Softvérové technológie

Toto je verzia elektronickej interaktívnej učebnice určenej na tlač. Táto verzia z pochopiteľných dôvodov neobsahuje všetky prílohy. Úplná verzia je zverejnená vo Vzdelávacom portáli Trnavskej univerzity: http://elearning.truni.sk/ a na http://cec.truni.sk/horvath/prievoznik/.

Autor: Mgr. Ing. Roman Horváth, PhD.
Katedra matematiky a informatiky

Recenzenti: prof. Ing. Veronika Stoffová, CSc. a PaedDr. Miroslav Ölvecký, PhD.
Jazyková korektúra: PaedDr. Ľubomír Rendár, PhD.

Za odbornú, jazykovú a štylistickú stránku týchto vysokoškolských učebných textov zodpovedajú autori.

ISBN 978-80-8082-696-3
EAN 9788080826963

Táto elektronická učebnica vznikla s podporou Európskej únie, zo štrukturálneho fondu ESF prostredníctvom operačného programu Vzdelávanie.

Obsah

Technické požiadavky

Na štúdium tohto materiálu potrebujete inštaláciu nasledujúceho softvéru:

- vývojové prostredie JDK (Java Development Kid) 6.0 alebo vyššie,
- programovací nástroj BlueJ 3.0 alebo vyšší,
- nakonfigurovanú slovenskú jazykovú mutáciu nástroja BlueJ s kódovaním UTF-8,
- triedy `Vstup` a `GRobot`.

Vstupné požiadavky

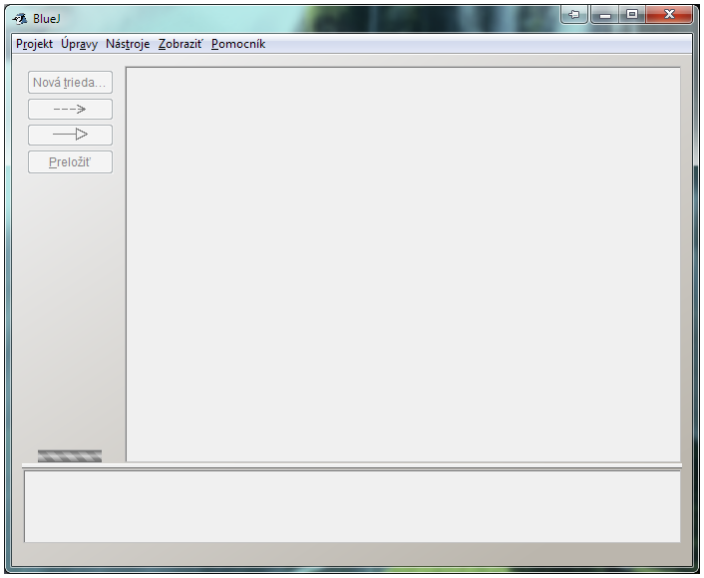
Štúdium tohto materiálu predpokladá **základné znalosti** (objektovo orientovaného) **programovania**. Nutnosťou sú vedomosti o základných riadiacich štruktúrach (vetvenia, cykly, podmienené spracovanie, spracovanie výnimiek…) a výhodou sú poznatky o základných princípoch objektovo orientovaného programovania (triedy, objekty, uzavretosť, dedičnosť, formy polymorfizmu…).

Návody

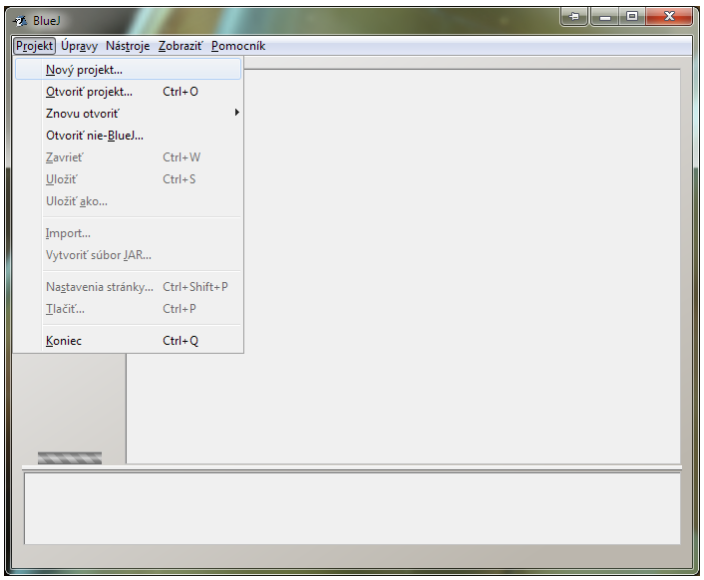
V tejto časti materiálu nájdete niekoľko rýchlych návodov, ktoré vám pomôžu, ak ste začiatočníkom v používaní nástroja BlueJ, prípadne ak ste nástroj používali, ale potrebujete osviežiť niektoré postupy…

Vytvorenie nového projektu

Prázdne hlavné okno nástroja BlueJ verzia 3.0 s nainštalovanou slovenčinou (návody na inštaláciu nie sú súčasťou tohto materiálu) vyzerá takto:

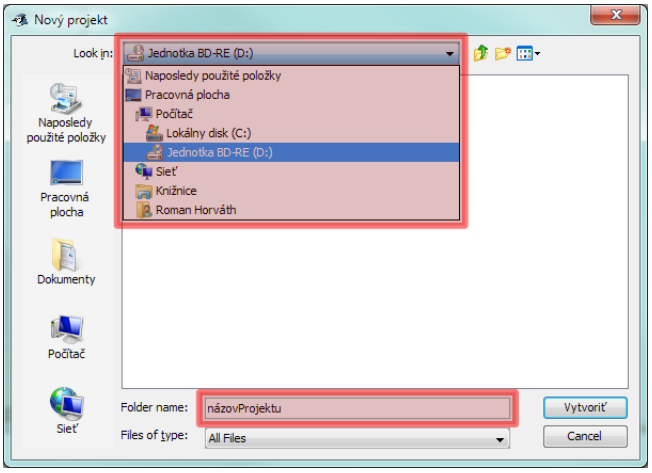


Nový projekt vytvoríme zvolením položky „Nový projekt“ z rolovacej ponuky „Projekt“:

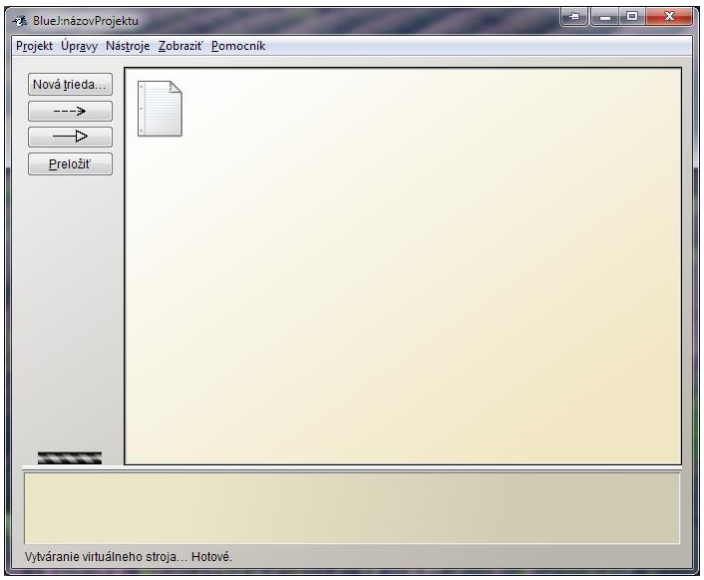


V zobrazenom dialógu musíme zvoliť umiestnenie a zadať názov projektu (na obrázku zvýraznené červenou). Popisy oboch položiek budú pravdepodobne v anglickom jazyku, pretože ich preklad nie je záležitosťou nástroja BlueJ, ale virtuálneho stroja Javy. V našej ukážke má umiestnenie popis „Look in:“ a názov projektu „Folder name:“.

Upozornenie: dobre si zapamätajte umiestnenie, kde projekt vytvárate!



Po potvrdení tlačidlom „Vytvoriť“ bude projekt vytvorený. Ovládacie prvky hlavného okna sa aktivujú a na ploche projektu sa zobrazí ikona reprezentujúca textový súbor „README.TXT“, ktorý je predvolenou súčasťou každého projektu:



Ďalším krokom je importovanie tried potrebných na fungovanie projektu.

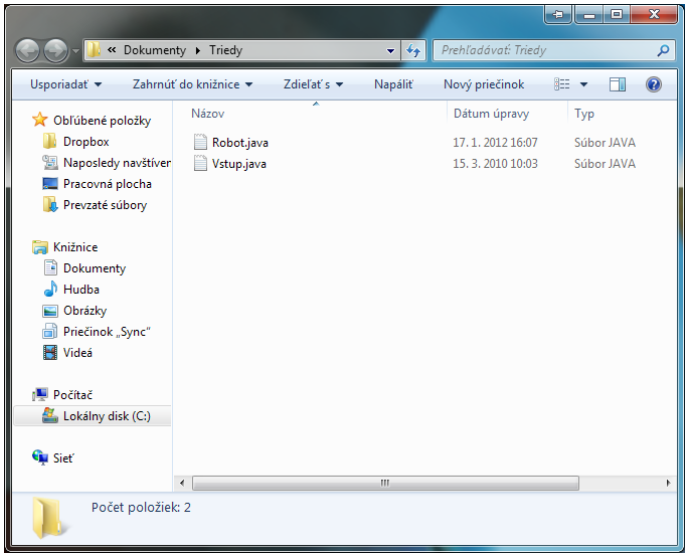
Importovanie tried

Na importovanie tried musíme poznať umiestnenie nášho projektu. Základom importu je skopírovanie tých tried, ktoré chceme importovať, do priečinka projektu. Môžeme to urobiť priamo z prostredia BlueJ alebo „ručne“, napríklad pomocou Prieskumníka Windows (v závislosti od použitého operačného systému).

Ručné kopírovanie

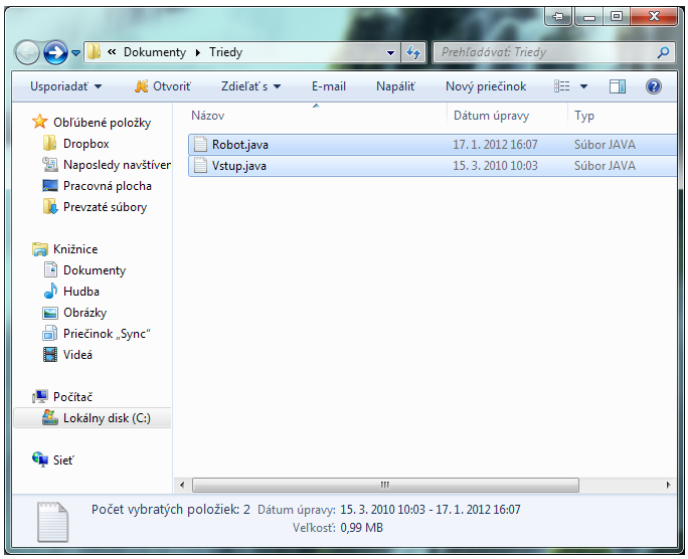
Výhodou tohto prístupu je úplná kontrola nad procesom importu. Použijeme ho vtedy, ak je v priečinku, z ktorého importujeme, umiestnených viacero tried a medzi nimi i také, ktoré importovať nechceme. Ak sme si istí, že v priečinku, z ktorého importujeme, sa nenachádzajú žiadne iné súbory, môžeme použiť priamy import…

Predpokladajme, že triedy, ktoré chceme importovať (GRobot.java[1] a Vstup.java) sú umiestnené v podpriečinku nazvanom „Triedy“ v dokumentoch aktuálneho používateľa:

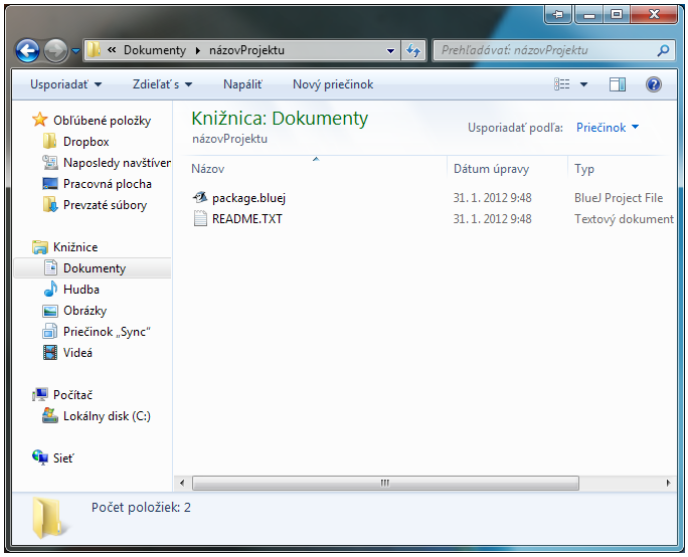


Skopírovať ich môžete ľubovoľným spôsobom, aký poznáte, napríklad klasicky – súbory označte, stlačte klávesovú skratku Ctrl + C:

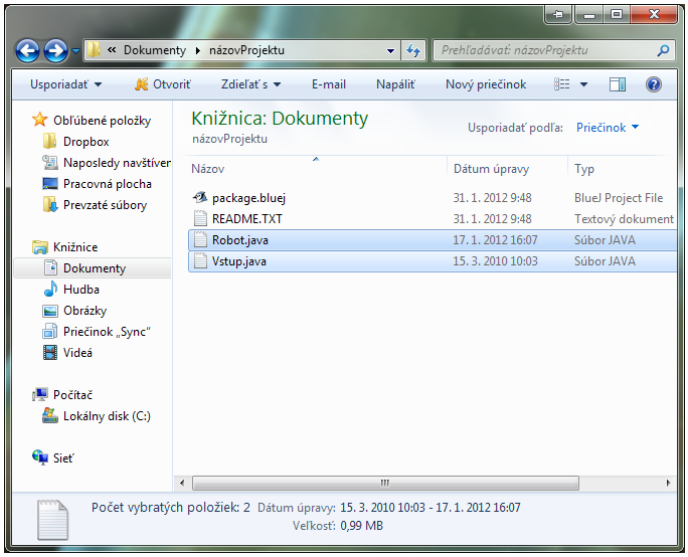
[1] Na obrázkoch sa vyskytuje staršie pomenovanie skupiny tried grafického robota (`Robot`).



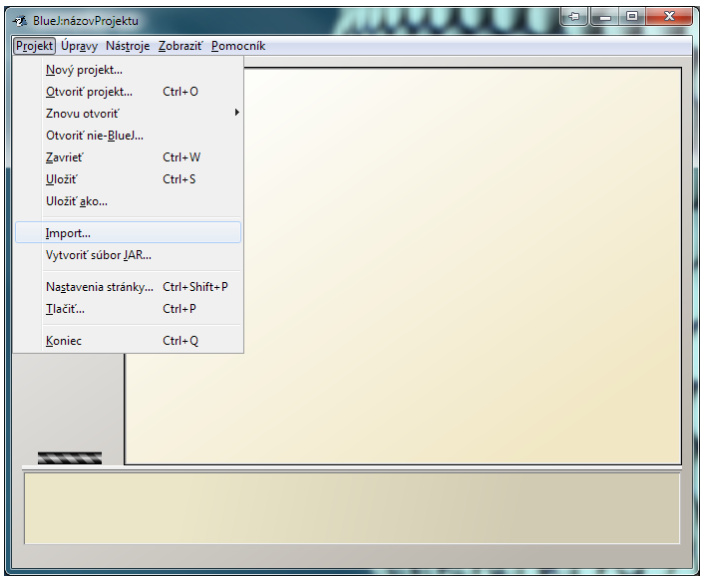
Prejdite do priečinka projektu:



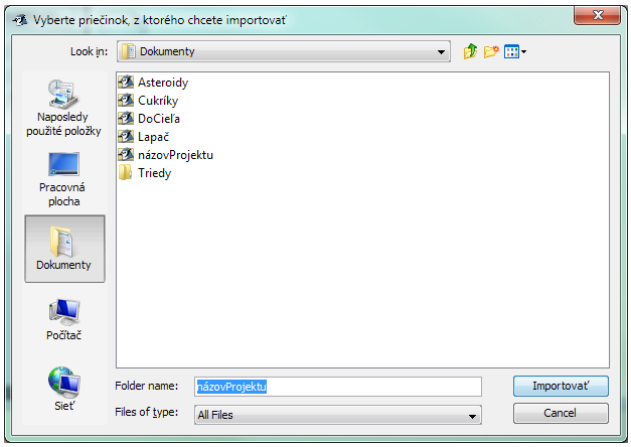
a stlačte klávesovú skratku Ctrl + V:



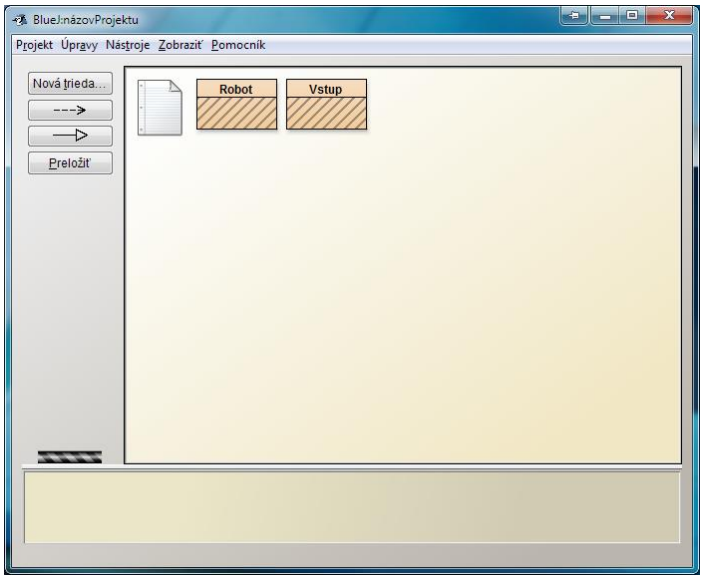
Teraz sa vráťte do okna projektu v BlueJ-i a zvoľte položku „Projekt“ » „Import…“:



V dialógu, ktorý sa následne otvorí, by mal byť predvolený náš projekt, ak nie je, vyhľadajte a zvoľte ho:



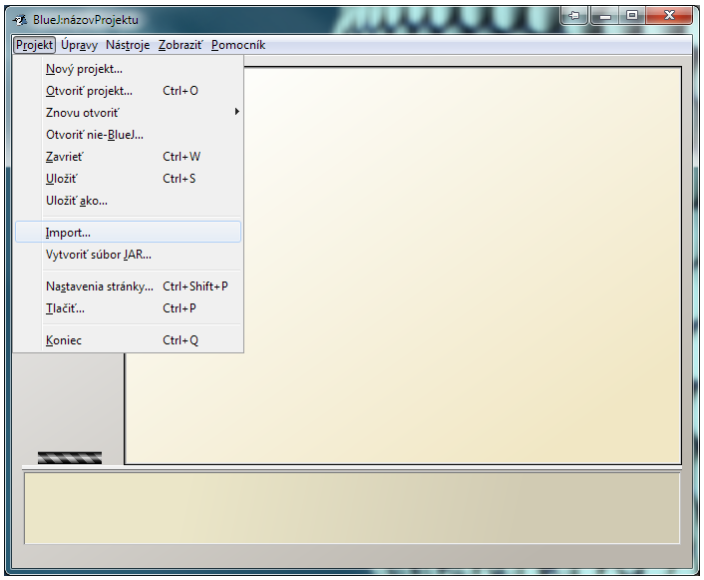
Zvoľte tlačidlo „Importovať“. Tým obnovíme stav projektu a triedy, ktoré sme predtým skopírovali do priečinka projektu, sa objavia na ploche projektu:



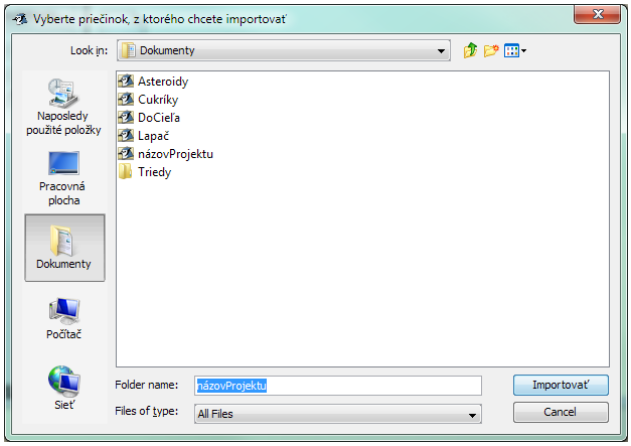
Keďže sme súbory ručne skopírovali priamo do nášho projektu, rovnaký efekt by sme dosiahli zatvorením a opätovným otvorením nášho projektu.

Priamy import

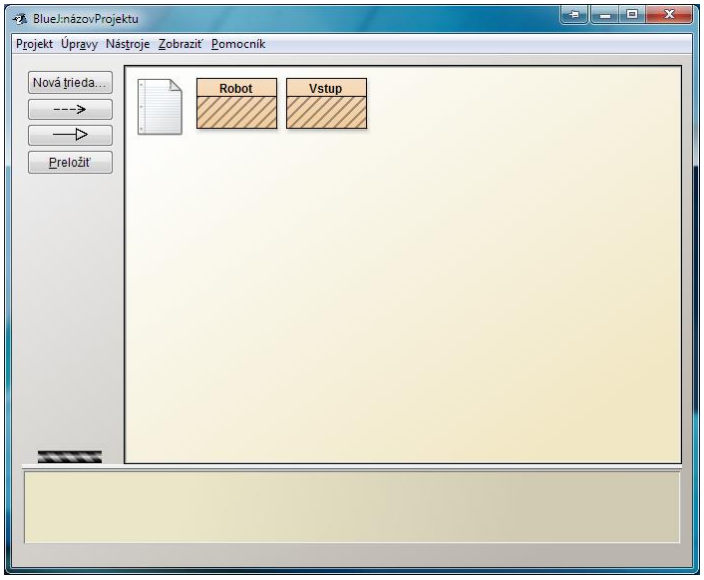
Pri použití priameho importu môžeme väčšinu krokov predchádzajúceho spôsobu vynechať. Stačí zvoliť položku „Projekt“ » „Import…“:



V dialógu, ktorý sa otvorí zvolíme priečinok, odkiaľ chceme importovať (napríklad priečinok „Triedy“):

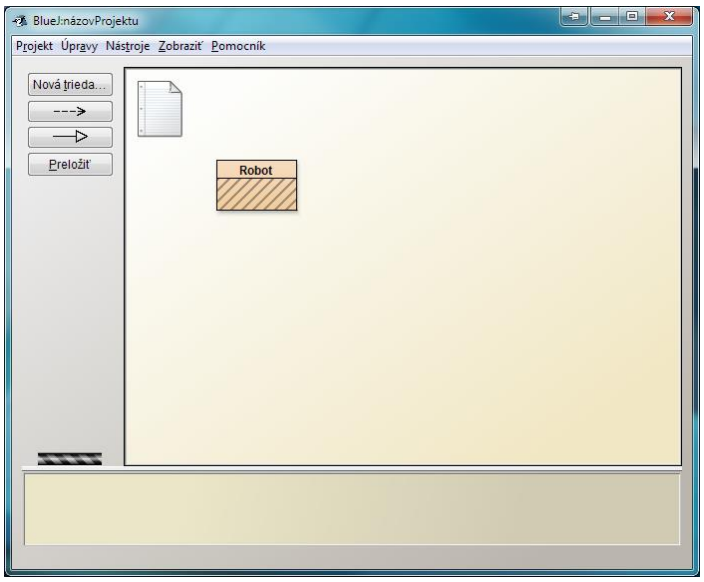


Potvrdíme tlačidlom „Importovať“, a ak všetko prebehne bez komplikácií, na ploche projektu sa objavia importované triedy:

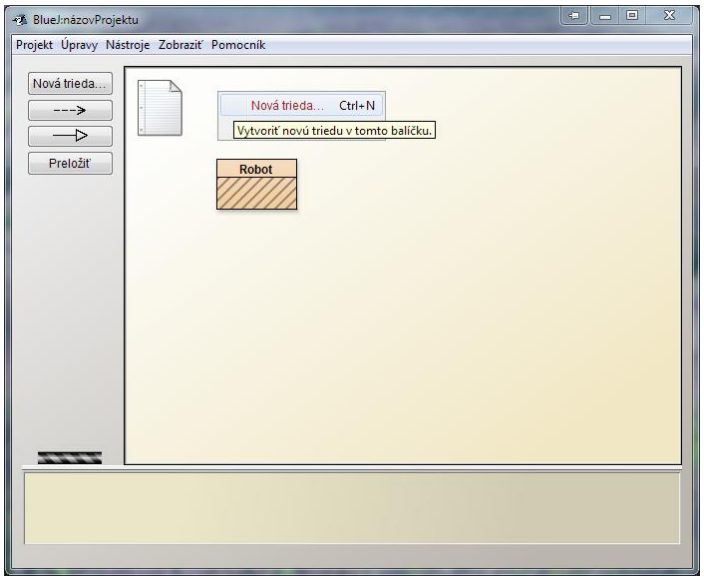


Vytvorenie novej triedy

Predpokladajme, že máme vytvorený nový projekt a importovanú triedu `GRobot`:

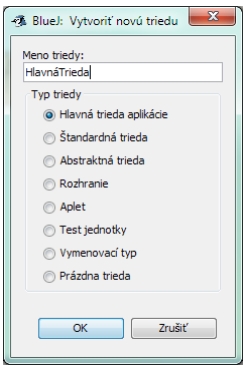


Novú triedu môžeme vytvoriť viacerými spôsobmi. Buď tlačidlom „Nová trieda…“ na ľavom paneli, alebo pomocou rovnomennej položky v kontextovej ponuke (vyvolanej sekundárnym[2] tlačidlom myši), prípadne stlačením klávesovej skratky Ctrl + N:

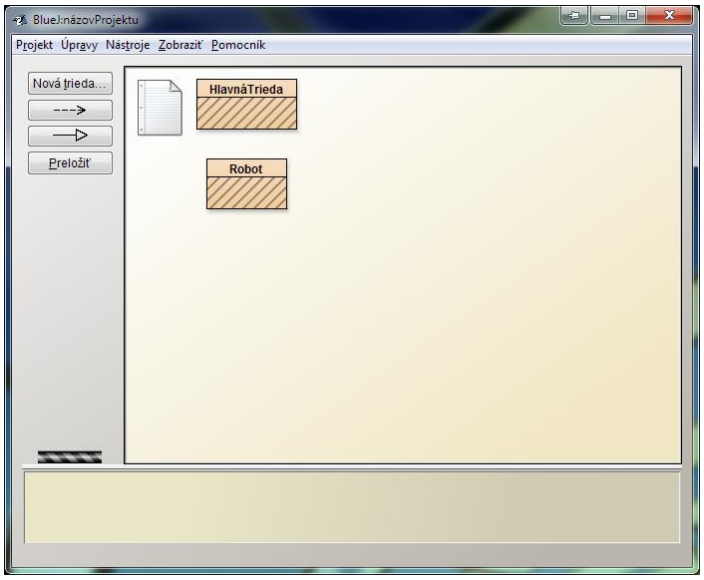


Po zvolení ľubovoľného spôsobu sa zobrazí dialóg na vytvorenie novej triedy, v ktorom musíme zadať meno triedy v súlade so zásadami tvorby identifikátorov v Jave (uvedené nižšie) a typ triedy (v tomto príklade „Hlavná trieda aplikácie“):

[2] Sekundárne tlačidlo myši je pri nastavení používania myši pre pravákov pravé tlačidlo myši a pri opačnom nastavení je to ľavé tlačidlo myši.



