```
push ebp      ; Save the old base pointer value.
mov ebp, esp ; Set the new base pointer value.
sub esp, 4    ; Make room for one 4-byte local variable.
push edi      ; Save the values of registers that the function
push esi      ; will modify. This function uses EDI and ESI.
; (no need to save EBX, EBP, or ESP)

; Subroutine Body
mov eax, [ebp+8] ; Move value of parameter 1 into EAX
mov esi, [ebp+12] ; Move value of parameter 2 into ESI
mov edi, [ebp+16] ; Move value of parameter 3 into EDI

mov [ebp-4], edi ; Move EDI into the local variable
add [ebp-4], esi ; Add ESI into the local variable
add eax, [ebp-4]
```



Vyšší programovací jazyky

- umožňují zápis bližší přirozenému jazyku
- DOKUD <platí něco> OPAKUJ <tohle>
- KDYŽ <platí tohle> TAK <udělej tohle>
- Pascal, C, Fortran, Algol, ...

Objektově orientované jazyky

- umožňují spojit data a operace do logických celků - tříd
- program je pak ještě "přirozenější"

```
Line l = new Line(1, 2);  
Line p = new Line(2, 3);  
Point x = l.intersection(p);
```

- Java, C#, C++, ...

- má-li se program vykonat, musí se přeložit (compile)
- ze zdrojového kódu vznikne spustitelný kód (instrukce pro procesor)
- kompilátor je program, musel ho někdo vytvořit

- procesory mají různé instrukční sady

- procesory mají různé instrukční sady
- procesory mají různé počty registrů

- procesory mají různé instrukční sady
- procesory mají různé počty registrů
- kompilace je složitý proces
 - je třeba využít vlastností cílové platformy

(pokus o) Řešení

- definujeme jednotnou sadu instrukcí, do kterých budeme překládat program na jakékoli platformě

(pokus o) Řešení

- definujeme jednotnou sadu instrukcí, do kterých budeme překládat program na jakékoli platformě

Vykonání programu

- 1 interpretace instrukcí
 - "simultánní překlad"
 - rychlý start (začneme hned)
 - pomalejší běh

(pokus o) Řešení

- definujeme jednotnou sadu instrukcí, do kterých budeme překládat program na jakékoli platformě

Vykonání programu

- 1 interpretace instrukcí
 - "simultánní překlad"
 - rychlý start (začneme hned)
 - pomalejší běh
- 2 Just-in-time kompilace
 - překlad do strojového jazyka konkrétního procesoru těsně před spuštěním
 - pomalejší start (kompilace nějakou dobu trvá)
 - rychlejší běh (kompilace může lépe volit instrukce)

.NET

.NET jazyk (C#, F#, Visual Basic, ...) → intermediate language (IL) → .NET Runtime

.NET

.NET jazyk (C#, F#, Visual Basic, ...) → intermediate language (IL) → .NET Runtime

Java

Java → bytecode → Java Runtime Environment(JRE)

.NET

.NET jazyk (C#, F#, Visual Basic, ...) → intermediate language (IL) → .NET Runtime

Java

Java → bytecode → Java Runtime Environment(JRE)

- stejný princip
- program spustíme na každém stroji, kde je k dispozici JRE, resp. .NET Runtime

.NET

.NET jazyk (C#, F#, Visual Basic, ...) → intermediate language (IL) → .NET Runtime

Java

Java → bytecode → Java Runtime Environment(JRE)

- stejný princip
- program spustíme na každém stroji, kde je k dispozici JRE, resp. .NET Runtime
- v praxi funguje vždy jen „tak trochu“ - problémy s knihovnami atd.

Příklad intermediate language

C#

```
using System;
class Program
{
    static void Main()
    {
        int i = 0;
        while (i < 10)
        {
            Console.WriteLine(i);
            i++;
        }
    }
}
```

Intermediate language

```
.method private hidebysig static void Main() cil managed
{
    .entrypoint
    .maxstack 2
    .locals init (
        [0] int32 num)
    L_0000: ldc.i4.0
    L_0001: stloc.0
    L_0002: br.s L_000e
    L_0004: ldloc.0
    L_0005: call void [mscorlib]System.Console::WriteLine(int32)
    L_000a: ldloc.0
    L_000b: ldc.i4.1
    L_000c: add
    L_000d: stloc.0
    L_000e: ldloc.0
    L_000f: ldc.i4.s 10
    L_0011: blt.s L_0004
    L_0013: ret
}
```

Objektově orientované programování (OOP) v C#

- datové typy: jednoduchá data
- jazyky umožňují sdružování dat do větších celků

- datové typy: jednoduchá data
- jazyky umožňují sdružování dat do větších celků

Pozorování 1

Většina operací dává smysl jen nad daty **s konkrétním významem**.

