

# OOP pokračování

- Objektový polymorfismus v jazycích Java, Object Pascal a C++
- Objektové konstrukce v programovacích jazycích
- Jak jsou dědičnost a polymorfismus implementovány v překladači
- Příklady využití polymorfismu
- Násobná dědičnost a rozhraní

Př. 1JavaZvirata

```

class Animal {
    String type ;
    Animal() {                type = "animal ";}
    String sounds() {        return "not known";}
    void prnt() {            System.out.println(type + sounds());}
}
class Dog extends Animal {
    Dog() {                  type = "dog "; }
    String sounds() {        return "haf"; }
}
class Cat extends Animal {
    Cat() {                  type = "cat "; }
    String sounds() {        return "miau"; }
}
public class Anim {
    public static void main(String [] args) {
        Animal notknown = new Animal();
        Dog filipes = new Dog();
        Cat tom = new Cat();
        tom.prnt();
        filipes.prnt();
        notknown.prnt();
    } } //konec

```

## Př.2PascalZvířata

```
program Zvirata;  
{$APPTYPE CONSOLE}  
uses SysUtils;  
type  
    Uk_Zvire = ^Zvire;  
    Zvire = object  
        Druh   : String;  
        procedure Inicializuj;  
        function Zvuky: String;  
        procedure Tisk;  
    end;  
    Uk_Pes = ^Pes;  
    Pes = object(Zvire)  
        procedure Inicializuj;  
        function Zvuky: String;  
    end;  
    Uk_Kocka = ^Kocka;  
    Kocka = object(Zvire)  
        procedure Inicializuj;  
        function Zvuky: String;  
    end;
```

```

{-----Implementace metod-----}
  procedure Zvire.Inicializuj;
  begin Druh := 'Zvire '
  end;
  function Zvire.Zvuky: String;
  begin Zvuky := 'nezname'
  end;
  procedure Zvire.Tisk;
  begin writeln(Druh, Zvuky);
  end;

  procedure Pes.Inicializuj;
  begin Druh := 'Pes '
  end;
  function Pes.Zvuky: String;
  begin Zvuky := 'steka'
  end;

  procedure Kocka.Inicializuj;
  begin Druh := 'Kocka '
  end;
  function Kocka.Zvuky: String;
  begin Zvuky := 'mnouka'
  end;

```

```

{-----Deklarace objektuu-----}
  var
    U1, U2, U3: Uk_Zvire;
    Nezname: Zvire;
    Micka: Kocka;
    Filipes: Pes;

{-----Hlavni program-----}
begin
  Filipes.Inicializuj; Filipes.Tisk; { !!!??? }
  new(U1); U1^.Inicializuj; U1^.Tisk;
  Micka.Inicializuj; writeln(Micka.Druh, Micka.Zvuky);
  readln;
end.

```

Konec Př.1ObjectPascalZvířata

Lze zařídit stejné chování i Java programu?

```

class Animal { //program AnimS
    String type ;
    Animal() { type = "animal ";}
    static String sounds() { return "not known";}
    void prnt() { System.out.println(type + sounds());}
}
class Dog extends Animal {
    Dog() { type = "dog "; }
    static String sounds() { return "haf"; }
}
class Cat extends Animal {
    Cat() { type = "cat "; }
    static String sounds() { return "miau"; }
}
public class Anim {
    public static void main(String [] args) {
        Animal notknown = new Animal();
        Dog filipes = new Dog();
        Cat tom = new Cat();
        tom.prnt();
        filipes.prnt();
        notknown.prnt();
    }
}

```

**?co to ted udela?**

```
class Animal { //
    String type ;
    Animal() {          type = "animal " ;}
    final String sounds() {    return "not known";}
    void prnt() {      System.out.println(type + sounds());}
}
class Dog extends Animal {
    Dog() {          type = "dog " ; }
    final String sounds() {    return "haf"; }
}
class Cat extends Animal {
    Cat() {          type = "cat " ; }
    final String sounds() {    return "miau"; }
}
public class Anim {
    public static void main(String [] args) {
        Animal notknown = new Animal();
        Dog filipes = new Dog();
        Cat tom = new Cat();
        tom.prnt();
        filipes.prnt();
        notknown.prnt();
    }
}
```

?co to ted udela?

```
class Animal { //
    String type ;
    Animal() {                type = "animal "};
    private final String sounds() {        return "not known";}
    void prnt() {        System.out.println(type + sounds());}
}
class Dog extends Animal {
    Dog() {                type = "dog "};
    private final String sounds() {        return "haf"; }
}
class Cat extends Animal {
    Cat() {                type = "cat "}; }
    private final String sounds() {        return "miau"; }
}
public class Anim {
    public static void main(String [] args) {
        Animal notknown = new Animal();
        Dog filipes = new Dog();
        Cat tom = new Cat();
        tom.prnt();
        filipes.prnt();
        notknown.prnt();
    }
}
```