Po potvrdení tlačidlom „OK“ sa nová trieda objaví na ploche projektu:



Trieda smie byť pomenovaná výhradne v súlade so zásadami tvorby identifikátorov jazyka Java, to znamená (v zátvorkách je použitá syntax regulárnych výrazov):

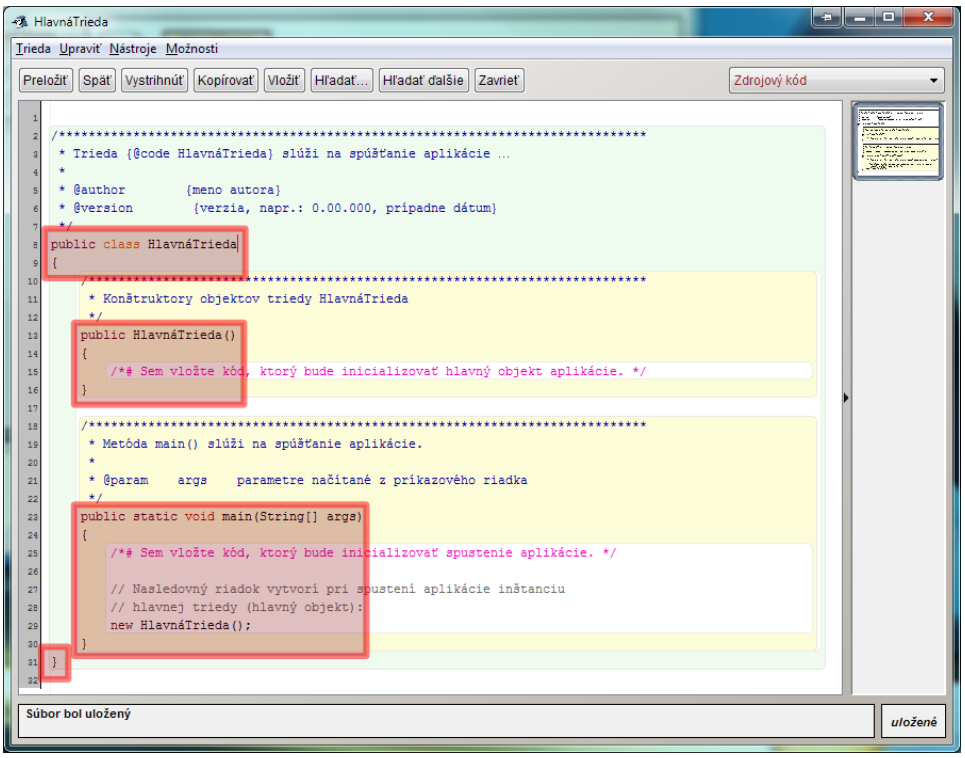
- prvý znak **nesmie** byť číslica (`0-9`), iba: písmeno anglickej (`a-z`, `A-Z`) alebo národnostnej abecedy (`č`, `ž`, `Š`, `Ľ`, `á`, `í`, `É`, `Ô`…) alebo znaky: spodná vodorovná čiara (`_`) a dolár (`$`),
- pre ďalšie znaky identifikátora platia rovnaké pravidlá, ale navyše môžeme použiť aj znaky číslic (`0-9`),
- ako identifikátor **nesmie** byť použité rezervované slovo Javy (`private`, `static`, `class`, `int`, `double`, `if`, `else`, `while`, `break`…).

Odvodenie triedy

Predpokladajme, že máme vytvorený nový projekt, importovanú triedu `GRobot` a vytvorenú novú triedu `HlavnáTrieda`. **Zdrojový kód** (text zapísaný pravidlami programovacieho jazyka) triedy potrebujeme otvoriť vždy, keď chceme do určitej triedy niečo zapísať (naprogramovať). Otvoríme ho buď dvojitým kliknutím na triedu, alebo zvolením položky „Otvoriť v editore“ v kontextovej ponuke triedy (sekundárne[3] tlačidlo myši). **Otvorte hlavnú triedu!**

V závislosti od verzie šablón, ktorú máte nainštalovanú vo vašej verzii BlueJ-a, sa môže zdrojový kód hlavnej triedy odlišovať. Kód vytvorený podľa šablóny nainštalovanej na našom stroji vyzeral tak, ako je zobrazený na nasledujúcom obrázku:

[3] Sekundárne tlačidlo myši je pri nastavení používania myši pre pravákov pravé tlačidlo myši a pri opačnom nastavení je to ľavé tlačidlo myši.



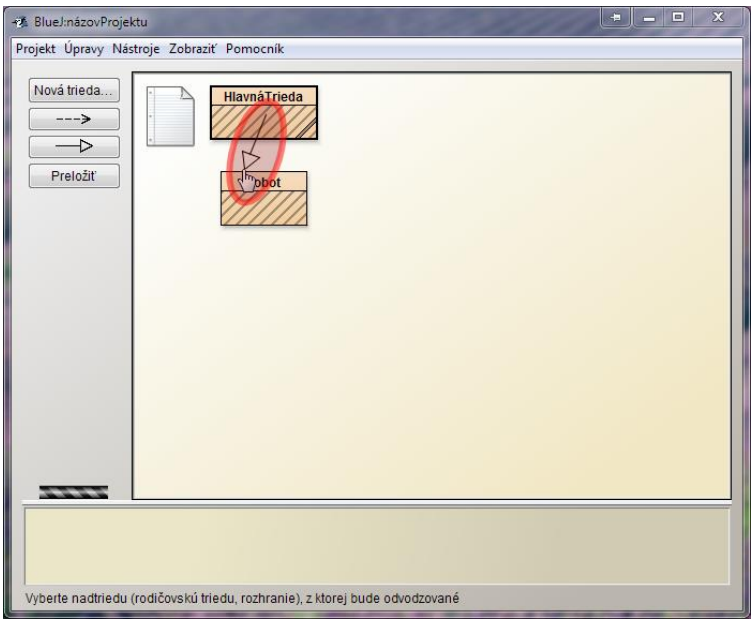
Najdôležitejšie časti sú v obrázku vyznačené červenou farbou. Keď si odmyslíme všetky komentáre, ide o túto časť kódu:

```
public class HlavnáTrieda
{
    public HlavnáTrieda()
    {

    }

    public static void main(String[] args)
    {
        new HlavnáTrieda();
    }
}
```

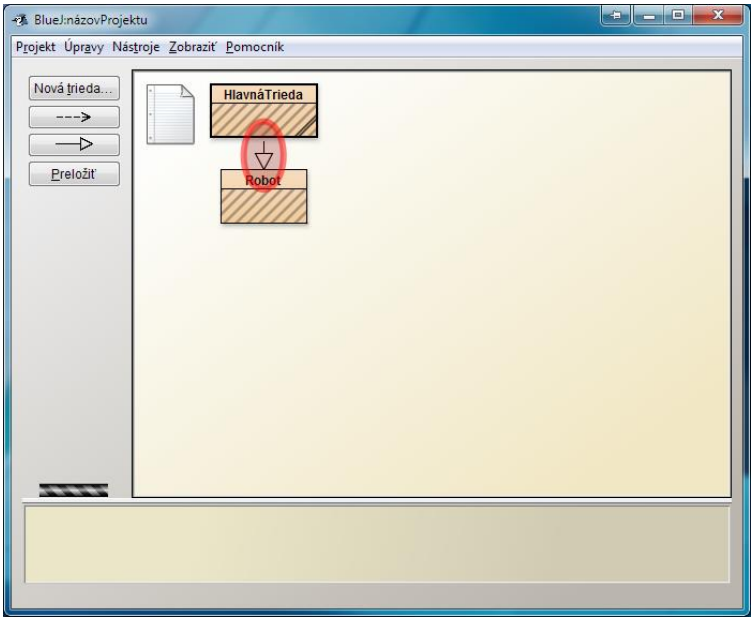
Čokoľvek potrebujeme v triede zmeniť, musíme vykonať tu, formou zápisu podľa pravidiel programovacieho jazyka Java. Jediný krok sa dá urobiť dvojako – **graficky** aj/alebo **zápisom** v zdrojovom kóde – odvodenie jednej triedy od druhej (v rámci projektu).

Prepnite sa do hlavného okna projektu! Kliknite na tretie tlačidlo zhora na ľavom paneli (tlačidlo so šípkou s neprerušovanou čiarou). Tým ste zapli režim odvádzania tried. Teraz stlačte a držte stlačené primárne[4] tlačidlo myši nad dcérskou triedou, ktorou bude v našom prípade `HlavnáTrieda`, potiahnite kurzor myši nad rodičovskú triedu, ktorou je v našom prípade trieda `GRobot` a uvoľnite tlačidlo myši. Jeden časový okamih procesu je zachytený na nasledujúcom obrázku (zvýraznený červenou elipsou):

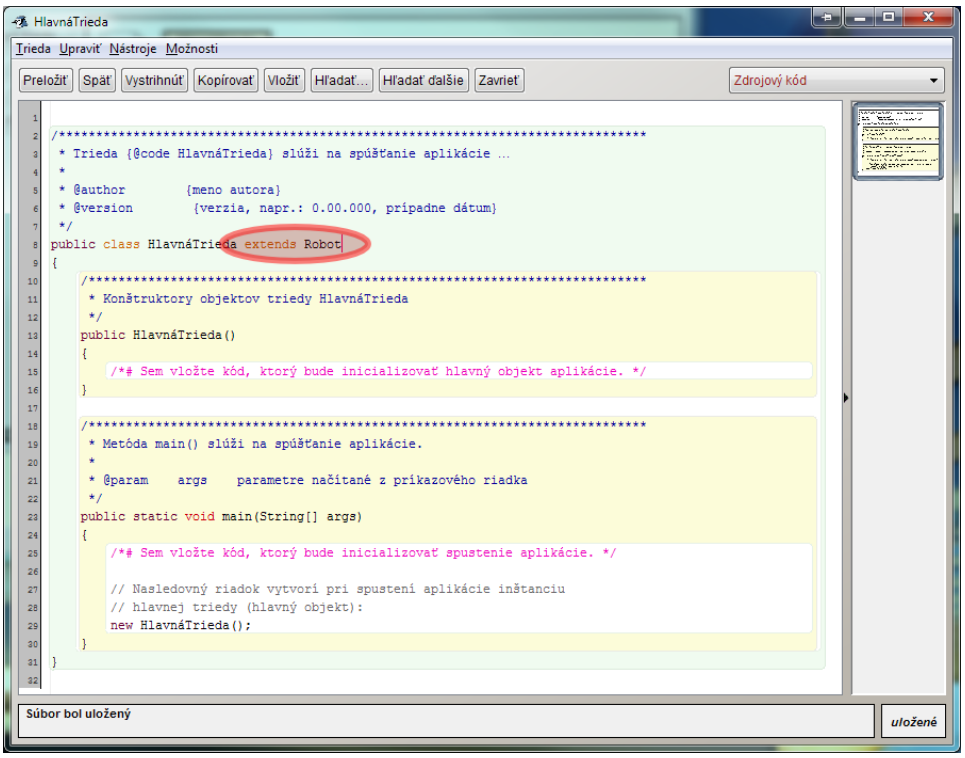
[4] Primárne tlačidlo myši je pri nastavení používania myši pre pravákov ľavé tlačidlo myši a pri opačnom nastavení je to pravé tlačidlo myši.



Výsledok procesu je zvýraznený červenou elipsou na nasledujúcom obrázku:



Odvodili ste triedu `HlavnáTrieda` od triedy `GRobot`. Keď teraz otvoríte zdrojový kód hlavnej triedy, uvidíte drobnú zmenu, ktorú sme tiež znázornili červenou elipsou na ďalšom obrázku:



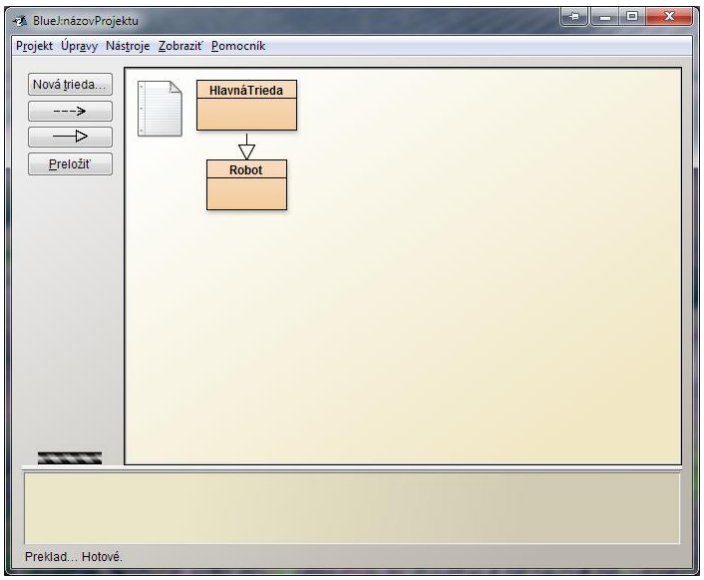
Slová `extends GRobot` (vo voľnom preklade „rozširuje“ triedu `GRobot`, resp. „je odvodené“ od triedy `GRobot`) by sme do zdrojového kódu mohli zapísať aj priamo ručne a nič by sme tým nepokazili. BlueJ by to automaticky rozpoznal a šípku medzi triedy by pridal za vás. Ak sa vám zdá byť napísanie textu rýchlejšie než ťahanie myši, môžete vykonať odvodenie aj priamym zápisom do zdrojového kódu. Oba spôsoby sú rovnocenné.

Kompilácia (preklad) a spustenie projektu

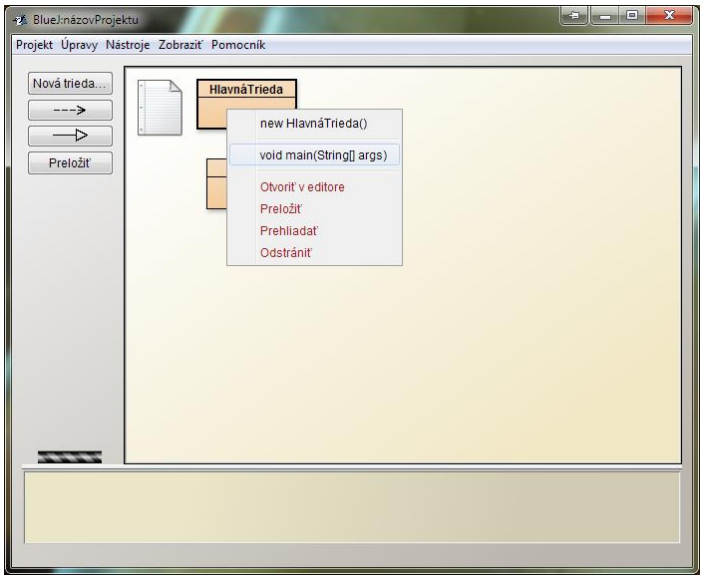
Na spustenie projektu je nevyhnutná bezchybná kompilácia všetkých súborov projektu (máme na mysli zdrojové súbory). Slovenský ekvivalent termínu kompilácia je (pri programovaní) „preklad“. (Doslovným ekvivalentom by bolo slovo „zloženie“, tento termín sa však v tomto kontexte nepoužíva.) Ak máte v programe syntaktickú alebo inú chybu, ktorá nedovolí jeho bezchybné preloženie, musíte ju najskôr odstrániť.

V BlueJ-i vykonáme preklad všetkých súborov projektu pomocou tlačidla „Preložiť“ v hlavnom okne projektu. Rovnaké tlačidlo nájdeme v každom okne zdrojového kódu, to však slúži iba na preloženie konkrétneho súboru, prípadne (ak je to potrebné) súborov, od ktorých je určitý súbor závislý. Namiesto tlačidla môžeme použiť rovnomennú položku v rolovacej ponuke „Nástroje“ alebo klávesovú skratku Ctrl + K.

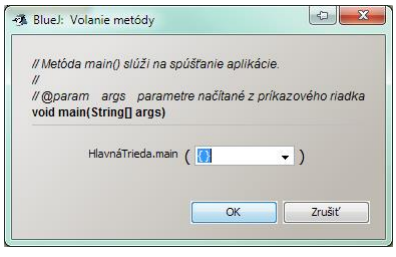
Stav po úspešnom preložení projektu môžeme vidieť na nasledujúcom obrázku:



Vstupným bodom každého nami vytváraného programu bude hlavná metóda `main`. Tá je automaticky vytvorená v šablóne hlavnej triedy. Projekt spustíte tak, že kliknete na hlavnú triedu sekundárnym[5] tlačidlom myši a zvolíte položku reprezentujúcu spustenie hlavnej metódy `main` (ak je všetko v poriadku, položka by mala obsahovať text „void main(String[] args)“). Kontextová ponuka by mala vyzerať tak, ako na nasledujúcom obrázku:

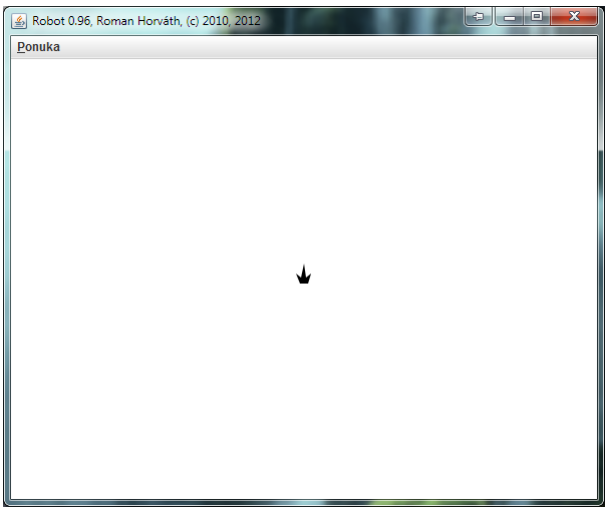


Zobrazí sa dialóg spustenia metódy s predvolenou hodnotou argumentov: {} (pozri nasledujúci obrázok). Túto hodnotu v dialógu ponechajte, ak by ste ju omylom zmazali, budete ju musieť ručne dopísať (ak sa nebude nachádzať v kombinovanom zozname argumentu). Dialóg potvrďte tlačidlom OK.



Prázdny projekt využívajúci triedu `GRobot`, bude vyzerať tak, ako je znázornené na nasledujúcom obrázku:

[5] Sekundárne tlačidlo myši je pri nastavení používania myši pre pravákov pravé tlačidlo myši a pri opačnom nastavení je to ľavé tlačidlo myši.



Príloha 1 – trieda Vstup

Trieda `Vstup`.

Príloha 2 – skupina tried GRobot

Skupina tried `GRobot`. (Odporúčame prevziať a použiť Generátor projektov BlueJ.)

Úvod

V rámci tohto materiálu sa budeme zaoberať dvoma spôsobmi riešenia rovnakého problému, konkrétne naprogramovania jednoduchej hry pre jedného hráča, keď má hráč za úlohu pomocou prievozníka/pltníka previezť na opačný breh rieky troch cestujúcich – vlka, kozu a kapustu, ktorí majú vlastné stravovacie návyky a špeciálne schopnosti. Príbeh znie takto:

> *„Prievozník vlastní plť, na ktorej prevezie okrem seba maximálne jedného pasažiera. Jedného dňa sa ocitol v situácii, keď mal bezpečne previezť na druhý breh rieky neobvyklú trojicu pasažierov. Vlka, kozu a kapustu. Vlk má chuť na kozu, avšak kapusta, ktorá ich sprevádza, je zázračná – dokáže kozu pred vlkom ochrániť. To síce koza vie, len čo sa však ocitne s kapustou sama, podľahne pokušeniu a zožerie ju. Prievozník sám je tiež autorita, takže v jeho bezprostrednej blízkosti nezje ani vlk kozu, ani koza kapustu. Problém teda je, že sa na jednom brehu rieky nesmú ocitnúť osamote vlk s kozou, ani koza s kapustou. V akom poradí ich previezť?“*

V prvej časti materiálu pôjdeme po „rýchlej línii“ – uspokojíme sa textovým režimom, v ktorom by profesionálny programátor mal mať problematiku vyriešenú za niekoľko hodín. (Samozrejme, že pri štúdiu nie je možné postupovať tempom profesionála.) Ukážeme si niekoľko princípov, ktoré neskôr využijeme pri naprogramovaní rovnakej hry v grafickom režime. Pri zapojení animovanej grafiky a zvuku do hry, by aj na profesionála čakala práca na niekoľko desiatok hodín… Do tohto času však zahŕňame aj riešenie problémov, ktoré sú v rámci tohto materiálu už vyriešené.

Textový projekt 1

Najskôr si ukážme riešenie tohto problému v textovom režime. Otvoríme BlueJ, vytvoríme nový projekt, importujeme triedu `Vstup` a vytvoríme novú triedu `VlkKozaKapusta`, ktorú odvodíme od triedy `Vstup`. Triedou `Vstup` sa podrobnejšie zaoberať nebudeme, je k dispozícii v prílohách tejto kapitoly. Vyčistite a upravte triedu `VlkKozaKapusta` do nasledujúceho stavu:

```
public class VlkKozaKapusta extends Vstup
{
    private VlkKozaKapusta()
    {

    }

    public static void main(String[] args)
    {
        new VlkKozaKapusta();
    }
}
```

Zo šablóny sme vymazali všetky komentáre (pre jednoduchosť, inak takýto postup **neodporúčame**) a zostala trieda, ktorá obsahuje iba konštruktor a hlavnú metódu. Hlavná metóda obsahuje jediný riadok – vytvorenie inštancie tejto triedy. Trieda má súkromný konštruktor, ten budeme považovať za vstupný bod programu.

Keďže budeme pracovať v textovom režime, vystačíme si s jednoduchými definíciami. Potrebujeme vytvoriť údajové štruktúry a definovať premenné, ktoré budú reprezentovať všetky stavy hry. Aktuálne stačí, ak si zapamätáme to, čo sa nachádza na jednotlivých brehoch rieky a pozíciu prievozníka. Na reprezentovanie obsahu brehov môžeme použiť dve jednoduché trojprvkové polia, ktoré môžeme hneď naplniť týmto spôsobom:

```
String[] ľavýBreh  = {"nič", "nič", "nič"};
String[] pravýBreh = {"vlk", "koza", "kapusta"};
```

V rámci dodržania dobrých programátorských návykov definujeme tieto polia ako súkromné a môžeme im priradiť aj modifikátor „finálny“, ktorý bude vypovedať o tom, že polia zostanú počas behu také, ako sme ich definovali prvýkrát – trojprvkové, pričom obsah jednotlivých prvkov stále môžeme meniť. Dostávame:

```
private final String[] ľavýBreh  = {"nič", "nič", "nič"};
private final String[] pravýBreh = {"vlk", "koza", "kapusta"};
```

Polohu prievozníka, keďže je iba dvojstavová (pravý alebo ľavý breh), môžeme definovať pomocou booleovskej premennej:

```
private boolean prievozníkJeNaPravomBrehu = true;
```

Teraz by sme mali definovať metódu, ktorá tieto premenné využije na výpis aktuálneho stavu hry, ktorý budeme musieť neustále vypisovať, aby sme pomohli predstavivosti hráča. Metódu definujeme ako súkromnú. Nazveme ju jednoducho – `vypíšStav`:

```
private void vypíšStav()
{

}
```

Do tela metódy vložíme kód, ktorý doslova „skonštruuje“ stav hry. Najprv vypíšeme obsah ľavého brehu. Všimnite si, že zatiaľ používame iba metódu `System.out.print`, ktorá zachováva výpis v jednom riadku. Na výpis všetkého, čo sa nachádza na ľavom brehu, budeme potrebovať využiť nielen pole `ľavýBreh`, ale i premennú `prievozníkJeNaPravomBrehu`, pretože aj polohu prievozníka musíme vhodným spôsobom oznámiť hráčovi.

Na výpis prvkov poľa použijeme cyklus `for`, v rámci ktorého odfiltrujeme iba zmysluplné položky, teda všetko okrem prázdneho `"nič"`. Spolu s výpisom prítomnosti prievozníka a výpisom úvodného textu získame takýto kód:

```
System.out.print("\nNa ľavom brehu je: ");

if (!prievozníkJeNaPravomBrehu)
{
    System.out.print(" [prievozník]");
}

for (int i = 0; i < ľavýBreh.length; ++i)
{
    if (!ľavýBreh[i].equals("nič"))
    {
        System.out.print(" " + ľavýBreh[i]);
    }
}
```

Čiže, ak prievozník nie je na pravom brehu, je automaticky na ľavom a vypíšme ho tam. Za ním sa vo výpise objaví každá položka, ktorá sa nerovná reťazcu `"nič"` (čiže buď vlk, koza, alebo kapusta). Na začiatku bude výpis prázdny, pretože všetko sa spočiatku nachádza na pravom brehu, no neskôr môže vyzerať napríklad takto:

```
Na ľavom brehu je:  [prievozník] koza
```

Podobným spôsobom vytvoríme aj výpis pravého brehu a na konci metódy môžeme zavolať príkaz `System.out.println`, ktorým odriadkujeme výpis pravého brehu. Mali by sme zabezpečiť, aby aj jednotlivé brehy boli vypísané na samostatných riadkoch. Môžeme to zabezpečiť napríklad znakovou entitou `\n` (pozri prvý riadok kódu na výpis ľavého brehu) umiestnenou na začiatok úvodného textu pre pravý breh.

Metódu nesmieme zabudnúť zavolať, pretože bez volania metódy, akoby ani nejestvovala. Volanie môžeme umiestniť do konštruktora, ktorý bude po tejto jednoduchej úprave vyzerať takto:

```
private VlkKozaKapusta()
{
    vypíšStav();
}
```

Ak budete postupovať správne, prvý výpis po skompilovaní a spustení programu, bude vyzerať takto (vrátane vynechaného prvého riadka):

```
Na ľavom brehu je:
Na pravom brehu je: [prievozník] vlk koza kapusta
```

Ak by ste si predsa len nevedeli dať rady s poskladaním príkladu, skompletizovanú verziu nájdete v prílohe.