- datové typy: jednoduchá data
- jazyky umožňují sdružování dat do větších celků

Pozorování 1

Většina operací dává smysl jen nad daty **s konkrétním významem**.

Příklad

- funkce $f(x, y) = \sqrt{x^2 + y^2}$ má smysl, pokud x a y jsou kartézské souřadnice bodu v 2D eukleidovském prostoru. Funkce má význam vzdálenosti bodu od počátku.
- pokud x je úhel a y poloměr v polárních souřadnicích, pak tato funkce **nemá smysl**, přestože podle vzorce vypočítat jde.

Pozorování 2

Často mají operace se stejným **smyslem** různou **implementaci** v závislosti na konkrétním druhu vstupních dat.

Pozorování 2

Často mají operace se stejným **smyslem** různou **implementaci** v závislosti na konkrétním druhu vstupních dat.

Příklad

- vzdálenost od počátku se pro bod v kartézských souřadnicích spočítá jako $d = \sqrt{x^2 + y^2}$, zatímco pro bod v polárních souřadnicích se určí jako $d = r$.
- bylo by užitečné mít možnost požadovat službu poskytující vzdálenost od počátku a nestarat se o konkrétní způsob reprezentace dat.

Pozorování 2

Často mají operace se stejným **smyslem** různou **implementaci** v závislosti na konkrétním druhu vstupních dat.

Příklad

- vzdálenost od počátku se pro bod v kartézských souřadnicích spočítá jako $d = \sqrt{x^2 + y^2}$, zatímco pro bod v polárních souřadnicích se určí jako $d = r$.
- bylo by užitečné mít možnost požadovat službu poskytující vzdálenost od počátku a nestarat se o konkrétní způsob reprezentace dat.

Motivace objektového programování

- oprostit se od mechanických detailů implementace
- umožnit programování soustředěné na **smysl** operací

Třída

- definuje data
 - jaké typy (číslo, řetězec, atd.)
 - jak se budou jmenovat
- definuje operace

Třída

- definuje data
 - jaké typy (číslo, řetězec, atd.)
 - jak se budou jmenovat
- definuje operace

Instance

- konkrétní případ, konkrétní hodnoty dat
- zabírá místo v paměti

Třída

- definuje data
 - jaké typy (číslo, řetězec, atd.)
 - jak se budou jmenovat
- definuje operace

Instance

- konkrétní případ, konkrétní hodnoty dat
- zabírá místo v paměti

Příklad

```
class Person{  
    String name;  
    int age;  
}
```

Referenční proměnná

- **ukazuje** na instanci třídy
- má své jméno ve zdrojovém kódu
- více referenčních proměnných může ukazovat na stejnou instanci
- nemusí ukazovat nikam - hodnota **null**

Referenční proměnná

- **ukazuje** na instanci třídy
- má své jméno ve zdrojovém kódu
- více referenčních proměnných může ukazovat na stejnou instanci
- nemusí ukazovat nikam - hodnota **null**

Příklad:

```
Person person1 = null;
```

- referenční proměnná se jmenuje `person1`
- může ukazovat na instanci třídy `Person`
- nyní neukazuje na žádnou (`null`)

- klíčové slovo `new`
- alokuje (zarezuje) místo v paměti pro jednu novou instanci třídy
- vrací referenci na novou instanci - můžeme ji přiřadit do referenční proměnné

- klíčové slovo **new**
- alokuje (zarezuje) místo v paměti pro jednu novou instanci třídy
- vrací referenci na novou instanci - můžeme ji přiřadit do referenční proměnné

Příklad:

```
Person p;  
p = new Person();
```


- klíčové slovo **new**
- alokuje (zarezuje) místo v paměti pro jednu novou instanci třídy
- vrací referenci na novou instanci - můžeme ji přiřadit do referenční proměnné

Příklad:

```
Person p;  
p = new Person();
```

Nebo zkráceně:

```
Person p = new Person();
```

- referenci na vytvořenou instanci nemusíme nutně ukládat do žádné proměnné

```
public void paint(Graphics g) {  
    g.setStroke(new BasicStroke(5));  
    ...  
}
```

- datové položky jednotlivých instancí
- můžeme k nim přistupovat tečkovou notací, přiřazovat, přepisovat atd.

- datové položky jednotlivých instancí
- můžeme k nim přistupovat tečkovou notací, přiřazovat, přepisovat atd.