Př.Zvirata1x řešení Borland Pascal

```
Zvire = object
    Druh : String;
    constructor Inicializuj;          {!!!}
    function Zvuky: String; virtual; {!!!}
    procedure Tisk;
end;
```

...

```
Pes = object(Zvire)
    constructor Inicializuj;          {!!!}
    function Zvuky: String; virtual; {!!!}
end;
```

...

```
Filipes.Inicializuj;
Filipes.Tisk;          { !!!??? }
```

```
new(U1);
U1^.Inicializuj;
U1^.Tisk; . . .
```

# Př.Zvirata1 řešení Object Pascal

```
Zvire = class
    Druh   : String;
    constructor Inicializuj;           {!!!}
    function Zvuky: String; virtual;  {!!!}
    procedure Tisk;
end;

Pes = class(Zvire)
    constructor Inicializuj;           {!!!}
    function Zvuky: String; override; {!!!}
end;

...

Filipes := Pes.Inicializuj;
Filipes.Tisk;                          { !!!??? }

new(U1);
U1^:= Zvire.Inicializuj;
U1^.Tisk;

Micka := Kocka.Inicializuj;
writeln(Micka.Druh, Micka.Zvuky); . . .
```

# Vizuální programování

## Kombinace OOP a Windows

Př.2DelphiZvirata

- Formulář
- Paleta komponent
- Objekty
- Vlastnosti
- Události

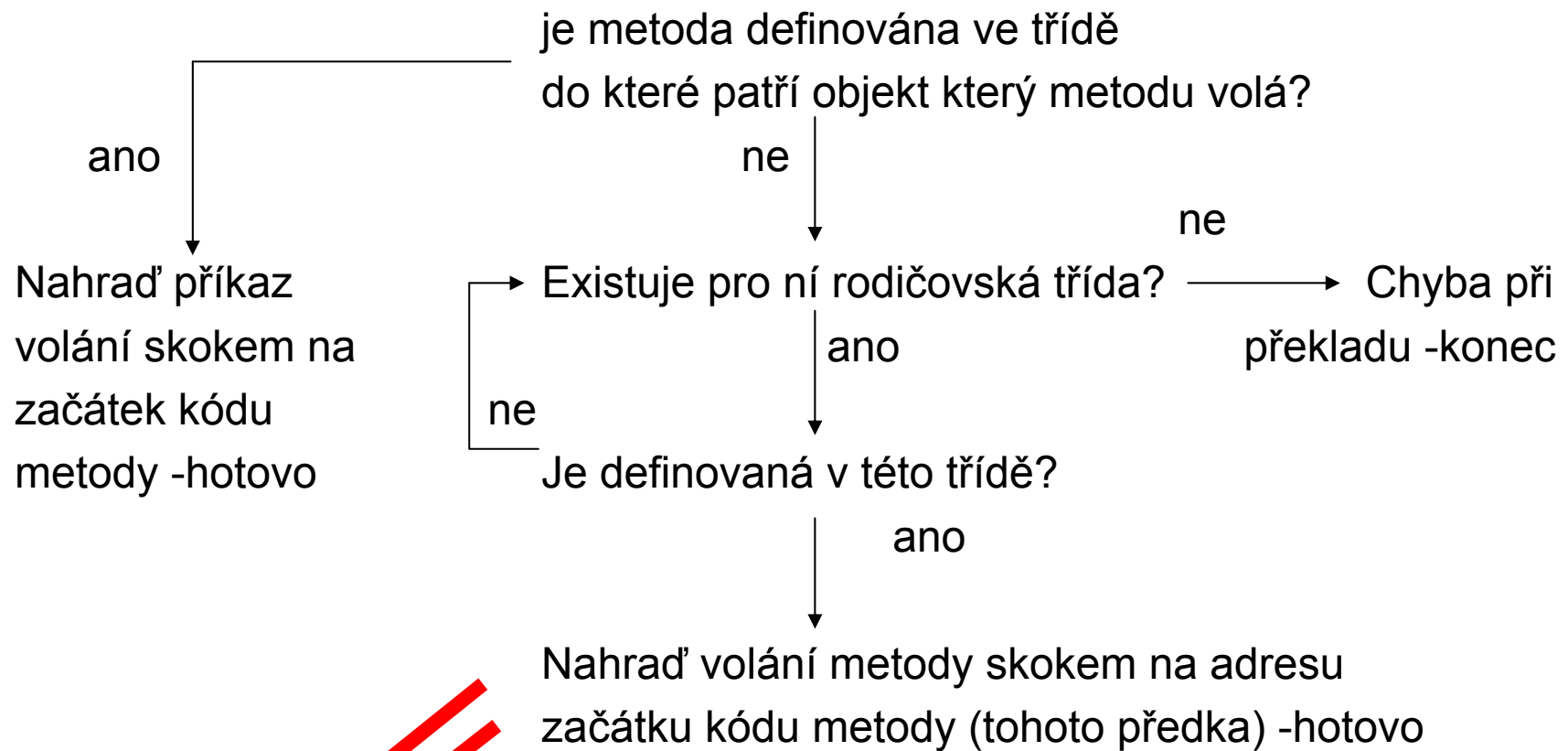
### Př.3CZvirata

```
class Zvire {public: char Druh[20]; void Inicializuj(); int Zvuky(); void Tisk(); };
class Pes:public Zvire {public: void Inicializuj(); int Zvuky(); };
class Kocka:public Zvire {public: void Inicializuj(); int Zvuky(); };
void Zvire::Inicializuj() { strcpy(Druh, "zvire");}
int Zvire::Zvuky() { return 10;}
void Pes::Inicializuj() { strcpy(Druh, "pes");}
int Pes::Zvuky() { return 11;}
void Kocka::Inicializuj() { strcpy(Druh, "kocka");}
int Kocka::Zvuky() { return 12;}
void Zvire::Tisk() {cout << Druh;
    if (Zvuky()==10) cout << " nezname";
    if (Zvuky()==11) cout << " steka";
    if (Zvuky()==12) cout << " mnouka";
    cout << "\n";
}
main()
{
    Zvire Nezname;
    Pes Filipes;
    Kocka Micka;
    Filipes.Inicializuj(); Filipes.Tisk();
    Micka.Inicializuj(); Micka.Tisk();
    getchar();
    return 0;
}
```