Príloha 3 – úvodná fáza textového projektu

```
public class VlkKozaKapusta extends Vstup
{
    private final String[] ľavýBreh  = {"nič", "nič", "nič"};
    private final String[] pravýBreh = {"vlk", "koza", "kapusta"};
    private boolean prievozníkJeNaPravomBrehu = true;
```

```
    private VlkKozaKapusta()
    {
        vypíšStav();
    }

    private void vypíšStav()
    {
        System.out.print("\nNa ľavom brehu je: ");

        if (!prievozníkJeNaPravomBrehu)
        {
            System.out.print(" [prievozník]");
        }

        for (int i = 0; i < ľavýBreh.length; ++i)
        {
            if (!ľavýBreh[i].equals("nič"))
            {
                System.out.print(" " + ľavýBreh[i]);
            }
        }

        System.out.print("\nNa pravom brehu je: ");

        if (prievozníkJeNaPravomBrehu)
        {
            System.out.print(" [prievozník]");
        }

        for (int i = 0; i < pravýBreh.length; ++i)
        {
            if (!pravýBreh[i].equals("nič"))
            {
                System.out.print(" " + pravýBreh[i]);
            }
        }

        System.out.println();
    }

    public static void main(String[] args)
    {
        new VlkKozaKapusta();
    }
}
```

Textový projekt 2

V predchádzajúcej kapitole sme vytvorili nový projekt, definovali sme základné stavy a zabezpečili ich výstup. Teraz postúpme o krok ďalej. Oživme kostru hry. Na to, aby hra dokázala reagovať na podnety hráča, musí prijímať vstupy. Na zabezpečenie vstupu použijeme funkcionalitu naprogramovanú v triede `Vstup`. V konštruktore deklarujeme reťazec `príkaz`, do ktorého budeme pomocou metódy `načítajReťazec` načítavať príkazy hráča:

```
String príkaz;
```

```
príkaz = načítajReťazec("Príkaz");
```

Táto dvojica riadkov spôsobí, že sa na obrazovke objaví nasledujúci automatický výpis výzvy pre hráča:

```
Príkaz (reťazec):
```

Po nej program čaká na zadanie textu a jeho potvrdenie klávesom Enter. Teraz potrebujeme rozpoznať „príkazy“ zadávané hráčom. Najjednoduchšie pre hráča by bolo, keby zadal a potvrdil jednoducho tú postavu, ktorá sa má previezť na druhý breh (vlk, koza, kapusta). Tomu prispôsobíme aj výzvu a namiesto textu „Príkaz“ sa hráča spýtame: „Čo má prievozník previezť?“. (So skloňovaním odpovede: vlk**a**, koz**u**, kapust**u**, sa nebudeme zaoberať.)

Dobrým zvykom je definovať na každý celistvý a dostatočne elementárny úkon samostatnú metódu. V súlade s tým vytvoríme samostatnú metódu na analýzu odpovede na otázku „Čo má prievozník previezť?“ Metódu naprogramujeme tak, aby zároveň spolu s analýzou vykonala aj odozvu (bude to komplexná metóda). Prijme reťazec zadaný používateľom, ktorý bude považovať za objekt určený na prevezenie na druhú stranu. Bude sa riadiť aktuálnou polohou prievozníka. Základná kostra metódy teda bude vyzerať takto:

```
private void prevez(String čo)
{
    if (prievozníkJeNaPravomBrehu)
    {

    }
    else
    {

    }
}
```

Keď je prievozník na pravom brehu, budeme vyhľadávať zadaný objekt v poli `pravýBreh`, v opačnom prípade v poli `ľavýBreh`. Úkon vyhľadávania je tiež dostatočne elementárny, takže je vhodné umiestniť ho do samostatnej metódy prijímajúcej reťazec a vracajúcej index prvku zhodného so zadaným reťazcom:

```
private int nájdiNaPravomBrehu(String čo)
{
    for (int i = 0; i < pravýBreh.length; ++i)
    {
        if (čo.equalsIgnoreCase(pravýBreh[i])) return i;
    }

    return -1;
}
```

Metóda v cykle prehľadá pole (pre pravý breh je to `pravýBreh`), na hľadanie použije metódu `equalsIgnoreCase` triedy `String`, ktorá porovnáva reťazce ignorujúc veľkosť písmen. Keď zadaný reťazec v poli nenájde, vráti hodnotu `-1`, čo signalizuje stav „nenájdené“. Inak získame index prvku, kde bol reťazec nájdený. Podobne definujeme aj metódu `nájdiNaĽavomBrehu`.

Obidve metódy potom využijeme v rámci metódy `prevez`, konkrétne v štruktúre jednoduchého vetvenia `if-else`. Keď je prievozník na pravom brehu, hľadáme zadaný objekt na pravom brehu a na ľavom brehu budeme hľadať prázdne miesto (`"nič"`):

```
int i = nájdiNaPravomBrehu(čo);
```

```
int j = nájdiNaĽavomBrehu("nič");
```

Na základe hodnôt, ktoré získame volaním metód `nájdiNaPravomBrehu` a `nájdiNaĽavomBrehu`, vykonáme výmenu prvkov medzi poľami `ĽavýBreh` a `pravýBreh` a presunieme prievozníka na opačnú stranu rieky (zmeníme hodnotu premennej `prievozníkJeNaPravomBrehu`). Musíme však najskôr vyriešiť prípady, keď niektorá z hľadaných hodnôt nebola nájdená (to znamená, že buď sa na pravom brehu nenachádza objekt, ktorý zadal hráč a ktorý máme uložený v argumente `čo`, alebo na ľavom brehu už nie je voľné miesto `"nič"`):

```
if (i == -1)
{
    System.out.println("Na ľavom brehu nie je " + čo + ".");
}
else if (j == -1)
{
    System.out.println("Na pravom brehu nie je voľné miesto.");
}
else
{
    ĽavýBreh[j] = pravýBreh[i];
    pravýBreh[i] = "nič";
    prievozníkJeNaPravomBrehu = !prievozníkJeNaPravomBrehu;
}
```

Celú túto viacnásobnú podmienku spolu s hľadaním objektov na brehoch (pozri dva riadky kódu vyššie) musíme umiestniť do hlavnej vetvy metódy `prevez` (na mieste, ktoré zvýrazňuje komentár):

```
private void prevez(String čo)
{
    if (prievozníkJeNaPravomBrehu)
    {
        /*# SEM! */
        // int i = …
        // …
        // if (i == -1…
    }
    else
    {

    }
}
```

Do vedľajšej vetvy (`else`) naprogramujeme protipól celého bloku. Získame kompletnú metódu `prevez`, ktorú môžeme využiť v konštruktore:

```
private VlkKozaKapusta()
{
    String príkaz;

    vypíšStav();
    príkaz = načítajReťazec("Čo má prievozník previezť?");
    prevez(príkaz);
    vypíšStav();
}
```

Ak teraz po spustení hry zadáme a potvrdíme reťazec „koza“, dostaneme takýto výstup:

```
Na ľavom brehu je:
Na pravom brehu je: [prievozník] vlk koza kapusta
Čo má prievozník previezť? (reťazec): koza

Na ľavom brehu je:  [prievozník] koza
Na pravom brehu je: vlk kapusta
```

Opäť, ak by ste mali problém so skompletizovaním príkladu, nájdete ho v prílohe.

Príloha 4 – druhá fáza textového projektu

```
public class VlkKozaKapusta extends Vstup
{
    private final String[] ľavýBreh  = {"nič", "nič", "nič"};
    private final String[] pravýBreh = {"vlk", "koza", "kapusta"};
    private boolean prievozníkJeNaPravomBrehu = true;

    private VlkKozaKapusta()
    {
        String príkaz;

        vypíšStav();
        príkaz = načítajReťazec("Čo má prievozník previezť?");
        prevez(príkaz);
        vypíšStav();
    }

    private void vypíšStav()
    {
        System.out.print("\nNa ľavom brehu je: ");

        if (!prievozníkJeNaPravomBrehu)
        {
            System.out.print(" [prievozník]");
        }

        for (int i = 0; i < ľavýBreh.length; ++i)
        {
            if (!ľavýBreh[i].equals("nič"))
            {
                System.out.print(" " + ľavýBreh[i]);
            }
        }

        System.out.print("\nNa pravom brehu je: ");

        if (prievozníkJeNaPravomBrehu)
        {
            System.out.print(" [prievozník]");
        }

        for (int i = 0; i < pravýBreh.length; ++i)
        {
            if (!pravýBreh[i].equals("nič"))
            {
                System.out.print(" " + pravýBreh[i]);
```

```
            }
        }

        System.out.println();
    }

    private int nájdiNaPravomBrehu(String čo)
    {
        for (int i = 0; i < pravýBreh.length; ++i)
        {
            if (čo.equalsIgnoreCase(pravýBreh[i])) return i;
        }

        return -1;
    }

    private int nájdiNaĽavomBrehu(String čo)
    {
        for (int i = 0; i < ľavýBreh.length; ++i)
        {
            if (čo.equalsIgnoreCase(ľavýBreh[i])) return i;
        }

        return -1;
    }

    private void prevez(String čo)
    {
        if (prievozníkJeNaPravomBrehu)
        {
            int i = nájdiNaPravomBrehu(čo);
            int j = nájdiNaĽavomBrehu("nič");

            if (i == -1)
            {
                System.out.println("Na ľavom brehu nie je " + čo + ".");
            }
            else if (j == -1)
            {
                System.out.println("Na pravom brehu nie je voľné miesto.");
            }
            else
            {
                ľavýBreh[j] = pravýBreh[i];
                pravýBreh[i] = "nič";
                prievozníkJeNaPravomBrehu = !prievozníkJeNaPravomBrehu;
            }
        }
        else
        {
            int i = nájdiNaĽavomBrehu(čo);
            int j = nájdiNaPravomBrehu("nič");

            if (i == -1)
            {
                System.out.println("Na pravom brehu nie je " + čo + ".");
            }
            else if (j == -1)
```

```
            {
                System.out.println("Na ľavom brehu nie je voľné miesto.");
            }
            else
            {
                pravýBreh[j] = ľavýBreh[i];
                ľavýBreh[i] = "nič";
                prievozníkJeNaPravomBrehu = !prievozníkJeNaPravomBrehu;
            }
        }
    }

    public static void main(String[] args)
    {
        new VlkKozaKapusta();
    }
}
```

Textový projekt 3

Projekt sme dostali do fázy, v ktorej sme schopní prenášať objekty z jedného brehu na druhý, avšak momentálne iba raz. Na zopakovanie celého doterajšieho procesu by sme potrebovali cyklus. Z troch druhov cyklov: `for`, `while` a `do-while` môžeme vylučovacou metódou vyselektovať ten pravý:

1. Nevieme na koľký raz sa individuálnemu hráčovi podarí hru vyriešiť, tým odpadá možnosť využitia cyklu `for`.
2. Cyklus budeme musieť vykonať minimálne raz (aby hranie hry bolo vôbec možné), a to nám indikuje použitie cyklu `do-while`.

Jediné, čo teraz potrebujeme, je vymyslieť podmienku (okolnosti), vďaka ktorej sa cyklus ukončí. Prvé, čo by nám malo napadnúť, je, že sa hra ukončí, keď bude úspešne alebo neúspešne dokončená, avšak tak ďaleko nie sme, a okrem toho, každá hra by mala hráčovi dovoliť ukončenie kedykoľvek počas hrania. Najmä pre druhý dôvod by sme mali vymyslieť univerzálny spôsob ukončenia. Navrhujem „špeciálny príkaz“, na ktorý hra zareaguje, to jest, keď hráč namiesto objektu, ktorý má byť prevezený, zadá špeciálne kľúčové slovo, napríklad „koniec“. Ak máme ujasnený takýto cieľ, môžeme sa posunúť ďalej:

Z nasledujúcich riadkov:

```
vypíšStav();
príkaz = načítajReťazec("Čo má prievozník previezť?");
prevez(príkaz);
vypíšStav();
```

Stačí umiestniť do cyklu posledné tri. Načítame príkaz, posunieme ho do metódy prevez, vypíšeme stav a celé to zopakujeme, kým hráč nezadá `koniec`:

```
vypíšStav();

do {
    príkaz = načítajReťazec("Čo má prievozník previezť?");
    prevez(príkaz);
    vypíšStav();
} while (!príkaz.equalsIgnoreCase("koniec"));
```

Avšak je absurdné, aby sa hra pokúšala „previezť na druhú stranu rieky príkaz koniec“. Preto to vyriešime dodatočnou podmienkou:

```
vypíšStav();

do {
    príkaz = načítajReťazec("Čo má prievozník previezť?");

    if (!príkaz.equalsIgnoreCase("koniec"))
    {
        prevez(príkaz);
        vypíšStav();
    }

} while (!príkaz.equalsIgnoreCase("koniec"));
```

Teraz môžeme prevážať objekty z jedného brehu na druhý a späť, kým nás to neomrzí. Keď zadáme „koniec“, hra sa skončí a funguje aj prechod pltníka naprázdno (keď zadáme „nič“):

```
Na ľavom brehu je:
Na pravom brehu je: [prievozník] vlk koza kapusta
Čo má prievozník previezť? (reťazec): koza

Na ľavom brehu je:  [prievozník] koza
Na pravom brehu je: vlk kapusta
Čo má prievozník previezť? (reťazec): nič

Na ľavom brehu je:  koza
Na pravom brehu je: [prievozník] vlk kapusta
Čo má prievozník previezť? (reťazec): kapusta
```

Bez toho, aby mohol prejsť pltník na prázdno na druhý breh, by sa hra nedala dokončiť. Momentálne však máme v hre jednu nelogickosť. Ak sa hneď na prvýkrát pokúsime prejsť na druhú stranu naprázdno, hra nám to nedovolí:

```
Na ľavom brehu je:
Na pravom brehu je: [prievozník] vlk koza kapusta
Čo má prievozník previezť? (reťazec): nič
Na ľavom brehu nie je nič.
```

Je síce malá pravdepodobnosť, že to hráč bude skúšať, ani úspešné dokončenie hry tým nie je nijako ohrozené. V podstate by sme to nemuseli riešiť, ale ak nám to prekáža, dá sa to veľmi jednoducho napraviť. Stačí rozšíriť definície polí jednotlivých brehov o jeden prázdny prvok. Hráč o ňom nemusí vedieť (prázdne prvky sa na výstupe nevypisujú), nikomu to prekážať nebude, iba sa tým mierne „poopraví“ fungovanie hry:

```
private final String[] ľavýBreh  = {"nič", "nič", "nič", "nič"};
private final String[] pravýBreh = {"vlk", "koza", "kapusta", "nič"};
```

To bolo mierne „kozmetické“ odbočenie, vráťme sa späť k téme. Do úplného dokončenia hry nám chýbajú dve podstatné veci:

1. Overenie prípadov, keď sa pasažieri požierajú navzájom.
2. Overenie úspešného dokončenia hry – všetci pasažieri sú na druhom brehu rieky.

Optimálne bude, keď pre každý bod naprogramujeme samostatnú metódu, ktorú vzápätí využijeme v hlavnom vlákne programu – v `do-while` cykle. Metódu overujúcu požieranie definujme s návratovou hodnotou typu `boolean`, ktorá bude vypovedať o tom, či bol niektorý z pasažierov zožraný. Nazvime ju

napríklad `niektoNiekohoZožral`. Rovnako i druhú metódu, tá bude slúžiť na overenie, či sa hra už úspešne skončila definujme s návratovou hodnotou typu boolean a nazvime ju napríklad `hraSkončila`.

Pre potreby obidvoch metód bude užitočné definovať pomocnú metódu zisťujúcu na ktorom brehu sa práve nachádza zadaný pasažier. Metóda bude fungovať rovnako ako metódy `nájdiNaPravomBrehu` a `nájdiNaĽavomBrehu`, ibaže namiesto celočíselnej hodnoty indexu bude vracať logickú hodnotu majúcu podobný význam ako premenná `prievozníkJeNaPravomBrehu`. Keďže pre potreby zisťovania úspešného dokončenia hry (metóda `hraSkončila`) je výhodnejšie vedieť, či je pasažier na ľavom brehu, definujeme túto metódu presne v tomto význame. Nazveme ju `jeNaĽavomBrehu` a jej definícia, ako bolo povedané, bude vyzerať podobne ako definícia metódy `nájdiNaĽavomBrehu`:

```
private boolean jeNaĽavomBrehu(String čo)
{
    for (int i = 0; i < ľavýBreh.length; ++i)
    {
        if (čo.equalsIgnoreCase(ľavýBreh[i])) return true;
    }

    return false;
}
```

Jej služby potom využijeme na začiatku metód `niektoNiekohoZožral` a `hraSkončila` – postup bude taký, že najskôr na začiatku každej metódy pre každého pasažiera zistíme, na ktorom brehu sa nachádza, a na základe toho vytvoríme logiku toho, či niektorý z nich niekoho zožral, alebo či sa už hra skončila. Prvý problém (detekcia vzájomného požierania pasažierov) vyžaduje buď dobrú predstavivosť a logiku, alebo pero a papier na zhrnutie všetkých možností. Ak zosumarizujeme fakty – máme štyri objekty, ktoré môžu jestvovať v dvoch stavoch: vlk, koza, kapusta i prievozník sa môžu nachádzať na pravom alebo ľavom brehu rieky – vyplynie nám z toho, že celkovo môže nastať 16 rôznych situácií.

Ťažšie by sa nám pracovalo, keby išlo o úplne abstraktné objekty, no i tak si môžeme ukázať, ako sa pri riešení takýchto problémov dá postupovať s pomocou tabuliek a zoskupovania možností. Zhrňme najskôr do jednej tabuľky všetky možné situácie:

Vlk	Koza	Kapusta	Prievozník	Komentáre
pravý	pravý	pravý	pravý	*[počiatočný stav]*
pravý	pravý	pravý	ľavý	pasažieri sa strážia navzájom
pravý	pravý	ľavý	pravý	kozu stráži prievozník
pravý	**pravý**	ľavý	ľavý	**vlk zožral kozu**
pravý	ľavý	pravý	pravý	nikto nikoho neohrozuje
pravý	ľavý	pravý	ľavý	nikto nikoho neohrozuje
pravý	**ľavý**	**ľavý**	pravý	**koza zožrala kapustu**
pravý	ľavý	ľavý	ľavý	kapustu stráži prievozník
ľavý	pravý	pravý	pravý	kapustu stráži prievozník
ľavý	**pravý**	**pravý**	ľavý	**koza zožrala kapustu**

Vlk	Koza	Kapusta	Prievozník	Komentáre
ľavý	pravý	ľavý	pravý	nikto nikoho neohrozuje
ľavý	pravý	ľavý	ľavý	nikto nikoho neohrozuje
ľavý	**ľavý**	pravý	pravý	**vlk zožral kozu**
ľavý	ľavý	pravý	ľavý	kozu stráži prievozník
ľavý	ľavý	ľavý	pravý	*alternatívny koniec hry (tento stav zrejme nenastane)*
ľavý	ľavý	ľavý	ľavý	*[koniec hry]*

Vidíme, že kritické sú štyri situácie – keď na hociktorom z brehov vlk požiera kozu alebo koza kapustu. Potrebujeme ich vyjadriť programovo – podmienkou, resp. logickým výrazom. To sme schopní urobiť najmenej dvoma rôznymi spôsobmi – buď zostavíme výraz, ktorý všetky štyri situácie „vymenuje“, alebo zostavíme „vzorec“, resp. logický výraz, ktorý bude poskytovať výsledok vyhovujúci (pravdivý) práve pre štyri kľúčové situácie. Pri tejto konfigurácii by boli z hľadiska počítača oba spôsoby približne rovnako výpočtovo náročné, ale z hľadiska človeka by sme sa náročnejšie dopracúvali k požadovanému výrazu pri zvolení druhého spôsobu, preto zvolíme prvý spôsob.

V úvode metód `niektoNiekohoZožral` a `hraSkončila` v prvom rade získame logické hodnoty prítomnosti na niektorom z brehov pre vlka, kozu a kapustu (pre prievozníka ju poznáme od začiatku – je uchovaná v premennej `prievozníkJeNaPravomBrehu`). Naprogramovali sme len metódu `jeNaĽavomBrehu`, čo sme zdôvodnili výhodnosťou pri použití v metóde `hraSkončila`. Mohli by sme naprogramovať rovnakú metódu `jeNaPravomBrehu`, ale z pohľadu logických hodnôt je to zbytočné. Jej naprogramovanie a použitie by do programu neprinieslo žiadne nové informácie (na rozdiel od dvojice metód `nájdiNaPravomBrehu` a `nájdiNaĽavomBrehu`, z ktorých každá poskytuje jedinečný údaj). Takže iba pripomíname, že pre premenné určujúce pozíciu pasažierov platí opačná logická hodnota než pre prievozníka:

```
boolean vlkNaĽavom = jeNaĽavomBrehu("vlk");
boolean kozaNaĽavom = jeNaĽavomBrehu("koza");
boolean kapustaNaĽavom = jeNaĽavomBrehu("kapusta");
```

(Názvy premenných sme zjednodušili, aby sme si ušetrili písanie.)

Tieto tri premenné spolu s premennou `prievozníkJeNaPravomBrehu` využijeme na detekciu toho, či vlk zožral kozu:

```
if ((vlkNaĽavom && kozaNaĽavom && !kapustaNaĽavom && prievozníkJeNaPravomBrehu) ||
    (!vlkNaĽavom && !kozaNaĽavom && kapustaNaĽavom && !prievozníkJeNaPravomBrehu))
{
    // Áno, vlk zožral kozu…
}
```

a rovnako na detekciu toho, či koza zožrala kapustu:

```
if ((kozaNaĽavom && kapustaNaĽavom && !vlkNaĽavom && prievozníkJeNaPravomBrehu) ||
    (!kozaNaĽavom && !kapustaNaĽavom && vlkNaĽavom && !prievozníkJeNaPravomBrehu))
{
    // Áno, koza zožrala kapustu…
}
```

Opačná situácia – úspešné dokončenie hry – sa dá detegovať ľahko. Stačí overiť, či sú všetci traja pasažieri na ľavom brehu:

```
if (vlkNaĽavom && kozaNaĽavom && kapustaNaĽavom)
{
    // Hurá, dokončili sme hru!
}
```

Teraz tieto tri situácie vložíme do jednotlivých metód. Najskôr naprogramujeme metódu `niektoNiekohoZožral`, ktorá bude vracať hodnotu `true` v prípade, že nastal ten prípad, keď niekto niekoho zožral a `false` v opačnom prípade. Pri tejto metóde **mierne porušíme dobré programátorské návyky** a popri vrátení hodnoty `true` vypíšeme na výstup správu o tom, kto koho zožral. V skutočnosti by sa vypísanie takejto správy malo odohrať **mimo tejto metódy** (pretože v tomto prípade nejde o komplexnú metódu, akou bola metóda `prevez`; naopak metóda `niektoNiekohoZožral` je typická elementárna metóda a tá by nemala vykonávať odozvy priamo pre používateľa), mali by sme hľadať iný spôsob komunikácie tejto metódy so svojím okolím (napríklad formou celočíselnej hodnoty alebo vymenovacieho typu), ale pre urýchlenie a zjednodušenie túto zásadu porušíme. Metóda bude vyzerať takto:

```
private boolean niektoNiekohoZožral()
{
    boolean vlkNaĽavom = jeNaĽavomBrehu("vlk");
    boolean kozaNaĽavom = jeNaĽavomBrehu("koza");
    boolean kapustaNaĽavom = jeNaĽavomBrehu("kapusta");

    if ((vlkNaĽavom && kozaNaĽavom &&
        !kapustaNaĽavom && prievozníkJeNaPravomBrehu) ||
        (!vlkNaĽavom && !kozaNaĽavom &&
        kapustaNaĽavom && !prievozníkJeNaPravomBrehu))
    {
        System.out.println("Vlk zožral kozu!");
        return true;
    }

    if ((kozaNaĽavom && kapustaNaĽavom &&
        !vlkNaĽavom && prievozníkJeNaPravomBrehu) ||
        (!kozaNaĽavom && !kapustaNaĽavom &&
        vlkNaĽavom && !prievozníkJeNaPravomBrehu))
    {
        System.out.println("Koza zožrala kapustu!");
        return true;
    }

    return false;
}
```

Pri metóde `hraSkončila` neporušíme žiadnu zo zásad, naopak, ukážeme si jedno zjednodušenie. Predpokladajme, že hocikde v programe vznikne v závere booleovskej metódy táto situácia:

```
if (true == booleovskáPremenná)
{
    return true;
}

return false;
```

Tá úplne korešponduje s nasledujúcou situáciou:

```
if (booleovskáPremenná)
{
    return true;
}
else
{
    return false;
}
```

čiže (inak povedané) v prípade, že hodnota premennej typu boolean je true, vráti metóda true, inak vracia false (to znamená že jej návratová hodnota presne korešponduje s aktuálnou hodnotou premennej). Vtedy (ako možno tušíte) môžeme celú štruktúru if-else zjednodušiť na jediný príkaz:

```
return booleovskáPremenná;
```

Presne to použijeme v metóde hraSkončila s drobným rozdielom – v príkaze return sa nevyskytne rýdzo booleovská premenná, ale celý booleovský výraz (princíp je rovnaký – výsledok je vždy hodnota typu boolean):

```
private boolean hraSkončila()
{
    boolean vlkNaĽavom = jeNaĽavomBrehu("vlk");
    boolean kozaNaĽavom = jeNaĽavomBrehu("koza");
    boolean kapustaNaĽavom = jeNaĽavomBrehu("kapusta");

    return vlkNaĽavom && kozaNaĽavom && kapustaNaĽavom;
}
```

So zjednodušovaním by sme mohli ísť ešte ďalej, ale to nechám na pozornom čitateľovi… Použitie metód ponecháme na poslednú kapitolu tejto série…

Skompletizovaný príklad sa nachádza v prílohe.

