První kontakt

Návrhové vzory – Design Patterns

- **Programátorský ekvivalent matematických vzorečků**

$$ax^2 + bx + c = 0 \qquad x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

- **Výhody:**
  - **Zrychlují návrh** (řešení se nevymýšlí, ale jenom použije)
  - **Zkvalitňují návrh**
  - Jsou ověřené, takže výrazně **snižují pravděpodobnost potenciálních chyb** typu *na něco jsme zapomněli*
  - **Zjednodušují a zpřesňují komunikaci mezi členy týmu** (větou, že diskriminant je záporný, řeknu znalým jednoduše řadu věcí, které bych musel jinak složitě vysvětlovat)

- **Znalost návrhových vzorů patří k povinné výbavě současného objektově orientovaného programátora**

Jednoduchá tovární metoda 010

- *Jednoduchá* (někdo používá termín *statická*) *tovární metoda* je zjednodušenou verzí návrhového vzoru *Tovární metoda*.
  Definuje statickou metodu nahrazující konstruktor.
  Používá se všude tam,
  kde potřebujeme získat odkaz na objekt,
  ale přímé použití konstruktoru
  není z nejrůznějších příčin optimálním řešením.

Charakteristika

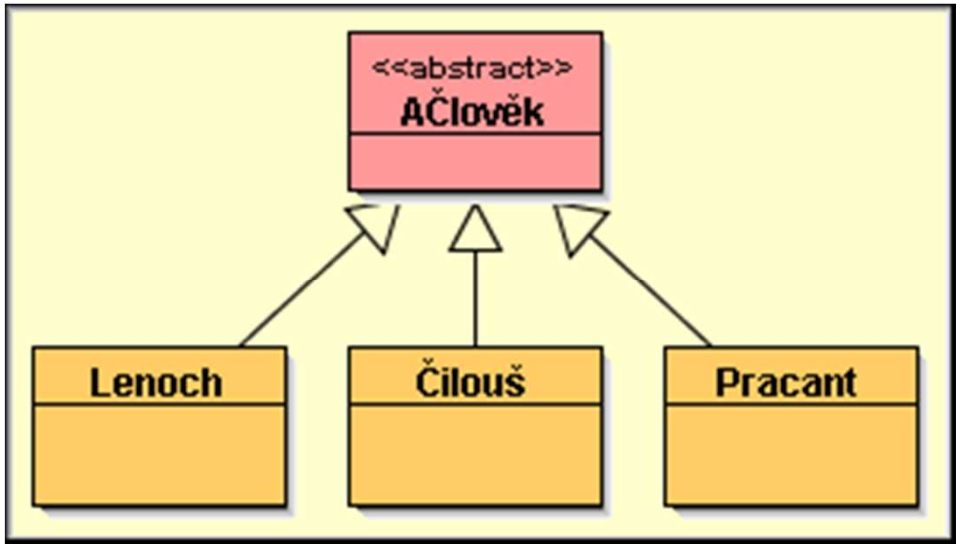
- **Speciální případ obecné tovární metody**
- **Náhražka konstruktoru v situacích, kdy potřebujeme funkčnost nedosažitelnou prostřednictvím konstruktorů**
- **Kdy po ni sáhneme**
  - Potřebujeme rozhodnout, zda se opravdu vytvoří nová instance
  - Potřebujeme vracet instance různých typů
  - Potřebujeme provést nějakou akci ještě před tím, než se zavolá rodičovský konstruktor
  - Potřebujeme více verzí se stejnými sadami parametrů (tovární metody mohou mít různé názvy)

Implementace

- Statická metoda vracející instanci daného typu
- Nemusí vracet vlastní instanci, ale může si vybrat potomka, jehož instanci následně vrátí
- Jmenné konvence
  - `getInstance`
  - `getXXX`, kde XXX je název vraceného typu
  - `valueOf`
  - Naznačující účel `probablePrime`

Příklad: AČlověk

- Tovární metoda může rozhodnout o skutečném typu vráceného objektu




```
public static AČlověk getČlověk() {
  switch ( index++ % 3 ) {
    case 0:  return new Lenoch();
    case 1:  return new Čilouš();
    case 2:  return new Pracant();
    default: throw  new RuntimeException(
                        "Špatně definované maximum" );
  }
}
```