## Př.4CZvirata

```
class Zvire {
public:
    char Druh[20];
    void Inicializuj();
    virtual int Zvuky() { return 10; };
    void Tisk() ;
};
class Pes:public Zvire {
public:
    void Inicializuj();
    int Zvuky() { return 11; };
};
. . .
int main()
{ Zvire Nezname;
  Pes Filipes;
  Kocka Micka;
  Filipes.Inicializuj(); Filipes.Tisk();
  Micka.Inicializuj(); Micka.Tisk();
  getchar();
  return 0;
}
```

# Dědičnost a statická (brzká) vazba



Obsahuje-li tato metoda volání další metody, je tato další metoda metodou předka, i když potomek má metodu, která ji překrývá – viz Zvirata

# Dědičnost a dynamická (pozdní) vazba

- Realizovaná pomocí virtuálních metod
- Při překladu se vytváří pro každou třídu tzv. datový segment obsahující:
  - údaj o velikosti instance a datových složkách
  - údaj o předku třídy
  - ukazatele na kódy metod s pozdní vazbou (VMT)
- Před prvním voláním metody musí být v dané instanci zavolána (případně implicitně) speciální inicializační metoda – constructor
- Constructor vytvoří spojení (při běhu programu) mezi instancí volající konstruktor a VMT. Součástí instance je místo pro ukazatel na VMT třídy, ke které patří instance
- Volání metody je realizováno nepřímým skokem přes VMT
- Pokud není znám typ instance (objektu) při překladu (viz případ ukazatel na typ předka lze použít k odkazu na potomka), umožní VMT polymorfní chování

# Dědičnost a dynamická (pozdní) vazba

Př CPP zápis.

```
class A { public: void f( ); virtual void g( ); ... private: double x, y; } ;
```

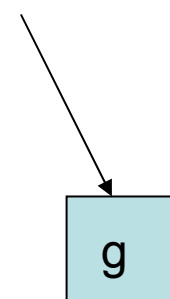
```
class B: public A { public: void f( ); virtual void h( ); ... private; double z; } ;
```

Alokované místo objektu a třídy A

Místo pro x

místo pro y

VMT ukazatel



VMT pro a

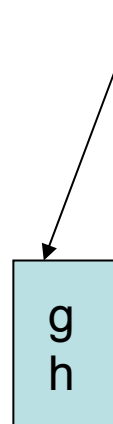
Alokované místo objektu b třídy B

Místo pro x

místo pro y

místo pro z

VMT ukazatel



VMT pro b



# OOP konstrukce C++

Třídy odvozeny ze struct C

```
class jméno třídy {  
    private: privátní položky a metody  
    protected: chráněné " "  
    public: veřejné " "  
}
```

