V Pascale rozoznávame **ordinálne** a **zložené** údajové typy. V Jave **primitívne** a **objektové,** pričom objektových je mnoho rôznych druhov (vymenovacie typy, parametrické typy, abstraktné triedy, rozhrania a tak podobne).

Ordinálne údajové typy (len Pascal):

- sú spočítateľné. Vždy vieme určiť nasledovníka a predchodcu čísla, napríklad `integer` (keďže hovoríme o Pascale; v Jave by to bol `int`).

Primitívne údajové typy (Java):

- sú všetky tie, ktoré nie sú objektové. Ich zoznam je dostupný na adrese https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html;
- treba poznať ich vnútornú reprezentáciu, čiže ako sú uložené/reprezentované v pamäti počítača.

Primitívne typy `byte` (8-bitový), `short` (16-bitový), `int` (32-bitový), `long` (64-bitový):

- v Pascale je `byte` reprezentovaný takto: 1 bit rezervovaný pre znamienko a 7 bitov pre číslo (hodnotu);
- treba ovládať doplnkový, priamy, inverzný kód.

Primitívny typ `float`:

- mantisa + exponent;
- pohyblivá rádová čiarka;
- 32-bitový údajový typ.

Primitívny typ `double`:

- 64-bitový údajový typ.

Primitívny typ `char`:

- 16-bitový Unicode znak.

Primitívny typ `boolean`:

- `true`, `false`.

• • •

Autoboxing (a unboxing)

Pri objektoch je za normálnych okolnosti striktne predpísané používať metódy na manipuláciu s atribútmi (t. j. vnútornými/súkromnými hodnotami). Vďaka autoboxingu (a unboxingu) smiem napísať `c = a + b`, aj keď ide o objekty (ale len niektoré preddefinované objekty). Autoboxing (a unboxing) je veľmi užitočná vlastnosť Javy. Bez nej by práca s objektovými ekvivalentmi primitívnych číselných typov bola komplikovanejšia. Čo to presne je?

Je to automatické spracovanie preddefinovaných číselných objektových typov tak, ako keby sme pracovali s primitívnymi typmi.

Najlepšie to ukážeme na príklade (nasledujúci príklad porovná primitívny typ `int` s jeho objektovým ekvivalentom `Integer`). Pri primitívnych údajových typoch môžeme bez váhania napísať toto:

```
int a = 10, b = 5;
int c;

c = a + b; // (Tento riadok je pri primitívnych typoch úplne prirodzený,
           // ale pri bežných objektoch by nefungoval.)

System.out.println("Súčet: " + c); // Vypíše: Súčet: 15
```

Ak by nejestvoval autoboxing (a unboxing), museli by sme pri použití objektových typov celý kód prepísať takto:

```
Integer a = new Integer(10), b = new Integer(5); // využitie konštruktorov
                                                 // triedy Integer
Integer c;

c = new Integer(a.intValue() + b.intValue()); // využitie „getterov“
                                              // a konštruktora triedy
                                              // Integer

System.out.println("Súčet: " + c.toString()); // volanie metódy prevádzajúcej
                                              // celé číslo na reťazec
```

Čiže nemohli by sme objektu číselného typu priradiť hodnotu bez vytvorenia novej inštancie cez konštruktor a nemohli by sme čítať hodnotu bez použitia prislúchajúcej metódy. Autoboxing (a unboxing) tieto akcie vykonávajú automaticky, takže môžeme napísať prakticky rovnaký kód ako pri primitívnych typoch:

```
Integer a = 10, b = 5;
Integer c;

c = a + b;

System.out.println("Súčet: " + c);
```

• • •

Podobne ako v Pascale, tak aj v Jave jestvujú zložené typy rôzneho druhu. Napríklad **parametrické údajové typy.** V programovacom rámci `GRobot` je jeden takýto typ – `Zoznam`. Vo všeobecnom zápise `Zoznam<Typ>` je `Typ` akákoľvek trieda, čiže nie povolené používať primitívne údajové typy (`int`, `long`, `double`…).

Parametrických údajových typov je v Jave veľa (strom `TreeSet<E>`, zásobník `Stack<E>`, front LinkedList<E>…). Napríklad balíček Javy `java.util` (https://docs.oracle.com/javase/8/docs/api/java/util/package-summary.html) definuje takmer 30 rozhraní a asi 70 tried parametrických údajových typov, ktoré implementujú rôzne druhy zoznamov, máp, zásobníkov a podobne. (To však nie sú jediné preddefinované parametrické typy Javy.)

Trieda

```
public class Trieda
{
    public Trieda()
    {

    }
}
```

- **trieda,** ktorá má rovnaký názov ako súbor, musí byť **verejná;**
- zoskupuje ostatné prvky ako atribúty, metódy, konštanty a ďalšie iné prvky (vnorená trieda, špeciálne typy tried – vymenovacie triedy (enumerácie), rozhrania):

```
public class «názov triedy» extends «rodičovská/nadradená trieda»
    implements «zoznam rozhraní, ktoré trieda implementuje»
```

- názov **triedy** a **konštruktora** musia byť zhodné;
- v Jave je povolená dedičnosť len od jedného priameho rodiča (za `extends` môže nasledovať len jeden názov triedy, ktorá bude rodičovská:

```
import knižnica.GRobot;

public class MojaTrieda extends GRobot    // dedíme z (alebo odvodzujeme od)
                                          // triedy GRobot
{
    public MojaTrieda()
    {
        // ...
    }
}
```

- v Jave môžeme definovať tzv. **rozhrania,** z ktorých môže trieda implementovať viac rodičov; slúžia na to, aby sme poskytli určitý „univerzálny predpis“ toho, akým spôsobom môžu pracovať tie triedy, ktoré rozhranie implementujú vďaka ktorému vedia ostatné triedy, ako s týmito triedami pracovať, príklad:

