OOP - Objektově Orientované Programování

Programovat lze klasicky neobjektově - tedy s využitím jen primitivních typů a statického kontextu - tj. statických atributů, statických metod, statických inicializátorů a výčtů.
Leč objektová metodika (analýza OOA, design OOD, programování OOP), výrazně usnadňuje vytváření složitého softwaru.
Objektový přístup je velmi přirozený, neboť pohlíží na softwarové entity a vztahy často tak, jakoby byly z materiálního světa, což programátorům usnadňuje pochopení a tvorbu konstrukcí pomocí analogií.

Java vyniká těmito důležitými rysy:

- **Zavedení jasného pořádku a etikety.**
- **Přísné typování jmen.**
- **Abstrakci (abstraction), tj. zobecnění pohledů a částečná řešení.**
- **Dědičnost (inheritance), tj. získávání vlastností a schopností děděním z vhodné třídy a případně i dat nebo částečných požadavků z interfejsů.**
- **Polymorfismus (polymorphism), tj. jednotné zacházení s různými (mnohotvarými) objekty mající některé společné zděděné schopnosti.**
- **Zapouzdření (encapsulation), tj. skrývání části tříd a interfejsů.**

Pořádek

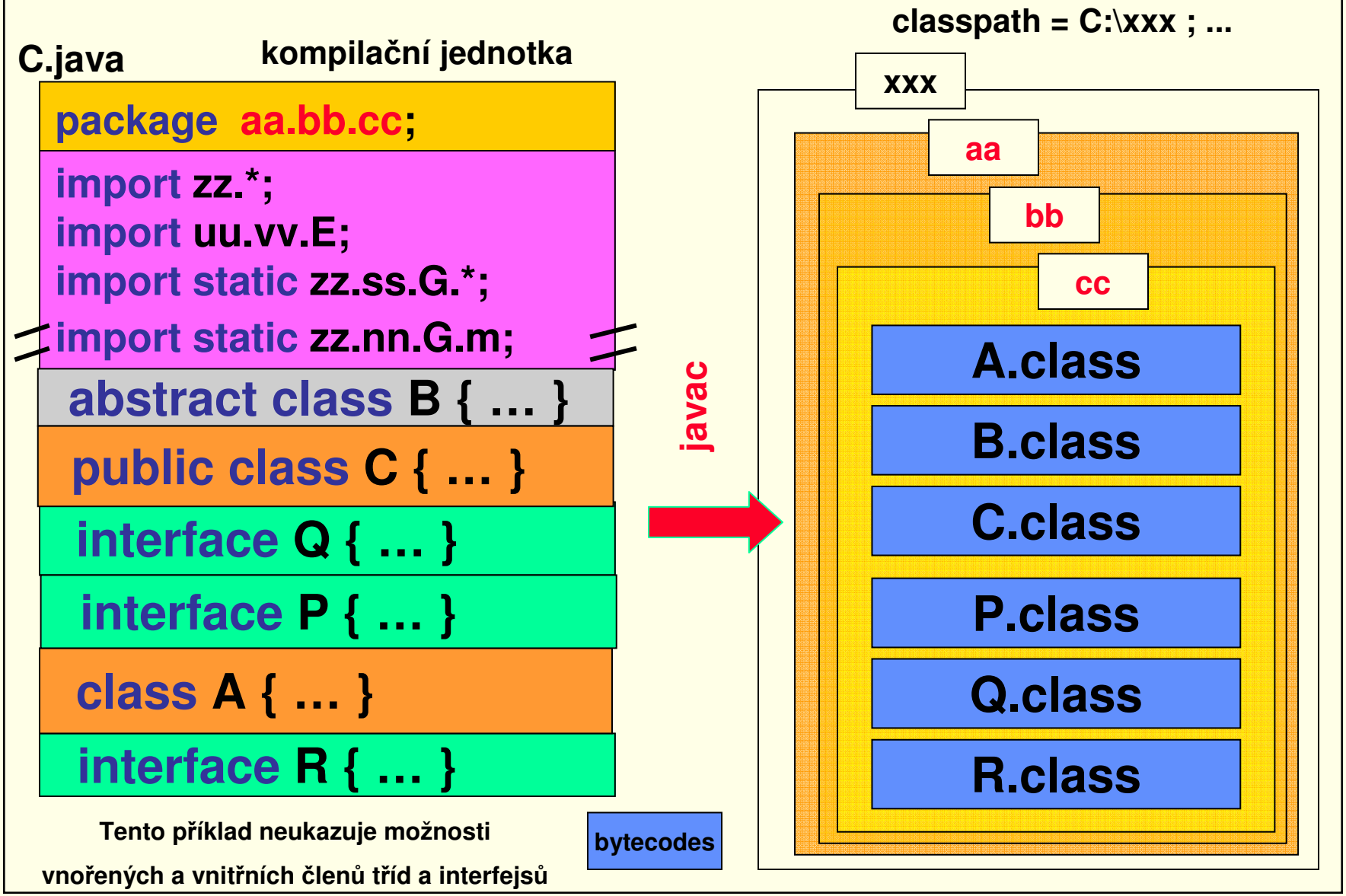
Třída (class), interfejs (interface, tj. rozhraní) a výčet (což je speciální třída enum) jsou v Javě základními stavebními kameny.
Ukládají se do tzv. balíčků (package), přístup k nim se usnadní import.