- data members = datové členy (elementy)
- member functions = členské funkce
- V class jsou implicitně private
- V struct jsou implicitně public
- V union jsou implicitně public a nedá se to změnit

instance třídy -statické  
-dynamické v haldě  
-dynamické v zásobníku

datové elementy mohou být dynamické (new / delete)

funkční elementy lze definovat:

- hlavička fce uvedena v definici třídy, deklarace fce je mimo třídu
- celé v definici třídy (včetně těla fce, inlined)

# OOP konstrukce C++

**Konstruktory** pojmenovány shodně s třídou

- inicializují datové elementy
- implicitně volatelné
- může jich být více pro třídu

**Destruktory** pojmenovány ~jméno třídy

- implicitně volatelné
- C++ nemá čistič paměti

**Řízení dědičnosti**

- konstrukcí friend
- private/public děděné elementy v potomkovi
- možnost násobné dědičnosti

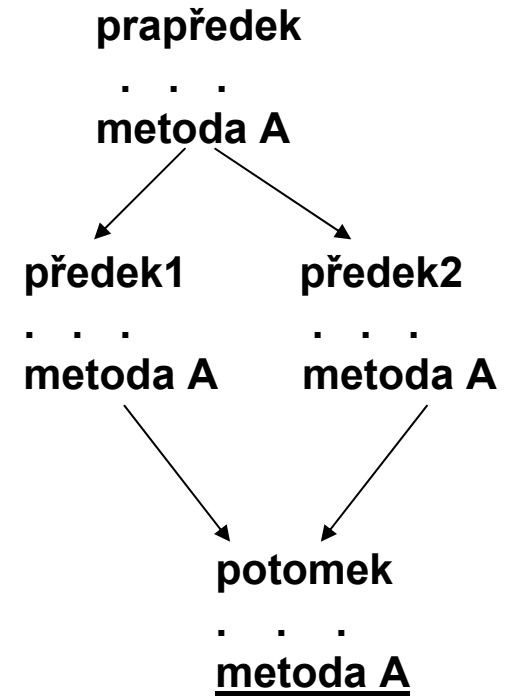
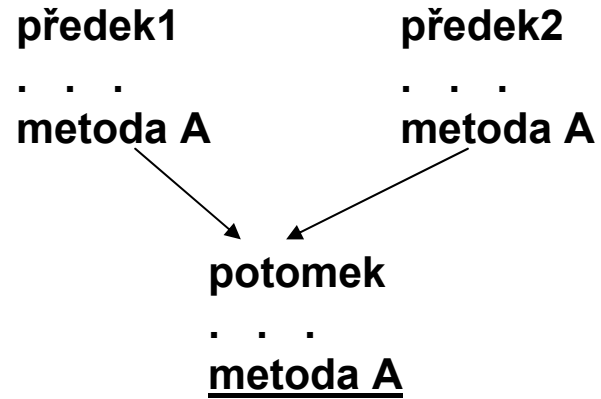
```
class potomek:    [virtual][private/public] předek1
                  [,virtual][private/public] předek2
                  ...
                  ]
{ data a metody
  friend přítel
};
```

# OOP konstrukce C++

přístup v rodiči	modifikace přístupu v potomkovi	
	public	private
public	public	private
protected	protected	private
private	nepřístupný	nepřístupný

```
Př.    CLASS Předek1 { PUBLIC: int p11;  
        PROTECTED: int p12;  
        PRIVATE: int p13;  
};  
CLASS Předek2 { PUBLIC: int p21;  
        PROTECTED: int p22;  
        PRIVATE: int p23;  
};  
CLASS Potomek: PUBLIC Předek1, PRIVATE Předek2; {  
        PUBLIC: int pot1;  
        PROTECTED: int pot2;  
        PRIVATE: int pot3;  
};
```

# OOP konstrukce C++



**Řešení:**

- plným jménem (jméno předka :: )
- virtual předek1,
- virtual předek2

## Grafický příklad - toolbox

Třída Location - umístění bodu na obrazovce

datové elementy -X souřadnice

-Y souřadnice

metody -konstruktor

-GetX

-GetY

Třída Point - potomek Location

datové elementy -DirX souřadnice x směru pohybu

-DirY souřadnice y směru pohybu

metody -konstruktor

-MoveTo(NewX, NewY) přesun do nového místa

-Move pohyb ve směru DirX,DirY

-Draw vykreslení bodu v novém místě

## Grafický příklad - toolbox

Třída Circle -potomek třídy Point  
datové elementy -R poloměr  
metody  
-konstruktor  
-Grow zvětšování kruhu  
-Move pohyb ve smeru DirX,DirY  
-Draw vykreslení kruhu v novém místě

