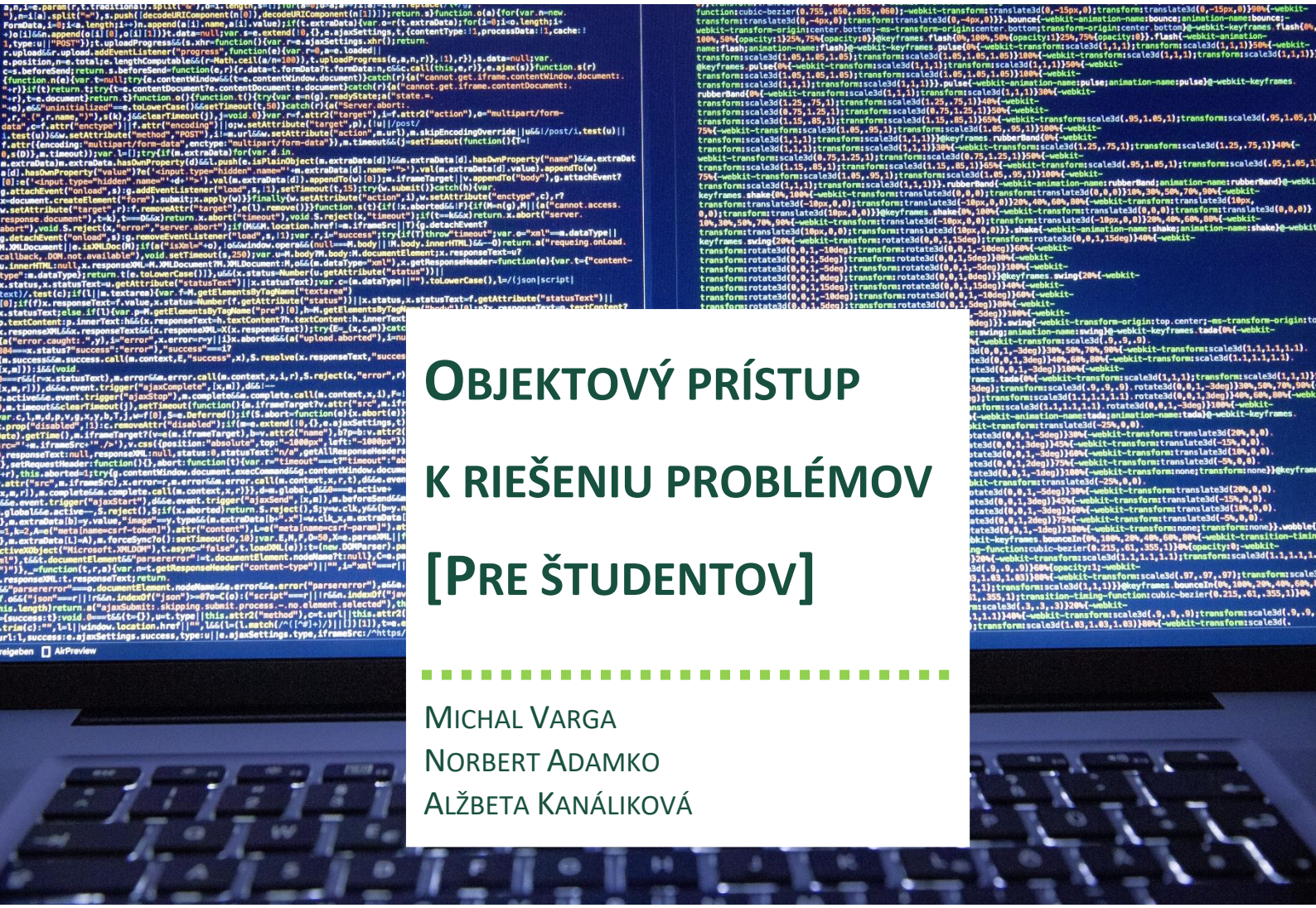


OBJEKTOVÝ PRÍSTUP K RIEŠENIU PROBLÉMOV [PRE ŠTUDENTOV]

MICHAL VARGA
NORBERT ADAMKO
ALŽBETA KANÁLIKOVÁ

EURÓPSKA ÚNIA
Európsky sociálny fond
Európsky fond regionálneho rozvoja

OPERAČNÝ PROGRAM ĽUDSKÉ ZDROJE

MINISTERSTVO ŠKOLSTVA, VEDY, VÝSKUMU A ŠPORTU SLOVENSKEJ REPUBLIKY

it akadémia

Tento projekt sa realizuje vďaka podpore z Európskeho sociálneho fondu a Európskeho fondu regionálneho rozvoja v rámci Operačného programu Ľudské zdroje

www.minedu.sk www.employment.gov.sk/sk/esf/ www.itakademia.sk

Objektový prístup k riešeniu problémov

Spracované v rámci národného projektu IT Akadémia – vzdelávanie pre 21. storočie

Bratislava, 2020

Objektový prístup k riešeniu problémov [pre študentov]
Spracované s finančnou podporou národného projektu IT Akadémia – vzdelávanie pre 21. storočie

Autori: Michal Varga, Norbert Adamko, Alžbeta Kanáliková

Recenzenti: Jana Jurinová, Peter Tomcsányi, Štefan Bocko

Neprešlo jazykovou úpravou.

Vydavateľ: Centrum vedecko-technických informácií SR, Bratislava
Rok vydania: 2020
Vydanie: 1.
ISBN: 978-80-89965-66-3 (verzia pre učiteľov)
EAN: 9788089965663 (verzia pre učiteľov)

Tento projekt sa realizuje vďaka podpore z Európskeho sociálneho fondu v rámci Operačného programu Ľudské zdroje.

OBSAH

1 ÚVOD DO PROSTREDIA GREENFOOT

KĽÚČOVÉ SLOVÁ

Objekt. Trieda. Inštancia. Konštruktor. Metóda. Svet. Aktor.