Príloha 5 – tretia fáza textového projektu

```
public class VlkKozaKapusta extends Vstup
{
    private final String[] ĽavýBreh  = {"nič", "nič", "nič", "nič"};
    private final String[] pravýBreh = {"vlk", "koza", "kapusta", "nič"};
    private boolean prievozníkJeNaPravomBrehu = true;

    private VlkKozaKapusta()
    {
        String príkaz;

        vypíšStav();

        do {
            príkaz = načítajReťazec("Čo má prievozník previezť?");

            if (!príkaz.equalsIgnoreCase("koniec"))
            {
                prevez(príkaz);
                vypíšStav();
            }
```

```
        } while (!príkaz.equalsIgnoreCase("koniec"));
    }

    private void vypíšStav()
    {
        System.out.print("\nNa ľavom brehu je: ");

        if (!prievozníkJeNaPravomBrehu)
        {
            System.out.print(" [prievozník]");
        }

        for (int i = 0; i < ľavýBreh.length; ++i)
        {
            if (!ľavýBreh[i].equals("nič"))
            {
                System.out.print(" " + ľavýBreh[i]);
            }
        }

        System.out.print("\nNa pravom brehu je: ");

        if (prievozníkJeNaPravomBrehu)
        {
            System.out.print(" [prievozník]");
        }

        for (int i = 0; i < pravýBreh.length; ++i)
        {
            if (!pravýBreh[i].equals("nič"))
            {
                System.out.print(" " + pravýBreh[i]);
            }
        }

        System.out.println();
    }

    private int nájdiNaPravomBrehu(String čo)
    {
        for (int i = 0; i < pravýBreh.length; ++i)
        {
            if (čo.equalsIgnoreCase(pravýBreh[i])) return i;
        }

        return -1;
    }

    private int nájdiNaĽavomBrehu(String čo)
    {
        for (int i = 0; i < ľavýBreh.length; ++i)
        {
            if (čo.equalsIgnoreCase(ľavýBreh[i])) return i;
        }

        return -1;
    }
```

```
    private void prevez(String čo)
    {
        if (prievozníkJeNaPravomBrehu)
        {
            int i = nájdiNaPravomBrehu(čo);
            int j = nájdiNaĽavomBrehu("nič");

            if (i == -1)
            {
                System.out.println("Na ľavom brehu nie je " + čo + ".");
            }
            else if (j == -1)
            {
                System.out.println("Na pravom brehu nie je voľné miesto.");
            }
            else
            {
                ľavýBreh[j] = pravýBreh[i];
                pravýBreh[i] = "nič";
                prievozníkJeNaPravomBrehu = !prievozníkJeNaPravomBrehu;
            }
        }
        else
        {
            int i = nájdiNaĽavomBrehu(čo);
            int j = nájdiNaPravomBrehu("nič");

            if (i == -1)
            {
                System.out.println("Na pravom brehu nie je " + čo + ".");
            }
            else if (j == -1)
            {
                System.out.println("Na ľavom brehu nie je voľné miesto.");
            }
            else
            {
                pravýBreh[j] = ľavýBreh[i];
                ľavýBreh[i] = "nič";
                prievozníkJeNaPravomBrehu = !prievozníkJeNaPravomBrehu;
            }
        }
    }

    private boolean jeNaĽavomBrehu(String čo)
    {
        for (int i = 0; i < ľavýBreh.length; ++i)
        {
            if (čo.equalsIgnoreCase(ľavýBreh[i])) return true;
        }

        return false;
    }

    private boolean niektoNiekohoZožral()
    {
```

```
        boolean vlkNaĽavom = jeNaĽavomBrehu("vlk");
        boolean kozaNaĽavom = jeNaĽavomBrehu("koza");
        boolean kapustaNaĽavom = jeNaĽavomBrehu("kapusta");

        if ((vlkNaĽavom && kozaNaĽavom &&
            !kapustaNaĽavom && prievozníkJeNaPravomBrehu) ||
            (!vlkNaĽavom && !kozaNaĽavom &&
            kapustaNaĽavom && !prievozníkJeNaPravomBrehu))
        {
            System.out.println("Vlk zožral kozu!");
            return true;
        }

        if ((kozaNaĽavom && kapustaNaĽavom &&
            !vlkNaĽavom && prievozníkJeNaPravomBrehu) ||
            (!kozaNaĽavom && !kapustaNaĽavom &&
            vlkNaĽavom && !prievozníkJeNaPravomBrehu))
        {
            System.out.println("Koza zožrala kapustu!");
            return true;
        }

        return false;
    }

    private boolean hraSkončila()
    {
        boolean vlkNaĽavom = jeNaĽavomBrehu("vlk");
        boolean kozaNaĽavom = jeNaĽavomBrehu("koza");
        boolean kapustaNaĽavom = jeNaĽavomBrehu("kapusta");

        return vlkNaĽavom && kozaNaĽavom && kapustaNaĽavom;
    }

    public static void main(String[] args)
    {
        new VlkKozaKapusta();
    }
}
```

Textový projekt 4

V predchádzajúcej kapitole sme tvorbu projektu zanechali vo fáze, v ktorej už nie je ďaleko do úplného úspešného dokončenia. V podstate nám chýba iba použitie naposledy naprogramovaných metód v praxi (`niektoNiekohoZožral` a `hraSkončila`) a vykonanie niekoľkých „kozmetických“ úprav. Jednou, z pohľadu algoritmizácie a princípov programovania, „kozmetickou“ úpravou je *vypísanie pravidiel pre hráča*. To, čo je z pohľadu funkčnosti softvéru menej podstatnou záležitosťou, je zároveň **veľmi podstatnou** záležitosťou pre hráča (alias používateľa). Pamätajte, že **softvér, ktorý nemá poriadne dopracovanú komunikáciu s používateľom, sa stáva nepoužiteľný**, nech funguje akokoľvek spoľahlivo.

Metóda, ktorá sa bude starať o vypísanie pravidiel, bude v tomto projekte veľmi jednoduchá. Momentálne pracujeme v textovom režime a vypísanie pravidiel bude pozostávať zo série príkazov `System.out.println`. Metódu nazvime `vypíšPravidlá`. Použijeme ju na vhodnom mieste v konštruktore.

Pravidlá by sa mali vypísať ako prvé – budú hrať úlohu úvodu do hry. Ak nazrieme do konštruktora, ponúkne sa nám umiestnenie volania tejto metódy pred metódu `vypíšStav`:

```
private VlkKozaKapusta()
{
    String príkaz;

    // SEM!        <----
    vypíšStav();

    do {
    // a tak ďalej…
```

Po čase by sa nám ukázalo, že vždy po volaní metódy `vypíšPravidlá`, je zároveň vhodné vypísať pre hráča aktuálny stav, preto môžeme dve činnosti spojiť do jednej. Pozor, stále si však potrebujeme zachovať možnosť vypísania stavu nezávisle od pravidiel. Takže urobíme to, že na konci metódy `vypíšPravidlá` zavoláme metódu `vypíšStav`. To nám dá možnosť volania metódy `vypíšStav` samostatne a zároveň nás odľahčí od povinnosti jej volania vždy tesne po metóde `vypíšPravidlá` (pretože sa fakticky zavolajú obe „naraz“, resp. jedna zavolá druhú). Nová metóda bude vyzerať takto (uvádzame mierne odľahčenú verziu):

```
private void vypíšPravidlá()
{
    System.out.println();
    System.out.println("Prievozník vlastní plť, na ktorej prevezie okrem ");
    System.out.println("seba maximálne jedného pasažiera. Jedného dňa sa ");
    System.out.println("ocitol v situácii, keď mal bezpečne previezť na ");
    // Ďalší text príbehu…

    System.out.println();
    System.out.println("Ak chcete, aby prievozník na druhú stranu nič " +
        "nepreviezol, zadajte „nič“.");
    System.out.println("Opätovný výpis týchto pravidiel získate príkazom " +
        "„pravidlá“.");
    System.out.println("Na predčasné ukončenie hry zadajte „koniec“.");
    System.out.println("Veľa šťastia!");
    vypíšStav();
}
```

Výpis príbehu sme skrátili (kompletnú verziu metódy nájdete v skompletizovanom kóde v prílohe) a ako môžete vidieť, do výpisu sme zahrnuli aj informácie o tom, ako nechať prievozníka prejsť na druhú stranu bez nákladu, ako hru predčasne ukončiť a (čo budeme riešiť o chvíľu) ako opätovne počas hry vypísať tieto pravidlá (keby to hráč potreboval).

Teraz môžeme metódu `vypíšPravidlá` použiť. Prvé použitie bude na začiatku hry a druhé v jej priebehu, čiže vo vnútri cyklu `do-while`. Druhé volanie musíme podmieniť želaním hráča, ktorý si môže vyžiadať výpis pravidiel zadaním príkazu „pravidlá“. Chceme, aby sa pri vypísaní pravidiel nedialo nič iné, preto zaradíme funkčný celok do zloženej štruktúry `if-else-if` (ide v podstate o riadiacu štruktúru `if-else`, ktorá má v alternatívnej vetve `else` umiestnené podmienené spracovanie `if`), čiže štruktúra by v rozvinutom tvare vyzerala takto:

```
if (prváPodmienka)
{
    // príkazy…
}
```

```
else
{
    if (druháPodmienka)
    {
        // príkazy…
    }
}
```

V praxi sa tento zápis často zjednodušuje týmto spôsobom:

```
if (prváPodmienka)
{
    // príkazy…
}
else if (druháPodmienka)
{
    // príkazy…
}
```

Tento zápis sme už v rámci tohto projektu použili, vtedy sme však na túto skutočnosť neupozorňovali. Celý konštruktor bude potom vyzerať takto:

```
private VlkKozaKapusta()
{
    String príkaz;
    vypíšPravidlá();

    do {
        príkaz = načítajReťazec("Čo má prievozník previezť?");

        if (príkaz.equalsIgnoreCase("pravidlá"))
        {
            vypíšPravidlá();
        }
        else if (!príkaz.equalsIgnoreCase("koniec"))
        {
            prevez(príkaz);
            vypíšStav();
        }

    } while (!príkaz.equalsIgnoreCase("koniec"));
}
```

Úplne poslednou časťou, ktorú potrebujeme doprogramovať, je využitie metód `niektoNiekohoZožral` a `hraSkončila` na detekciu prípadov neúspešného a úspešného dokončenia hry. Táto detekcia sa musí udiať vždy po prevezení pasažiera (prípadne prechode prievozníka na druhý breh naprázdno), preto obidve metódy využijeme v zloženej štruktúre `if-else-if`, ktorú umiestnime za príkazy zodpovedné za prevezenie pasažiera a výpis aktuálneho stavu. Z algoritmického hľadiska nie je medzi týmito dvoma ukončeniami rozdiel. Pre hráča však (opäť) hrá významnú rolu výpis, ktorý mu oznámi úspech alebo neúspech. A to je pri terajšom (textovom) prístupe jediný rozdiel. Pri metóde `niektoNiekohoZožral` (ak si spomínate) sa nemusíme zaoberať problémom výpisu, pretože sme ho (porušiac pravidlá „čistého programovania“) umiestnili priamo do metódy. Zostáva teda vyriešiť výpis oznamujúci úspech. Uvádzame len tú časť konštruktora, ktorej sa to dotýka:

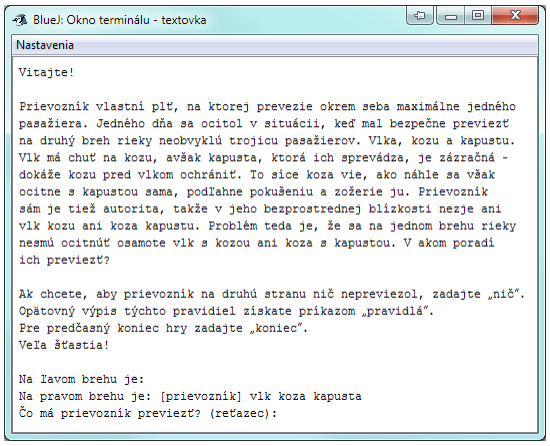
```
// …pokračovanie
else if (!príkaz.equalsIgnoreCase("koniec"))
```

```
{
    prevez(príkaz);
    vypíšStav();

    if (niektoNiekohoZožral())
    {
        break;
    }
    else if (hraSkončila())
    {
        System.out.println("Hotovo! Blahoželám!");
        break;
    }
}
// pokračovanie…
```

Príkaz `break` ukončí cyklus, ktorý je k nemu najbližšie, momentálne je to cyklus `do-while` (iný cyklus v našom konštruktore ani nemáme). Ak by sme mali v programe viacero vnorených cyklov, museli by sme hľadať iný spôsob ukončenia. Ponúka sa možnosť zariadiť to tak, aby prestala platiť podmienka na úrovni najvyššieho cyklu (výraz: `!príkaz.equalsIgnoreCase("koniec")`) a to sa dá v tomto prípade zariadiť nasledujúcim príkazom, ktorým by sme nahradili každý `break`:

```
príkaz = "koniec";
```

Môžete to skúsiť. Efekt bude rovnaký.



Príloha 6 – finálna fáza textového projektu

```
public class VlkKozaKapusta extends Vstup
{
    private final String[] ľavýBreh  = {"nič", "nič", "nič", "nič"};
    private final String[] pravýBreh = {"vlk", "koza", "kapusta", "nič"};
    private boolean prievozníkJeNaPravomBrehu = true;

    private VlkKozaKapusta()
    {
        String príkaz;

        System.out.println("Vitajte!");
        vypíšPravidlá();

        do {
            príkaz = načítajReťazec("Čo má prievozník previezť?");
```

```
            if (príkaz.equalsIgnoreCase("pravidlá"))
            {
                vypíšPravidlá();
            }
            else if (!príkaz.equalsIgnoreCase("koniec"))
            {
                prevez(príkaz);
                vypíšStav();

                if (niektoNiekohoZožral())
                {
                    break;
                }
                else if (hraSkončila())
                {
                    System.out.println("Hotovo! Blahoželám!");
                    break;
                }
            }

        } while (!príkaz.equalsIgnoreCase("koniec"));

        System.out.println("Dovidenia!");
    }

    private void vypíšPravidlá()
    {
        System.out.println();
        System.out.println("Prievozník vlastní plť, na ktorej prevezie okrem " +
            "seba maximálne jedného ");
        System.out.println("pasažiera. Jedného dňa sa ocitol v situácii, keď " +
            "mal bezpečne previezť ");
        System.out.println("na druhý breh rieky neobvyklú trojicu " +
            "pasažierov. Vlka, kozu a kapustu. ");
        System.out.println("Vlk má chuť na kozu, avšak kapusta, ktorá ich " +
            "sprevádza, je zázračná – ");
        System.out.println("dokáže kozu pred vlkom ochrániť. To síce koza " +
            "vie, len čo sa však ");
        System.out.println("ocitne s kapustou sama, podľahne pokušeniu " +
            "a zožerie ju. Prievozník ");
        System.out.println("sám je tiež autorita, takže v jeho " +
            "bezprostrednej blízkosti nezje ani ");
        System.out.println("vlk kozu, ani koza kapustu. Problém teda je, " +
            "že sa na jednom brehu rieky ");
        System.out.println("nesmú ocitnúť osamote vlk s kozou, ani koza " +
            "s kapustou. V akom poradí ");
        System.out.println("ich previezť?");
        System.out.println();
        System.out.println("Ak chcete, aby prievozník na druhú stranu nič " +
            "nepreviezol, zadajte „nič“.");
        System.out.println("Opätovný výpis týchto pravidiel získate príkazom " +
            "„pravidlá“.");
        System.out.println("Na predčasné ukončenie hry zadajte „koniec“.");
        System.out.println("Veľa šťastia!");
        vypíšStav();
    }
```

```
    private void vypíšStav()
    {
        System.out.print("\nNa ľavom brehu je: ");

        if (!prievozníkJeNaPravomBrehu)
        {
            System.out.print(" [prievozník]");
        }

        for (int i = 0; i < ľavýBreh.length; ++i)
        {
            if (!ľavýBreh[i].equals("nič"))
            {
                System.out.print(" " + ľavýBreh[i]);
            }
        }

        System.out.print("\nNa pravom brehu je: ");

        if (prievozníkJeNaPravomBrehu)
        {
            System.out.print(" [prievozník]");
        }

        for (int i = 0; i < pravýBreh.length; ++i)
        {
            if (!pravýBreh[i].equals("nič"))
            {
                System.out.print(" " + pravýBreh[i]);
            }
        }

        System.out.println();
    }

    private int nájdiNaPravomBrehu(String čo)
    {
        for (int i = 0; i < pravýBreh.length; ++i)
        {
            if (čo.equalsIgnoreCase(pravýBreh[i])) return i;
        }

        return -1;
    }

    private int nájdiNaĽavomBrehu(String čo)
    {
        for (int i = 0; i < ľavýBreh.length; ++i)
        {
            if (čo.equalsIgnoreCase(ľavýBreh[i])) return i;
        }

        return -1;
    }

    private void prevez(String čo)
    {
```

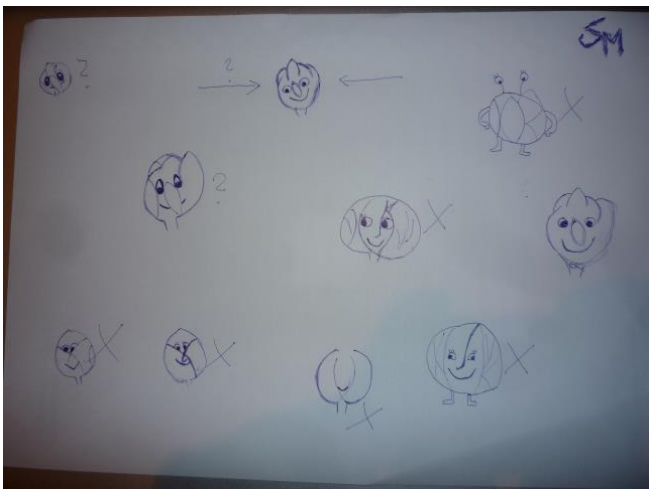
```
        if (prievozníkJeNaPravomBrehu)
        {
            int i = nájdiNaPravomBrehu(čo);
            int j = nájdiNaĽavomBrehu("nič");

            if (i == -1)
            {
                System.out.println("Na ľavom brehu nie je " + čo + ".");
            }
            else if (j == -1)
            {
                System.out.println("Na pravom brehu nie je voľné miesto.");
            }
            else
            {
                ľavýBreh[j] = pravýBreh[i];
                pravýBreh[i] = "nič";
                prievozníkJeNaPravomBrehu = !prievozníkJeNaPravomBrehu;
            }
        }
        else
        {
            int i = nájdiNaĽavomBrehu(čo);
            int j = nájdiNaPravomBrehu("nič");

            if (i == -1)
            {
                System.out.println("Na pravom brehu nie je " + čo + ".");
            }
            else if (j == -1)
            {
                System.out.println("Na ľavom brehu nie je voľné miesto.");
            }
            else
            {
                pravýBreh[j] = ľavýBreh[i];
                ľavýBreh[i] = "nič";
                prievozníkJeNaPravomBrehu = !prievozníkJeNaPravomBrehu;
            }
        }
    }

    private boolean jeNaĽavomBrehu(String čo)
    {
        for (int i = 0; i < ľavýBreh.length; ++i)
        {
            if (čo.equalsIgnoreCase(ľavýBreh[i])) return true;
        }

        return false;
    }

    private boolean niektoNiekohoZožral()
    {
        boolean vlkNaĽavom = jeNaĽavomBrehu("vlk");
        boolean kozaNaĽavom = jeNaĽavomBrehu("koza");
        boolean kapustaNaĽavom = jeNaĽavomBrehu("kapusta");
```

```
        if ((vlkNaĽavom && kozaNaĽavom &&
            !kapustaNaĽavom && prievozníkJeNaPravomBrehu) ||
            (!vlkNaĽavom && !kozaNaĽavom &&
            kapustaNaĽavom && !prievozníkJeNaPravomBrehu))
        {
            System.out.println("Vlk zožral kozu!");
            return true;
        }

        if ((kozaNaĽavom && kapustaNaĽavom &&
            !vlkNaĽavom && prievozníkJeNaPravomBrehu) ||
            (!kozaNaĽavom && !kapustaNaĽavom &&
            vlkNaĽavom && !prievozníkJeNaPravomBrehu))
        {
            System.out.println("Koza zožrala kapustu!");
            return true;
        }

        return false;
    }

    private boolean hraSkončila()
    {
        boolean vlkNaĽavom = jeNaĽavomBrehu("vlk");
        boolean kozaNaĽavom = jeNaĽavomBrehu("koza");
        boolean kapustaNaĽavom = jeNaĽavomBrehu("kapusta");

        return vlkNaĽavom && kozaNaĽavom && kapustaNaĽavom;
    }

    public static void main(String[] args)
    {
        new VlkKozaKapusta();
    }
}
```

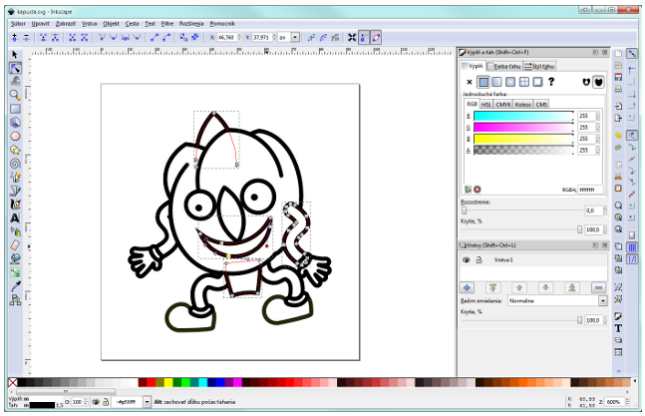
Príprava grafického projektu 1

Pri riešení grafického projektu budeme predpokladať, že **poznáme všetky princípy spomínané počas tvorby textovej verzie** a pridáme viacero ďalších. Pred začatím samotnej fázy programovania, vykonáme niekoľko prípravných fáz. V prvej fáze zostavíme knižnicu grafických a zvukových súborov, ktoré použijeme v rámci projektu. V tejto kapitole stručne opíšeme jeden z možných postupov prípravy, pričom pripravená knižnica je k dispozícii na prevzatie na jej konci.

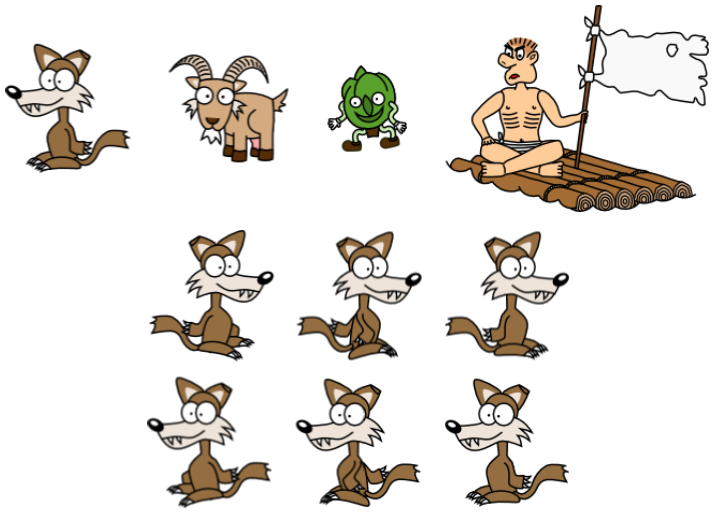
Netreba pripomínať, že tvorba grafiky je tvorivý proces. Koniec koncov, aj programovanie je tvorivá činnosť, avšak vyžaduje iný druh tvorivosti užšie spojenej s procesom myslenia než predstavivosti… Pri tvorbe grafiky môžeme využiť rozmanité prístupy. Záleží od našich možností a schopností. Vždy je dobré urobiť si najskôr skice pripravovaného materiálu, a potom zvoliť vhodný spôsob jeho prevedenia do grafickej podoby.



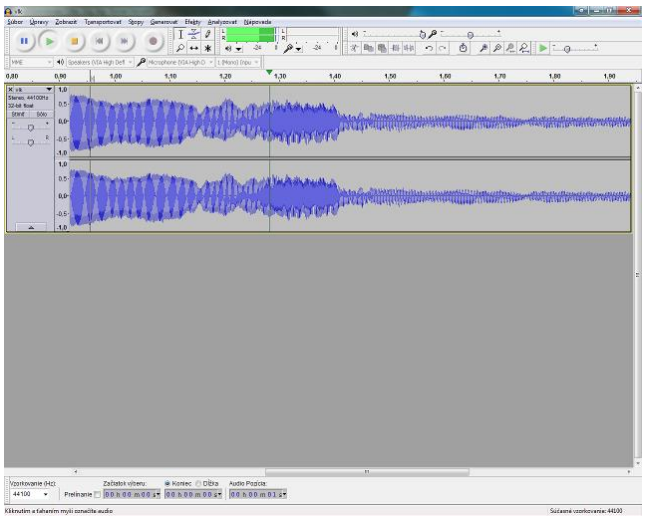
My sme zvolili vektorový spôsob kreslenia grafiky. Na vytvorenie všetkých postavičiek sme použili voľne dostupný vektorový grafický editor Inkscape (http://inkscape.org/). Skice sme naskenovali, vložili na podklad kresby a postupne prekreslili do vektorovej grafickej podoby, ktorú sme neskôr rozanimovali.



Postupne, postavičku po postavičke, fázu po fáze sme vytvorili 20 samostatných grafických súborov vo formáte PNG, na ktorých sú zachytené fázy pohybu pasažierov pohybujúcich sa doľava a doprava a prievozník obrátený hlavou doprava a doľava.



Na úpravu zvukových súborov sme použili voľne dostupný zvukový editor Audacity (http://audacity.sourceforge.net/). Spracovali sme 6 samostatných zvukových efektov vo formáte WAV určených pre ozvučenie pasažierov, prievozníka, požierania a víťazného záveru hry.



Všetky súbory sme pomenovali takým spôsobom, aby sme ich mohli v projekte čo najjednoduchšie načítať. Dotýka sa to najmä názvov grafických súborov, pre ktoré sme si zvolili jednotný predpis – meno súboru sa začína názvom grafického objektu (napríklad vlk), potom nasleduje označenie smeru, akým je objekt obrátený (ľ – doľava; p – doprava) a nakoniec číslo fázy pohybu (napr.: vlk-ľ-1.png, koza-p-2.png…). Ak je číslo fázy pohybu zamlčané (napr.: prievozník-ľ.png, kapusta-p.png…), znamená to, že tento grafický súbor buď zobrazuje objekt v statickej polohe, alebo objekt vôbec nie je animovaný. Štyri zvukové súbory určené na ozvučenie objektov pri pohybe pozostávajú iba z názvu objektu (napr.: prievozník.wav, vlk.wav…). Dva zvukové súbory sprevádzajúce úspešný a neúspešný záver hry sme nazvali: potlesk.wav a jedenie.wav. Spôsob pomenovania súborov budeme považovať internú dohodu platnú v rámci projektu. Ešte sa k nej v niektorej z nasledujúcich kapitol vrátime.

Príloha 7 – knižnica súborov grafického projektu

Knižnica grafických a zvukových súborov.

Príprava grafického projektu 2

V ďalšej prípravnej fáze urobíme predbežný návrh hierarchie tried, ktoré budeme v rámci projektu definovať, určíme si východiská a premyslíme postup.

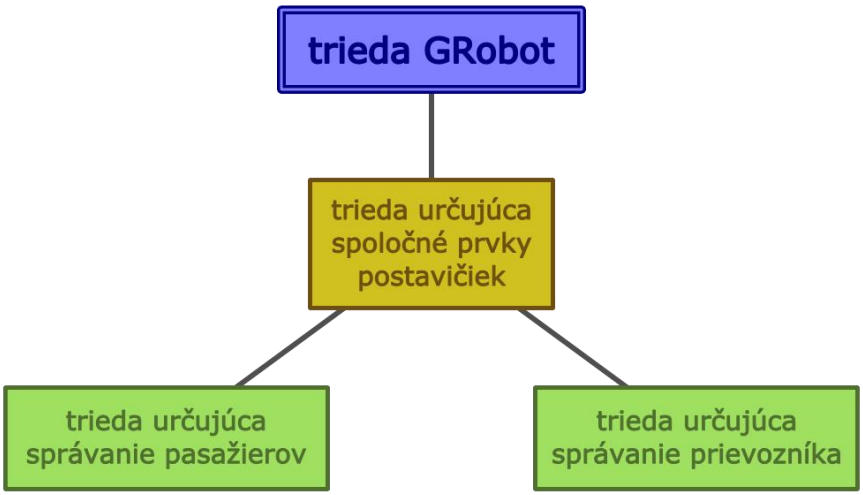
Stabilným bodom bude pre nás trieda `GRobot`. Od nej budeme odvodzovať všetky triedy, od ktorých očakávame prácu s grafikou alebo inými prvkami sveta grafického robota. Počas tvorby textovej verzie hry sme získali predstavu prinajmenšom o tom, aké figúrky by mali hru tvoriť a ako by sa mali správať. V grafickej verzii budeme očakávať, aby sa figúrky pohybovali a vydávali zvuky. (To ich predurčuje byť potomkami triedy `GRobot`.)

Povedzme, že z hľadiska funkčnosti a princípov, ktoré bude potrebné pre jednotlivé figúrky naprogramovať, budeme odlišovať pasažierov od prievozníka. Pre prievozníka bude stačiť, ak sa bude po obrazovke kĺzať. Animácia kráčania pasažierov bude postačovať aj veľmi jednoduchá (tú sme v podstate určili fázami pohybu na obrázkoch vytvorených v predchádzajúcej prípravnej kapitole). Z pohľadu pohybu pasažierov bude potrebné zabezpečiť, aby sa figúrky boli schopné (za súčasnej animácie kráčania) dostať z brehu na plť

a späť (platí pre oba brehy), a zároveň, aby sa počas prítomnosti na plti pohybovali súčasne s ňou. Prostredie hry (brehy a rieka) môže byť nakreslené ako statické pozadie. Pokiaľ ide o ostatné princípy, budeme sa inšpirovať i textovou verziou hry.