Třída Square -potomek třídy Point  
datové elementy -Site délka strany  
metody  
-konstruktor  
-Grow zvětšování čtverce  
-Move pohyb ve smeru DirX,DirY  
-Draw vykreslení čtverce v novém místě

Př. 5DelphiObrazce

Př. 6CPPObrazce

# Objektové vlastnosti C#

- Používá **class** i **struct**
- Pro dědění při definici tříd používá CPP syntax  

```
public class NovaTrida : RodicovskaTrida { . . . }
```
- V podtřídě lze nahradit metodu zděděnou od rodiče zápisem  

```
new definiceMetody;
```

ta pak zakryje děděnou metodu stejného jména.  
Metodu z rodiče lze ale přesto volat pomocí zápisu např.  

```
base.vykresli( );
```
- Dynamická vazba je u metody báze třídy povinně označena  

```
virtual
```

a u metod odvozených tříd povinně označena  

```
override
```

(převzato z Objekt Pascalu)

Př.

# Objektové vlastnosti C#

```
public class Obrazec {  
    public virtual void Vykresli ( ) { . . . }  
    . . .  
}  
public class Kruh : Obrazec {  
    public override void Vykresli ( ) { . . . }  
    . . .  
}  
public class Ctverec : Obrazec {  
    public override void Vykresli ( ) { . . . }  
    . . .  
}
```

- **Má abstraktní metody, nepř, abstract public void Vykresli( );  
ty pak musí být implementovány ve všech potomcích a třídy s abstract metodou musí  
být označeny abstract**
- **Kořenem všech tříd je Object jako u Javy**
- **Nestatické vnořené třídy nejsou zavedeny**



# Java rozhraní

- Posouvá na vyšší úroveň princip abstraktních tříd
- Rozhraní částečně nahrazuje násobnou dědičnost
- Klíčovým slovem *interface* lze oddělit rozhraní třídy od třídy, která je implementuje
- Je „obdobou“ abstraktní třídy
  - ❖ definuje jen hlavičky metod, všechny bez implementace
  - ❖ Nemůže deklarovat žádné proměnné
  - ❖ **Třída může implementovat = „zdědit“ více než jedno rozhraní**
  - ❖ Nepříbuzné třídy mohou implementovat stejné rozhraní (nezávisle na stromu hierarchie tříd), tj. rozhraní nevynucuje příbuzenské vztahy a způsob jeho implementace se může lišit
  - ❖ Rozhraní může dědit jiné rozhraní pomocí *extends*. Implementující třída pak musí implementovat i metody z předka rozhraní
- Uplatní se pokud třídy nemohou mít společného předka (např. jsou odvozeny z knihovnických tříd) a mají vykonávat podobné funkce

# Java rozhraní

## Deklarace

```
[ public ] interface Jméno {           // pri public je dostupne vsem z baliku
    hlavicka metody1;                 // pouze jmeno, typ a pocet parametru
    typ jmenopromenne_1 = hodnota;     //implicitne je public,final,static
    hlavicka metody2;
    . . .
    hlavicka metodyn;
    typ jmenopromenne_m = hodnota;     //neni to promenna instance
}
```

## Implementace

```
[ public ] class Trida [ extends Nadtrida ] implements Jméno {
    tělo Třídy //včetně implementace všech metod rozhraní
               //metody musí být deklarovány jako veřejné
}
```

Při implementaci z více rozhraní se uvede **Jméno1, Jméno2, Jméno\_n**

# Java rozhraní

- Lze deklarovat referenční proměnnou typu *rozhraní* (které implementuje nějaká třída)
- Ta může odkazovat na jakýkoliv objekt třídy implementující rozhraní
- Pomocí této referenční proměnné lze ale přistupovat jen k těm metodám instance, které deklarace rozhraní definuje
- Volání metody prostřednictvím proměnné referující na rozhraní způsobí realizaci verze metody spojené s objektem, který je instancí třídy implementující toto rozhraní

Př. 7RozhraniZvirata

[Soubor Zvuky.java](#)

```
public interface Zvuky { public String sounds(); }
```

[Soubor RAnim.java](#)

```

class Animal implements Zvuky { //
    String type ;
    Animal() {                type = "animal ";}
    public String sounds() {   return "not known";}
    void prnt() {              System.out.println(type + sounds());}
}
class Dog extends Animal implements Zvuky {
    Dog() {                    type = "dog "; }
    public String sounds() {    return "haf"; }
}
class Cat extends Animal implements Zvuky {
    Cat() {                    type = "cat "; }
    public String sounds() {    return "miau"; }
}
public class RAnim {
    public static void main(String [] args) {
        Animal notknown = new Animal();
        Zvuky z = new Animal();
        Dog filipes = new Dog();
        Cat tom = new Cat();
        System.out.println(z.sounds());
        z = filipes;
        System.out.println(z.sounds());
    }
}

```

?co to ted udela?