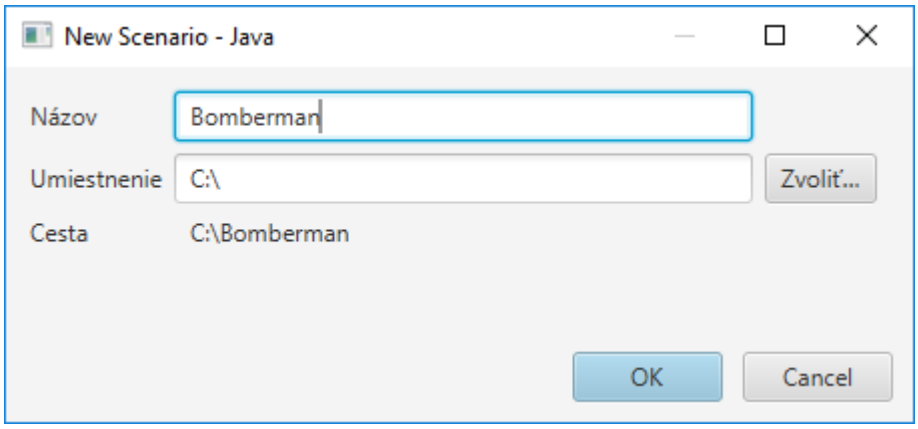
CIELE

Cieľom prvej kapitoly je poznať základné princípy objektovo orientovaného programovania a naučiť sa orientovať v prostredí Greenfoot [1]. Naučíme sa, ako je prostredie rozložené, ako je možné spustiť aplikáciu a kde sa edituje zdrojový kód. Preskúmame možnosti, ktoré prostredie Grenfoot ponúka na prácu s objektmi. Naučíme sa vytvoriť vlastné triedy a meniť ich grafické reprezentácie.

OBSAH

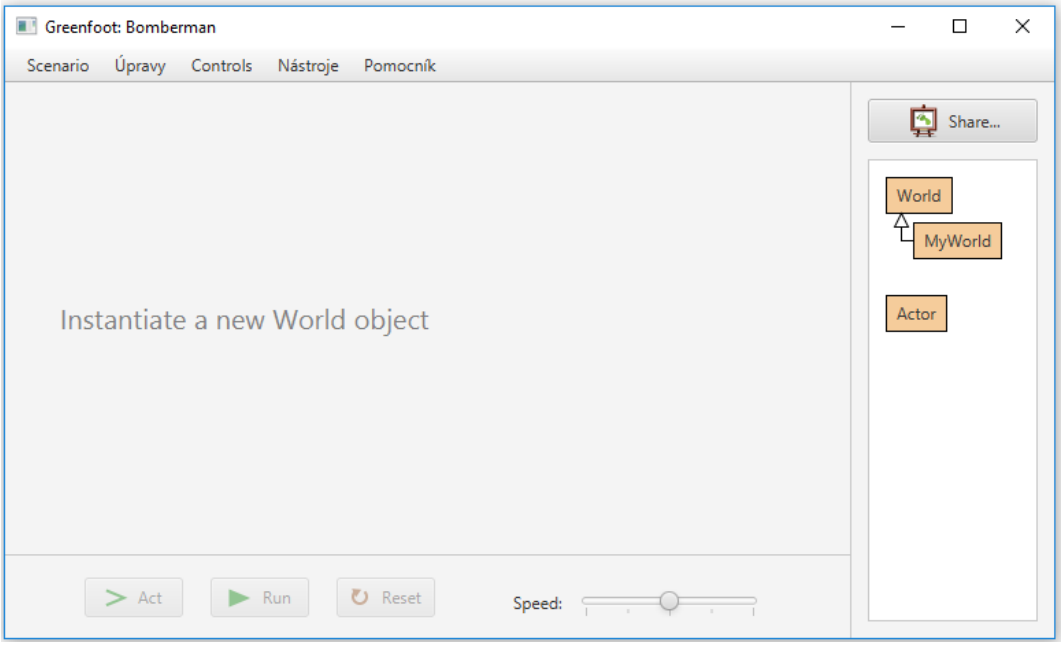
1.1 Prostredie Greenfoot

Aby sme mohli začať pracovať, musíme najprv vytvoriť náš nový projekt. Po spustení nástroja Greenfoot vyberieme v menu **`Scenario`** položku **`New Java Scenario...`**. Po jej zvolení vidíme dialóg, ktorý je zobrazený na obrázku 1.1. Ako názov projektu vypíšeme *Bomberman*. Následne si zvolíme zložku, do ktorej sa náš projekt uloží (napr. do zložky Dokumenty).



Obrázok 1.1: Vytvorenie nového projektu

Každý projekt v prostredí Greenfoot sa odohráva vo svete (po anglicky **`World`**). Všetko, čo sa vo svete pohybuje sa nazýva aktor (po anglicky **`Actor`**, čo môžeme vyložiť ako herec alebo aktívny účastník). Svet si môžeme predstaviť ako javisko, po ktorom sa pohybujú a hrajú herci podľa pravidiel, ktoré naprogramujeme. Projekt bez hercov a s jednoduchým javiskom (teda bez aktorov a s jediným svetom) vidíme na obrázku 1.2.



Obrázok 1.2: Prostredie Greenfoot s prázdnym projektom

Základným konceptom, s ktorým budeme počas celej doby tvorby projektu pracovať, je **objektovo orientované programovanie**. Znamená to, že budeme programovať objekty a využívať ich vlastnosti, funkcie a vzájomnú spoluprácu na to, aby sme vytvorili funkčný systém.