- **Základním balíčkem je java.lang, bez něho by Java vůbec nefungovala.**

- **Zdrojové soubory, tj. kompilační jednotky, mají extenzi .java a obsahují:**

  - **Jeden příkaz package - je vždy první. Lze ho sice vynechat – pak se jedná o tzv. noname space – nedoporučuje se.**
  - **Několik (i žádný) příkazů [static] import pro snadnější přístup k jiným balíčkům (import java.lang.*; je default).**
  - **Definice tříd, interfejsů a výčtů v libovolném pořadí.**
  - **Dokumentační komentáře před třídou, interfejsem a jejich složkami.**
  - **Ostatní komentáře (řádkové nebo blokové) kdekoli .**

Kompilace



Tento příklad neukazuje možnosti vnořených a vnitřních členů tříd a interfejsů

Pořádek

- Ve zdrojovém souboru může být nanejvýš jedna vrcholná (top) třída, výčet či interfejs s modifikátorem public - pak musí být jméno zdrojového souboru shodné se jménem oné třídy, výčtu či interfejsu.

- Kompilátor vytvoří tolik souborů bytekódu, kolik je ve zdrojovém tříd, výčtů a interfejsů (i vnitřních). Vzniklé soubory mají odpovídající jména s extenzí .class.

  (Vnitřní třídy či interfejsy mají vlastní jména složená ze sekvence jmen vnějších tříd či interfejsů oddělených dolarem.)

- Bytekódové soubory jsou přijatelné pro JVM, jen tehdy leží-li v těch adresářích jejichž cesta odpovídá operandu v příkazu package.

  Tzv. classpath, musí odkazovat na adresář v němž se nalézá adresář shodného jména s první složkou jména uvedeného v příkazu package.

Účel tříd a interfejsů

Třídy slouží jako:

- **Úložiště statického kontextu tj. statických položek tříd.**
- **Plán pro tvorbu objektů obsahující nestatické položky.**
- **Zdroj neprivátních členů pro dědictví potomkům.**
- **Privátní zdroj funkcionality potomků.**
- **Brána pro spuštění aplikace pomocí public static void main(String ...**

Interfejsy slouží jako norma či požadavek.

- **Nemají funkcionalitu.**
- **Nemají konstruktory ani inicializátory.**
- **Mohou mít jen abstraktní metody.**
- **Mohou mít jen finální statické atributy.**
- **Mohou být i prázdné – tzv. tagging Interfejsy.**

Typy

- **Jménům se v definici přiřadí neměnitelný typ, který může být:**
  - **primitivní – primitivních typů je jen 8.**
  - **referenční, tj. dle:**
    - **dle třídy nebo interfejsu,**
    - **pole (primitivních hodnot anebo referencí).**
  - **void (prázdný) - jen jako návratový typ metod.**

- **Konstruktory nemají vyznačený návratový typ – ten je dán jménem třídy - a musejí být nazvány přesně podle třídy.**

  Java nemá destruktory – ničení objektů zajišťuje garbage collector.

Jména

- **Všechny položky programu musejí být rozlišitelné různými jmény.**
  V některých případech se rozlišují položky shodných jmen dle principu "košile-kabát".