## Java rozhraní Př. 71 RozhraniFronta

Pomocí rozhraní realizuje fronty pro ukládání znaků

- Pevné délky
- Zkruhované
- Zvětšující délku pokud je třeba

```
// interface fronta znaku soubor IZF.java
```

```
public interface IZF {  
    // Put znak do fronty  
    void put(char ch);  
  
    // Get znak z fronty  
    char get();  
}
```

## Java rozhraní Př. 71RozhraniFronta

```
//Fronta s pevnou delkou Soubor FixF.java
class FixF implements IZF {
    private char q[]; // array pro frontu
    private int putloc, getloc; // put a get indexy
    public FixF(int size) { // vytvori prazdnou frontu delky size
        q = new char[size+1]; // alokace pameti pro frontu
        putloc = getloc = 0;
    }
    public void put(char ch) { // Put znak do fronty
        if(putloc==q.length-1) {
            System.out.println(" -- Fronta je plna!!");
            return;
        }
        putloc++;
        q[putloc] = ch;
    }
    public char get() { // Get znak z fronty
        if(getloc == putloc) {
            System.out.println(" -- Fronta je prazdna!!");
            return (char) 0;
        }
        getloc++;
        return q[getloc];
    }
}
```

```

//Kruhova fronta Soubor KruhF.java
class KruhF implements IZF {
    private char q[]; // array pro frontu
    private int putloc, getloc; // put a get indexy
    public KruhF(int size) { // vytvoreni prazdne fronty
        q = new char[size+1]; // alokace pameti pro frontu
        putloc = getloc = 0;
    }
    public void put(char ch) { // Putznak do fronty
        if(putloc+1==getloc | // Test plnosti fronty
            ((putloc==q.length-1) & (getloc==0))) {
            System.out.println(" -- Fronta je plna!!");
            return;
        }
        putloc++;
        if(putloc==q.length) putloc = 0; //zkruhovani
        q[putloc] = ch;
    }
    public char get() { // Get znak z fronty
        if(getloc == putloc) {
            System.out.println(" -- fronta je prazdna!!");
            return (char) 0;
        }
        getloc++;
        if(getloc==q.length) getloc = 0; // zkruhovani
        return q[getloc];
    }
}

```

```

// dynamicka fronta soubor DynF.java
class DynF implements IZF {
    private char q[]; // array pro frontu
    private int putloc, getloc; // put a get indexy
    public DynF(int size) { // vytvoreni prazdne fronty
        q = new char[size+1]; // alokace pameti pro frontu
        putloc = getloc = 0;
    }
    public void put(char ch) { // Putznak do fronty.
        if(putloc==q.length-1) { // zvetsi velikost fronty
            char t[] = new char[q.length * 2];
            for(int i=0; i < q.length; i++) //kopiruje prvky do nove fronty
                t[i] = q[i];
            q = t;
        }
        putloc++;
        q[putloc] = ch;
    }
    public char get() { // Get znak z fronty
        if(getloc == putloc) {
            System.out.println(" -- fronta je prazdna!!");
            return (char) 0;
        }
        getloc++;
        return q[getloc];
    }
}

```



```

//Hlavni program Soubor IFDemo.java
class IFDemo {
    public static void main(String args[]) {
        FixF q1 = new FixF(10);
        DynF q2 = new DynF(5);
        KruhF q3 = new KruhF(10);
        IZF iQ;
        char ch;
        int i;
        iQ = q1;
        for(i=0; i < 5; i++) // Put znaky do fronty pevne delky
            iQ.put((char) ('J' + i));
        System.out.print("Ve fronte je: "); // Vypis obsah fronty
        for(i=0; i < 5; i++) {
            ch = iQ.get();
            System.out.print(ch);
        }
        System.out.println();
        iQ = q2;
        for(i=0; i < 10; i++) // Put znaky do dynamicke fronty
            iQ.put((char) ('Z' - i));
        System.out.print("Ve fronte je: "); // Vypis obsah fronty
        for(i=0; i < 10; i++) {
            ch = iQ.get();
            System.out.print(ch);
        }
        System.out.println();
    }
}