Majme nasledujúce dve rozhrania:

```
public interface Farebnosť
{
    public Farba farba();
}

public interface Poloha
{
    public Bod poloha();
    public double polohaX();
    public double polohaY();
}
```

Všetky triedy, ktoré chcú implementovať jedno alebo obidve rozhrania, musia implementovať všetky metódy, ktoré rozhrania deklarujú (predpisujú). Napríklad:

```
public class GRobot implements Farebnosť, Poloha
{
    private Farba f;
    private int x, y;

    public Farba farba() // implementácia metódy z rozhrania Farebnosť
    {
        return f;
    }

    // Implementácia metód z rozhrania Poloha:

    public Bod poloha()
    {
        return new Bod(x, y);
    }

    public double polohaX()
    {
        return x;
    }

    public double polohaY()
    {
        return y;
    }
}
```

Keď odvodíme novú triedu od inej (napr. `GRobot`), používame vopred naprogramovaný základ – obrazne povedané „neobjavujeme znova Ameriku“. **Dedičnosť** dáva programátorom možnosť, nerobiť množstvo vecí odznova – môžu využívať preddefinované prvky zvoleného jazyka alebo programovacieho rámca.

Konštruktor sa vždy spúšťa pri tvorbe inštancií. Až keď napíšeme `new MojaTrieda()`, vzniká v pamäti počítača inštancia, ktorá „žije“, ktorá má svoju polohu, farbu atď.

Deštruktor

Java má vlastný zberač odpadkov a preto tento jazyk nemá žiaden explicitný mechanizmus deštrukcie. Implicitný mechanizmus zberača odpadkov kontroluje, či sa niečo využíva a je to ešte potrebné. Ak zberač zistí, že nie je, sám to pamäťové miesto uvoľní (v iných jazykoch máme konštruktor <–> deštruktor, deštruktor v nich uvoľňuje pamäť).

(*Poznámka:* Modifikátor `public` súvisí s uzavretosťou, podobne ako modifikátory `private` a `protected`.)

Inicializácia atribútov pri konštrukcii

Máme viacero možností, ako inicializovať atribúty. Napríklad:

```
public class MojaTrieda extends GRobot
{
    // Inicializácia pri deklarácii – inicializácia definíciou:
    private «typ» «názov atribútu» «= priradenie hodnoty (nepovinné)»;

    public MojaTrieda()
    {
        // Inicializácia v konštruktore:
        «názov atribútu» = «priradenie hodnoty»;
    }


    {
        // Inicializácia v anonymnom bloku (ktorého použitie nemá pri
        // dynamických atribútoch nezastupiteľný význam, naopak, odporúča
        // sa namiesto toho použiť inicializáciu v konštruktore;
        // nezastupiteľný význam majú iba statické anonymné bloky pri
        // inicializácii niektorých statických atribútov):
        «názov atribútu» = «priradenie hodnoty»;
    }
}
```

Príklad:

```
public class MojaTrieda extends GRobot
{
    // Jedna možnosť je inicializovať atribúty priamo pri deklarácii
    // (čiže vlastne vykonať definíciu):
    private int môjStav = 5;
}
```

Ale odporúčané je inicializovať atribúty prostredníctvom konštruktorov:

```
public class MojaTrieda extends GRobot
{
    private int môjStav;

    public MojaTrieda(int môjStav) // úplný konštruktor
    {
        this.môjStav = môjStav;
    }

    public MojaTrieda() // predvolený konštruktor
    {
        this.môjStav = 5;
    }
}
```

Predvolený konštruktor nemá parametre, nastavuje (inicializuje) hodnoty všetkých atribútov na predvolené stavy.

Úplný konštruktor: prijíma (v parametroch) hodnoty všetkých atribútov a nastavuje (inicializuje) nimi svoje (súkromné) atribúty.

- Každý konštruktor musí nastavovať vybrané atribúty. Konštruktor je špeciálny typ metódy. V Jave sa musí volať rovnako ako trieda a nemá návratový typ (ani `void`).
- Ďalšie špeciálne metódy, tzv. **„gettery“** a **„settery“** sú metódy na čítanie a zápis hodnôt atribútov a v niektorých jazykoch atribút + konštruktory + „getter“ a „setter“ tvoria tzv. vlastnosti (properties).
- Tieto *vlastnosti* tvoria podstatnú časť **uzavretosti:** atribúty by mali byť (síce nie vždy sú, ale základom OOP je, že by mali byť) definované ako súkromné a metódy, ktoré s nimi pracujú ako verejné:

```
public class MojaTrieda extends GRobot
{
    private int môjStav;     // atribút (field), v niektorých jazykoch
                             // sa používa aj termín „inštančná premenná“

    ...konštruktory...

    public int getMôjStav()    // getter
    {
        return môjStav;
    }

    public void setMôjStav(int novýStav)    // setter
    {
        if (...kontrola stavu...)
        {
            ...
            môjStav = novýStav;
        }
    }
}
```

- Párujúcim termínom k termínu **parameter** je **argument** (to je to, čo uvedieme na miesto parametra pri volaní konkrétnej metódy).
- **Parameter** je v podstate deklarácia a **argument** je výraz (môže byť aj veľmi jednoduchý, napríklad literál `1`, ale vždy to je výraz).
- Zjednodušene môžeme povedať, že **trieda** je „niečo všeobecné“ a **inštancia** zase „niečo konkrétne“. Inštancia je konkrétny objekt, napríklad inštancia nového robota, ktorú vytvoríme príkazom: `new GRobot()`.