1.2 Objekty a triedy

Základným stavebným kameňom objektovo orientovaného programovania je **objekt**. Zamyslime sa, čo je to objekt v reálnom svete. Jednoduchá odpoveď by mohla byť, že je to akýkoľvek predmet (osoba alebo vec). Podobne objekt definuje aj Slovník slovenského jazyka [2]. Neskôr však spoznáme, že z pohľadu programovania môžu byť objektmi aj veci, ktoré nie sú hmotné (nemôžeme sa ich dotknúť). Každý objekt však môže mať svoje vlastnosti a určité správanie (môže vedieť vykonať isté činnosti, napríklad reagovať na podnety z okolia). Objektom môže byť napríklad človek. Jeho vlastnosti môžu byť vek, výška, váha, vzdelanie, ale napríklad aj to, kto je jeho otec, mama či súrodenci. Aktuálne hodnoty týchto vlastností určujú stav objektu. Človek tiež dokáže konať mnoho vecí – rozprávať, chodiť, jesť, programovať.

ÚLOHA 1.1

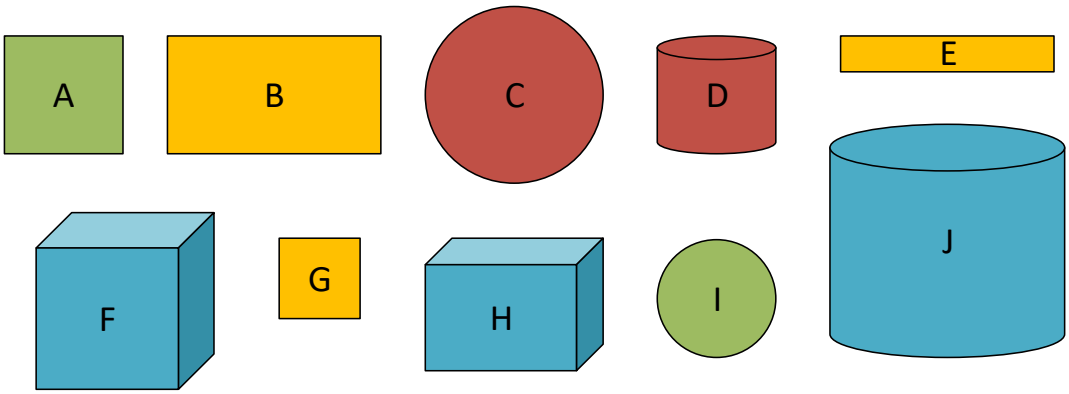
Identifikujte vo svojom okolí objekty a vypíšte ich vlastnosti a činnosti, ktoré vedia oni sami vykonať. Dokážete identifikovať objekty, ktoré nemajú žiadne vlastnosti? Dokážete identifikovať objekty, ktoré nedokážu nič vykonať? Dokážete identifikovať nehmotné objekty (také, ktorých sa nevieme fyzicky dotknúť)?

Uveďme niekoľko jednoduchých príkladov:

Televízor má vlastnosti výška, šírka, hĺbka, hmotnosť, uhlopriečka, rozlíšenie, počet a typ portov, atď. Televízor môžeme zapnúť, vypnúť, môžeme prepnúť stanicu či vstup, vie ladiť stanice.

Za objekt bez vlastností môžeme považovať jednoduchú softvérovú kalkulačku – dokáže vykonávať matematické operácie s operátormi, ktoré zadáme, avšak sama nie je charakterizovaná žiadnou vlastnosťou.
Objekt, ktorý nedokáže nič vykonať je napr. pohár. Je vyrobený z nejakého materiálu, má objem, farbu, ale sám nič nerobí.
Nehmotný objekt je napr. čas. Nedokážeme sa ho dotknúť, ale vieme určiť jeho vlastnosti (hodiny, minúty, sekundy) a aj operácie (môžeme napr. pridať sekundu).

Na nasledujúcom obrázku vidíme niekoľko geometrických útvarov. Identifikujme základné vlastnosti jednotlivých útvarov a skúsme zoskupiť útvary podľa ich spoločných vlastností.



Obrázok 1.3: Geometrické útvary

Identifikujeme nasledujúce typy útvarov a ich vlastnosti:

1) **Obdĺžnik** (zelený štvorec A, žltý štvorec G a žlté obdĺžniky B a E) je charakterizovaný farbou, dĺžkou a výškou.
2) **Kruh** (červený kruh C a zelený kruh I) je charakterizovaný farbou a polomerom.
3) **Kváder** (modré kvádre F a H) je charakterizovaný farbou, dĺžkou, šírkou a výškou.
4) **Valec** (červený valec D a modrý valec J) je charakterizovaný farbou, polomerom a výškou.

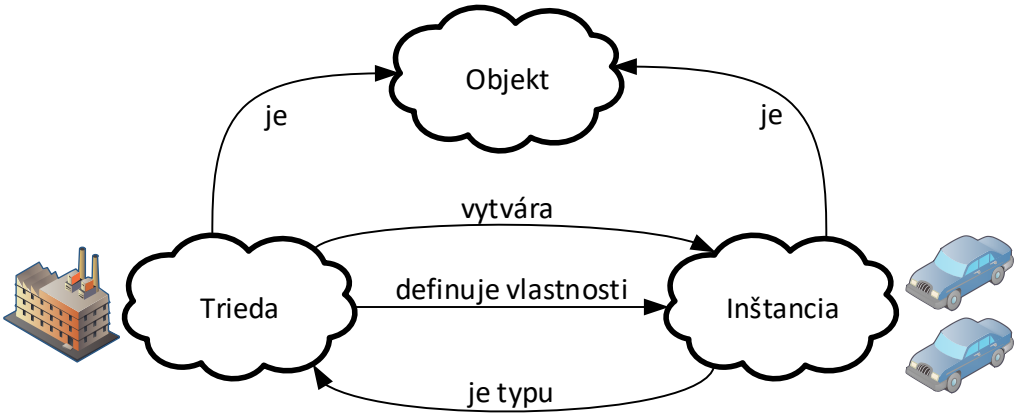
Vidíme, že napriek tomu, že geometrické objekty vyzerajú rozlične, dokážeme im priradiť ich spoločnú príslušnosť – geometrický tvar a aj spoločné vlastnosti. To, čím sa objekty navzájom líšia je hodnota jednotlivých vlastností.