```

```

iQ = q3;
    for(i=0; i < 10; i++) // Put znaky do kruhove fronty
        iQ.put((char) ('A' + i));
    System.out.print("Obsah kruhove fronty: ");
    for(i=0; i < 10; i++) {
        ch = iQ.get();
        System.out.print(ch);
    }
    System.out.println();
    for(i=10; i < 20; i++) //Vloz dalsi znaky do kruhove fronty
        iQ.put((char) ('A' + i));
    System.out.print("Obsah kruhove fronty: ");
    for(i=0; i < 10; i++) {
        ch = iQ.get();
        System.out.print(ch);
    }
    System.out.println("Vyber z kruhove fronty.");
    for(i=0; i < 20; i++) {
        iQ.put((char) ('A' + i));
        ch = iQ.get();
        System.out.print(ch);
    }
}
}
//Konec Souboru IFDemo.java

```

# Java rozhraní

- Pokud třída Předek implementuje rozhraní, pak z ní děděním vzniklá třída Potomek zdědí toto rozhraní beze změny. Metody z rozhraní budou (pokud je nepřekryjeme) přístupné v obou třídách. Budou to metody z Předka.
- Zdědění třídy, která implementuje rozhraní neomezuje Potomka v možnosti implementovat libovolně další rozhraní
- Za běhu lze operátorem *instanceof* testovat, zda určitá třída implementuje určité rozhraní. Zápis podmínky má tvar:

jménoObjektu instanceof JménoRozhraní

to dovoluje vytvářet a překládat programy, které využívají zatím ještě neimplementované metody

# Java rozhraní

Př.

if (filipes instanceof Zvuky) ...

if (q2 instanceof Zvuky) ...

- Operátor instanceof lze použít i k testování, zda referenční proměnná je typu příslušné třídy

Př.

if (filipes instanceof Pes) ...

# OOP konstrukce v jazyce ADA

--Objektový typ je založen na konstrukci záznamu

```
type Datum is tagged
  record
    Den      : Den_Subtyp ;
    Mesic    : Mesic_Subtyp ;
    Rok      : Rok_Subtyp ;
  end record ;
```

--Konstrukce potomka

```
type Komplet_Datum is new Datum with
  record
    Den_v_tydnu : Den_v_tydnu_Typ ;
  end record ;
```

--Popis metod následuje za konstrukcí record

```
procedure Display(D: Datum) is . . . ;
procedure Display( CD : in Complete_Datum ) is
  Display(Datum(CD)); . . .
```

--Lze konvertovat objekty odvozeného typu na rodičovský typ

```
Text_IO.Put(Day_Of_Week_Type'Image(CD.Day_Of_Week));
end Display;
```

# OOP konstrukce v jazyce ADA

- Zapouzdření realizuje pomocí *private části recordu*
- Polymorfismus realizuje pomocí *Class-wide typu, ten* umožňuje odkazovat na celou rodinu typů

```
D : Datum'Class := Datum'(6, Jul, 1415); --nutna inicializace (zpřesní typ)
D := Complete_Datum'(6, Jul, 1415, Mon) ; --neni již dovoleno,měnilo by typ
type Ptr is access Datum'Class ; --může ukazovat na objekty obou typů
```

```
procedure Dynamic_Dispatching_Demo is
  A : array(1 .. 2) of Ptr ;
begin
  A(1) := new Datum'(6, Jul, 1415);
  A(2) := new Complete_Datum'(14, Jul, 1789, Wen) ;
  for I in A'Range loop
    Display( A(I).all ) ; --rozhodne se až při výpočtu
  end loop ;
end Dynamic_Dispatching_Demo ;
```

```
procedure Show(X : in Datum'Class) is
begin
  Display(X) ; --rozhodne až při výpočtu
end Show; -- Použití v parametrech podprogramu:
```

# OOP konstrukce v jazyce ADA PŘ. 8ZVÍŘATA

```
with TEXT_IO; use TEXT_IO;
```

```
package ZVIRATA is
```

```
  type STRING10 is new STRING(1..10);
```

```
  type ZVIRE is tagged
```

```
    record DRUH: STRING10;
```

```
  end record;
```

```
  type UK_ZVIRE is access ZVIRE;
```

```
  procedure INICIALIZUJ(O: out ZVIRE);
```

```
  function ZVUKY(O:ZVIRE) return STRING10;
```

```
    procedure TISK(O:ZVIRE); --musi byt u zvirete
```

```
  type PES is new ZVIRE with record MAJITEL: STRING10; end record;
```

```
  type UK_PES is access PES;
```

```
  procedure INICIALIZUJ(O: out PES);
```

```
  function ZVUKY(O:PES) return STRING10;
```

```
end ZVIRATA;
```