- **Třídy a interfejsy jsou rozlišitelné pomocí úplných jmen. Ta jsou dána prefixem dle příkazu package a jejich vlastním jménem.**

- **Atributy přejímají úplná jména tříd či interfejsů jako prefix vlastních jmen.**

- **Metody a konstruktory přejímají úplná jména tříd či interfejsů jako prefix vlastních signatur. Signatura metod i konstruktorů se skládá z jejich vlastních jmen a sekvenci typů jejich parametrů.**

- **Metody i konstruktory v téže třídě mohou mít shodná vlastní jména, avšak různé signatury - pak jsou tzv. přetížené (overload).**
  To je vhodné tehdy, mají-li obdobnou funkcionalitu.

Abstrakce a konkretizace

- **Abstrakce je důležitým pohledem na oba světy: ten reálný i ten virtuální, je základem taxonomie složitých systémů.**
- **Abstrakce má tvar stromu: směrem ke kořeni abstrahujeme (vypouštíme detaily), směrem k listům konkretizujeme (přidáváme a realizujeme detaily).**
- **Pomocí abstrakce lze vytvářet velmi užitečné neúplné popisy a plány. Jednak vyjasňují strukturu systému a jednak po pozdějším doplnění se stanou úplnými a tedy konkrétními.**
- **Jen podle konkrétních plánů (tříd) lze vytvářet výrobky (objekty).**
- **Konkrétní třída může zabránit konstrukci objektů privatizací všech svých konstruktorů modifikátorem private a případně ani nepřipustí jejich použití jiným způsobem. (Např. java.lang.Math.)**

Abstraktní třídy, metody a interfejsy

- **Abstraktní třída má modifikátor abstract - konkrétní třída nikoli.**
  Má alespoň jeden konstruktor, nelze však podle ní objekty vytvářet.
  Nemusí mít žádnou abstraktní metodu
  Abstraktní třída nemůže být finální.

- **Statický kontext (tj. static atributy, metody, inicializátory, výčty) abstraktní třídy je funkční.**

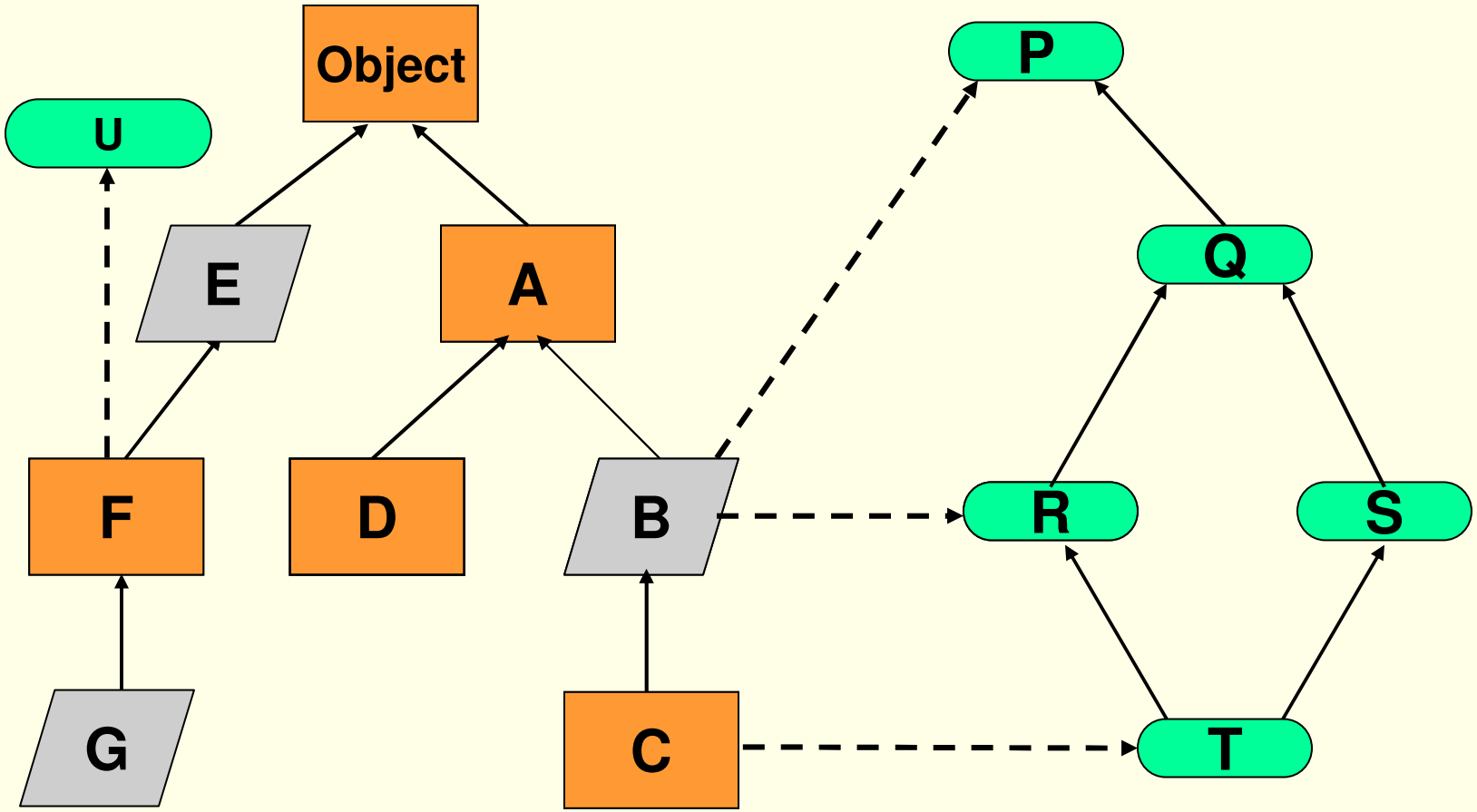
- **Abstraktní metoda má v definici místo těla jen středník, např.:**

  ```
  public abstract int compareTo( Object s ) ;
  ```