Dá sa očakávať, že všetky figúrky budú mať určitú množinu vlastností spoločnú a že budú jestvovať podmnožiny rozdielov pre pasažierov a prievozníka. Pri textovej verzii nám stačilo pamätať si jediný údaj vyjadrujúci na ktorom brehu sa figúrka nachádza. V grafickom režime musíme počítať s omnoho väčším objemom údajov pre každú postavičku. Aktuálna poloha objektu na obrazovke, rýchlosť pohybu, cieľová poloha, fáza animácie kráčania… to všetko sú informácie, ktoré musia byť niekde uchované. Mnohé z nich dokážeme pokryť funkčnosťou triedy grafického robota, určite však budeme musieť vytvárať niektoré vlastné.

Objektovo orientovaný prístup v programovaní nám dovoľuje oddeliť spoločné vlastnosti figúrok do samostatnej nadtriedy a rozdiely upraviť v odvodených triedach. Vytvoríme preto hierarchiu tried, kde nadtrieda bude priamym potomkom triedy `GRobot` a z nej budú odvodené triedy určujúce samostatné správanie pre prievozníka a pasažierov. Hierarchiu ilustruje nasledujúci obrázok:



Zvoľme si pre nich názvy. Triedu určujúcu spoločné prvky postavičiek nazvime `AnimovanýObjekt`. Bude uchovávať informácie o „kotviacich“ bodoch postavičky na jednotlivých brehoch, o súboroch (grafických a zvukových), ktoré postavička používa, rôzne pomocné atribúty (napríklad pre animáciu, zobrazenie popisu nad postavičkou) a iné. Triedu prievozníka nazvime `Prievozník` a triedu pre pasažierov nazvime `Pasažier`.

`AnimovanýObjekt` bude zhŕňať všetko, čo bude spoločné pre všetky animované objekty v hre (vlk, koza, kapusta a prievozník). Rozdiely v správaní zapracujeme do odvodených tried. Pri kategorizácii všetkých objektov hry, by sme zistili, že môžeme vytvoriť dve kategórie animovaných objektov (pričom jedna z nich by obsahovala jeden jediný prvok). Takže postačí naprogramovať dve triedy odzrkadľujúce odlišnosti v správaní – `Prievozník` a `Pasažier`. Toto je princíp objektovej dedičnosti (**inheritance**). Z týchto tried následne vytvoríme konkrétne inštancie (objekty). Z triedy `Prievozník` vytvoríme jedinú inštanciu a z triedy `Pasažier` tri.

Okrem týchto tried bude v projekte určite figurovať hlavná trieda (označená identifikátorom `HlavnáTrieda`), v ktorej vytvoríme a prepojíme inštancie postavičiek a vymenovacia trieda (enumerácia – trieda typu `enum`), pomocou ktorej budeme vyjadrovať rôzne stavy animovaných objektov.

Vymenovacie triedy v Jave slúžia, okrem iného, na definovanie rôznych stavov objektov. Majú širokú funkcionalitu, ale nám bude stačiť použitie ich základného (najjednoduchšieho) tvaru. Definujeme vymenovaciu triedu pomocou ktorej budeme vyjadrovať logickú pozíciu postavičky. Nazveme ju `Pozícia`.

Definícia triedy je jednoduchá – niekoľkoriadková. Pri vytváraní novej triedy zvolíme „Vymenovací typ“ a hodnoty vytvorené šablónou (dni v týždni) nahradíme vlastnými hodnotami. Zdrojový kód vymenovacej triedy `Pozícia` bude vyzerať takto:

```
public enum Pozícia
{
    pravýBreh, ľavýBreh, prievozník
}
```

S jej pomocou budeme schopní vyjadriť tri hraničné situácie, v ktorých sa môže postavička nachádzať: prítomnosť na niektorom z brehov alebo prítomnosť „na“ prievozníkovi (na jeho plti, pričom tento stav nemá zmysel pre samotného prievozníka). Takže pri grafickom projekte nestačí vyjadriť pozíciu premennou typu `boolean` (pravý breh/ľavý breh). Potrebujeme ešte jeden „medzistav“ na vyjadrenie situácie, keď sa postavička nachádza „na“ prievozníkovi.

Na záver si vysvetlime programátorskú techniku, ktorú budeme často používať v nasledujúcich kapitolách. K jej vysvetleniu sa nebudeme vracať, preto si ju dobre zapamätajte! Ide o takzvané prekrývanie metód (niekedy nazývané „prepisovanie“ metód). Dotýka sa to objektového polymorfizmu, čiže „viactvarosti“ objektov (**polymorphism**). Prekrývanie metód je technika, pri ktorej v odvodenej triede napíšeme nové telo metódy definovanej predtým v rodičovskej triede – prekryjeme ju. To má za následok to, že keď metódu zavoláme pre inštanciu rodičovskej triedy, zavoláme originálnu metódu, no keď ju zavoláme pre inštanciu odvodenej triedy, bude volaná nová verzia metódy, ktorou sme prekryli jej rovnomennú predchodkyňu.

Vysvetlíme si to názorne na situácii úzko súvisiacej s naším projektom. Skupina tried `GRobot` má preddefinovaných množstvo stavov a (niekedy prázdnych) metód, ktoré môžu odvodené triedy využívať podľa uváženia. Okrem bežných stavov a vlastností typu poloha, orientácia (smer), rýchlosť, farba, hrúbka čiary a podobne, sú tu aj niektoré metódy čakajúce na prekrytie. Predvolene sú prázdne – nerobia nič. Ak ich odvodená trieda prekryje, zmení správanie robota, čím získa kontrolu nad určitými záležitosťami súvisiacimi so správaním a fungovaním robota. Typickým príkladom je prekrytie metódy `aktivita`:

```
@Override public void aktivita()
{
}
```

Získame tým kontrolu nad správaním aktívneho robota. Čokoľvek napíšeme do prekrytej (symbolizuje to klauzula `@Override`) metódy, bude vykonané pre aktívne inštancie odvodenej triedy. Klauzula `@Override` síce nie je (z pohľadu syntaxe Javy) povinná, ale ak ju neuvedieme a urobíme nechcený preklep v hlavičke metódy, program nebude správne fungovať napriek tomu, že bol preložený bez chýb. Dôvodom je, že nedôjde ku korektnému prekrytiu želanej metódy. Také chyby sa ťažko hľadajú, preto budeme z pohľadu programátorskej korektnosti považovať túto klauzulu za povinnú!

Dôležité upozornenie! Na to, aby sme v novej vytváranej triede mohli používať vlastnosti robota a prekrývať jeho metódy, musí byť trieda odvodená od triedy `GRobot`! Takže ak napríklad definujeme novú triedu `AnimovanýObjekt`, v ktorej prekrývame metódu aktivita, musíme vždy skontrolovať, či je trieda skutočne odvodená od triedy `GRobot` (buď vizuálne – prítomnosť šípky v diagrame hlavného okna BlueJ, alebo kontrolou zápisu v zdrojovom kóde). V programe (zdrojovom kóde triedy) sa táto skutočnosť prejaví zápisom `extends GRobot` pri hlavičke triedy:

```
public class AnimovanýObjekt extends GRobot
```

Vymenovacie typy vytvorené v rámci tejto kapitoly sú príliš jednoduché na to, aby bolo nutné ich pripájať v samostatných prílohách tejto kapitoly.

Grafický projekt 1

V rámci tejto kapitoly si vysvetlíme postup začatia tvorby triedy `AnimovanýObjekt`, ktorá bude základom pre ďalšie grafické objekty hry. Pripomíname, že tak ako všetko v rámci tohto materiálu, aj toto je len jedno z možných riešení. Programovanie je v širokej miere tvorivá činnosť a pri tomto type úloh je málokedy možné nájsť jednoznačné riešenie úlohy. My ukážeme náš spôsob riešenia a poskytneme vysvetlenie, čo nás k tomu spôsobu riešenia viedlo, kam tým smerujeme a kde jednotlivé časti triedy využijeme v rámci celého projektu.

V tomto materiáli sú niektoré procesy tvorby urýchlené. V praxi je často tvorba podobného projektu zdĺhavejšia. Funkcionalita tried je rozširovaná postupne, podľa potrieb projektu. Často sa postupuje tak, že sa priebežne spisujú požiadavky na rozšírenie funkcionality tried – spracúva sa zoznam požiadaviek (**wish-list**). Na základe neho sú triedy postupne rozširované, až kým nie je projekt dokončený. Zoznam požiadaviek je dôležitým komunikačným prostriedkom pri projektoch, na ktorých spolupracuje tím viacerých programátorov. Každá požiadavka musí byť sformulovaná veľmi presne!

Jestvujú určité všeobecné pravidlá spolupráce. V rámci nich je dôležité naučiť sa dobre posúdiť, čo má programátor programovať sám, a naopak, aké požiadavky posielať kolegom programujúcim iné časti softvéru (iné triedy). Zásada je, aby to, čo sa dotýka vnútra (správania a stavov) každej triedy, bolo implementované priamo v nej, nie nejakou „obchádzkou“ zvonka. Ak potrebujeme zistiť stav, ovplyvniť správanie objektu určenej triedy, riešime to naprogramovaním metódy alebo skupiny metód vo vnútri konkrétnej triedy, nikdy nie zvonka nej (v inej triede). Porušili by sme princíp objektovej uzavretosti (**encapsulation**). Ak nie sme autormi triedy, spíšeme požiadavku adresovanú jej autorovi.

Náš projekt je síce produktom jedného autora, spisovanie požiadaviek však vždy dobre poslúži na zlepšenie organizácie práce. Takže zoznam požiadaviek dokáže využiť každý (i samostatný) programátor. Odporúčame založiť si zoznam požiadaviek pri každom projekte čo najskôr. V tomto vzdelávacom materiáli budeme vo väčšine prípadov predpokladať, že zoznam požiadaviek jestvuje a je automaticky priebežne aktualizovaný.

Tvorba každej novej triedy sa začína naprogramovaním konštruktora inicializujúceho objekt. Základný konštruktor (**default constructor**) neprijíma žiadne argumenty, úplný konštruktor (**full constructor**) prijíma zoznam všetkých súkromných inštančných premenných, ktoré na základe toho inicializuje. Jednoduchá verzia triedy `AnimovanýObjekt` s úplným konštruktorom by mohla vyzerať takto:

```
public class AnimovanýObjekt extends GRobot
{
    private double posunX, posunY;

    public AnimovanýObjekt(double posunX, double posunY)
    {
        this.posunX = posunX;
        this.posunY = posunY;
    }
}
```

Tento jednoduchý konštruktor iba kopíruje hodnoty prijímaných argumentov do súkromných inštančných premenných majúcich (spravidla) rovnaký názov. Na odlíšenie medzi argumentmi a súkromnými premennými slúži ukazovateľ `this`.

Úplný konštruktor o chvíľu stratí prívlastok „úplný“, pretože budeme potrebovať triedu `AnimovanýObjekt` rozšíriť. Viacero inštančných premenných budeme inicializovať automaticky. Stanovíme si na to vlastný predpis. Využijeme hodnoty ostatných inštančných premenných. V nasledujúcej tabuľke sú vymenované

všetky vnútorné inštančné premenné triedy `AnimovanýObjekt`. Tabuľka zároveň obsahuje stručné vysvetlenie účelu každej premennej.

Údajový typ	Názov premennej	Účel
`Pozícia`	`pozícia`	aktuálne logické umiestnenie tohto objektu (brehy/plť); vymenovací typ `Pozícia` sme definovali v predchádzajúcej kapitole
`String`	`meno`	pomenovanie objektu (nájde využitie na viacerých miestach v rámci tejto triedy)
`String`	`zvuk`	názov súboru so zvukom vydaného objektom pri pohybe
`String[]`	`obrázokDoľava`, `obrázokDoprava`	polia uchovávajúce názvy súborov s obrázkami využívanými na animáciu chôdze pri pohybe doprava a doľava, pričom prvý prvok (s indexom 0) rezervujeme pre statický obrázok objektu
`AnimovanýObjekt`	`cieľ`	ukazovateľ na cieľový objekt, ku ktorému sa má tento objekt pohybovať (bude využité pri presune postáv na plť)
`double`	`posunX`, `posunY`	relatívna poloha grafiky (obrázkov vlastného tvaru objektu) voči skutočným súradniciam (stredu) objektu
`double`	`ľavýX`, `ľavýY`	cieľová poloha objektu na ľavom brehu rieky
`double`	`pravýX`, `pravýY`	cieľová poloha objektu na pravom brehu rieky
`double`	`posunPopisuX`, `posunPopisuY`	relatívna poloha popisu (mena objektu) zobrazovaného pri presunutí ukazovateľa myši ponad neho
`int`	`fázaChôdze`	číslo určujúce aktuálnu fázu animácie chôdze

Na dodržanie princípu objektovej uzavretosti, budú všetky premenné súkromné. Tie inštančné premenné, ktoré nebudú inicializované v konštruktore, inicializujeme priamo pri ich definícii:

```
private Pozícia pozícia = Pozícia.pravýBreh;
private String meno, zvuk;
// …
private double posunPopisuY;
private int fázaChôdze = 0;
```

Skúste doplniť deklarácie a definície ostatných premenných na miesto označené komentárom. Správe riešenie nájdete v prílohách kapitoly…

Pre triedu definujeme jediný konštruktor prijímajúci najdôležitejšie údaje, podľa ktorých dokážeme inicializovať viaceré ďalšie vnútorné premenné triedy. Konštruktor bude prijímať meno objektu, posun grafiky a posun popisu. Spomínanú inicializáciu viacerých vnútorných premenných môžeme vykonať vďaka dohode stanovenej pre naše potreby. Bude sa dotýkať spôsobu pomenovania (nomenklatúry) grafických a zvukových súborov. V jednej z prípravných kapitol grafického projektu sme už naznačili, ako bude nomenklatúra vyzerať. Budeme sa pridržiavať nasledujúcej schémy:

Účel súboru	Schéma pre názov
obrázok nehybného objektu obráteného doľava	`meno + "-ľ.png"`
obrázok nehybného objektu obráteného doprava	`meno + "-p.png"`
fáza animácie číslo `i` pri pohybe objektu doľava	`meno + "-ľ-" + i + ".png"`
fáza animácie číslo `i` pri pohybe objektu doprava	`meno + "-p-" + i + ".png"`
zvuk vydaný objektom pri pohybe	`meno + ".wav"`

Všetky názvy využívajú meno objektu. Premenná `i` bude nadobúdať číselné hodnoty v rozmedzí 1 až počet fáz chôdze (ktorý definujeme neskôr). Ešte pridáme konvenciu ukladania všetkých súborov do spoločného priečinka s názvom súbory. Pri nomenklatúre súborov to zohľadníme tak, že pred každý prvok schémy pridáme text `"súbory/"`. Nomenklatúru mediálnych súborov projektu môžeme považovať za vyriešenú.

Animácia nebude povinná pre všetky objekty, preto inicializáciu s ňou súvisiacich prvkov ponecháme na samostatnú metódu volanú v prípade potreby. Práve sme zhromaždili všetky informácie potrebné na zostavenie konštruktora. Ten bude za týchto okolností vyzerať takto:

```
public AnimovanýObjekt(String meno, double posunX, double posunY,
    double posunPopisuX, double posunPopisuY)
{
    zdvihniPero();

    obrázokDoľava[0] = "súbory/" + meno + "-ľ.png";
    obrázokDoprava[0] = "súbory/" + meno + "-p.png";
    zvuk = "súbory/" + meno + ".wav";

    this.meno = meno;
    this.posunX = posunX;
    this.posunY = posunY;
    this.posunPopisuX = posunPopisuX;
    this.posunPopisuY = posunPopisuY;

    spôsobKreslenia(KRESLI_NA_STRED);
    rýchlosť(10, false);
}
```

Objektu sme nastavili konštantnú rýchlosť 10 bodov za jeden tik[6] a spôsob kreslenia „na stred“, vďaka čomu nebudú obrázky vlastného tvaru robota pootočené v aktuálnom smere robota. (V neskoršej fáze projektu si môžete skúsiť, ako by to vyzeralo, keby sme túto vlastnosť nenastavili.) Taktiež sme zdvihli pero, aby objekty pri presune nekreslili na plátno čiaru.

Môžeme prejsť na definovanie vlastného tvaru animovaného objektu (odvodeného od robota). Aj keď prekrytie metódy `kresliTvar` nie je optimálnym spôsobom kreslenia vlastného tvaru (pozri dokumentáciu skupiny tried `GRobot`), rozhodli sme sa ho (na zjednodušenie) v rámci toho projektu využiť. Vlastný tvar robota (alias animovaného objektu) bude kreslený podľa aktuálnej orientácie robota:

```
@Override public void kresliTvar()
{
```

[6] Tik je udalosť časovača. Vzniká v pravidelných intervaloch určených aktuálnym nastavením časovača.

```
    skoč(posunX, posunY);

    if (smer() >= 90 && smer() <= 270)
        obrázok(obrázokDoľava[0]);
    else
        obrázok(obrázokDoprava[0]);
}
```

(**Na zamyslenie:** prečo je podmienka `if` stanovená práve takto? Na aký účel sme do tela metódy umiestnili prvý príkaz `skoč`?)

Na to, aby sme mohli do metódy dopracovať animáciu pohybu, musíme vykonať niekoľko ďalších definícií. Najprv definujeme konštantu určujúcu maximálny počet fáz animácie:

```
private final static int fázChôdze = 2;
```

Tú spätne využijeme pri definícii inštančných polí `obrázokDoľava` a `obrázokDoprava`. Opäť skúste urobiť definíciu najskôr samostatne.

Pre potreby vnútorného riadenia triedy zavedieme konvenciu. Ak bude hodnota premennej `fázaChôdze` nenulová, bude to pre nás znamenať, že objekt je animovaný a budeme sa spoliehať, že polia obsahujú názvy súborov s animáciou chôdze.

> *Takouto konvenciou môžeme ušetriť objem vnútorných premenných, prípadne optimalizovať kód, avšak treba konať premyslene. Každú takúto dohodu treba zapracovať do komentárov v zdrojovom kóde. Nikdy si nemôžeme byť na sto percent istí, že triedu po nás nebude niekto v budúcnosti upravovať…*

Na tento účel je vhodné definovať samostatnú metódu. Prostredníctvom nej získa objekt možnosť zapnutia animácie i v rámci odvodených tried:

```
public void umožniAnimáciuChôdze()
{
    for (int i = 1; i <= fázChôdze; ++i)
    {
        obrázokDoľava[i] = "súbory/" + meno + "-ľ-" + i + ".png";
        obrázokDoprava[i] = "súbory/" + meno + "-p-" + i + ".png";
    }
    fázaChôdze = 1;
}
```

Pred dokončením metódy `kresliTvar` definujeme ešte jednu pomocnú metódu `vPohybe`, ktorú umiestnime v rámci projektu všade tam, kde budeme potrebovať detegovať, či sa objekt momentálne pohybuje. Momentálne je jej obsah jednoduchý – objekt sa pohybuje, keď je aktívny. Na prvý pohľad sa zdá, že metóda je zbytočná, pretože namiesto nej by sme mohli použiť priamo metódu `aktívny`. Napriek tomu odporúčam zvyknúť si tvoriť podobné metódy. Ich benefit sa často prejaví neskôr, napríklad ak treba upraviť správanie odvodenej triedy alebo ak by sme zrazu spätne potrebovali preprogramovať (zovšeobecniť alebo rozšíriť) podmienky detekcie pohybujúceho sa objektu:

```
public boolean vPohybe()
{
    return aktívny();
}
```

Z rovnakého dôvodu definujeme metódu `myšV`, ktorú využijeme všade tam, kde budeme potrebovať zistiť, či sa ukazovateľ myši nachádza nad objektom:

```
public boolean myšV()
{
    return myšVElipse(50, 60);
}
```

Obe metódy využijeme aj v rámci kreslenia. Pre tie objekty, ktorým sme umožnili animáciu v pohybe (samozrejme pri spoľahnutí sa, že súbory s fázami animácie jestvujú – vytvárali sme ich počas prípravy grafického projektu), naprogramujeme mechanizmus animácie a nad tými objektmi, nad ktorými sa práve nachádza ukazovateľ myši, zobrazíme ich meno. Úplná verzia metódy `kresliTvar` bude vyzerať takto:

```
@Override public void kresliTvar()
{
    skoč(posunX, posunY);

    if (vPohybe() && fázaChôdze != 0)
    {
        if (smer() >= 90 && smer() <= 270)
            obrázok(obrázokDoľava[fázaChôdze]);
        else
            obrázok(obrázokDoprava[fázaChôdze]);
    }
    else if (smer() >= 90 && smer() <= 270)
        obrázok(obrázokDoľava[0]); else obrázok(obrázokDoprava[0]);

    domov();
    if (myšV())
    {
        skoč(posunPopisuX, posunPopisuY);
        text(meno);
    }
}
```

Implementácia animovaného objektu nie je len o grafickej stránke. Zvyšnú funkcionalitu dopracujeme v nasledujúcej kapitole.

Príloha 8 – prvá fáza tvorby triedy AnimovanýObjekt

```
public class AnimovanýObjekt extends GRobot
{
    // Fixný počet fáz animácie pri chôdzi.
    private final static int fázChôdze = 2;

    private Pozícia pozícia = Pozícia.pravýBreh;
    // Premenné zvuk, obrázokDoľava a obrázokDoprava budú obsahovať
    // názvy súborov podľa vnútorne dohodnutej schémy.
    private String meno, zvuk;
    private String obrázokDoľava[] = new String[1 + fázChôdze];
    private String obrázokDoprava[] = new String[1 + fázChôdze];
    private AnimovanýObjekt cieľ;
    private double posunX, posunY;
    private double ľavýX = 0, ľavýY = 0;
    private double pravýX = 0, pravýY = 0;
    private double posunPopisuX;
    private double posunPopisuY;
    private int fázaChôdze = 0; // Ak je hodnota nulová, znamená to,
```

```
                                    // že objekt nie je animovaný.

    public AnimovanýObjekt(String meno, double posunX, double posunY,
        double posunPopisuX, double posunPopisuY)
    {
        zdvihniPero();

        // Inicializujem názvy súborov podľa dohody.
        obrázokDoľava[0] = "súbory/" + meno + "-ľ.png";
        obrázokDoprava[0] = "súbory/" + meno + "-p.png";
        zvuk = "súbory/" + meno + ".wav";

        this.meno = meno;
        this.posunX = posunX;
        this.posunY = posunY;
        this.posunPopisuX = posunPopisuX;
        this.posunPopisuY = posunPopisuY;

        spôsobKreslenia(KRESLI_NA_STRED);
        rýchlosť(10, false);
    }

    @Override public void kresliTvar()
    {
        skoč(posunX, posunY);

        if (vPohybe() && fázaChôdze != 0)
        {
            if (smer() >= 90 && smer() <= 270)
                obrázok(obrázokDoľava[fázaChôdze]);
            else
                obrázok(obrázokDoprava[fázaChôdze]);
        }
        else if (smer() >= 90 && smer() <= 270)
            obrázok(obrázokDoľava[0]); else obrázok(obrázokDoprava[0]);

        domov();
        if (myšV())
        {
            skoč(posunPopisuX, posunPopisuY);
            text(meno);
        }
    }

    public boolean vPohybe()
    {
        return aktívny();
    }

    public boolean myšV()
    {
        return myšVElipse(50, 60);
    }

    public void umožniAnimáciuChôdze()
    {
```

```
        for (int i = 1; i <= fázChôdze; ++i)
        {
            obrázokDoľava[i] = "súbory/" + meno + "-ľ-" + i + ".png";
            obrázokDoprava[i] = "súbory/" + meno + "-p-" + i + ".png";
        }
        fázaChôdze = 1;
    }
}
```

Grafický projekt 2

V predchádzajúcej kapitole sme začali s definíciou triedy `AnimovanýObjekt`. V nej naprogramujeme čo najviac všeobecných vlastností animovaných objektov, ktoré neskôr upravíme pre pasažierov a prievozníka. V rámci tejto kapitoly dokončíme základ triedy `AnimovanýObjekt`.

Úvodom dokončíme možnosti inicializácie objektu. Predvolené hodnoty všetkých vlastností sú nastavované buď priamo definične, alebo v konštruktore. Doteraz sme poskytli možnosť zapnutia animácie pri chôdzi (pričom sme si zároveň stanovili vlastné vnútorné pravidlá). Ešte budeme potrebovať umožniť nastavenie fyzickej cieľovej pozície objektu na ľavom a pravom brehu. To znamená pozície určujúce pevné body na obrazovke so súradnicami x, y, na ktoré objekt na obrazovke dokráča potom, čo ho pošleme na jeden z brehov. Pre prievozníka to bude stabilná pozícia pri brehu a pre postavičky na brehu. Vyrobíme si na to dvojicu metód:

```
public void nastavĽavý(double x, double y)
{
    ĽavýX = x;
    ĽavýY = y;
}

public void nastavPravý(double x, double y)
{
    pravýX = x;
    pravýY = y;
}
```

> *(**Zamyslenie:** dokážem si predstaviť, ako by sme ich v rámci iného projektu vedeli využiť v tele cyklu, ktorý by umiestňoval postavičky zo zoznamu, konkrétne `Zoznam<AnimovanýObjekt>`, v jednom rade, ale pre náš projekt to nebude využiteľné, my budeme mať tri samostatné inštancie, pre ktoré sa neopláca definovať zoznam. Preto budeme nastavovať pozíciu „ručne“ – pri inicializácii jednotlivých inštancií.)*

Tým sme uzavreli potreby a možnosti inicializácie animovaných objektov. Ďalej doprogramujeme trojicu jednoduchých metód na zisťovanie stavov objektu. V podstate sa všetky tri budú dotýkať tej istej inštančnej premennej `pozícia`, to znamená aktuálneho logického umiestnenia objektu (niektorý breh alebo paluba plte). Metódy umožnia rýchle overenie toho, či sa objekt nachádza na niektorom z brehov, alebo zistenie konkrétne definovaného stavu a vďaka tomu budeme môcť stavy porovnávať (budeme to potrebovať – spomeňte si, ako sme pri programovaní textového projektu overovali úspešnosť a neúspešnosť hráčovho počínania):

```
public boolean pravýBreh()
{
    return pozícia == Pozícia.pravýBreh;
}
```

```
public boolean ľavýBreh()
{
    return pozícia == Pozícia.ľavýBreh;
}

public Pozícia pozícia()
{
    return pozícia;
}
```

Ďalšou záležitosťou na vyriešenie je posielanie objektov na konkrétne pozície na obrazovke – rozanimovanie objektov. Prvú časť tejto funkcionality sme už naprogramovali v rámci metódy `kresliTvar`, čo zahŕňalo vykonanie všetkých potrebných definícií. Ďalšie dve časti sa budú dotýkať naprogramovania metód zahajujúcich (odštartujúcich) animáciu a prekrytia metód robota starajúcich sa o tie aktivity robota, ktoré fyzicky zabezpečujú animovanie objektu. Najskôr implementujme metódy slúžiace na posielanie objektov na jednotlivé brehy (čiže na kotviace pozície na brehoch definované pomocou premenných `pravýX`, `pravýY`, `ľavýX` a `ľavýY`) a na palubu plte. Všetko potrebné na naprogramovanie dvojice metód, ktoré spustia fyzický presun robota na určený breh už máme k dispozícii. Stačí poslať robota na cieľovú pozíciu, čím dôjde automaticky k jeho aktivácii, a o ostatné sa už postará funkcionalita robota. Popritom zároveň zmeníme hodnotu vnútornej premennej `pozícia`, čiže logickú pozíciu robota (animovaného objektu):

```
public void choďNaĽavý()
{
    cieľ(ľavýX, ľavýY);
    pozícia = Pozícia.ľavýBreh;
}

public void choďNaPravý()
{
    cieľ(pravýX, pravýY);
    pozícia = Pozícia.pravýBreh;
}
```

Pokusne vytvorme inštanciu animovaného objektu pre vlka a postupne zavolajme jeho metódy `kresliTvar` (tú preto, lebo animovaný objekt v čase vytvorenia sveta (vrátane kresliaceho plátna) ešte nemá inicializované všetky premenné a nemôže sa správne vykresliť), `umožniAnimáciuChôdze`, `nastavPravý` a `choďNaPravý`:

```
AnimovanýObjekt vlk = new AnimovanýObjekt("vlk", 0, 0, 0, 0);
vlk.kresliTvar();
vlk.umožniAnimáciuChôdze();
vlk.nastavPravý(100, 0);
vlk.choďNaPravý();
```

> *Príkazy môžeme zadať do anonymného statického bloku prázdnej pokusnej triedy (`public class Pokus { static { … }}`), z ktorej následne vytvoríme inštanciu, alebo vytvoriť inštanciu triedy `AnimovanýObjekt` v prostredí BlueJ-a – kliknutím pravého tlačidla nad preloženou triedou `AnimovanýObjekt` a voľbou položky „new…“ a následne spustíme želané metódy nad vytvoreným objektom – kliknutím pravého tlačidla nad (červenou) inštanciou v spodnej časti hlavného okna a voľbou želanej metódy. Prvý spôsob znamená menej klikania.*

Uvidíme vlka pohybujúceho sa zo stredu doprava (na súradnicu x = 100 bodov), pričom počas chôdze bude mať stále zdvihnutú len jednu labu. Striedanie fáz animácie zabezpečíme v prekrytej metóde `aktivita`:

```
@Override public void aktivita()
{
    if (fázaChôdze != 0 && ++fázaChôdze > fázChôdze) fázaChôdze = 1;
}
```

Tento zápis zabezpečuje, aby sa hodnota premennej `fázaChôdze` menila v rozsahu 1 až `fázChôdze`, avšak len v prípade, že jej hodnota je nenulová. (Pretože iba vtedy máme istotu, že sú inicializované všetky potrebné premenné.) Ak vytvoríme vlka teraz a vykonáme všetky vyššie uvedené príkazy, uvidíme vlka kráčať. Tým sme zabezpečili, aby sa postavička mohla z ľubovoľného miesta na plátne presunúť na stabilnú pozíciu na brehu a aby bola počas toho animovaná jej činnosť (chôdza). Podobne budeme potrebovať zabezpečiť posielanie postavičiek na palubu plte.