Neměnné objekty (Immutable objects) 012

- ***Neměnný objekt*** **je hodnotový objekt, u nějž není možno změnit jeho hodnotu.**

Primitivní a objektové typy

Java pro zvýšení efektivnosti dělí datové typy na:

- **Primitivní**
  někdo je označuje jako hodnotové, protože se u nich pracuje přímo s hodnotou daného objektu (číslo, znak, logická hodnota)
- **Objektové**
  někdo je označuje za referenční, protože se u nich pracuje pouze s odkazem („referencí“) na objekt
  - Označování odkazových typů jako referenční je z jazykového hlediska stejná chyba, jako překládat control (řídit) → kontrolovat (check)
  - Reference (česky) = zpráva, dobrozdání, doporučení, posudek
    - Viz Akademický slovník cizích slov, ISBN 80-200-0524-2, str. 652
  - Reference (anglicky) = zmínka, narážka, odkaz, vztah, ...
    - Viz Velký anglicko-český slovník, nakl. ČSAV 21-055-85, str. 1181
- **Dělení na hodnotové a odkazové není terminologicky vhodné protože objektové typy dále dělíme na hodnotové a odkazové**

Dělení objektů

- **Odkazové objektové datové typy (reference object data types)**
  - Neuvažujeme o hodnotách, objekt představuje sám sebe, nemá smysl hovořit o ekvivalenci dvou různých objektů
  - Příklady
    - Geometrické tvary
    - Vlákna
- **Hodnotové objektové datové typy (value object data types)**
  - Objekt zastupuje nějakou hodnotu
  - Objekty se stejnou hodnotou se mohou vzájemně zastoupit => má smysl hovořit o jejich ekvivalenci
  - Příklady
    - Obalové typy, zlomky, velká čísla a další matematické objekty
    - Barvy, časy, data
  - Překrývají metody `equals(Object)` a `hashCode()`
    - Tyto metody je vždy vhodné překrýt obě, jinak hrozí problémy