**package body ZVIRATA is**

**procedure INICIALIZUJ(O: out ZVIRE) is**  
    **begin O.DRUH := "zvire ";**  
    **end;**

**procedure INICIALIZUJ(O: out PES) is**  
    **begin O.DRUH := "pes ";**  
    **end;**

**function ZVUKY(O:ZVIRE) return STRING10 is**  
    **begin return "nezname ";**  
    **end;**

**function ZVUKY(O:PES) return STRING10 is**  
    **begin return "steka ";**  
    **end;**

**procedure TISK(O:ZVIRE) is**  
    **begin PUT\_LINE(STRING(O.DRUH));**  
        **PUT\_LINE(STRING(ZVUKY(O)));**  
    **end;**

**end ZVIRATA;**



# OOP konstrukce v jazyce ADA Příklad 8 ZVÍŘATA

```
with ZVIRATA; use ZVIRATA;  
procedure POK_ZV is
```

```
    UZ: UK_ZVIRE;  
    UP: UK_PES;  
    NEZNAME: ZVIRE;  
    FILIPES: PES;
```

```
begin  
    INICIALIZUJ(FILIPES);  
    INICIALIZUJ(NEZNAME);  
    TISK(NEZNAME);  
    TISK(FILIPES);  
    UP:=new PES;  
    INICIALIZUJ(UP.all);  
    TISK(UP.all);  
end POK_ZV;
```

Co tiskne?

# OOP konstrukce v jazyce ADA Př. 9ZVÍŘATA1

```
with TEXT_IO; use TEXT_IO;
```

```
package ZVIRATA1 is
```

```
  type STRING10 is new STRING(1..10);
```

```
  type ZVIRE is tagged
```

```
    record DRUH: STRING10;
```

```
  end record;
```

```
  type UK_ZVIRE is access ZVIRE;
```

```
  procedure INICIALIZUJ(O: out ZVIRE);
```

```
  function ZVUKY(O:ZVIRE) return STRING10;
```

```
  type PES is new ZVIRE with record MAJITEL: STRING10; end record;
```

```
  type UK_PES is access PES;
```

```
  procedure INICIALIZUJ(O: out PES);
```

```
  function ZVUKY(O:PES) return STRING10;
```

```
  procedure TISK(O:ZVIRE'CLASS);
```

```
end ZVIRATA1;
```

package body ZVIRATA1 is –Hl.program je opet POK\_ZV.adb jako u 8ADAZVIRATA

```
procedure INICIALIZUJ(O: out ZVIRE) is
  begin O.DRUH := "zvire  ";
  end;
```

```
procedure INICIALIZUJ(O: out PES) is
  begin O.DRUH := "pes  ";
  end;
```

```
function ZVUKY(O:ZVIRE) return STRING10 is
  begin return "nezname  ";
  end;
```

```
function ZVUKY(O:PES) return STRING10 is
  begin return "steka  ";
  end;
```

```
procedure TISK(O:ZVIRE'CLASS) is
  begin PUT_LINE(STRING(O.DRUH));
        PUT_LINE(STRING(ZVUKY(O)));
  end;
```

```
end ZVIRATA1;
```

# OOP konstrukce v jazyce ADA

Abstraktní typy a abstraktní podprogramy

```
package P is
    type Abstract_Datum is abstract tagged null record ;
    procedure Display(AD : Abstract_Datum) is abstract ;
end P ;
```

Abstraktní typ je použitelný jen k odvozování typů (ne k deklaraci)

Abstraktní procedura nemá tělo

```
with P ; use P ;
package Q is
    . . .
    type Ptr is access Abstract_Datum ;
    type Datum is new Abstract_Datum with record . . . --D,M,R
    type Complete_Datum is new Datum with record . . . --Den_v_tydnu
    procedure Display(D : Datum) ; --v těle modulu bude její tělo
    procedure Display(CD : Complete_Datum) ;-- “
end Q ;
```

Můžeme psát modul P se všemi jeho abstraktními procedurami před tím, než programujeme modul Q s detailním tvarem odvozených typů

Vlastnosti jazyka	Delphi	Java	C++	Visual Basic	ADA
Dědičnost	+	+	+		+
Násobná dědičnost			+		
Rozhraní	+	+			
Virtuální metody	+	+	+		+
Generické typy		Budou	+		+
Statické proměnné		+	+		
Statické metody	+	+	+		+
Abstraktní metody	+	+	+		+
Podpora paralelních výpočtů	+	+			+
Čistič paměti		+			+
Variantní typy	+			+	+
Statická kontrola typů	+	+	+		+
Zpracování výjimek	+	+	+	+	+
Přetěžování funkcí	+	+	+		+
Přetěžování operátorů			+		+