Na to, aby sme vlka (alebo akúkoľvek postavičku) poslali na palubu plte, potrebujeme mať definovaného prievozníka. Ten síce v súčasnosti nejestvuje, ale môžeme si vopred vytvoriť „tieňový“ objekt, s ktorým môžeme narábať bez ohľadu na to, či momentálne jestvuje alebo nie. Inak povedané, môžeme naprogramovať metódy pracujúce s prievozníkom bez toho, že by prievozník jestvoval. Ibaže vyskúšať ich funkčnosť budeme môcť až v ďalšej fáze programovania – v čase, keď vytvoríme skutočného prievozníka.

Definujme statickú vlastnosť `prievozník` – inicializujeme ju neskôr, v čase vytvorenia skutočného prievozníka. (Ako sme povedali, vlastnosť budeme v súčasnosti chápať ako prievozníkov „tieň“ a prechod na palubu plte si budeme môcť prakticky vyskúšať neskôr, keď bude pltník fyzicky jestvovať.) Vlastnosť `prievozník` bude ukazovateľom na pltníka, čiže všade, kde budeme potrebovať pracovať s animovaným objektom reprezentujúcim prievozníka, použijeme túto vlastnosť. V súlade s princípom objektovej uzavretosti (**encapsulation**), bude vlastnosť ukazovateľa na pltníka realizovaná trojicou: súkromná premenná, verejné metódy (jedna na zápis, druhá na čítanie):

```
private static AnimovanýObjekt prievozník;

public static void prievozník(AnimovanýObjekt prievozník)
{
    AnimovanýObjekt.prievozník = prievozník;
}

public static AnimovanýObjekt prievozník()
{
    return prievozník;
}
```

Metóda inicializujúca prechod postavičky na palubu plte bude túto vlastnosť využívať predpokladajúc, že v čase jej spustenia už bude pltník jestvovať:

```
public void choďNaPrievozníka()
{
    cieľ(cieľ = prievozník);
    pozícia = Pozícia.prievozník;
}
```

Metóda (okrem zmeny pozície) jedným razom nastaví nový cieľ animovaného objektu (to jest súkromný cieľ animovanej postavičky – to je vlastnosť definovaná v rámci animovaného objektu pre naše potreby) na prievozníka a začne prechádzanie na jeho aktuálnu pozíciu (to jest – zároveň nastaví aktuálny cieľ zdedený

po robotovi). Metóda `cieľ` (pozri dokumentáciu triedy `GRobot`) je schopná začať presun smerom na cieľový objekt (napr. prievozníka), ale iba na jeho aktuálnu polohu v čase jej spustenia. Ak by sa počas presunu postavičky na palubu pltník pohol, postavička by dorazila na pozíciu, kde bol pltník v čase začatia svojho pohybu, a tam by zastavila. Ak chceme túto nelogickosť vyriešiť, pridáme do metódy aktivita pravidelnú aktualizáciu cieľa:

```
@Override public void aktivita()
{
    if (null != cieľ) upravCieľ(cieľ);
    if (fázaChôdze != 0 && ++fázaChôdze > fázChôdze) fázaChôdze = 1;
}
```

Teraz by malo byť všetko v poriadku, testovaním by sme však rýchlo zistili zvláštne správanie. Konkrétne, keby sme chceli postavičku poslať na ľubovoľné súradnice na plátne (to jest, realizovať vystúpenie postavičky z plte na breh), mala by stále tendenciu zostať na palube plte, pretože jej cieľ by bol stále aktualizovaný smerom na pltníka. Stalo by sa to hneď na prvom riadku metódy `aktivita` hovoriacom: „ak je môj súkromný cieľ neprázdny, uprav aktuálny cieľ na tento súkromný cieľ.“ Tento riadok je v poriadku, iba sa musíme postarať o to, aby sa nevykonal vtedy, keď nechceme. Postačí zabezpečiť to, aby sa súkromný cieľ vždy po jeho dosiahnutí vynuloval. To môžeme urobiť prekrytím metódy `dosiahnutieCieľa` obsahujúcej jediný riadok kódu:

```
@Override public void dosiahnutieCieľa()
{
    cieľ = null;
}
```

Teraz sme schopní poslať ľubovoľný animovaný objekt na palubu plte prievozníka (ktorého musíme vopred zvoliť) a na jeden z brehov. O animáciu sa čiastočne starajú mechanizmy robota a čiastočne nami naprogramované vlastnosti triedy `AnimovanýObjekt`. Napohľad by sme mohli povedať, že trieda `AnimovanýObjekt` je hotová. Avšak nepredbiehajme, je možné, že počas programovania odvodených tried – `Prievozník` a `Pasažier` – vzniknú ďalšie požiadavky, ktoré vo fáze plánovania projektu nie sme vždy schopní predvídať…

Príloha 9 –druhá fáza tvorby triedy AnimovanýObjekt

```
public class AnimovanýObjekt extends GRobot
{
    private static AnimovanýObjekt prievozník;

    public static void prievozník(AnimovanýObjekt prievozník)
    {
        AnimovanýObjekt.prievozník = prievozník;
    }

    public static AnimovanýObjekt prievozník()
    {
        return prievozník;
    }

    // Fixný počet fáz animácie pri chôdzi.
    private final static int fázChôdze = 2;

    private Pozícia pozícia = Pozícia.pravýBreh;
```

```
    // Premenné zvuk, obrázokDoľava a obrázokDoprava budú obsahovať
    // názvy súborov podľa vnútorne dohodnutej schémy.
    private String meno, zvuk;
    private String obrázokDoľava[] = new String[1 + fázChôdze];
    private String obrázokDoprava[] = new String[1 + fázChôdze];
    private AnimovanýObjekt cieľ;
    private double posunX, posunY;
    private double ľavýX = 0, ľavýY = 0;
    private double pravýX = 0, pravýY = 0;
    private double posunPopisuX;
    private double posunPopisuY;
    private int fázaChôdze = 0; // Ak je hodnota nulová, znamená to,
                                // že objekt nie je animovaný.

    public AnimovanýObjekt(String meno, double posunX, double posunY,
        double posunPopisuX, double posunPopisuY)
    {
        zdvihniPero();

        // Inicializujem názvy súborov podľa dohody.
        obrázokDoľava[0] = "súbory/" + meno + "-ľ.png";
        obrázokDoprava[0] = "súbory/" + meno + "-p.png";
        zvuk = "súbory/" + meno + ".wav";

        this.meno = meno;
        this.posunX = posunX;
        this.posunY = posunY;
        this.posunPopisuX = posunPopisuX;
        this.posunPopisuY = posunPopisuY;

        spôsobKreslenia(KRESLI_NA_STRED);
        rýchlosť(10, false);
    }

    @Override public void kresliTvar()
    {
        skoč(posunX, posunY);

        if (vPohybe() && fázaChôdze != 0)
        {
            if (smer() >= 90 && smer() <= 270)
                obrázok(obrázokDoľava[fázaChôdze]);
            else
                obrázok(obrázokDoprava[fázaChôdze]);
        }
        else if (smer() >= 90 && smer() <= 270)
            obrázok(obrázokDoľava[0]); else obrázok(obrázokDoprava[0]);

        domov();
        if (myšV())
        {
            skoč(posunPopisuX, posunPopisuY);
            text(meno);
        }
    }
```

```
    public boolean vPohybe()
    {
        return aktívny();
    }

    public boolean myšV()
    {
        return myšVElipse(50, 60);
    }

    public boolean pravýBreh()
    {
        return pozícia == Pozícia.pravýBreh;
    }

    public boolean ľavýBreh()
    {
        return pozícia == Pozícia.ľavýBreh;
    }

    public Pozícia pozícia()
    {
        return pozícia;
    }

    public void nastavĽavý(double x, double y)
    {
        ľavýX = x;
        ľavýY = y;
    }

    public void nastavPravý(double x, double y)
    {
        pravýX = x;
        pravýY = y;
    }

    public void umožniAnimáciuChôdze()
    {
        for (int i = 1; i <= fázChôdze; ++i)
        {
            obrázokDoľava[i] = "súbory/" + meno + "-ľ-" + i + ".png";
            obrázokDoprava[i] = "súbory/" + meno + "-p-" + i + ".png";
        }
        fázaChôdze = 1;
    }

    public void choďNaĽavý()
    {
        cieľ(ľavýX, ľavýY);
        pozícia = Pozícia.ľavýBreh;
    }

    public void choďNaPravý()
    {
        cieľ(pravýX, pravýY);
        pozícia = Pozícia.pravýBreh;
```

```
    }

    public void choďNaPrievozníka()
    {
        cieľ(cieľ = prievozník);
        pozícia = Pozícia.prievozník;
    }

    @Override public void dosiahnutieCieľa()
    {
        cieľ = null;
    }

    @Override public void aktivita()
    {
        if (null != cieľ) upravCieľ(cieľ);
        // Nenulová hodnota fázy animácie znamená, že sme vykonali potrebné
        // nastavenia, aby mohol byť objekt animovaný.
        if (fázaChôdze != 0 && ++fázaChôdze > fázChôdze) fázaChôdze = 1;
    }
}
```

Grafický projekt 3

V predchádzajúcej kapitole sme dokončili triedu `AnimovanýObjekt`, resp. prinajmenšom sme ju pripravili na základný rámec fungovania tried `Prievozník` a `Pasažier`, ktoré z nej odvodíme. V rámci tejto kapitoly implementujeme štartovacie verzie spomenutých tried. Pripomíname, že hlavným zmyslom existencie triedy `AnimovanýObjekt` je poskytnutie spoločného rozhrania (základne) pre všetky animované objekty v hre.

Vytvorme dve nové prázdne triedy `Prievozník` a `Pasažier` (pozri kapitolu Vytvorenie novej triedy – v dialógu na vytvorenie novej triedy zvolíme „Prázdna trieda“). Obe odvoďme od prievozníka (pozri kapitolu Odvodenie triedy). Ich kód bude vyzerať takto (samozrejme v samostatných súboroch):

```
public class Prievozník extends AnimovanýObjekt
{
}

public class Pasažier extends AnimovanýObjekt
{
}
```

Nepôjdu preložiť. Je to preto, lebo nemajú definovaný konštruktor v súlade s požiadavkami nadradenej triedy `AnimovanýObjekt`. Musíme zabezpečiť správnu inicializáciu. Začneme triedou `Prievozník`, tá je ústrednejšia a budú ju využívať aj postavičky pasažierov. Pri tvorbe triedy `AnimovanýObjekt` sme vytvorili všeobecnú vlastnosť `prievozník`, ktorá mala hru vopred pripraviť na existenciu prievozníka. Teraz na to budeme pamätať. Keďže prievozník bude len jeden, môžeme v konštruktore bez obáv predvoliť všetky jeho vlastnosti. Tým získame bezparametrový konštruktor prievozníka, ktorý bude volať konštruktor nadradenej triedy s konkrétnymi hodnotami:

```
public Prievozník()
{
    super("prievozník", -30, 10, -30, 115);
}
```

> *Volaniu nadradeného konštruktora sa nevyhneme, Java by nám nedovolila program preložiť. Preklad by sa zastavil na konštruktore tejto triedy s chybovým hlásením oznamujúcim, že počet argumentov nadradenej triedy sa líši od „aktuálne zadaného“ (nezadali sme žiadny). Nepomohlo by ani vytvorenie parametrického konštruktora s rovnakým počtom argumentov ako má nadradená trieda, pretože Java parametre konštruktorov neprenáša automaticky. Ak neurčíme inak, volá sa bezparametrový konštruktor nadradenej triedy, ktorý momentálne pre triedu* `AnimovanýObjekt` *nejestvuje.*

Teraz už trieda `Prievozník` preložiť pôjde. Okrem povinného volania nadradeného konštruktora nastavíme v rámci procesu inicializácie niektoré vlastnosti prievozníka zdedené ešte od nadradených tried `AnimovanýObjekt` a `GRobot` (domovskú pozíciu, zrýchlenie, kotviace pozície). Nakoniec zavoláme statickú metódu `prievozník` (definovali sme ju v triede `AnimovanýObjekt`) s argumentom `this` (tento), čím oznámime prostrediu, že „toto je prievozník“ – „**ja** som prievozník“.

```
public Prievozník()
{
    super("prievozník", -30, 10, -30, 115);

    domov(50, -25, 0);
    zrýchlenie(2, false);
    nastavĽavý(domaX() - 50, domaY());
    nastavPravý(domaX() + 50, domaY());

    prievozník(this);
}
```

Prakticky všetky parametre prievozníka sú tu zadané „napevno“ – numerickými literálmi (terminologická pripomienka – používame termín „literál“, termínu „konštanta“ by sme sa mali v tomto kontexte vyhýbať, pretože tento termín v programovaní označuje „nemennú hodnotu označenú identifikátorom“). Flexibilitu by sme mohli zvýšiť tak, že niektoré parametre by sme určili pomocou premenných, argumentov či konštánt. Napríklad polohu prievozníka na jednotlivých brehoch by sme mohli určiť pomocou konštanty určujúcej rozptyl vzhľadom na domovskú pozíciu (čiže to, o koľko bodov doľava, resp. doprava je umiestnený kotviaci bod ľavého, resp. pravého brehu voči domovskej pozícii). Momentálne nastavujeme kotviace body na 50 bodov od domovskej pozície (volaním metód `nastavĽavý` a `nastavPravý`). Ak sme si istí, že rozptyl bude za každých okolností rovnaký, definujeme konštantu `rozptylBrehov`:

```
private final static int rozptylBrehov = 50;
```

ktorú použijeme vo volaní metód `nastavĽavý` a `nastavPravý` v konštruktore:

```
nastavĽavý(domaX() - rozptylBrehov, domaY());
nastavPravý(domaX() + rozptylBrehov, domaY());
```

Teraz, hocikedy, keď budeme potrebovať zväčšiť rozptyl kotviacich bodov na brehoch, postačí zvýšiť hodnotu z 50 napríklad na 90. Konštantami môžeme zároveň zvýšiť čitateľnosť programu. Lepšie sa číta kód, kde sú konštantné hodnoty pomenované, než kód plný čísiel. Ak je definícia konštanty vyslovene zbytočná, môžeme zvýšiť čitateľnosť kódu komentármi. Porovnajte verziu vyššie uvedeného konštruktora s nasledujúcou verziou doplnenou o definované konštanty:

```
private final static int rozptylBrehov = 90;
private final static int posunGrafikyX = -30;
private final static int posunGrafikyY = 10;
private final static int posunPopisuX = -30;
private final static int posunPopisuY = 115;
```

```
public Prievozník()
{
    super("prievozník", posunGrafikyX, posunGrafikyY, posunPopisuX, posunPopisuY);

    domov(50, -25, 0);    // domovská pozícia [x, y] a domovský uhol
    zrýchlenie(2, false);
    nastavĽavý(domaX() - rozptylBrehov, domaY());
    nastavPravý(domaX() + rozptylBrehov, domaY());

    prievozník(this);
}
```

V predchádzajúcej verzii bol prvý riadok plný čísiel bez jasného významu. Teraz obsahuje slová opisujúce význam hodnôt. Význam konštánt sa násobí, ak potrebujeme v rámci programu používať rovnakú hodnotu v tom istom kontexte.

Keďže väčšinu funkcionality sme naprogramovali v triede `AnimovanýObjekt`, máme základ triedy `Prievozník` hotový. Ešte ju obohatíme o možnosť automatického prechádzania prievozníka z jedného brehu na druhý. Naprogramujeme metódu `prejdiRieku`:

```
public void prejdiRieku()
{
    if (pravýBreh())
        choďNaĽavý();
    else
        choďNaPravý();
}
```

Čiže: „ak som na pravom brehu, prejdi na ľavý a naopak.“ Ak teraz v BlueJ-i vytvoríme novú inštanciu prievozníka a spustíme metódu `prejdiRieku`, uvidíme pohybujúceho sa prievozníka.

Pristúpme k tvorbe triedy `Pasažier`. Pri jednotlivých pasažieroch budeme chcieť definovať všetky argumenty prijímané konštruktorom triedy `AnimovanýObjekt` individuálne, preto definujeme konštruktor, ktorý ich bude všetky prijímať a posúvať nadradenému konštruktoru. Zároveň, keďže budeme chcieť, aby bola chôdza každého pasažiera animovaná (ako sme si to pokusne vyskúšali pri vlkovi), môžeme priamo do konštruktora umiestniť volanie metódy `umožniAnimáciuChôdze`:

```
public Pasažier(String meno, double posunX, double posunY,
    double posunPopisuX, double posunPopisuY)
{
    super(meno, posunX, posunY, posunPopisuX, posunPopisuY);
    umožniAnimáciuChôdze();
}
```

Pre jednotlivých pasažierov budeme potrebovať nastaviť rôzne kotviace pozície na brehoch. Podobne ako pri prievozníkovi si môžeme situáciu uľahčiť zrkadlovým umiestnením. Môžeme na to vyrobiť jednoduchú metódu prijímajúcu súradnice bodu na pravom brehu, z ktorých odvodíme zrkadlovú pozíciu na ľavom brehu:

```
public void pozíciaNaBrehu(double x, double y)
{
    nastavĽavý(-x, y);
    nastavPravý(x, y);
}
```

V tomto momente je najvyšší čas navrhnúť štýl ovládania hry. Navrhujem vyriešiť to čo najjednoduchšie. Hráč klikne na postavičku, s ktorou chce manipulovať, a tá sa automaticky pohne podľa toho, v akom je momentálnom rozpoložení. Čiže ak bude na niektorom z brehov, overí, či sa prievozník nachádza na rovnakom brehu, a ak áno, prejde na palubu plte, a naopak, ak je na palube, vystúpi na ten breh, pri ktorom sa prievozník momentálne nachádza. To vyžaduje detekciu troch rôznych stavov. Na to je vhodné použiť riadiacu štruktúru `switch`. Metódu nazvime `choď`. Všetko potrebné na jej implementáciu máme pripravené. Stačí to správne použiť:

```
public void choď()
{
    switch (pozícia())
    {
        case pravýBreh:
            if (prievozník().pravýBreh())
                choďNaPrievozníka();
            break;

        case ľavýBreh:
            if (prievozník().ľavýBreh())
                choďNaPrievozníka();
            break;

        case prievozník:
            if (prievozník().ľavýBreh())
                choďNaĽavý();
            else
                choďNaPravý();
            break;
    }
}
```

Teraz môžeme pristúpiť k vytvoreniu hlavnej triedy, pomocou ktorej všetky doteraz naprogramované komponenty prepojíme. Vytvoríme hlavnú triedu (v dialógu na vytvorenie triedy zvolíme „Hlavná trieda aplikácie“) s názvom `HlavnáTrieda`. Po prečistení a úpravách vygenerovaného kódu získame takýto zjednodušený tvar:

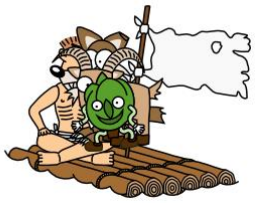
```
public class HlavnáTrieda extends GRobot
{
    private HlavnáTrieda()
    {
    }

    public static void main(String[] args)
    {
        new HlavnáTrieda();
    }
}
```

Do neho budeme postupne vpisovať potrebný kód. V prvom rade je nutné vytvoriť všetky inštancie postáv hry, s ktorými budeme manipulovať. Súradnice relatívneho posunu grafiky a popisného textu pri pasažieroch by sme predbežne stanovili odhadom a neskôr prispôsobili – metódou pokusu a omylu. Vo výsledku by mohli byť definície takéto:

```
private final Prievozník prievozník = new Prievozník();
```

```
private final Pasažier vlk = new Pasažier("vlk", 0, 0, 0, 65);
private final Pasažier koza = new Pasažier("koza", 0, -10, -5, 50);
private final Pasažier kapusta = new Pasažier("kapusta", 0, -25, 0, 25);
```

Ak teraz spustíme hlavnú triedu, všetky postavičky sa zobrazia vzájomne prekryté v strede plátna:



Samozrejme, zatiaľ nijaká z nich nereaguje (napr. na klikanie), to sme zatiaľ neriešili. Môžeme ich aspoň príkazovo navigovať k jednému z brehov. Ak do konštruktora napíšeme:

```
prievozník.choďNaPravý();
vlk.choďNaPravý();
koza.choďNaPravý();
kapusta.choďNaPravý();
```

Keď triedu spustíme, zistíme, že zo všetkých postáv sa iba prievozník vydá k neviditeľnému pravému brehu. Na čo sme zabudli…? Pozrime sa podrobne na triedu `Pasažier`. Zdá sa, že všetko máme… Všetko… až na to, že sme doteraz nikde nevolali metódu `pozíciaNaBrehu`…(!) Ak metódu nezavoláme, akoby nejestvovala. Je to ako napísať recept, ale nečítať ho, iba ho založiť do kuchárskej knihy a nikdy nepoužiť… Máme niekoľko možností, ako situáciu doriešiť. Buď metódu zavoláme v konštruktore `HlavnáTrieda`, alebo vygenerujeme náhodné súradnice v určitom rozsahu a volanie pridáme na koniec konštruktora `Pasažier`, alebo pridáme ďalšie dva argumenty do konštruktora `Pasažier` a nimi inicializujeme pozície na brehoch. Osobne by som volil tú poslednú možnosť.

Pozastavme sa teraz nachvíľu. Predstavme si, že nie sme autormi triedy `Pasažier`. O akékoľvek zmeny musíme požiadať autora, spoluriešiteľa v rámci tímu. Požiadavka musí byť čo najpresnejšia. Mala by obsahovať, s čím zmena súvisí, aké poskytujeme vstupy, aké očakávame výstupy, prípadne stručné zdôvodnenie požiadavky a podobne. Požiadavku zaradíme do zoznamu požiadaviek (**wish-list**) a ak je to možné, naprogramujeme náš kód tak, ako by už bolo požiadavke vyhovené (ibaže s testovaním budeme musieť vyčkať).

Požiadavka na novú funkcionalitu by mohla vyzerať relatívne jednoducho:

- **Umiestnenie:** trieda `Pasažier`, konštruktor;
- **Opis:** vytvorenie novej verzie konštruktora alebo úprava jestvujúcej verzie – pridanie dvoch nových vstupov;
- **Vstupy:** pozícia na brehu určená dvoma premennými typu `double`;
- **Výstupy:** objekt s inicializovanými kotviacimi pozíciami na oboch brehoch;
- **Zdôvodnenie:** žiadame pridanie inicializácie kotviacich bodov na brehoch do konštruktora; zjednoduší sa tým pre nás proces inicializácie nových objektov typu `Pasažier`.

Formát požiadavky môže vyzerať v rámci každej organizácie rôzne. Individuálne rozdiely sa môžu dotýkať formátovania, rozloženia, niektoré položky môžu byť nepovinné a podobne. Požiadavku by sme odoslali kolegovi (autorovi triedy) a počas čakania by sme sa museli zaoberať inými záležitosťami, pretože kým by našej požiadavke nebolo vyhovené, nemohli by sme podľa nej upraviť svoj kód. Zoznam požiadaviek vždy

môžeme využiť aj na zlepšenie organizácie našej vlastnej práce. Pomôže nám pri orientácii v množstve úloh, ktoré často čakajú pri programovaní na jedného pracovníka.

Predpokladajme, že uplynul nejaký čas a kolega našej požiadavke vyhovel: konštruktoru `Pasažier` pridal dva argumenty typu `double` – `pozíciaNaBrehuX` a `pozíciaNaBrehuY`, ktoré teraz využijeme. V rámci nášho projektu musíme kolegovu prácu zastúpiť – pridajte požadované argumenty do hlavičky konštruktora a na koniec tela konštruktora zaraďte volanie metódy `pozíciaNaBrehu`:

```
pozíciaNaBrehu(pozíciaNaBrehuX, pozíciaNaBrehuY);
```

Teraz môžeme upraviť definície pasažierov v hlavnej triede. Pridajme na záver každej inicializácie dve hodnoty (opäť, súradnice zistíme empiricky):

```
private final Pasažier vlk = new Pasažier("vlk", 0, 0, 0, 65, 200, 10);
private final Pasažier koza = new Pasažier("koza", 0, -10, -5, 50, 280, 10);
private final Pasažier kapusta = new Pasažier("kapusta", 0, -25, 0, 25, 360, 10);
```

Spustime hru. Teraz sa na svoju pozíciu vydajú všetky postavičky, ibaže zatiaľ akosi nemajú dostatok miesta. Kapusta vyjde úplne za hranice okna, až k okraju plátna, ktorý uvidíme až keď okno zväčšíme. Ak sa nechceme uspokojiť s predvolenými rozmermi plátna a okna, môžeme ich upraviť podľa potrieb v konštruktore hlavnej triedy. (Upozorňujeme, že volanie nadriadeného konštruktora – `super()` – musí byť vždy úplne prvým príkazom konštruktora odvodenej triedy!) Hlavná trieda je odvodená priamo od triedy `GRobot`. Jej konštruktory umožňujú pozmeniť predvolenú veľkosť plátna, avšak túto zmenu môžeme urobiť v rámci jedného projektu iba raz. Ďalším príkazom `svet.zbaľ` prispôsobíme veľkosť okna rozmerom plátna. Zmeňme plátno s oknom na panoramatický rozmer:

```
super(800, 300, "Prievozník…");
svet.zbaľ();
```

V tejto fáze kapitolu ukončíme. V najbližšej kapitole sa budeme venovať ovládaniu hry.