Základným konceptom objektovo orientovaného programovania je objekt, ktorý môže reprezentovať čokoľvek. V objektovo orientovanom programovaní rozlišujeme dva druhy objektov - **triedy** a **inštancie**.

Trieda definuje spoločné vlastnosti **inštancií** tej istej triedy (napr. trieda `Kruh` definuje, že všetky inštancie triedy `Kruh` budú mať vlastnosti `farba` a `polomer`. Každá inštancia však môže mať inú farbu a iný polomer.).

Inštancia je „skutočný objekt“ danej triedy. Triedu môžeme považovať za továreň svojich inštancií, pričom počet inštancií, ktoré z triedy vytvoríme, nie je obmedzený (napr. môžeme mať viac inštancií triedy `Obdlznik` – na obrázku vyššie sú to tvary A, B, E a G). Taktiež môžeme

povedať, že každá inštancia triedy je typu trieda (**`modrý kruh`** má typ triedy **`Kruh`**). Každá inštancia má všetky vlastnosti definované v triede a nastavuje im konkrétne, sebe vlastné, hodnoty (napríklad každý objekt obdĺžnik môže mať inú farbu). Vlastnosti objektov budeme nazývať **atribúty**. Hodnoty všetkých atribútov v objekte definujú aktuálny **stav** objektu.



Obrázok 1.4: Vzťah medzi triedou a inštanciou

Rôzne druhy objektov definované v bodoch 1 až 4 môžeme označiť za triedy. Tieto definujú spoločné vlastnosti a správanie pre určité objekty. Jednotlivé geometrické útvary sú potom inštancie týchto tried. **`Zelený štvorec A`**, je inštanciou triedy **`Obdĺžnik`**, pričom jeho **`výška`** a **`dĺžka`** sú rovnaké a jeho **`farba`** je zelená. **`Žltý obdĺžnik B`** je tiež inštanciou tej istej triedy **`Obdĺžnik`**, no hodnoty jeho vlastností sú iné (špecifické iba pre tento konkrétny útvar), má rôznu **`dĺžku`** a **`šírku`** a má žltú **`farbu`**.

ZAPAMÄTAJTE SI!

Základným stavebným kameňom objektovo orientovaného programovania je **objekt**. Pri programovaní aplikácií budeme využívať hlavne **inštancie tried**. Triedu môžeme chápať ako predpis pre vytvorenie svojej inštancie, ktorý určuje jej vlastnosti a správanie

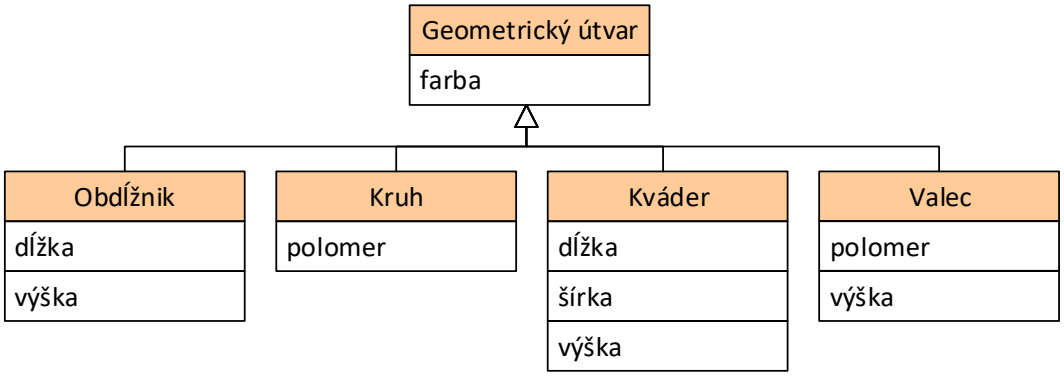
Pri tvorbe aplikácií, založených na princípoch objektovo orientovaného programovania, sa najčastejšie pracuje s inštanciami. Preto je často zaužívaná konvencia, označovať pojmom **objekt** inštanciu triedy, tieto pojmy sa teda používajú ako ekvivalenty. Túto konvenciu budeme dodržovať aj v tomto texte, hoci takto zaužívané pomenovanie nie je úplne správne.

Zamyslime sa nad príkladom s geometrickými tvarmi ešte raz. Podobne ako v skutočnom svete (napríklad všetci ľudia na svete majú vek, všetci občania Slovenska majú rodné číslo, hráči jedného športového tímu majú všetci číslo dresu), aj triedy dokážu preberať isté vlastnosti a správanie (napríklad každý futbalista vie kopnúť do lopty) všeobecnejších tried. Všetky štyri naše triedy definujú ten istý atribút – **`farba`**. Keby každá naša trieda definovala tento atribút samostatne, bolo by takéto riešenie neefektívne a, ako uvidíme neskôr, aj dosť obmedzujúce. S využitím tried si však dokážeme jednoducho poradiť. Definujme všeobecnejšiu triedu (napríklad **`Geometrický útvar`**) s atribútom **`farba`** a ostatným triedam určíme, aby prevzali jej vlastnosti. Takémuto vzťahu medzi triedami, v ktorom jedna trieda (potomok) preberá vlastnosti inej triedy (predok) sa hovorí **dedičnosť**.

ZAPAMÄTAJTE SI!

Dedičnosť patrí medzi základné koncepty objektovo orientovaného programovania. Je to vzťah medzi dvoma triedami – predkom a potomkom. Potomok preberá všetky vlastnosti predka, ktoré môže ďalej rozširovať.