Příklad:

```
Person p = new Person();  
p.age = 35;  
p.name = "Josef_Novak";  
int a = p.age;  
a += 1;
```

Referenční proměnná

(značně) Nepřesná, leč instruktivní představa

Hodnotou referenční proměnné je **adresa v paměti**

- např. číslo paměťové buňky, kde začínají data nějaké instance

Referenční proměnná

(značně) Nepřesná, leč instruktivní představa

Hodnotou referenční proměnné je **adresa v paměti**

- např. číslo paměťové buňky, kde začínají data nějaké instance

```
Point p1 = new Point(3, 2);  
Point p2 = new Point(5, 7);  
int x = 8;  
Point p3 = p1;
```

Referenční proměnná

(značně) Nepřesná, leč instruktivní představa

Hodnotou referenční proměnné je **adresa v paměti**

- např. číslo paměťové buňky, kde začínají data nějaké instance

```
Point p1 = new Point(3, 2);  
Point p2 = new Point(5, 7);  
int x = 8;  
Point p3 = p1;
```

číslo paměťové buňky →	0	1	2	3	4	5	6	7
hodnota paměťové buňky →	4	6	8	4	3	2	5	7
název v programu →	p1	p2	x	p3				

```
int n1 = 10;  
int n2 = n1;  
n2 = 20;  
Console.WriteLine(n1); // 10  
Console.WriteLine(n2); // 20
```



```
int n1 = 10;  
int n2 = n1;  
n2 = 20;  
Console.WriteLine(n1); // 10  
Console.WriteLine(n2); // 20
```

- operátor přiřazení (=) kopíruje data
- všechny proměnné (pojmenované) mají primitivní datový typ
- referenční proměnná **je primitivní datový typ**
- její hodnotou (tj. co se zkopíruje) je **odkaz** (reference)!

Kopírování dat - příklad

```
Person p1 = new Person();  
p1.age = 35;
```

Kopírování dat - příklad

```
Person p1 = new Person();  
p1.age = 35;
```

```
Person p2 = p1;
```

Kopírování dat - příklad

```
Person p1 = new Person();  
p1.age = 35;
```

```
Person p2 = p1;
```

```
p2.age = 25;
```

Kopírování dat - příklad

```
Person p1 = new Person();  
p1.age = 35;
```

```
Person p2 = p1;
```

```
p2.age = 25;
```

```
Console.WriteLine(p1.age);
```

Kopírování dat - příklad

```
Person p1 = new Person();  
p1.age = 35;
```

```
Person p2 = p1;
```

```
p2.age = 25;
```

```
Console.WriteLine(p1.age); // 25!
```

- podprogramy definované nad daty instance
- mají přístup ke všem datovým položkám
- referenční proměnná `this` - ukazuje na instanci, nad kterou byla metoda volána

Příklad:

```
public void printArea() {  
    Console.WriteLine(this.a*this.b);  
}
```

- identifikátor `this` je možno vynechat, pokud nevznikne dvojznačnost

Návratová hodnota

- hodnota předávaná ven z metody
- klíčové slovo `return`
- pokud žádná není, uvádí se `void`

Návratová hodnota

- hodnota předávaná ven z metody
- klíčové slovo `return`
- pokud žádná není, uvádí se `void`

Příklad:

```
public double area() {  
    return (this.a*this.b);  
}  
...  
double a = r.area();
```

Parametry metody

- data předávaná do metody
- uvnitř se chovají jako lokální proměnné
- při volání se musí určit jejich hodnoty

Patametry metody

- data předávaná do metody
- uvnitř se chovají jako lokální proměnné
- při volání se musí určit jejich hodnoty

Příklad:

```
public double cylinderVolume(double radius, double height){  
    double volume = Math.PI * radius * radius * height;  
    return(volume);  
}  
  
...  
double a = cylinderVolume(3+7, h);
```

Patametry metody

- data předávaná do metody
- uvnitř se chovají jako lokální proměnné
- při volání se musí určit jejich hodnoty

Příklad:

```
public double cylinderVolume(double radius, double height){  
    double volume = Math.PI * radius * radius * height;  
    return(volume);  
}  
  
...  
double a = cylinderVolume(3+7, h);
```

- předání parametrů funguje podobně jako přiřazení: určí se hodnota, vloží se do proměnné

Příklad

```
class Document{
    public void loadFromFile(string fileName){
        ...
    }

    public void print(){
        ...
    }
}

class PrintFile{
    static void Main(String[] args){
        if (args.Length>0){
            Document d = new Document();
            d.loadFromFile(args[0]);
            d.print();
        }
    }
}
```