Príloha 10 – prvé verzie tried Prievozník, Pasažier a HlavnáTrieda

Prievozník

```
public class Prievozník extends AnimovanýObjekt
{
    private final static int rozptylBrehov = 90;
    private final static int posunGrafikyX = -30;
    private final static int posunGrafikyY = 10;
    private final static int posunPopisuX = -30;
    private final static int posunPopisuY = 115;

    public Prievozník()
    {
        super("prievozník", posunGrafikyX, posunGrafikyY,
            posunPopisuX, posunPopisuY);

        domov(50, -25, 0);    // domovská pozícia [x, y] a domovský uhol
        zrýchlenie(2, false);
        nastavĽavý(domaX() - rozptylBrehov, domaY());
        nastavPravý(domaX() + rozptylBrehov, domaY());

        prievozník(this);
```

```
    }

    public void prejdiRieku()
    {
        if (pravýBreh())
            choďNaĽavý();
        else
            choďNaPravý();
    }
}
```

Pasažier

```
public class Pasažier extends AnimovanýObjekt
{
    public Pasažier(String meno, double posunX, double posunY,
        double posunPopisuX, double posunPopisuY,
        double pozíciaNaBrehuX, double pozíciaNaBrehuY)
    {
        super(meno, posunX, posunY, posunPopisuX, posunPopisuY);
        umožniAnimáciuChôdze();
        pozíciaNaBrehu(pozíciaNaBrehuX, pozíciaNaBrehuY);
    }

    public void pozíciaNaBrehu(double x, double y)
    {
        nastavĽavý(-x, y);
        nastavPravý(x, y);
    }

    public void choď()
    {
        switch (pozícia())
        {
            case pravýBreh:
                if (prievozník().pravýBreh())
                    choďNaPrievozníka();
                break;

            case ľavýBreh:
                if (prievozník().ľavýBreh())
                    choďNaPrievozníka();
                break;

            case prievozník:
                if (prievozník().ľavýBreh())
                    choďNaĽavý();
                else
                    choďNaPravý();
                break;
        }
    }
}
```

HlavnáTrieda

```
public class HlavnáTrieda extends GRobot
```

```
{
    private final Prievozník prievozník = new Prievozník();
    private final Pasažier vlk = new Pasažier("vlk", 0, 0, 0, 65, 200, 10);
    private final Pasažier koza = new Pasažier("koza", 0, -10, -5, 50, 280, 10);
    private final Pasažier kapusta = new Pasažier("kapusta", 0, -25, 0, 25, 360, 10);

    private HlavnáTrieda()
    {
        super(800, 300, "Prievozník…");
        svet.zbaľ();

        prievozník.choďNaPravý();
        vlk.choďNaPravý();
        koza.choďNaPravý();
        kapusta.choďNaPravý();
    }

    public static void main(String[] args)
    {
        new HlavnáTrieda();
    }
}
```

Grafický projekt 4

V predchádzajúcej kapitole sme vytvorili základ prostredia s postavičkami na bielom pozadí. Zastavili sme sa pred problémom ovládania postavičiek. Tým sa budeme zaoberať v tejto kapitole.

V triede `AnimovanýObjekt` sme definovali metódu `myšV`, ktorú sme využili v metóde `kresliTvar`. Možno ste si všimli, že keď počas presúvania sa postavičiek z domovskej pozície na svoje kotviace pozície na brehu prejdete myšou ponad niektorú z nich, objaví sa nad jej hlavou popis. Len čo sa všetky animované objekty zastavia, tento mechanizmus prestane fungovať. Je to preto, že svet robota nemá dôvod prekresľovať grafiku, keď nie je žiadny robot aktívny. Mechanizmus popisov v skutočnosti funguje naďalej, ibaže to nevidíme. Situáciu môžeme vyriešiť aktiváciou hlavného robota.

> ***Poznámka:*** *hlavný robot je vždy prvý robot vytvorený vo svete grafických robotov. Pre nás je výhodné, ak ním je robot hlavnej triedy. Preto sme žiadny z animovaných objektov nedefinovali staticky. Máme tak istotu, že prvým robotom bude priamo robot hlavnej triedy. Možno ste si ho už všimli. Ak ste videli typický tvar trojzubca po celý čas nehybne stojaci uprostred plátna, tak to je on.*

Na konci konštruktora hlavnej triedy prikážeme:

```
aktivuj();
```

Po spustení hry bude zobrazovanie a miznutie popisov nad postavičkami fungovať aj potom, čo sa všetky zastavia. Zopakujme si teraz, ako pracuje metóda `myšV` naprogramovaná v predchádzajúcich kapitolách. Použili sme ju na zobrazovanie popisov nad hlavami postavičiek a budeme ju potrebovať pri ovládaní hry myšou. Metóda `myšV` má za úlohu overiť, či sú súradnice ukazovateľa myši v rámci plochy grafiky animovaného objektu. Ide o relatívne primitívny spôsob overenia – metóda pracuje jednoducho: iba vyhodnocuje, či sa kurzor myši nachádza v ploche pomyselnej elipsy so stredom v strede objektu. Ak chceme presne vidieť, ktorá plocha je považovaná za plochu objektu, jednoducho nakreslíme rovnakú elipsu v rámci kreslenia vlastného tvaru animovaného objektu. Pozrime sa do metódy `myšV` v triede

`AnimovanýObjekt`. Tu vidíme rozmery pomyselnej elipsy: `myšVElipse(50, 60)`. Pridajme na začiatok metódy `kresliTvar` (v rámci tej istej triedy) nasledujúce dva príkazy:

```
farba(0, 0, 100, 100);
vyplňElipsu(50, 60);
```

Po spustení hry, uvidíme pri (resp. na alebo pod – záleží kam príkaz kreslenia umiestnite) všetkých objektoch elipsy vyplnené priesvitnou modrou farbou:

Elipsa je pre každý objekt rovnaká a nie celkom presne vystihuje jeho rozmery a tvar. Keby sme chceli byť presní, museli by sme pre potreby metódy `myšV` vytvoriť tzv. klikaciu mapu, v ktorej by boli aktívne len reálne viditeľné body postavičky (na obrazovke). Nebudeme však komplikovať spôsob fungovania tejto metódy, iba mierne upravíme rozmery elipsy pre niektoré objekty, aby približne pokryli ich tvar. Pre vlka a kozu nám tvar elipsy celkom vyhovuje. Inak je to pri kapuste a prievozníkovi. Kapusta je menšia, viac-menej kruhová a potrebovali by sme elipsu posunúť o niečo nižšie, aby lepšie pokryla rozmery kapusty. Skúsme najprv odhadnúť veľkosť a umiestnenie elipsy použitej pre kapustu. Zmeňme príkazy, ktorými kreslíme priehľadnú modrú elipsu takto:

```
farba(0, 0, 100, 100);
skoč(0, -20);
vyplňElipsu(50, 50);
skoč(0, 20);
```

Prvým príkazom `skoč` sme posunuli stred elipsy o niečo nižšie a druhým sme posunutie vrátili, aby sa správanie ostatných príkazov nijako nezmenilo. Vrátenie súradníc na pôvodné hodnoty je dôležité. Ak by sme na to zabudli, hra by sa mohla začať správať „čudne“…

Spustime hru a nevšímajme si to, že nakreslená elipsa (resp. kruh) sa zmenila pre všetky objekty. Ide len o vizualizáciu – správanie metódy `myšV` sme predsa nemenili a všetky objekty reagujú tak ako doteraz. Kreslenie elíps aj tak o chvíľu vymažeme, len čo zistíme, aké rozmery a umiestnenie jej tvaru najlepšie vystihujú kapustu a prievozníka.

Súradnice pre kapustu by sme potrebovali posunúť ešte o niečo nižšie a aj elipsa by mohla byť o niečo menšia. Nakoniec zistíme, že elipsu bude lepšie nahradiť kruhom. Skúsme použiť kruh s rozmerom 40 bodov, ktorý posunieme o 30 bodov dole (nezabudnime hodnotu 30 prepísať aj v druhom príkaze – zišla by sa definícia konštanty; to ponechám na zváženie čitateľovi):

```
farba(0, 0, 100, 100);
skoč(0, -30);
kruh(40);
skoč(0, 30);
```

Je to lepšie. Ešte jeden pokus – nakoniec sa uspokojíme s rozmerom kruhu 45 bodov. Čím viac pokusov by sme vykonali, tým by bol náš odhad presnejší.

Java umožňuje upravovať správanie každého objektu individuálne pri jeho konštrukcii. Prejdime do hlavnej triedy na miesto, kde inicializujeme objekt kapusta:

```
private final Pasažier kapusta = new Pasažier("kapusta", 0, -25, 0, 25, 360, 10);
```

Tesne pred bodkočiarku vložme blok, do ktorého napíšeme novú verziu metódy `myšV` platnú len pre kapustu:

```
private final Pasažier kapusta = new Pasažier("kapusta", 0, -25, 0, 25, 360, 10)
{
    @Override public boolean myšV()
    {
        skoč(0, -30);
        boolean myšV = myšVKruhu(45);
        skoč(0, 30);
        return myšV;
    }
};
```

Návratovú hodnotu metódy `myšVKruhu` sme museli uložiť do pomocnej premennej, aby sme mohli vrátiť späť posunutie príkazom `skoč`. Ak teraz spustíme hru, ľahko si overíme, že kapusta reaguje na upravené správanie. Využili sme polymorfický mechanizmus prekrývania metód (**overriding**). Odteraz sa všade tam, kde je použitá metóda `myšV` pre kapustu, bude volať nami upravená verzia metódy.

Podobne budeme postupovať pri prievozníkovi. Keďže ten je iba jeden, môžeme metódu `myšV` prekryť priamo v triede `Prievozník`. Principiálne je to v tomto prípade (pri prievozníkovi) jedno. Ale vo všeobecnosti:

- ak potrebujeme zmeniť správanie plošne, pre všetky inštancie určitej triedy, prekrývame v triede;
- ak potrebujeme úpravy vykonať individuálne, pre konkrétnu inštanciu (ako to bolo pri kapuste), prekrývame pri konštrukcii.

Po niekoľkých pokusoch sme zistili, že plochu prievozníka uspokojivo pokrýva elipsa posunutá od stredu doľava o 30 bodov, vysoká 90 bodov a široká 120 bodov:

```
@Override public boolean myšV()
{
    skoč(-30, 0);
    boolean myšV = myšVElipse(90, 120);
    skoč(30, 0);
    return myšV;
}
```

> *Podotýkame, že naše animované objekty sa pohybujú smerom „doprava“ a „doľava“, čiže pod uhlami 0° a 180°. Obidva smery sú oproti základnému smeru robota (čo je 90° – „hore“), pootočené o 90°, takže výška a šírka elipsy sú „vymenené“. Inak povedané: „výšku“ (inak povedané „veľkosť vedľajšej poloosi“) zadávame ako prvý argument a „šírku“ (inak povedané „veľkosť hlavnej poloosi“) ako druhý argument metód robota: `myšVElipse`, `vyplňElipsu`… (Pozri dokumentáciu skupiny tried `GRobot`.)*

Teraz môžeme kreslenie elipsy z metódy `kresliTvar` odstrániť. Obsluhu hry myšou realizujeme vytvorením obsluhy udalostí v konštruktore hlavnej triedy. Umiestnime do neho reakciu na kliknutie myšou a do neho umiestnime štyri jednoduché riadky riadiace animované objekty. Slovne vyjadrené – ak je kurzor myši v rámci konkrétneho objektu, nech sa pohne:

```
new ObsluhaUdalostí()
{
    @Override public void klik()
    {
        if (vlk.myšV()) vlk.choď();
        if (koza.myšV()) koza.choď();
        if (kapusta.myšV()) kapusta.choď();
        if (prievozník.myšV()) prievozník.prejdiRieku();
    }
};
```

Po spustení hry, budú všetky objekty reagovať na kliknutie ľubovoľným tlačidlom myši. Správanie hry zatiaľ nie je dokonalé. Objekty sa presúvajú niekedy nelogicky a niekedy sa pohnú viaceré naraz. To, aby sa nehýbali viaceré naraz, zabezpečíme jednoducho. Vytvoríme viacúrovňovú štruktúru `if-else`:

```
if (vlk.myšV()) vlk.choď();
else if (koza.myšV()) koza.choď();
else if (kapusta.myšV()) kapusta.choď();
else if (prievozník.myšV()) prievozník.prejdiRieku();
```

Po tej to úprave zareaguje na kliknutie myšou vždy len jeden objekt. Priorita je určená hierarchiou štruktúry `if-else`. Ďalej je potrebné zabezpečiť, aby sa objekty na palube plte hýbali súbežne s plťou a aby sa na palubu plte nemohli dostať viacerí pasažieri naraz. Najjednoduchšie riešenie je prekrytie metódy `pasivita` v triede `AnimovanýObjekt`, do ktorej vložíme jediný riadok kódu:

```
@Override public void pasivita()
{
    if (pozícia == Pozícia.prievozník) skočNa(prievozník);
}
```

Slovami vyjadrené: „ak je objekt na palube plte prievozníka, nech sa pohybuje súbežne s ním, resp. nech skočí na súradnice objektu prievozníka.“ Rýchle a účinné riešenie. Avšak keď potrebujeme využiť služby metódy `vPohybe`, ktorá môže spresňovať okolnosti detekcie pohybujúcich sa objektov, nemáme inú možnosť, než prekrytie metódy `pracuj`. Avšak, pri tom pozor! Ako je uvedené v dokumentácii robota: *„prekrytím tejto metódy by sme mohli úplne prepracovať správanie robota – aktívneho aj neaktívneho, avšak odporúčame ponechať predvolené správanie a iba korigovať správanie robotov prekrývaním metód aktivita a pasivita.“* Metóda `pracuj` úplne determinuje správanie robota. Preto, ak ju prekrývame, musíme v jej tele zabezpečiť volanie pôvodnej metódy `pracuj`, inak by sme stratili funkcionalitu robota! Tento spôsob bude vyzerať takto:

```
@Override public void pracuj()
{
    if (!vPohybe() && pozícia == Pozícia.prievozník)
    {
        skočNa(prievozník);
    }

    // Zavoláme verziu metódy „pracuj“ nadradenej triedy:
    super.pracuj();
}
```

Nikdy nezabudnite pri prekrývaní tejto metódy volať pôvodnú verziu metódy pracuj: `super.pracuj();`

Druhý naznačený problém – zabránenie vstúpenia viacerých pasažierov na palubu naraz bude vyžadovať hlbšie zamyslenie. Na to, aby sme na palubu plte zabránili vstupu ďalšieho pasažiera, musíme byť najskôr schopní zistiť, či sa už na palube niekto nachádza. Vyhotovme (pre svoje potreby) požiadavku na spracovanie novej vlastnosti triedy `Prievozník`, ktorá bude spravovať obsah paluby:

- **Umiestnenie:** trieda `Prievozník`;
- **Opis:** vytvorenie novej vlastnosti spravujúcej obsah paluby; nech metóda na zápis buď nedovolí vykonanie ďalších príkazov, ak paluba nie je prázdna, alebo zabezpečí uvoľnenie paluby;
- **Vstupy:** metóda na zápis bude prijímať objekt typu `Pasažier`;
- **Výstupy:** metóda alebo metódy na čítanie budú umožňovať najmenej jeden z nasledujúcich spôsobov detekcie obsahu paluby: 1. vrátenie hodnoty typu `boolean` vyjadrujúcej (ne)prázdnosť paluby; 2. vrátenie hodnoty typu `Pasažier`, ktorá je v prípade prázdnej paluby rovná hodnote `null`;
- **Zdôvodnenie:** hra vyžaduje, aby na palubu plte prievozníka mohol vstúpiť výhradne jeden pasažier; pomocou tejto vlastnosti ľahšie detegujeme, či je paluba voľná.

Požiadavku postupne naplníme. V prvom rade potrebujeme súkromnú premennú typu `Pasažier`. V nej budeme uchovávať aktuálneho pasažiera prítomného na palube. Ak bude jej hodnota rovná `null`, bude to znamenať, že paluba je prázdna. Na začiatku bude (samozrejme) paluba prázdna:

```
private Pasažier naPalube = null;
```

Premennú môžeme hneď využiť v metóde detegujúcej obsadenosť paluby. Jej realizácia bude jednoduchá. Stačí overiť, či je obsah premennej `naPalube` rovný `null`:

```
public boolean niektoNaPalube()
{
    return null != naPalube;
}
```

Uvoľnenie paluby bude tiež veľmi jednoduché. Stačí do premennej vložiť hodnotu `null`:

```
public void uvoľniPalubu()
{
    naPalube = null;
}
```

Kľúčové je pre nás nastupovanie pasažierov na palubu. Palubu môžeme obsadiť až po overení, či je prázdna. Prípadne lepšie: zabezpečíme, aby sa paluba pred vstupom ďalšieho pasažiera na palubu automaticky uvoľnila.

```
public void naPalubu(Pasažier ktorý)
{
    // <-- sem treba vložiť kód overujúci alebo zabezpečujúci „prázdnosť“ paluby
    naPalube = ktorý;
}
```

Keby sme chceli iba overiť, či je paluba prázdna a na základe toho ju (ne)naplniť, museli by sme vrhnúť výnimku, ktorú by sme museli zachytiť a ošetriť. Skúsme sa namiesto toho pozrieť na to, či nie sme schopní zabezpečiť uvoľnenie paluby (v prípade, že to bude situácia vyžadovať). Postupnou analýzou všetkých metód, ktoré máme k dispozícii, vyhľadáme tú, ktorá by mala vyhovieť našim požiadavkám. Metóda `choď` definovaná v triede `Pasažier`, sa správa takto:

- ak je pasažier na brehu, nastúpi na palubu

- a naopak, ak je na palube plte, vystúpi na najbližší breh.

Jednu z týchto akcií vykoná určite, bez obmedzujúcich podmienok. Nech ju voláme v ľubovoľnom čase. Čiže, keď metódu spustíme pre pasažiera prítomného na palube, určite palubu opustí:

```
if (null != naPalube) naPalube.choď();
```

Vďaka tomu, že si týmto správaním metódy môžeme byť istí, môžeme ho využiť v tele metódy `naPalubu`:

```
public void naPalubu(Pasažier ktorý)
{
    if (null != naPalube) naPalube.choď();
    naPalube = ktorý;
}
```

Predbežne budeme považovať okolie vlastnosti „práca s palubou“ za dokončené. Ak by sme potrebovali ďalšie metódy pracujúce v tomto kontexte, dokončíme ich neskôr. V tomto okamihu treba pouvažovať o tom, kde a ako tieto vlastnosti použijeme. Optimálne by bolo využiť ich vždy, keď pasažierovi (objektu) prikážeme, aby sa posunul. Keď vstupuje na palubu, treba overiť, či je plť prázdna, a keď z nej vystupuje, treba palubu uvoľniť.

Mohli by sme prekryť metódy animovaného objektu v triede `Pasažier` (konkrétne metódy `choďNaĽavý`, `choďNaPravý` a `choďNaPrievozníka`). Nič však nepokazíme ani tým, keď prekrývanie nevyužijeme a funkcionalitu doprogramujeme priamo do triedy `AnimovanýObjekt`. Prekrývanie je povinné v tých prípadoch, keď potrebujeme zároveň zachovať neporušenú pôvodnú funkcionalitu triedy a zároveň ju meniť v odvodených triedach.

Ak správanie uvedených troch metód upravíme, tak i keď je určené najmä pre pasažierov, nijako negatívne neovplyvní ani prievozníka. Do každej metódy umiestnime jediný podmienený príkaz. Je to bezpečné riešenie, ani výpočtová zložitosť sa príliš nezhorší. (Pri prekrytí metód by sme museli i tak volať nadradenú verziu v tele každej z nich, čo by sa násobilo počtom pasažierov. Takto budeme mať menej náročné podmienené spracovanie, ktoré bude nadbytočné iba pre jediný objekt – prievozníka.) V metódach `choďNaĽavý` a `choďNaPravý` potrebujeme zariadiť uvoľnenie paluby. To podmienime prítomnosťou postavy na palube:

```
public void choďNaĽavý()
{
    if (pozícia == Pozícia.prievozník)
        prievozník.uvoľniPalubu();

    cieľ(ľavýX, ľavýY);
    pozícia = Pozícia.ľavýBreh;
}

public void choďNaPravý()
{
    if (pozícia == Pozícia.prievozník)
        prievozník.uvoľniPalubu();

    cieľ(pravýX, pravýY);
    pozícia = Pozícia.pravýBreh;
}
```

Inak povedané: ak postava pri požiadavke presunutia sa na kotviacu pozíciu na brehu opúšťa palubu, automaticky uvoľňuje jej priestor, čo signalizuje volaním metódy `uvoľniPalubu`.

Naproti tomu v metóde `choďNaPrievozníka` potrebujeme zistiť, či je paluba voľná a následne do príslušnej premennej vložiť odkaz na aktuálneho pasažiera. Pri našom riešení obsah paluby automaticky vyprázdňujeme, ak treba. Pred chvíľou sme si pripravili metódu schopnú priestor paluby uvoľniť. Využijeme ju (môžeme to však vykonať len pre objekty, ktoré sú inštanciami triedy `Pasažier`):

```
public void choďNaPrievozníka()
{
    if (this instanceof Pasažier)
        prievozník().naPalubu((Pasažier)this);

    cieľ(cieľ = prievozník);
    pozícia = Pozícia.prievozník;
}
```

Teraz by sa nikdy nemali na palube vyskytnúť dvaja pasažieri naraz. Bolo by to v poriadku, keby bol program preložiteľný. Preklad programu sa zastaví v tele metódy `choďNaĽavý`. Chybové hlásenie pri príkaze `prievozník.uvoľniPalubu();` oznamuje, že metódu `uvoľniPalubu` nie je možné nájsť. Skutočne, metódu sme definovali v triede `Prievozník`, ale inštancia `prievozník` v triede `AnimovanýObjekt` je typu `AnimovanýObjekt`, nie `Prievozník`. V čase, keď sme prievozníka definovali, sme nemali na výber. Museli sme využiť jestvujúce triedy. Teraz, keď trieda `Prievozník` fyzicky jestvuje, môžeme zmeniť údajový typ prievozníka na taký, aký mu prináleží:

```
private static Prievozník prievozník;

public static void prievozník(Prievozník prievozník)
{
    AnimovanýObjekt.prievozník = prievozník;
}

public static Prievozník prievozník()
{
    return prievozník;
}
```

Po tejto zmene je program preložiteľný bez chýb a zdá sa, že funguje, ako má. Menší problém nastáva, keď náhodou pošleme na palubu plte dvoch pasažierov v rýchlom slede za sebou (tak, aby boli obaja ešte v pohybe). Prvý pasažier sa síce na chvíľu „zatvári“, že sa vracia späť na svoje miesto, no vzápätí sa obráti späť na palubu, a tam zostane stáť. Prievozník ho však na druhú stranu neprevezie. Zdá sa, že objekt „si len myslí“, že je na palube plte.

Je síce malá šanca, že hráč bude postupovať takto a ide iba o vizuálny problém, ale nemali by sme nič zanedbávať. Čo je koreňom problému? Je to rovnaký problém, aký sme pred časom riešili prekrytím metódy `dosiahnutieCieľa`. Metódy `choďNaĽavý` a `choďNaPravý` síce pošlú pasažiera na breh (implicitne predpokladajúc, že je na palube plte), ale nijako neoverujú ani nezabezpečia zrušenie sledovania vnútorného cieľa (pre prípad, že by bol pasažier ešte len na ceste na plť). Riešenie opäť nie je žiadnou veľkou záhadou. Stačí do oboch metód pridať riadok kódu, ktorým zrušíme vnútorný cieľ animovaného objektu:

```
cieľ = null;
```

Tým sme uzavreli kapitolu ovládania. Z hľadiska funkčnosti hry nám zostáva vyriešiť jediné: detegovať, situácie vzájomného požierania sa pasažierov a situáciu úspešného dokončenia hry. Z vizuálneho hľadiska chýba doriešenie vykresľovania grafiky prostredia. Oboma záležitosťami sa budeme zaoberať v nasledujúcej kapitole. Rovnako pridáme aj jednoduché ozvučenie postáv. Potrebné súbory sme si pripravili v úvodných kapitolách grafickej časti.