- **Má-li třída abstraktní metodu (vlastní definovanou či zděděnou a nepřepsanou na konkrétní), pak musí být označena jako abstraktní.**

- **Interfejs lze považovat za značně omezenou abstraktní třídu, která má všechny metody abstraktní (netřeba u nich uvádět modifikátor abstract).**
  Interfejs umožňuje jednak částečný pohled na třídu a jednak slouží s výhodou jako norma či požadavek.

Abstrakce, dědičnost a polymorfismus



Typy:

- konkrétní plán
- abstraktní plán
- norma, předpis, požadavek

Abstrakce (po šipkách) - úží pohled

Vztahy:

- extends (rozšiřuje)
- implements (splňuje)

Polymorfismus

Polymorfismus se týká jen překrytých nestatických metod - umožňují objektům různým podtypům daného nadtypu specifické chování.

Polymorfismus je relace (reflexivní, asymetrická a transitivní) mezi referenčním typem a množinou referenčních typů, které lze oním typem referovat.

Existuje-li dráha (po kterýchkoli šipkách) z typu x k typu y pak:

- **Typ y je abstrakcí typu x, označme: y ← x**
- **Typ x lze referovat typem y – čili lze říci, že x je také y.**
- **Množina typů *Y* = { x U y | y ← x } je polymorfismem typu y.**

Tři příklady polymorfismu z předchozího obrázku:

A **= { A, B, C, D },** ***Q*** **= { Q, B, C, R, S, T },** ***U*** **= { U, F, G }**

Zda objekt o patří do určitého typu lze otestovat výrazem:

- **o instanceof Typ má hodnotu true nebo false, přičemž**
- **o instanceof Object má vždy hodnotu true .**

Dědičnost (inheritance)

- **Dědičnost velmi usnadňuje:**
  - **Přístup k existujícím třídám a interfejsům a jejich pochopení.**
  - **Vytváření nových tříd a interfejsů s využitím vlastností a schopností již existujících, ověřených a dokumentovaných.**

- **Všechny třídy (bez ohledu na balíčky) tvoří jediný, dědičný, kořenový strom orientovaný směrem ke kořeni, kterým je třída java.lang.Object. Tedy každá třída, kromě java.lang.Object, má právě jednoho přímého předka (superclass = nadtřída), jehož je přímým potomkem (subclass = podtřída). Svoji přímou nadtřídu vyznačuje podtřída ve své hlavičce za klíčovým slovem extends, jinak je přímým předkem java.lang.Object.**

- **Konkrétní třída java.lang.Object nemá žádného předka (je to jakoby prapředek Adam) a tedy nic nezdědila avšak definuje 11 „generálních“ metod. Všechny ostatní třídy jsou tedy jejími přímými či nepřímými potomky a zaručeně mají tyto „generální“ metody.**

- **Třída může také dědit od interfejsů uvedených v seznamu za klíčovým slovem implements.**