Vzťahy medzi triedami sa často vyjadrujú graficky. Trieda má typicky tvar obdĺžnika, v ktorom je napísaný jej názov. Voliteľne môže obsahovať zoznam atribútov a činností, ktoré môže vykonávať. Prvý vzťah medzi triedami, ktorý sme uviedli, je dedičnosť. Značí sa pomocou šípky  smerujúcej od potomka k predkovi. Diagram tried predstavujúci geometrické entity vidíme na nasledujúcom obrázku. Konvenciou je nazývať triedy veľkým začiatočným písmenom.

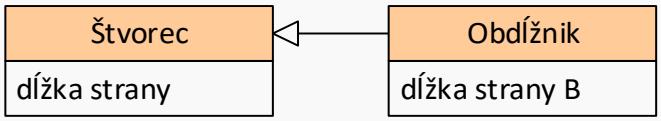


Obrázok 1.5: Hierarchia tried predstavujúcich geometrické útvary

Dedičnosť medzi dvoma triedami funguje správne iba vtedy, ak potomka môžeme označiť ako predka. Teda **`Obdĺžnik`** je vhodným potomkom triedy **`Geometrický útvar`**, lebo každý obdĺžnik je možné považovať za geometrický útvar (je zrejmé, že naopak to neplatí).

ÚLOHA 1.2

Na nasledujúcom obrázku je hierarchia tried Štvorec a Obdĺžnik. Je takáto hierarchia dobrá?



Ak by bola trieda **`Štvorec`** predkom triedy **`Obdĺžnik`**, tak by to znamenalo, že každý obdĺžnik je štvorcom. To, samozrejme, nie je pravda, preto by takto namodelovaný vzťah dvoch tried nebol správny.

Vráťme sa ku geometrickým útvarom a ich triedam. Povedali sme si, že trieda slúži ako továreň svojich inštancií. Trieda definuje atribúty (vlastnosti) a správanie, ktoré budú mať jej inštancie. Nie vždy je však rozumné vytvárať inštancie triedy, niekedy to nemá zmysel. Ako by v našom prípade vyzerala inštancia triedy **`Geometrický útvar`**? Čo by to bolo? Vidíme, že v tomto prípade nemá zmysel tvoriť inštancie triedy, napriek tomu, že samotná trieda zmysel má –

zastrešuje spoločné vlastnosti pre všetky geometrické útvary. Triedy, ktorých inštancie nemá zmysel vytvárať, budeme nazývať **abstraktné triedy**.

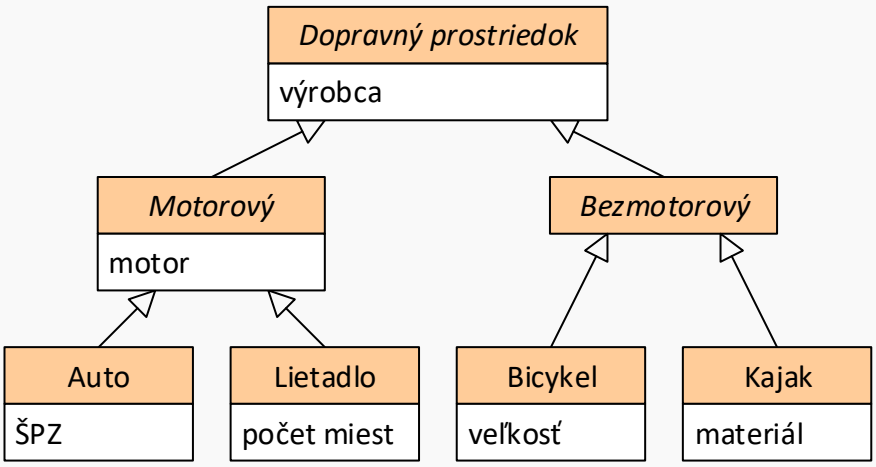
Pripomeňme, že podobný termín - nehmotný - sme používali už skôr v úlohe 1.1. Tam sme chápali nehmotný objekt v takom zmysle, že ho nevieme uchopiť do rúk. Takéto chápanie však nesúvisí s pojmom abstraktná trieda tak, ako je chápaná v objektovom programovaní. Napr. čas je nehmotný (abstraktný) z hľadiska reálneho sveta (nevieme ho chytiť do rúk), avšak trieda, ktorou čas modelujeme, nie je abstraktná (vieme vytvoriť jej inštanciu).

Poznamenajme ešte, že potreba zovšeobecnenia a abstrakcie na úrovni objektov môže vychádzať nielen z toho, že objekty majú spoločné vlastnosti, ale aj z ich spoločného správania. Predstavme si napríklad orchester, v ktorom sú spolu združení hudobníci hrajúci na rôzne nástroje. Každý z nich môže mať úplne iné vlastnosti (atribúty), ktoré sú špecifické pre ten ktorý druh hudobného nástroja (hudobníci dokonca nemusia zdieľať ani jeden spoločný atribút), no všetci vedia hrať. Aj v tomto prípade je vhodné takéto triedy hudobníkov združiť a zaviesť triedu spoločného predka pre všetkých hudobníkov (napr. **`Hudobník`**) , ktorá bude definovať činnosť nazvanú napríklad **`hraj`**. Neskôr si ukážeme, že objektové programovanie nám umožňuje efektívne zabezpečiť, aby každá odvodená trieda od triedy **`Hudobník`** mohla zahrať na hudobný nástroj pre ňu špecifickým spôsobom.