Dělení hodnotových objektů

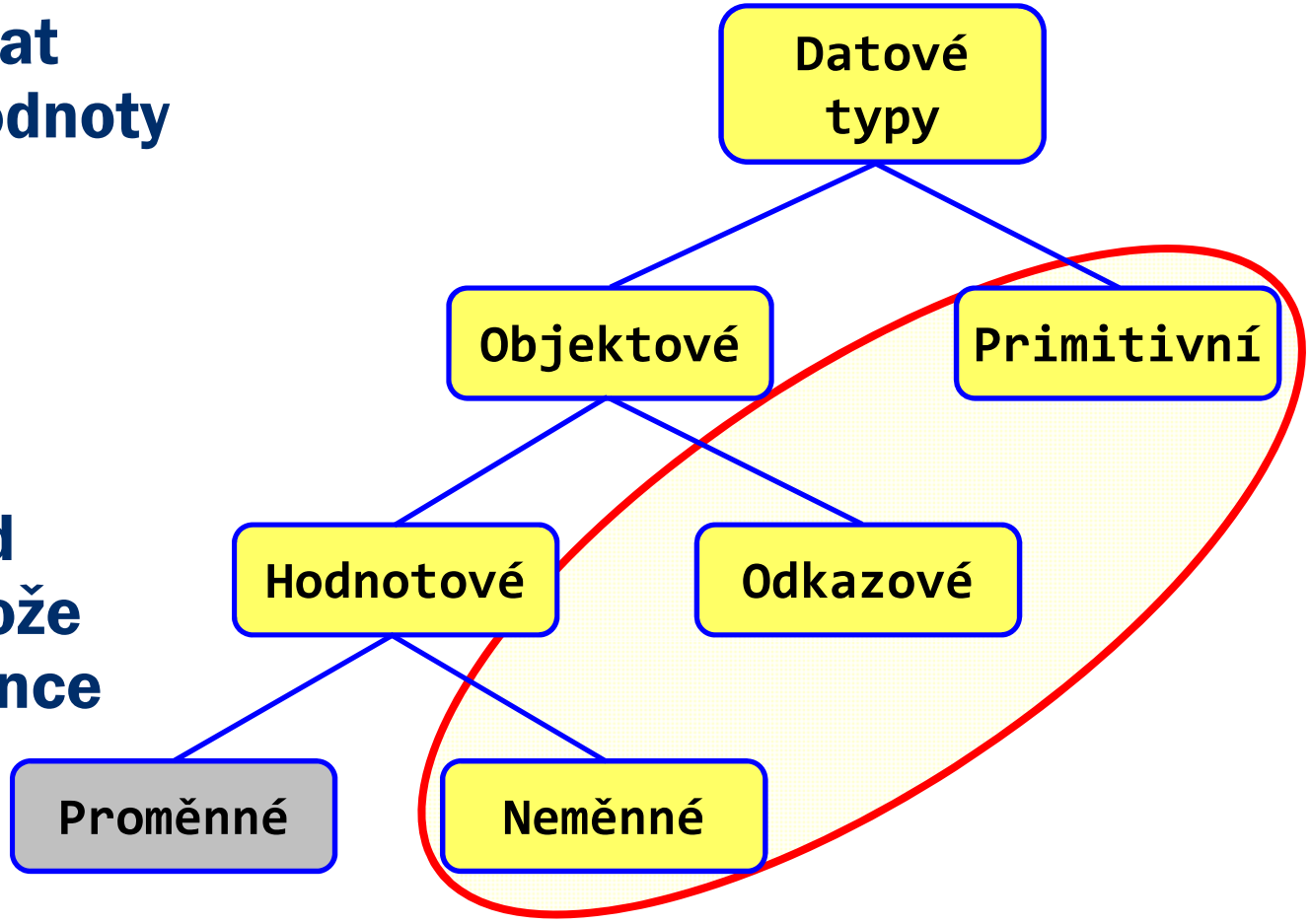
- **Proměnné**
  - Jejich hodnota se může v průběhu života změnit
  - Nesmějí se používat v některých situacích
- **Neměnné**
  - Hodnotu, která jim byla přiřazena „při narození“ zastupují až do své „smrti“
  - Chovají se obdobně jako hodnoty primitivních typů a také je s nimi možno obdobně zacházet
  - Metody, které mají měnit hodnotu objektu, musejí vracet jiný objekt s touto změněnou hodnotou
- **Všechny hodnotové typy bychom měli definovat jako neměnné**
  - Neplatí jen pro Javu, ale i pro jazyky, které umožňují pracovat přímo s objekty a ne jenom s odkazy (C++, C#)

Důsledky proměnnosti objektů

- Hodnota objektu se může změnit uprostřed výpočtu (jako kdyby se z pětek staly šestky)
- Se změnou hodnoty objektu se obecně mění i jeho hash-code =>
- Uložíme-li objekt do kontejneru implementovaného pomocí hashové tabulky a po změně hodnoty jej tam už nenajdeme
- Používání proměnných hodnotových objektů výrazně snižuje robustnost a bezpečnost programů

Doporučené k použití

- Neměnné musí zůstávat jen atributy, jejichž hodnoty používají metody `equals(Object)` a `hashCode()`
- Atributy neovlivňující výsledky těchto metod se mohou měnit, protože na nich hodnota instance nezávisí
- Příklad:
  Atributy počítající počet použití dané instance, počet volání některé metody atd.
- V hodnotových třídách by měly všechny „změnové metody“ vytvářet nové instance



Příklad: Zlomek

```
public class Zlomek extends Number
{
    private final int c; //čitatel
    private final int j; //jmenovatel;

    public Zlomek plus(Zlomek z) {  //this.
        return new Zlomek( this.c*z.j + z.c*j,
                           this.j*z.j );
    }

    public Zlomek krát(Zlomek z) {
        return new Zlomek( c * z.c,  j * z.j );
    }
//...
```

Přepravka (Crate, Transport Object) 013

- Vzor *Přepravka* využijeme při potřebě sloučení několika samostatných informací do jednoho objektu, prostřednictvím nějž je pak možno tyto informace jednoduše ukládat nebo přenášet mezi metodami.