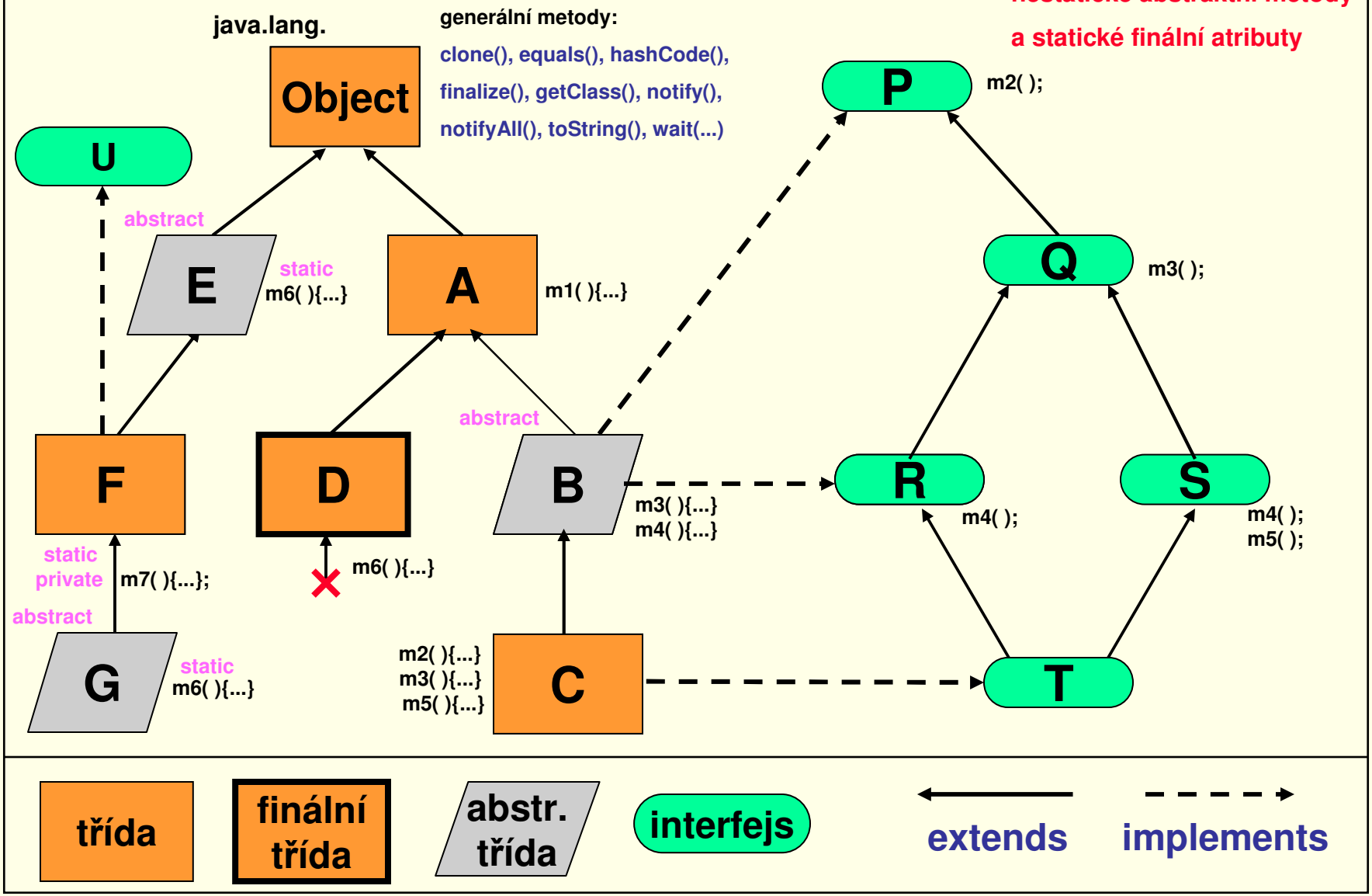
Dědičnost (inheritance)

- **Interfejsy tvoří nesouvislý acyklický graf. (Hrany jsou jen typu extends.)**
- **Není žádný základní interfejs (co obdoba třídy java.lang.Object)**
- **Interfejs:**
  - **nemusí mít žádného předka,**
  - **může být přímým potomkem více interfejsů uvedených v seznam za klíčovým slovem extends,**
  - **nemůže být potomkem žádné třídy,**
  - **nemůže být finální.**