Referenční proměnná jako parametr metody

- kopíruje se hodnota (jako každý jiný primitivní datový typ)
- hodnotou je odkaz

Referenční proměnná jako parametr metody

- kopíruje se hodnota (jako každý jiný primitivní datový typ)
- hodnotou je odkaz

```
void increaseAge(Person p, int i){  
    p.age = p.age + i;  
    i = i+1;  
}
```

Referenční proměnná jako parametr metody

- kopíruje se hodnota (jako každý jiný primitivní datový typ)
- hodnotou je odkaz

```
void increaseAge(Person p, int i){  
    p.age = p.age + i;  
    i = i+1;  
}
```

```
...  
Person p1 = new Person();  
p1.age = 25;  
int i1 = 5;  
increaseAge(p1, i1);  
Console.WriteLine(p1.age);
```


Referenční proměnná jako parametr metody

- kopíruje se hodnota (jako každý jiný primitivní datový typ)
- hodnotou je odkaz

```
void increaseAge(Person p, int i){  
    p.age = p.age + i;  
    i = i+1;  
}
```

```
...  
Person p1 = new Person();  
p1.age = 25;  
int i1 = 5;  
increaseAge(p1, i1);  
Console.WriteLine(p1.age);    // 30!
```

Referenční proměnná jako parametr metody

- kopíruje se hodnota (jako každý jiný primitivní datový typ)
- hodnotou je odkaz

```
void increaseAge(Person p, int i){  
    p.age = p.age + i;  
    i = i+1;  
}
```

```
...  
Person p1 = new Person();  
p1.age = 25;  
int i1 = 5;  
increaseAge(p1, i1);  
Console.WriteLine(p1.age);    // 30!  
Console.WriteLine(i1);
```

Referenční proměnná jako parametr metody

- kopíruje se hodnota (jako každý jiný primitivní datový typ)
- hodnotou je odkaz

```
void increaseAge(Person p, int i){  
    p.age = p.age + i;  
    i = i+1;  
}
```

```
...  
Person p1 = new Person();  
p1.age = 25;  
int i1 = 5;  
increaseAge(p1, i1);  
Console.WriteLine(p1.age);    // 30!  
Console.WriteLine(i1);        // 5  
...
```

Přetížení metod

- několik metod může mít stejné jméno
- **musí** se lišit počtem, typem nebo pořadím parametrů
- **nesmí** se lišit návratový typ
- mohou volat jedna druhou (šetří to kód)

Přetížení metod

- několik metod může mít stejné jméno
- **musí** se lišit počtem, typem nebo pořadím parametrů
- **nesmí** se lišit návratový typ
- mohou volat jedna druhou (šetří to kód)

Příklad:

```
double area(double a, double b) {  
    return a*b; // Rectangle  
}
```

Přetížení metod

- několik metod může mít stejné jméno
- **musí** se lišit počtem, typem nebo pořadím parametrů
- **nesmí** se lišit návratový typ
- mohou volat jedna druhou (šetří to kód)

Příklad:

```
double area(double a, double b) {  
    return a*b; // Rectangle  
}
```

```
double area(double a) {  
    return area(a,a); // Square  
}
```

Vlastní konstruktor

- metoda bez návratového typu (ani `void`!)
- název shodný se jménem třídy
- uvede nově vytvořenou instanci do konzistentního stavu

```
public Person() {  
    this.name = "Josef_Novak";  
    this.age = 35;  
}
```

Konstruktor s parametry

- vhodný pro inicializaci datových položek

Příklad:

```
public Person(String name, int age) {  
    this.name = name;  
    this.age = age;  
}
```

- třída může mít libovolný počet konstruktorů
- musí se lišit počtem, typem nebo pořadím parametrů, aby se poznalo, který se má použít

- má ho každá třída automaticky
- inicializuje datové položky
 - čísla na nulu
 - referenční proměnné na `null`
- je-li definován jakýkoli vlastní konstruktor, pak implicitní konstruktor **nelze použít**

Toto lze:

```
public Person(String name, int age) {  
    this.name = name;  
    this.age = age;  
}  
public Person() {  
    this.name = "Josef_Novak";  
    this.age = 35;  
}  
...  
Person p1 = new Person("Frantisek_Dvorak", 45);  
Person p2 = new Person(); // Novak
```

Toto **nelze**:

```
public Person(String name, int age) {  
    this.name = name;  
    this.age = age;  
}  
...  
Person p1 = new Person("Frantisek_Dvorak", 45);  
Person p2 = new Person(); // beze jmena
```