Príloha 11 – ďalšie verzie tried AnimovanýObjekt, Prievozník a HlavnáTrieda

AnimovanýObjekt

```
public class AnimovanýObjekt extends GRobot
{
    private static Prievozník prievozník;

    public static void prievozník(Prievozník prievozník)
    {
        AnimovanýObjekt.prievozník = prievozník;
    }

    public static Prievozník prievozník()
    {
        return prievozník;
    }

    // Fixný počet fáz animácie pri chôdzi.
    private final static int fázChôdze = 2;

    private Pozícia pozícia = Pozícia.pravýBreh;
    // Premenné zvuk, obrázokDoľava a obrázokDoprava budú obsahovať
    // názvy súborov podľa vnútorne dohodnutej schémy.
    private String meno, zvuk;
    private String obrázokDoľava[] = new String[1 + fázChôdze];
    private String obrázokDoprava[] = new String[1 + fázChôdze];
    private AnimovanýObjekt cieľ;
    private double posunX, posunY;
    private double ľavýX = 0, ľavýY = 0;
    private double pravýX = 0, pravýY = 0;
    private double posunPopisuX;
    private double posunPopisuY;
    private int fázaChôdze = 0; // Ak je hodnota nulová, znamená to,
                                // že objekt nie je animovaný.

    public AnimovanýObjekt(String meno, double posunX, double posunY,
        double posunPopisuX, double posunPopisuY)
    {
        zdvihniPero();

        // Inicializujem názvy súborov podľa dohody.
        obrázokDoľava[0] = "súbory/" + meno + "-ľ.png";
        obrázokDoprava[0] = "súbory/" + meno + "-p.png";
        zvuk = "súbory/" + meno + ".wav";

        this.meno = meno;
        this.posunX = posunX;
        this.posunY = posunY;
```

```
        this.posunPopisuX = posunPopisuX;
        this.posunPopisuY = posunPopisuY;

        spôsobKreslenia(KRESLI_NA_STRED);
        rýchlosť(10, false);
    }

    @Override public void kresliTvar()
    {
        skoč(posunX, posunY);

        if (vPohybe() && fázaChôdze != 0)
        {
            if (smer() >= 90 && smer() <= 270)
                obrázok(obrázokDoľava[fázaChôdze]);
            else
                obrázok(obrázokDoprava[fázaChôdze]);
        }
        else if (smer() >= 90 && smer() <= 270)
            obrázok(obrázokDoľava[0]); else obrázok(obrázokDoprava[0]);

        domov();
        if (myšV())
        {
            skoč(posunPopisuX, posunPopisuY);
            text(meno);
        }
    }

    public boolean vPohybe()
    {
        return aktívny();
    }

    public boolean myšV()
    {
        return myšVElipse(50, 60);
    }

    public boolean pravýBreh()
    {
        return pozícia == Pozícia.pravýBreh;
    }

    public boolean ľavýBreh()
    {
        return pozícia == Pozícia.ľavýBreh;
    }

    public Pozícia pozícia()
    {
        return pozícia;
    }

    public void nastavĽavý(double x, double y)
    {
        ľavýX = x;
```

```
        ľavýY = y;
    }

    public void nastavPravý(double x, double y)
    {
        pravýX = x;
        pravýY = y;
    }

    public void umožniAnimáciuChôdze()
    {
        for (int i = 1; i <= fázChôdze; ++i)
        {
            obrázokDoľava[i] = "súbory/" + meno + "-ľ-" + i + ".png";
            obrázokDoprava[i] = "súbory/" + meno + "-p-" + i + ".png";
        }
        fázaChôdze = 1;
    }

    public void choďNaĽavý()
    {
        if (pozícia == Pozícia.prievozník)
            prievozník.uvoľniPalubu();

        cieľ = null;
        cieľ(ľavýX, ľavýY);
        pozícia = Pozícia.ľavýBreh;
    }

    public void choďNaPravý()
    {
        if (pozícia == Pozícia.prievozník)
            prievozník.uvoľniPalubu();

        cieľ = null;
        cieľ(pravýX, pravýY);
        pozícia = Pozícia.pravýBreh;
    }

    public void choďNaPrievozníka()
    {
        if (this instanceof Pasažier)
            prievozník().naPalubu((Pasažier)this);

        cieľ(cieľ = prievozník);
        pozícia = Pozícia.prievozník;
    }

    @Override public void dosiahnutieCieľa()
    {
        cieľ = null;
    }

    @Override public void aktivita()
    {
        if (null != cieľ) upravCieľ(cieľ);
        // Nenulová hodnota fázy animácie znamená, že sme vykonali potrebné
```

```
        // nastavenia, aby mohol byť objekt animovaný.
        if (fázaChôdze != 0 && ++fázaChôdze > fázChôdze) fázaChôdze = 1;
    }

    // Jednoduchšie riešenie:
    // @Override public void pasivita()
    // { if (pozícia == Pozícia.prievozník) skočNa(prievozník); }

    @Override public void pracuj()
    {
        if (!vPohybe() && pozícia == Pozícia.prievozník)
        {
            skočNa(prievozník);
        }

        // Zavoláme verziu metódy „pracuj“ nadradenej triedy:
        super.pracuj();
    }

    // Urobíme výnimku a publikujeme jednu metódu predčasne. Toto bude jediná
    // zmena, ktorú vykonáme v tejto triede v nasledujúcej kapitole.
    public void vydajZvuk()
    {
        svet.zvuk(zvuk);
    }
}
```

Prievozník

```
public class Prievozník extends AnimovanýObjekt
{
    private final static int rozptylBrehov = 90;
    private final static int posunGrafikyX = -30;
    private final static int posunGrafikyY = 10;
    private final static int posunPopisuX = -30;
    private final static int posunPopisuY = 115;

    public Prievozník()
    {
        super("prievozník", posunGrafikyX, posunGrafikyY,
            posunPopisuX, posunPopisuY);

        domov(50, -25, 0);    // domovská pozícia [x, y] a domovský uhol
        zrýchlenie(2, false);
        nastavĽavý(domaX() - rozptylBrehov, domaY());
        nastavPravý(domaX() + rozptylBrehov, domaY());

        prievozník(this);
    }

    public void prejdiRieku()
    {
        if (pravýBreh())
            choďNaĽavý();
        else
            choďNaPravý();
    }
```

```
    @Override public boolean myšV()
    {
        skoč(-30, 0);
        boolean myšV = myšVElipse(90, 120);
        skoč(30, 0);
        return myšV;
    }


    // --- Vlastnosť „na palube“ spravujúca obsah paluby ---

    private Pasažier naPalube = null;

    public boolean niektoNaPalube()
    {
        return null != naPalube;
    }

    public void uvoľniPalubu()
    {
        naPalube = null;
    }

    public void naPalubu(Pasažier ktorý)
    {
        if (null != naPalube) naPalube.choď();
        naPalube = ktorý;
    }

}
```

HlavnáTrieda

```
public class HlavnáTrieda extends GRobot
{
    private final Prievozník prievozník = new Prievozník();
    private final Pasažier vlk = new Pasažier("vlk", 0, 0, 0, 65, 200, 10);
    private final Pasažier koza = new Pasažier("koza", 0, -10, -5, 50, 280, 10);
    private final Pasažier kapusta = new Pasažier("kapusta", 0, -25, 0, 25, 360, 10)
    {
        @Override public boolean myšV()
        {
            skoč(0, -30);
            boolean myšV = myšVKruhu(90, 120);
            skoč(0, 30);
            return myšV;
        }
    };

    private HlavnáTrieda()
    {
        super(800, 300, "Prievozník…");
        svet.zbaľ();

        prievozník.choďNaPravý();
        vlk.choďNaPravý();
```

```
        koza.choďNaPravý();
        kapusta.choďNaPravý();

        new ObsluhaUdalostí()
        {
            @Override public void klik()
            {
                if (vlk.myšV()) vlk.choď();
                else if (koza.myšV()) koza.choď();
                else if (kapusta.myšV()) kapusta.choď();
                else if (prievozník.myšV()) prievozník.prejdiRieku();
            }
        };

        aktivuj();
    }

    public static void main(String[] args)
    {
        new HlavnáTrieda();
    }
}
```

Grafický projekt 5

V predchádzajúcej kapitole sme dokončili ovládanie hry. V tejto kapitole sa budeme zaoberať tromi záležitosťami: 1. úspešným a neúspešným dokončením hry, 2. vykreslením grafiky prostredia a 3. jednoduchému ozvučeniu postáv.

Prvou zo spomenutých záležitostí sme sa v rámci tohto materiálu už raz zaoberali. V textovom projekte sme riešili rovnaký problém. Ak si spomeniete, v metódach `niektoNiekohoZožral` a `hraSkončila` textového projektu sme porovnávali prítomnosť postáv na jednom či druhom brehu. Spôsob riešenia bude čiastočne rovnaký i v grafickej verzii projektu. To, na ktorom z brehov sa pasažier práve nachádza, vieme určiť ľahko. Trieda `AnimovanýObjekt` má na to definované potrebné metódy. Stačí ich vhodne použiť.

> ***Pozor!*** *Dokončenie hry neznamená ukončenie aplikácie. Také riešenie by som nepovažoval sa správne! (Už som sa s niečím takým stretol…)*

Najjednoduchšie bude zaradiť kontrolu do prekrytej metódy `aktivita` v hlavnej triede. Začnime detekciou dokončenia hry, pretože podmienka na úspešné dokončenie hry je jednoduchšia. Všetci pasažieri musia byť na ľavom brehu:

```
@Override public void aktivita()
{
    if (vlk.ľavýBreh() && koza.ľavýBreh() && kapusta.ľavýBreh())
    {
        deaktivuj();
        svet.správa("Všetci pasažieri sú úspešne prevezení na ľavý breh!");
    }
}
```

Dokončenie sme zariadili jednoducho, deaktivovali sme hlavného robota a vypísali sme správu oznamujúcu úspech. Skúsme previezť všetkých pasažierov na druhý breh. Môžeme v ľubovoľnom poradí. Hra oznámi hráčov úspech trochu predčasne: hneď, ako prikážeme poslednému pasažierovi, aby vystúpil. Nepôsobí to prirodzene. Zariaďme to preto tak, aby kontrola (ne)úspešnosti hráčovho počínania si prebiehala len vtedy,

ak nie je žiadny pasažier v pohybe. Nasledujúci riadok kódu umiestnený na začiatku metódy `aktivita` to zabezpečí:

```
if (vlk.vPohybe() || koza.vPohybe() || kapusta.vPohybe()) return;
```

Možno by bolo vhodné pridať do podmienky aj prievozníka… (To nechám na posúdenie čitateľa.)

Keď pasažierov prevezieme na ľavý breh teraz, správa sa zobrazí až po zastavení posledného z nich. Hlavný robot je deaktivovaný, preto prestane fungovať mechanizmus skrývania a zobrazovania popisov postavičiek, ale blokovanie ich ovládania sme zatiaľ nijako nezabezpečili. Ak klikneme na hociktorú postavu (vrátane prievozníka), pohne sa. Otázka znie: kam umiestniť vhodný kód na zablokovanie ovládania postáv?

Postavy ovládame myšou prostredníctvom reakcie `klik` v obsluhe udalostí. Deaktivovanie ktoréhokoľvek z robotov nemá žiadny vplyv na to, či sa spustí alebo nespustí obsluha udalosti kliknutia myšou. Taktiež nemáme možnosť priameho blokovania obsluhy udalostí. Jediný spôsob je umiestniť na začiatok reakcie (reakcie `klik`) obmedzujúcu podmienku, ktorou ju „odstavíme“. Zamyslime sa nad tým, čo bude najvhodnejšie využiť na tento účel. V podstate potrebujeme, aby sa ovládanie zablokovalo **«po»** dokončení hry. Dokončenie hry sa snažíme signalizovať hlavnému robotovi jeho deaktivovaním. Presne tento stav – stav aktivity hlavného robota – môžeme využiť na odstávku reakcie na kliknutie. Ak je hlavný robot neaktívny, nech sa reakcia obsluhy udalostí nevykoná:

```
if (neaktívny()) return;
```

Podobne zabezpečíme problém neúspechu hráča. Pozrime sa, ako sme to riešili v textovej verzii projektu:

```
private boolean niektoNiekohoZožral()
{
    boolean vlkNaĽavom = jeNaĽavomBrehu("vlk");
    boolean kozaNaĽavom = jeNaĽavomBrehu("koza");
    boolean kapustaNaĽavom = jeNaĽavomBrehu("kapusta");

    if ((vlkNaĽavom && kozaNaĽavom &&
        !kapustaNaĽavom && prievozníkJeNaPravomBrehu) ||
        (!vlkNaĽavom && !kozaNaĽavom &&
        kapustaNaĽavom && !prievozníkJeNaPravomBrehu))
    {
        System.out.println("Vlk zožral kozu!");
        return true;
    }

    if ((kozaNaĽavom && kapustaNaĽavom &&
        !vlkNaĽavom && prievozníkJeNaPravomBrehu) ||
        (!kozaNaĽavom && !kapustaNaĽavom &&
        vlkNaĽavom && !prievozníkJeNaPravomBrehu))
    {
        System.out.println("Koza zožrala kapustu!");
        return true;
    }

    return false;
}
```

Skúsme využiť rovnaké podmienky, iba prepíšme premenné na vlastnosti objektov:

```
if ((vlk.ľavýBreh() && koza.ľavýBreh() &&
```

```
    !kapusta.ľavýBreh() && prievozník.pravýBreh()) ||
    (!vlk.ľavýBreh() && !koza.ľavýBreh() &&
    kapusta.ľavýBreh() && !prievozník.pravýBreh()))
{
    deaktivuj();
    svet.správa("Vlk zožral kozu!");
}

if ((koza.ľavýBreh() && kapusta.ľavýBreh() &&
    !vlk.ľavýBreh() && prievozník.pravýBreh()) ||
    (!koza.ľavýBreh() && !kapusta.ľavýBreh() &&
    vlk.ľavýBreh() && !prievozník.pravýBreh()))
{
    deaktivuj();
    svet.správa ("Koza zožrala kapustu!");
}
```

Kód umiestnime do metódy aktivita hlavného robota a hru spustíme. Nespráva sa korektne. Dôvodom je tretí stav určujúci prítomnosť postavy na palube plte. Ten situáciu značne odlišuje od tej, akú sme mali v textovej verzii. Pri nej sme rozlišovali iba dva stavy: prítomnosť na ľavom alebo pravom brehu rieky. Zamyslime sa nad situáciou z iného pohľadu. Keď sa pozrieme na tabuľku stavov hry prevzatú z kapitoly textový projekt 3:

Vlk	Koza	Kapusta	Prievozník	Komentáre
pravý	pravý	pravý	pravý	*[počiatočný stav]*
pravý	pravý	pravý	ľavý	pasažieri sa strážia navzájom
pravý	pravý	ľavý	pravý	kozu stráži prievozník
pravý	**pravý**	ľavý	ľavý	**vlk zožral kozu**
pravý	ľavý	pravý	pravý	nikto nikoho neohrozuje
pravý	ľavý	pravý	ľavý	nikto nikoho neohrozuje
pravý	**ľavý**	**ľavý**	pravý	**koza zožrala kapustu**
pravý	ľavý	ľavý	ľavý	kapustu stráži prievozník
ľavý	pravý	pravý	pravý	kapustu stráži prievozník
ľavý	**pravý**	**pravý**	ľavý	**koza zožrala kapustu**
ľavý	pravý	ľavý	pravý	nikto nikoho neohrozuje
ľavý	pravý	ľavý	ľavý	nikto nikoho neohrozuje
ľavý	**ľavý**	pravý	pravý	**vlk zožral kozu**
ľavý	ľavý	pravý	ľavý	kozu stráži prievozník
ľavý	ľavý	ľavý	pravý	*alternatívny koniec hry*
ľavý	ľavý	ľavý	ľavý	*[koniec hry]*

a zároveň sa nad situáciou zamyslíme z praktického hľadiska, nájdeme riešenie problému hľadaním odpovede na otázku: „Čo majú spoločné tie stavy hry, ktoré nás v tomto okamihu zaujímajú, čiže situácie znamenajúce hráčov neúspech?“

Pri neúspechu sú vždy kritickí pasažieri na spoločnom brehu (ich stavy sa rovnajú), pričom ostatné dve postavy hry musia byť buď na opačnom brehu rieky, alebo na plti (prevzatá tabuľka síce s plťou pre jednoduchosť nepočíta, ale my v skrytosti budeme). Pripomeňme, že máme dve dvojice kritických pasažierov a naraz môžeme kontrolovať len jednu z nich. Vyberme si jednu kritickú dvojicu (vlk a koza alebo koza a kapusta) a vytvorme podmienku.

- Prvá časť podmienky striehnucej na hráčovo pochybenie vzájomne porovná stavy oboch postáv kritickej dvojice (či sa rovnajú).
- Ďalšie dve časti podmienky spojenej operátorom „a súčasne“ budú porovnávať stavy ostatných dvoch postáv so stavom ľubovoľnej postavy z kritickej dvojice (či sú rôzne).

(Preto môžeme porovnávať ostatné postavy s ľubovoľnou postavou z kritickej dvojice, lebo ak má útočník svoju obeť zožrať, musia byť obaja – obeť i útočník – na rovnakom brehu, čiže je jedno, s ktorou z nich ďalšiu postavu porovnávame.)

Stav každej postavy dokážeme zistiť pomocou metódy `pozícia` z triedy `AnimovanýObjekt`. Pre kritickú dvojicu vlka a kozy bude kód overujúci hráčove pochybenie vyzerať takto:

```
if (vlk.pozícia() == koza.pozícia() &&
    kapusta.pozícia() != vlk.pozícia() &&
    prievozník.pozícia() != vlk.pozícia())
{
    deaktivuj();
    svet.správa("Vlk zožral kozu!");
}
```

Rovnaký manéver vykonáme pre druhú kritickú dvojicu:

```
if (koza.pozícia() == kapusta.pozícia() &&
    vlk.pozícia() != koza.pozícia() &&
    prievozník.pozícia() != kapusta.pozícia())
{
    deaktivuj();
    svet.správa ("Koza zožrala kapustu!");
}
```

Otestujme správanie sa hry. Hlásenie o neúspechu je zobrazené okamžite po zmene stavu pltníka (jeho vyštartovaní na opačný breh). Riešenie som už naznačoval – nájsť ho bude vašou úlohou.

V poradovníku sa ocitlo dokončenie ďalšej záležitosti – grafiky prostredia. Na nakreslenie prostredia využijeme hlavého robota. V hlavnej triede naprogramujeme súkromnú metódu, do ktorej budeme umiestňovať všetky príkazy kreslenia. V prvom rade skryjeme hlavného robota:

```
private void nakresliProstredie()
{
    skry();
}
```

Volanie metódy nezabudneme umiestniť na záver konštruktora. Príkaz môžeme umiestniť tesne pred aktiváciu hlavného robota. Najskôr nakreslíme oblohu a vodu. Oblohu nakreslíme vyplnením podlahy („podlahou“ máme na mysli názov jedného z kresliacich plátien vo svete robota – pozri dokumentáciu

skupiny tried `GRobot`) tyrkysovou farbou a na nakreslenie vody použijeme obdĺžnik vyplnený modrou farbou:

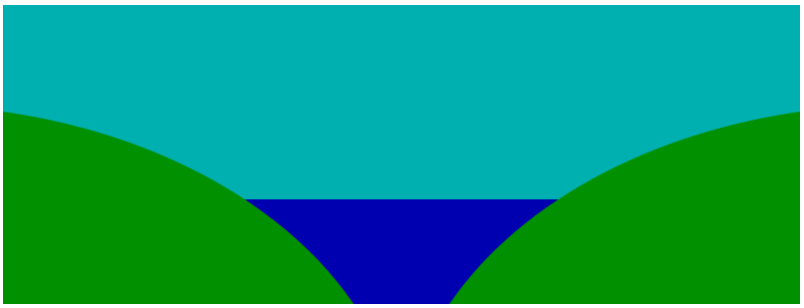
```
// Obloha
podlaha.vyplň(tyrkysová);

// Voda
skočNa(0, -100);
farba(modrá);
vyplňObdĺžnik(405, 55);
```

Výsledok bude jednoduchý:

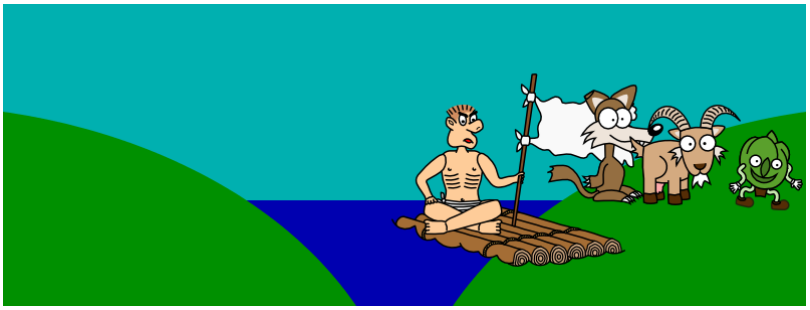
Prvý nápad na nakreslenie brehov je použiť dve tmavozelené elipsy:

```
// Brehy
skočNa(-500, -300);
farba(tmavozelená);
vyplňElipsu(500, 350);

skočNa(500, -300);
vyplňElipsu(500, 350);
```

Výsledok nie je ideálny:

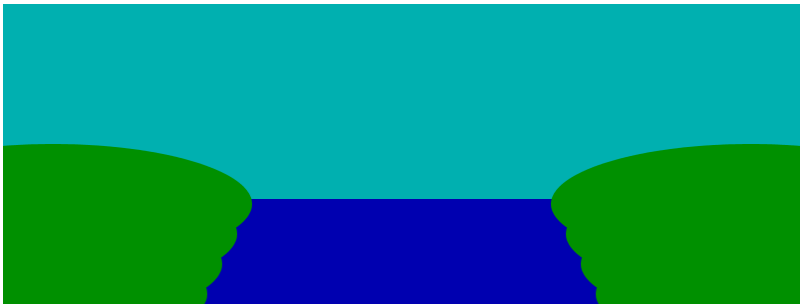


Po pridaní grafických objektov postáv, pôsobí tento tvar brehov ako z inej perspektívy:



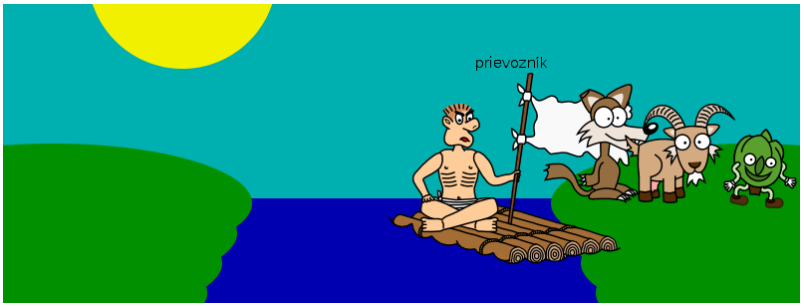
Nie je to prirodzené. Nepožadujeme dokonalosť, ale skúsme aspoň naznačiť tok rieky rozširujúci sa smerom k nám. Použijeme niekoľko elíps nakreslených pod seba so zväčšujúcou sa vzdialenosťou od stredu. Ak chceme využiť relatívny posun robota, musíme nakresliť každý breh v samostatnom cykle:

```
// Ľavý breh
farba(tmavozelená);
skočNa(-350, -50);

for (int i = 0; i < 5; ++i)
{
    vyplňElipsu(200, 60);
    choď(-15, -30);
}

// Pravý breh
skočNa(350, -50);

for (int i = 0; i < 5; ++i)
{
    vyplňElipsu(200, 60);
    choď(15, -30);
}
```



Nakoniec dokreslime slnko a prostredie máme hotové:

```
// Slnko
farba(svetložltá);
skočNa(-220, 180);
kruh(95);
```



Po grafickej stránke je hra v podstate dokončená. V tomto stave ide už iba o dolaďovanie detailov. Doteraz sa všetko dotýkalo grafiky a ovládania. Hráči istotne ocenia spestrenie hry zvukom. Hoci veľmi jednoduchým. Takmer všetko potrebné na jednoduché ozvučenie už máme pripravené. Do triedy `AnimovanýObjekt` ešte doprogramujme jednoduchú metódu:

```
public void vydajZvuk()
{
```

```
    svet.zvuk(zvuk);
}
```

Využijeme ju na to, aby postavička vydala zvuk vždy vtedy, keď na ňu hráč klikne, v dôsledku čoho sa pohne. Keby sme chceli, aby postavička vydala zvuk pri každom pohybe, umiestnili by sme volania metódy `vydajZvuk` na do tela každej z metód `choďNaĽavý`, `choďNaPravý` a `choďNaPrievozníka`. Ibaže to si neželáme. Preto volania umiestnime do obsluhy udalostí v hlavnej triede – do reakcie `klik`:

```
if (vlk.myšV())
{
    vlk.choď();
    vlk.vydajZvuk();
}
else if (koza.myšV())
{
    koza.choď();
    koza.vydajZvuk();
}
else if (kapusta.myšV())
{
    kapusta.choď();
    kapusta.vydajZvuk();
}
else if (prievozník.myšV())
{
    prievozník.prejdiRieku();
    prievozník.vydajZvuk();
}
```

Principiálne môžeme považovať hru za dokončenú. Zlepšovanie detailov, či už grafických alebo funkčných, ponecháme na čitateľovi. Z tých funkčných môžeme niektoré zlepšenia naznačiť: počiatočné umiestnenie postáv (teraz všetky dokráčajú na počiatočné miesto zo stredu plátna), pridanie ponuky (s možnosťami a predvoľbami, napríklad s možnosťou opätovného spustenia hry…), zmiznutie alebo animovanie zožratej postavy a podobne. V súlade s tým by sme mohli využiť nadobudnuté poznatky a definovať, napríklad, novú vymenovaciu triedu na účely rozlišovania rôznych stavov hry:

```
public enum Stav
{
    prebieha, hotovo, jemKapustu, jemKozu, kapustaZjedená, kozaZjedená
}
```

Môžeme definovať ďalšie grafické objekty, ale to už skutočne ponecháme na čitateľa.

Veríme, že ste sa pri tomto sprievodcovi veľa naučili. Tešíme sa na ďalšie programovacie stretnutie pri podobnom materiáli v budúcnosti…

Príloha 12 – posledná verzia triedy HlavnáTrieda

HlavnáTrieda

```
public class HlavnáTrieda extends GRobot
{
    private final Prievozník prievozník = new Prievozník();
    private final Pasažier vlk = new Pasažier("vlk", 0, 0, 0, 65, 200, 10);
    private final Pasažier koza = new Pasažier("koza", 0, -10, -5, 50, 280, 10);
```

```
    private final Pasažier kapusta = new Pasažier("kapusta", 0, -25, 0, 25, 360, 10)
    {
        @Override public boolean myšV()
        {
            skoč(0, -30);
            boolean myšV = myšVKruhu(90, 120);
            skoč(0, 30);
            return myšV;
        }
    };

    private HlavnáTrieda()
    {
        super(800, 300, "Prievozník…");
        svet.zbaľ();

        prievozník.choďNaPravý();
        vlk.choďNaPravý();
        koza.choďNaPravý();
        kapusta.choďNaPravý();

        new ObsluhaUdalostí()
        {
            @Override public void klik()
            {
                if (vlk.myšV())
                {
                    vlk.choď();
                    vlk.vydajZvuk();
                }
                else if (koza.myšV())
                {
                    koza.choď();
                    koza.vydajZvuk();
                }
                else if (kapusta.myšV())
                {
                    kapusta.choď();
                    kapusta.vydajZvuk();
                }
                else if (prievozník.myšV())
                {
                    prievozník.prejdiRieku();
                    prievozník.vydajZvuk();
                }
            }
        };

        nakresliProstredie();
        aktivuj();
    }

    @Override public void aktivita()
    {
        if (vlk.vPohybe() || koza.vPohybe() || kapusta.vPohybe() ||
            prievozník.vPohybe()) return;
```

```
        if (vlk.ľavýBreh() && koza.ľavýBreh() && kapusta.ľavýBreh())
        {
            deaktivuj();
            svet.správa("Všetci pasažieri sú úspešne prevezení na ľavý breh!");
        }

        if (vlk.pozícia() == koza.pozícia() &&
            kapusta.pozícia() != vlk.pozícia() &&
            prievozník.pozícia() != vlk.pozícia())
        {
            deaktivuj();
            svet.správa("Vlk zožral kozu!");
        }

        if (koza.pozícia() == kapusta.pozícia() &&
            vlk.pozícia() != koza.pozícia() &&
            prievozník.pozícia() != kapusta.pozícia())
        {
            deaktivuj();
            svet.správa ("Koza zožrala kapustu!");
        }
    }

    private void nakresliProstredie()
    {
        skry();

        // Obloha
        podlaha.vyplň(tyrkysová);

        // Voda
        skočNa(0, -100);
        farba(modrá);
        vyplňObdĺžnik(405, 55);

        // Ľavý breh
        farba(tmavozelená);
        skočNa(-350, -50);

        for (int i = 0; i < 5; ++i)
        {
            vyplňElipsu(200, 60);
            choď(-15, -30);
        }

        // Pravý breh
        skočNa(350, -50);

        for (int i = 0; i < 5; ++i)
        {
            vyplňElipsu(200, 60);
            choď(15, -30);
        }

        // Slnko
        farba(svetložltá);
```

```
        skočNa(-220, 180);
        kruh(95);
    }

    public static void main(String[] args)
    {
        new HlavnáTrieda();
    }
}
```