Motivace

- **Některé vlastnosti sestávají z více jednoduchých hodnot**
  - Pozice je definována jednotlivými souřadnicemi
- **Při nastavování takových hodnot se používá více parametrů**
  - Příklad: `setPozice( int x, int y )`
- **Takovéto hodnoty se špatně zjišťují, protože Java neumožňuje vrácení několika hodnot současně**
- **Možnosti:**
  - Zjišťovat každou složku zvlášť (`getX()` + `getY()`)
  - Definuje se speciální třída, jejíž instance budou sloužit jako přepravky pro přenášení dané sady hodnot
- **Druhou možnost doporučuje návrhový vzor *Přepravka***
- **V anglické literatuře bývá někdy označována *Messenger* nebo *Transport Object***

Vlastnosti přepravky

- Přepravka je objekt určený k uložení skupiny údajů „pod jednu střechu“ a jejich společnému transportu
- Přepravku řadíme mezi kontejnery = objekty určené k uložení jiných objektů
- Oproti standardům definuje své atributy jako veřejné, aby tak zjednodušila a zefektivnila přístup k jejich hodnotám
- Aby nebylo možno atributy nečekaně měnit, definuje je jako konstantní
- Velmi často pro ni definujeme pouze konstruktor s parametrem pro každý atribut, ale žádné metody
- V řadě případů však přepravky definují přístupové metody, ale i řadu dalších užitečných metod

Definice přepravky `Pozice`

```
public class Pozice
{
    public final int x;
    public final int y;

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

Použití přepravky `Pozice`

```
public class Světlo
{
    private static final Barva ZHASNUTÁ = Barva.ČERNÁ;
    private final Elipsa žárovka;
    private final Barva  barva;

    // ... Konstruktory a dříve definované metody

    public Pozice getPozice() {
        return new Pozice( žárovka.getX(), žárovka.getY() );
    }