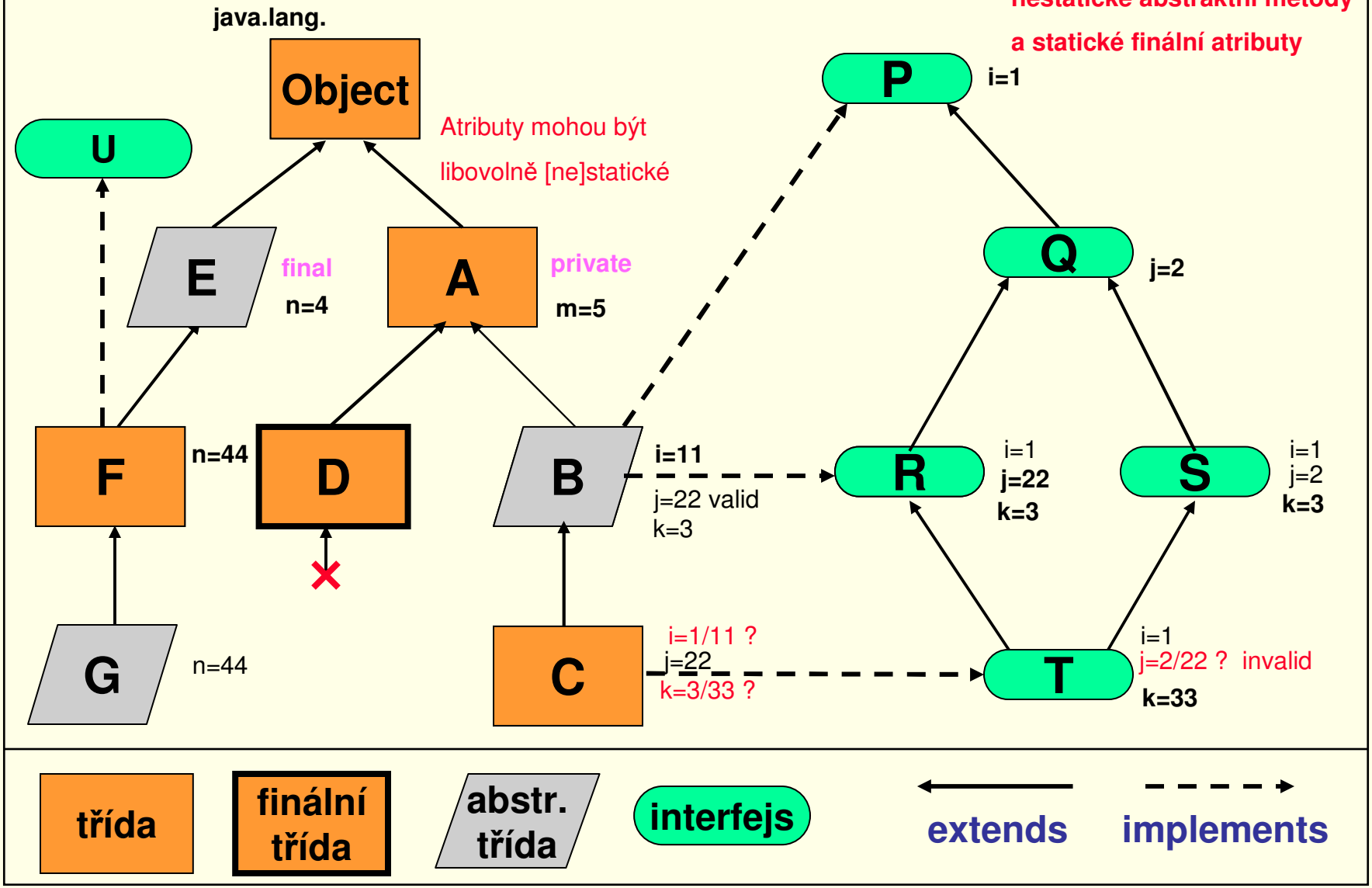
Dědí se pouze viditelné členy (tj. atributy, metody, vnitřní třídy, vnitřní interfejsy, výčty), jež nemají příliš restriktivní přístupový modifikátor:
(public – protected – *default=friend* – private tj.
vždy – v balíčku či potomek v jiném balíčku – v témže balíčku – nikdy).