Toto **nelze**:

```
public Person(String name, int age) {  
    this.name = name;  
    this.age = age;  
}
```

...

```
Person p1 = new Person("Frantisek_Dvorak", 45);
```

```
Person p2 = new Person(); // beze jména
```

(je definován vlastní konstruktor, konstruktor bez parametrů ale nikoli, znamenalo by to volání implicitního konstruktoru)

Přístupové modifikátory

týkají se metod, atributů a konstruktorů

Přístupové modifikátory

týkají se metod, atributů a konstruktorů

```
public int x;
```

atribut mohou modifikovat jiné třídy

Přístupové modifikátory

týkají se metod, atributů a konstruktorů

```
public int x;
```

atribut mohou modifikovat jiné třídy

```
public int compute(int a, int b) { ... }
```

metodu mohou volat jiné třídy

Přístupové modifikátory

týkají se metod, atributů a konstruktorů

```
public int x;
```

atribut mohou modifikovat jiné třídy

```
public int compute(int a, int b) {...}
```

metodu mohou volat jiné třídy

```
private Person(int a, int b) {...}
```

konstruktor může volat jen třída sama

Speciální třídy, které reprezentují výjimečný stav, do kterého se program může dostat

výjimka se „vyhodí“ klíčovým slovem `throw`

metoda, která za nějakých okolností vyhazuje výjimky se označuje klíčovým slovem `throws`

- běh programu dál nepokračuje
- výjimku může „odchytit“ jiná část programu, ze které se metoda vyhazující výjimky volala
- pokud výjimka není zachycena, program končí výpisem parametrů vyhozené výjimky.

(poněkud abstraktní) příklad výjimky

```
class Matrix{  
    ...  
    public Vector solve (Vector rhs) throws Exception {  
        ...  
        if (singular(this))  
            throw new Exception("Matrix_is_singular");  
        ...  
        return solution;  
    }  
}
```

(poněkud abstraktní) příklad výjimky

```
for (int i = 0; i < matrices.length; i++)
{
    try
    {
        Console.WriteLine(matrices[i].solve(r));
    }
    catch (Exception e)
    {
        Console.WriteLine("Could not solve one of the matrices,
        but pushing on anyway");
    }
}
```

Zlomek

```
class Fraction{  
    int numer; //citatel  
    int denom; //jmenovatel  
    ...  
}
```

Konstruktor

- neměl by umožnit nulový jmenovatel

```
public Fraction() {  
    denom = 1;  
}  
  
public Fraction(int n, int d) throws Exception {  
    if (d == 0)  
        throw new Exception();  
    numer = n;  
    denom = d;  
}
```

Zjednodušení

```
void simplify() {  
    int a = numer;  
    int b = denom;  
    while (b!=0) {  
        int nb = a%b;  
        a = b;  
        b = nb;  
    }  
    numer /= a; // a je nejvetsi spolecny delitel  
    denom /= a;  
}
```

Vypsání

```
void print() {  
    Console.WriteLine(numer + "/" + denom);  
}
```

```
public Fraction multiply(Fraction f) {  
    int n = numer * f.numer;  
    int d = denom * f.denom;  
    Fraction r = new Fraction(n, d);  
    r.simplify();  
    return result;  
}
```

```
public Fraction multiply(Fraction f) {  
    int n = numer * f.numer;  
    int d = denom * f.denom;  
    Fraction r = new Fraction(n, d);  
    r.simplify();  
    return result;  
}
```

```
public Fraction add(Fraction f) {  
    int n = numer * f.denom + f.numer*denom;  
    int d = denom * f.denom;  
    return new Fraction(n,d);  
}
```



```
...  
Fraction f1 = new Fraction(3, 5);  
Fraction f2 = new Fraction(2, 5);  
Fraction f3 = f1.add(f2);  
f3 = f3.multiply(f1);  
f3.print(); // 3/5  
...
```

```
...  
Fraction f1 = new Fraction(3, 5);  
Fraction f2 = new Fraction(2, 5);  
Fraction f3 = f1.add(f2);  
f3 = f3.multiply(f1);  
f3.print(); // 3/5  
...
```

- nemusím vědět co je uvnitř metod `add` a `multiply`
- nemusím vědět že metoda `simplify` vůbec existuje
- zápis programu je poměrně abstraktní - zapisujeme výroky/příkazy o zlomcích namísto výroků/příkazů o číslech

- organizace paměti
- dědičnost tříd