    public void setPozice( int x, int y ) {
        žárovka.setPozice( x, y );
    }

    public void setPozice( Pozice p) {
        žárovka.setPozice( p.x, p.y );
    }
}
```

Převádí akci na svoji přetíženou verzi

Služebník (Servant) 014

- Návrhový vzor *Služebník* použijeme v situaci, kdy chceme skupině tříd nabídnout nějakou další funkčnost, aniž bychom zabudovávali reakci na příslušnou zprávu do každé z nich.
- *Služebník* je třída, jejíž instance (případně i ona sama) poskytují metody, které si vezmou potřebnou činnost (službu) na starost, přičemž objekty, s nimiž (nebo pro něž) danou činnost vykonávají, přebírají jako parametry.

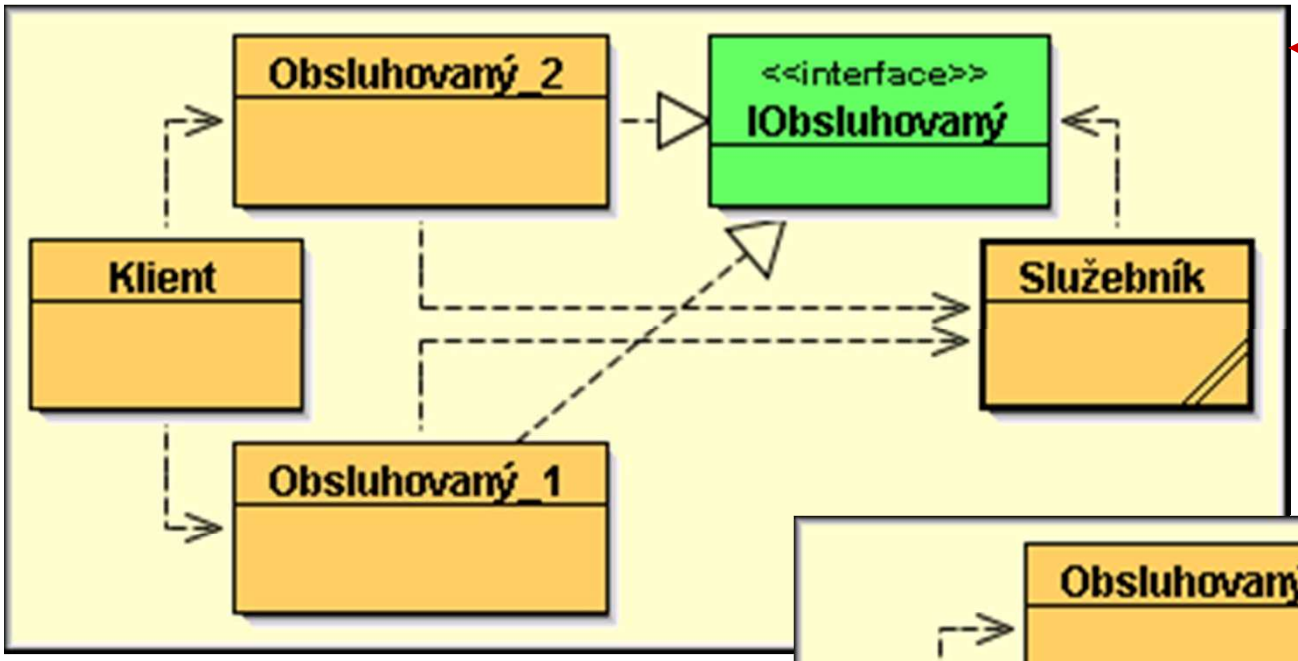
Motivace

- Několik tříd potřebuje definovat stejnou činnost
  a nechceme definovat na několika místech stejný kód
- Objekt má úkol, který naprogramovat buď neumíme,
  nebo bychom jej sice zvládli, ale víme,
  že je úloha již naprogramovaná jinde
- Řešení: Definujeme či získáme třídu, jejíž instance (služebníci)
  budou obsluhovat naše instance a řešit úkoly místo nich
  - Řešení pak bude na jednom místě a bude se snáze spravovat
- Postup se hodí i když připravujeme řešení,
  které chceme definovat dostatečně obecné,
  aby je mohli používat všichni,
  kteří je budou v budoucnu potřebovat,
  a přitom nevíme, kdo budou ti potřební

Implementace

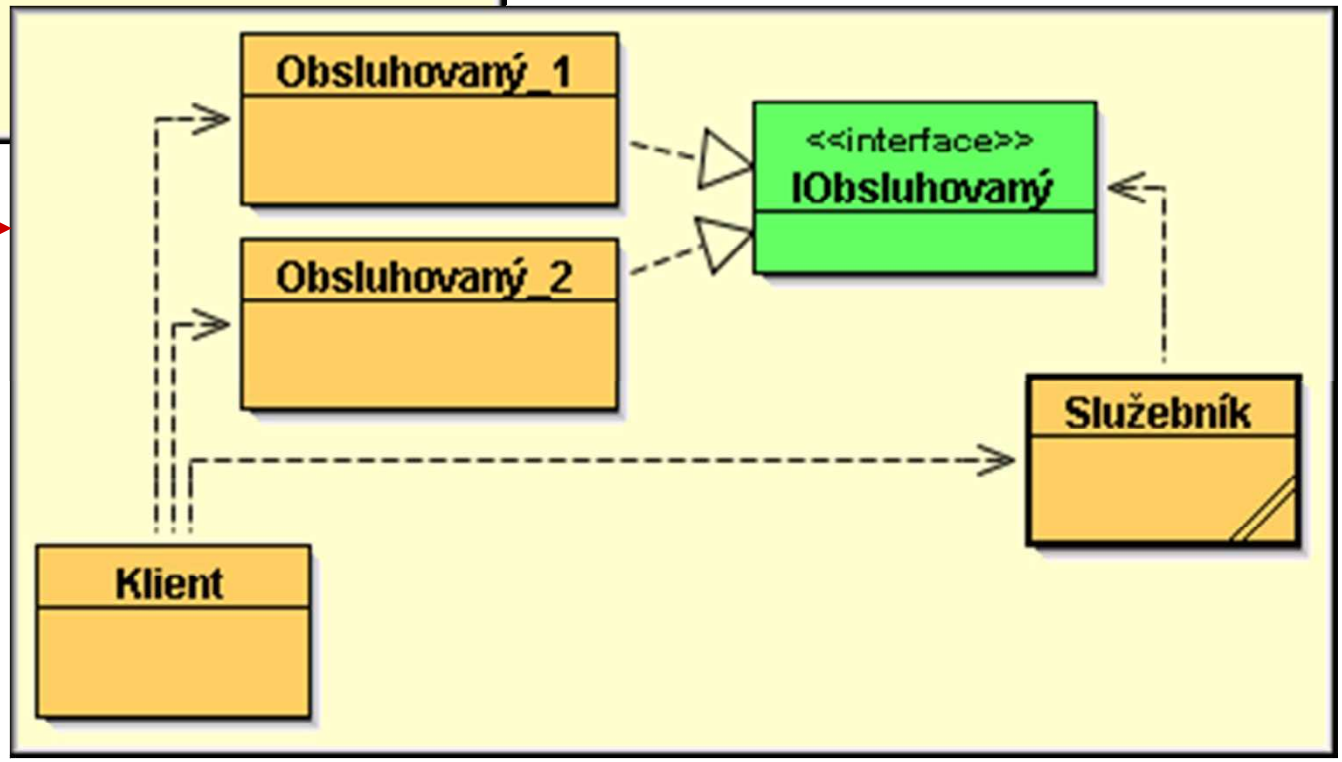
- **Služebník nepracuje sám, ale komunikuje s obsluhovanými instancemi**
- **Aby mohl instance bezproblémově obsluhovat, klade na ně požadavky, co všechno musejí umět**
- **Služebník proto:**
  - Definuje rozhraní (`interface`), v němž deklaruje své požadavky
  - Jeho obslužné metody budou jako své parametry akceptovat pouze instance deklarovaného rozhraní
- **Instance, která chce být obsloužena:**
  - Musí být instancí třídy implementující dané rozhraní, aby se mohla vydávat za jeho instanci
  - Implementací rozhraní deklaruje, že umí to, co od ní služebník k její plnohodnotné obsluze požaduje

Služebník – diagramy tříd



Klient posílá zprávu obsluhované instanci a ta předá požadavek služebníku, který jej realizuje; klient nemusí o služebníku vědět