Potomek nemůže dědictví odmítnout a tedy není nikdy chudší než předek – zpravidla bývá mnohem bohatší.

Dědičnost metod



Dědičnost atributů



Syntaxe třídy

```
[ public | protected | private ] [ abstract | final ] [ strictfp ]  static
class  JménoTřídy  [ <~> ]                                          //  hlavička třídy
            [ extends JménoNadTřídy [ <~> ] | extends java.lang.Object ]
            [ implements JménoInterfejsu [ <~> ]  , ... ]
 {                                                                  //  tělo třídy
        ....  //  atributy statické a nestatické
        ....  //  inicializátory statické a nestatické
        ....  //  konstruktory
        ....  //  metody statické a nestatické
        ....  //  vnořené a vnitřní třídy
        ....  //  výčty  enum
        ....  //  vnořené interfejsy
        ....  //  anotace
 }
```

- pro vnořené (nested)
- typové parametry – od v. 1.5
- meta symboly

Syntaxe interfejsu

```
[ public | protected | private ] [ strictfp ] [ static ] [ abstract ]
interface  JménoInterfejsu  [ <~> ]                          //  hlavička interfejsu
        [ extends JménoNadInterfejsu [ <~> ] ,  ... ]
{                                                            //  tělo interfejsu
        // [ public static final ] atributy inicializované již v definici
        // [ public abstract ] metody ale ne static final strictfp native
        //  vnořené a vnitřní třídy
        //  výčty  enum
        //  vnořené interfejsy
        //  anotace
}
```

- jen pro vnořené (nested)
- typové parametry – od v. 1.5

- Metody stejných signatur kolidují jen při nekompatibilitě návratových typů.
- Interfejs označený public, má atributy také public - což netřeba uvádět.
- Při vícečetném dědění stejnojmenných atributů nastane kolize.

Syntaxe anonymní třídy

Anonymní třída nemá hlavičku třídy, tudíž ani jméno, nemůže mít potomka a slouží k vytvoření jediného objektu.
Má však jediného přímého předka tím může být nejen třída, ale i jediný interfejs (v tom případě je nadtřídou java.lang.Object).

- **TypT t = new Třída (...) { // tělo anonymní třídy**
  .... // nestatické atributy, inicializátory, metody a třídy
  }
  TypT může být Třída anebo její nadtypy.

- **TypZ z = new Interfejs ( ) { // tělo anonymní třídy**
  .... // nestatické atributy, inicializátory, metody a třídy
  }
  TypZ může být Interfejs nebo jeho nadinterfejsy anebo Object.

Dědictví tříd

- **V nově definované třídě lze:**
  - **zdědit od nadtřídy jen nestatické členy, které jsou public či protected a patří-li potomek do stejného balíčku, pak i default členy. Privátní členy se nedědí.**
  - **přidat nové členy.**
  - **přepsat zděděné metody novými metodami (override), nelze však přepsat statickou nestatickou a naopak.**
  - **zastínit zděděné atributy stejnojmennými novými atributy (shadow),**
  - **definovat vlastní konstruktory a inicializátory.**

- **Zdědění abstraktních metod, není " výhodné dědictví ", leč břemeno, se kterým se může třída vyrovnat jedním z těchto způsobů:**
  - **přepsat všechny zděděné abstraktní metody na konkrétní,**
  - **ponechat některé abstraktní - a bude tudíž abstraktní třídou.**

Dědictví interfejsu

- **Při dědění metod nemůže dojít ke kolizi, neboť všechny jsou abstraktní, tj. nemají tělo – místo něho mají středník.**

- **Při dědění stejnojmenných atributů od více předků-interfejsů dojde k víceznačnosti a to i když jsou inicializovány na stejnou hodnotu.**

  Ke kolizi dojde i při dědění do třídy stejnojmenných atributů z nadtřídy a z interfejsu či jen z více interfejsů. Dědictví z interfejsů do třídy se neuplatní tehdy, je-li ve třídě jméno atributu definováno (ne zděděno). Může záležet i na pořadí v jakém jsou interfejsy uvedeny v implements.

Zastínění a překrytí

- **Zděděný atribut, lze v potomkovi zastínit stejnojmenným atributem, přičemž na typech a modifikátorech přístupu nezáleží.**

  Nefinální statickou metodu lze zastínit.

  Statické atributy a metody jsou přístupné přes jméno třídy.