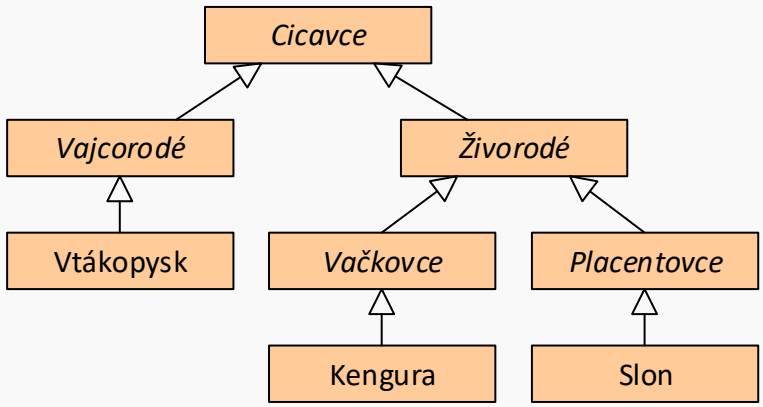
ÚLOHA 1.3

Vytvorte hierarchiu tried a) dopravných prostriedkov, b) zvierat a c) súčiastok počítača. Uveďte vlastnosti, ktoré dané triedy definujú. Ktoré triedy sú v návrhu abstraktné?

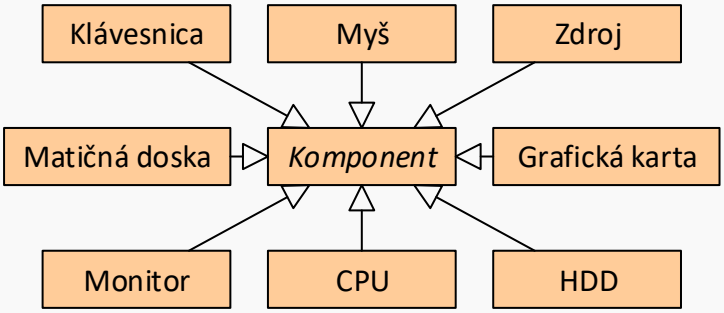
Uvedieme iba jednoduché príklady. Atribúty neuvádzame. Abstraktné triedy majú názov zobrazený kurzívou



V prípade zvierat sa môžeme orientovať biologickou klasifikáciou živočíchov.



Komponenty tvoriace počítač môžu mať všetky jediného spoločného predka.



1.3 Tvorba inštancie sveta

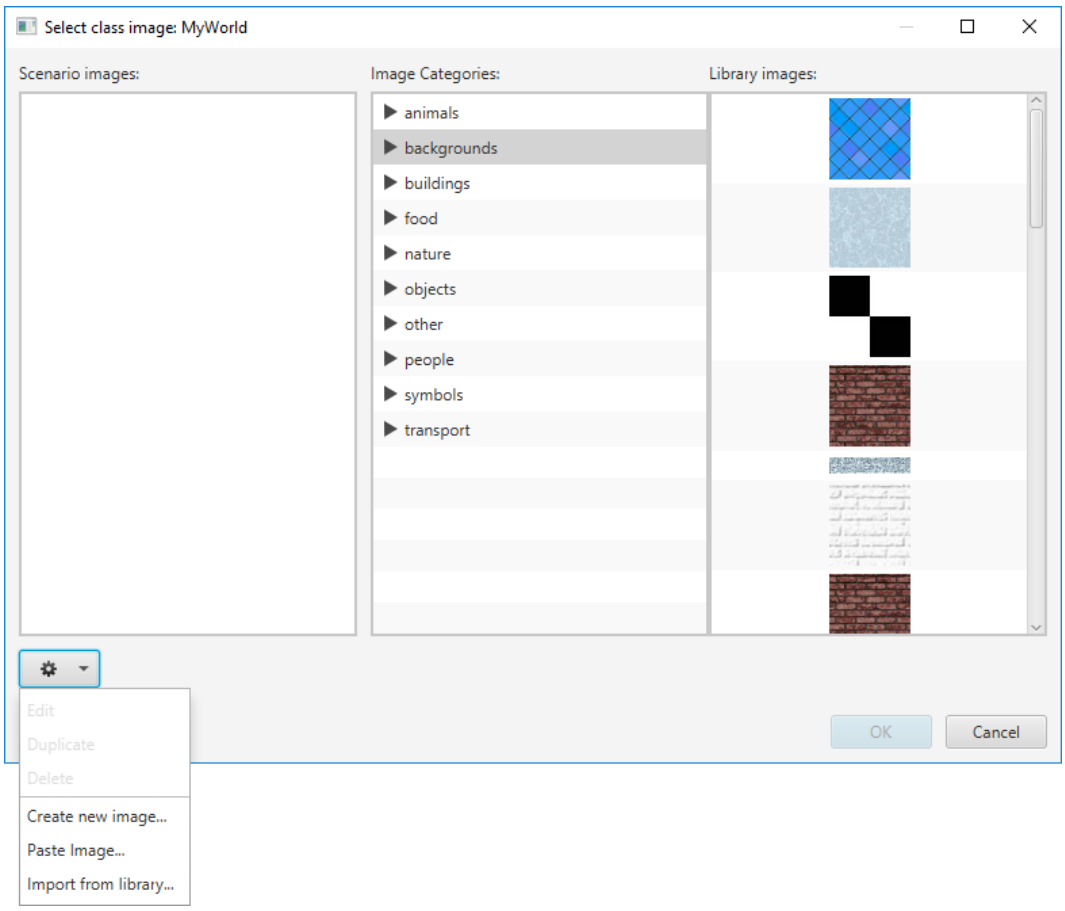
Vráťme sa naspäť do prostredia Greenfoot. Na pravej strane vidíme diagram tried, ktorý obsahuje dve abstraktné triedy – **`World`** a **`Actor`** a jedného potomka triedy **`World`** – triedu **`MyWorld`**. Ako sme už uviedli, potomkovia triedy **`World`** tvoria javisko (scénu), na ktorej hrajú aktori podľa toho, ako ich naprogramujeme.

Svet v prostredí Greenfoot sa skladá z buniek. Bunky sú vždy štvorcového tvaru, dĺžku strany bunky vieme nastaviť. Počet buniek v oboch smeroch definuje potomok triedy **`World`**. Svet (javisko) si teda môžeme predstaviť ako veľkú šachovnicu.