Klient posílá zprávu přímo služebníku a v parametru mu současně předává instanci, kterou má obsloužit



Příklad 1: Plynule posuvné objekty

- Objekty projektu Tvary se umí přesouvat pouze skokem
- Pokud bychom chtěli, aby se přesouvaly plynule, museli bychom do každého z nich přidat příslušné metody, které by však byly u všech tříd téměř totožné
- Seženeme si služebníka – instanci třídy `Přesouvač`
- Tito služebníci vyžadují od obsluhovaných objektů implementaci rozhraní `IPosuvný` deklarujícího metody:
  - `Pozice getPozice();`
  - `void setPozice( Pozice pozice );`
  - `void setPozice( int x, int y  );`
- Zato nabízí možnost volat metody:
  - `void přesunO ( IPosuvný ip, int dx, int dy );`
  - `void přesunNa( IPosuvný ip, int  x, int  y );`
  - `void přesunNa( IPosuvný ip, Pozice pozice  );`

Příklad 2: Blikající světlo

- Chceme po instancích třídy `Světlo`,
  aby uměly blikat zadanou dobu nezávisle na jiné činnosti
- Cyklus je nepoužitelný,
  protože po dobu jeho provádění ostatní činnosti stojí
  - Takto není možno naprogramovat ukazatel směru jedoucího auta
- Využijeme služeb instancí třídy `Opakovač`, jejichž metody umějí opakovat klíčové činnosti svých parametrů
- Opakovač od obsluhovaných vyžaduje,
  aby implementovali rozhraní `IAkční` s metodou `akce()`,
  kterou bude opakovač zadaný-počet-krát opakovat
- Demo: D03-2: Blikání světla

Prázdný objekt (Null Object) 015

- *Prázdný objekt* je platný, formálně plnohodnotný objekt, který použijeme v situaci, kdy by nám použití klasického prázdného ukazatele null přinášelo nějaké problémy – např. by vyžadovalo neustále testování odkazuj na prázdnost nebo by vedlo k vyvolání výjimky `NullPointerException`.

Motivace

- Vrací-li metoda v nějaké situaci prázdný odkaz (`null`), musí volající metoda tuto možnost kontrolovat, než získaný odkaz použije
- Vrácením odkazu na plnohodnotný objekt umožníme zrušit tyto testy
- Příklady:
  - Žádná barva
  - Žádný směr
  - Prázdný iterovatelný objekt – viz další stránka
  - Třída `java.util.Collections`
    - `public static final <T> Set<T>  emptySet()`
    - `public static final <T> List<T> emptyList()`
    - `public static final <T> Map<T>  emptyMap()`