- **Překrytí metod ve třídě**

  Nevyhovuje-li potomkovi zděděná nefinální metoda, lze v potomkovi deklarovat metodu, která:

  - **má shodnou signaturu,**
  - **má kovariantní návratový typ, typ či subtyp (od 1.5) – ne primitivní,**
  - **nemění statičnost – nestatičnost,**
  - **nezužuje modifikátor přístupu,**
  - **nerozšiřuje množinu kontrolovaných výjimek udaných za throws.**

  Překrytím metody v potomkovi se metoda v předkovi nezmění. Přepsaná nestatická metoda je přístupná z přímého potomka pomocí klíčového slova super. Statická metody je přístupná jsou přístupné přes jméno třídy.

Dědění versus kompozice

Vytvářet bohatší třídy lze:

- **Děděním tj. přidáváním zejména metod, např.:**

```
class B extends A {            // relace: B is A
    void m1( ... ) { ... }
    void m2( ... ) { ... }
}
```

- **Kompozicí tj. přidávání zejména referenčních atributů**

```
class C {                      // relace: C has B and C
    B b = new B( ... );
    C c = new C( ... );
}
class B {
    void m1( ... ) { ... }
    void m2( ... ) { ... }
}
```

Doporučuje se dávat přednost kompozici, dědění používat obezřetně.

Zapouzdření (encapsulation)

- Ve třídách a objektech lze ukrýt atributy, metody, konstruktory, vnitřní třídy a vnitřní interfejsy a tím podpořit spořádanost a bezpečnost.
- Inicializátory jsou skryté již tím, že nemají jméno.

- Míru zapouzdření určuje modifikátor viditelnosti členu či konstruktoru.

modifikátor		viditelnost
public	-	odevšad
protected	-	ve vlastním balíčku a v potomcích v jiných balíčcích
	-	modifikátor se nepíše, chápe se jako package private tj. viditelný ve vlastním balíčku (“friend” v C/C++)
private	-	jen ve vlastní třídě (a i z vnitřních tříd)

- K nedostupným členům nějaké třídy lze přistupovat z jiné třídy jen pomocí neprivátních metod oné třídy.

- V interfejsech lze zapouzdřovat jen atributy jen úrovní “default”.

Getry a setry

Dobrovolná jmenná konvence pro přístupové metody k zejména privátním atributům, které jejich hodnotu vydávají (tzv. getter, accessor) anebo mění (tzv. setter, mutator) usnadňuje tvorbu nadstavbového softwaru. Tento způsob je podmínkou pro tvorbu tzv. Java beans.

```
private  int  cenaKusu;                                  //  atribut
public int getCenaKusu( )  { return cenaKusu; }          //  getter
public void setCenaKusu ( int cenaKusu ) {               //  setter
   ....  this.cenaKusu = cenaKusu;
}

private Point  vrcholHory;                               //  atribut
public Point getVrcholHory ( ) { return vrcholHory ; }   //  getter
public void setVrcholHory ( Point  vrcholHory ) {        //  setter
  ....  this.vrcholHory = vrcholHory ;
}
```

Getry a setry

Getry vracející boolean často začínají "is" resp. "has" – např.:

isVisible( ) resp. hasNext( ).

V případě jednorozměrných polí se používá např.:

```
private String[ ] line =  ...  ;
public String getLine( int index );
public void setLine( int index, String line )
```

Inicializace statických atributů

Když JVM zavede třídu či interfejs do paměti, inicializuje jejich statické atributy. U tříd i tzv. statickými inicializátory - ty připomínají metody bez hlavičky, mají přístup jen ke statickému kontextu, nic nevracejí, nemají signaturu, nevyznačují vyhazování výjimek a provádějí se jen jednou a to v pořadí zápisu. Nelze je volat, nejsou členy třídy a nedědí se.

Syntakticky jsou velmi prosté: `static { ... }`

```
class Priklad {
   static int  j = 10, k, n;
   static  {
       String value1 = System.getProperty( "key1" ) ;
       n = Integer.parse( value1 ) ;
       k = Integer.parseInt( System.getProperty( "key2" ) ) ;
    }
    static Color[ ] c = new Color[ n ] ;
    static {   for ( int i = 0; i < c.length;  i++ )   c[ i ] = new Color( i, i*j, 0 );  }
}   //  Execute as:  java   -Dkey1=value1  -Dkey2=value2  MyProgram
```