Poďme urobiť náš svet krajší. Kliknite na svet (hnedý obdĺžnik **`MyWorld`** v pravej časti prostredia Greenfoot) pravým tlačidlom myši a z kontextového menu vyberte možnosť **`Set image...`** Greenfoot nám ponúkne dialóg, v ktorom je možné rôznym spôsobom zvoliť obrázok:

1) Vybrať si obrázok, ktorý je pripravený (v sekcii **`Image categories`** vyberieme **`background`**).
2) Vytvoriť vlastný obrázok, ktorý bude ponúknutý v sekcií **`Scenario images`**. Toto je možné spraviť tromi spôsobmi pomocou tlačidla a menu vľavo dole:

3) Nakresliť si vlastný obrázok v externom editore a využiť funkciu **`Import from library`** prípadne ho ručne umiestniť do adresára **`images`** v projektovom adresári.
4) Využiť funkciu **`Paste image...`**. V tomto prípade musí byť obrázok skopírovaný v schránke (napr. pomocou CTRL+C). Greenfoot obrázok zo schránky automaticky vyberie a predtým, ako ho zaradí do sekcie **`Scenario images`** vyzve užívateľa na zadanie jeho mena.
5) Využiť funkciu **`Create new image...`**. Greenfoot v tomto prípade vyzve na zadanie rozmerov a názvu obrázka, ktorý automaticky zaradí do **`Scenario images`**. Potom otvorí externý editor, v ktorom je možné obrázok upraviť.



Obrázok 1.6: Nastavenie obrázka sveta

Po výbere obrázka sa tento priradí svetu. Aby sa však zmena vizuálne prejavila, je potrebné vytvoriť novú inštanciu sveta (teda požiadať triedu **`MyWorld`**, aby vytvorila objekt). To urobíme tak, že z kontextového menu triedy **`MyWorld`** vyberieme položku new **`MyWorld()`**. Obrázok sa potom opakovane zobrazí na celej ploche sveta. Je dôležité si uvedomiť, že veľkosť bunky a veľkosť obrázka spolu nijako nesúvisia. Obrázok sa aplikuje na svet a nie na jeho jednotlivé bunky. Ak teda chceme, aby boli bunky viditeľne a správne ohraničené, musíme zvoliť rovnakú veľkosť obrázka a veľkosť buniek.

ÚLOHA 1.4

V grafickom editore vytvorte dlaždicu, ktorá bude graficky reprezentovať bunku sveta. Zvoľte štvorcovú reprezentáciu, ideálne 60x60 pixelov. Obrázok importujte alebo uložte v projektovom priečinku do priečinka **images**. Nastavte obrázok ako obrázok sveta. Všimnite si, že trieda **MyWorld** získala v diagrame tried malú ikonku reprezentujúcu jej grafickú podobu.

Obrázok nakreslite v grafickom editore, ktorý vám vyhovuje. Importovanie obrázka je popísané v kapitole vyššie.

Nastavenému obrázku je potrebné prispôsobiť veľkosť sveta. To urobíme pomocou editora kódu. Dvojklikom na triedu **MyWorld** sa nám zobrazí editor kódu, v ktorom vidíme kód triedy **MyWorld**.

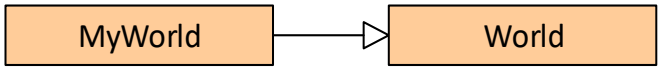
```
public class MyWorld extends greenfoot.World
{
    /**
    * Constructor for objects of class MyWorld.
    *
    */
    public MyWorld()
    {
        // Create a new world with 600x400 cells with a cell size of 1x1 pixels.
        super(600, 400, 1);
    }
}
```

Kód triedy **MyWorld** (ako i akejkoľvek inej triedy) môžeme rozdeliť na dve časti:

1. Hlavička
2. Telo

Hlavička triedy začína dvojicou kľúčových slov **public class**, čím špecifikujeme, že sa jedná o verejnú (**public**) triedu (**class**). Nasleduje identifikátor (názov triedy), ktorý si môžeme zvoliť. Ďalej nasleduje nepovinné kľúčové slovo **extends**, čím dávame najavo, že trieda **MyWorld** je potomkom triedy **World** (teda ju rozširuje – **extends**).

```
public class MyWorld extends greenfoot.World
```



Telo triedy bude v budúcnosti obsahovať definície atribútov a činností, ktoré budú vedieť inštancie triedy vykonávať. V súčasnosti sa v tele nachádza iba automaticky vytvorený **konštruktor triedy**. Zatiaľ sa týmto kódom nemusíme zaoberať. Stačí nám vedieť, že konštruktor tvorí postupnosť príkazov, pomocou ktorej trieda inicializuje svoje inštancie (v našom prípade sú to príkazy, ktorými sa inicializuje inštancia triedy **`MyWorld`**). Pomocou konštruktora nastaví novo vznikajúci objekt hodnoty svojich atribútov tak, aby bol v uspokojivom počiatočnom stave. Všimnite si tiež rovnaký názov konštruktora a triedy (toto platí vždy). V súčasnosti sa v konštruktore nachádza jediný riadok:

```
super(600, 400, 1);
```

Týmto riadkom vyjadrujeme, že chceme vytvoriť svet o veľkosti 600 x 400 buniek, pričom každá bunka bude mať veľkosť 1 pixel.

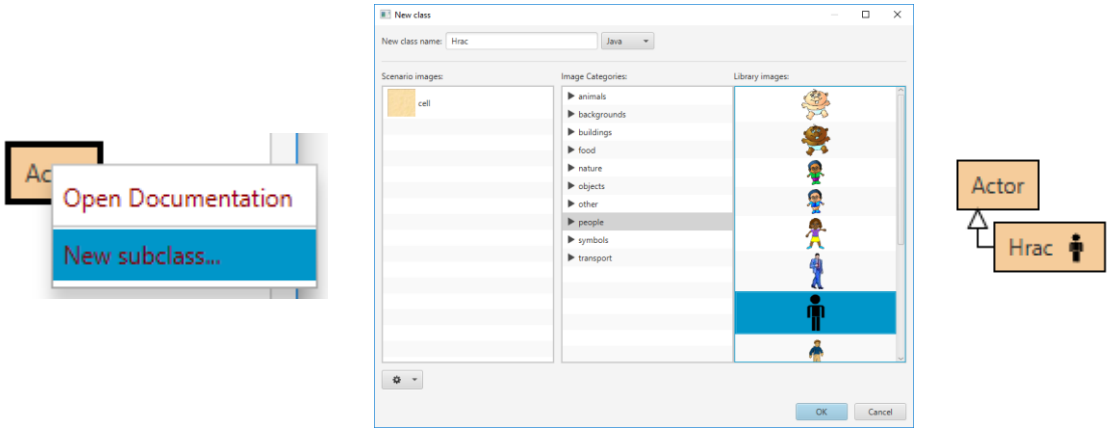
ÚLOHA 1.5

Upravte konštruktor triedy **`MyWorld`** tak, aby sa vytvoril svet o veľkosti 25x15 buniek, pričom každá bunka bude veľká 60 pixelov. Ako by bolo potrebné upraviť obrázok, aby svet vyzeral ako šachovnica (teda aby sa striedali rôzne farebné políčka)?

```
super(25, 15, 60);
```

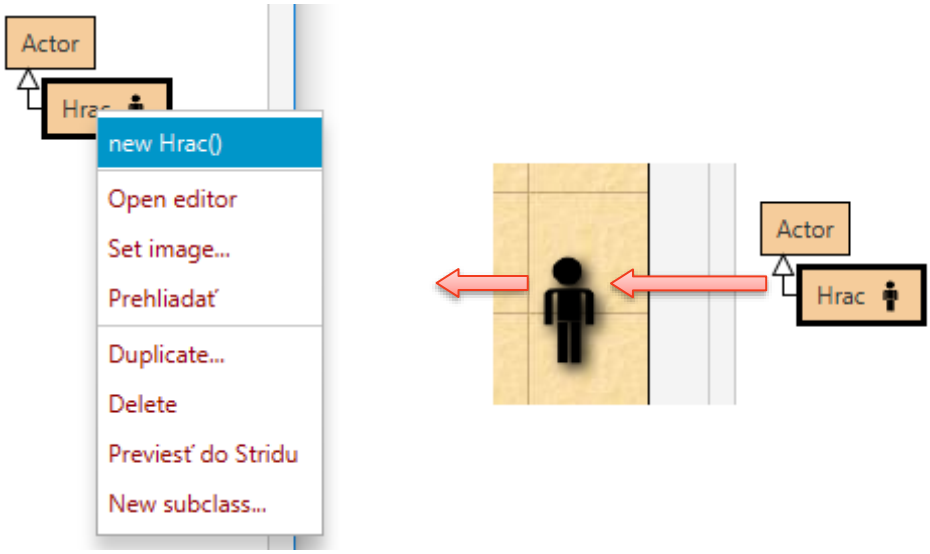
1.4 Vytvorenie hráča

Hráči budú predstavovať človekom ovládaných bombermanov, ktorí budú súperiť o víťazstvo vo svete. Ako všetko, čo sa chce pohybovať vo svete, musí byť aj hráč odvodený od triedy **`Actor`**. Vytvoríme si teda novú triedu – klikneme pravým tlačidlom myši na triedu **`Actor`** a z menu zvolíme **`New subclass...`**. V zobrazenom dialógu zadáme názov novej triedy bez diakritiky **`Hrac`**. Ďalej, podobne ako pri triede **`MyWorld`**, zvolíme pre danú triedu obrázok. Vyberieme obrázok hráča (napr. z kategórie **`people`**). Po stlačení tlačidla OK vznikne nová trieda **`Hrac`**, ktorá je potomkom triedy **`Actor`**.



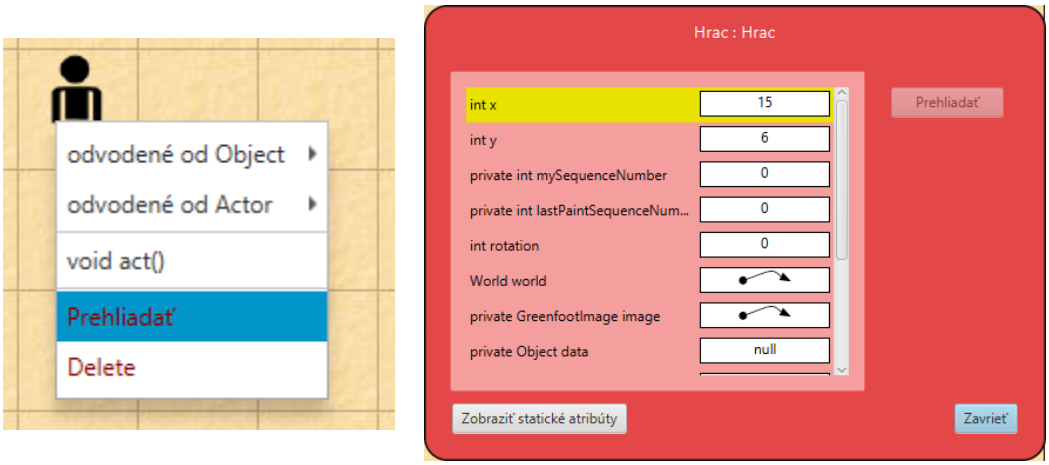
Obrázok 1.7: Postup pri tvorbe novej triedy

Po vytvorení triedy môžeme vytvoriť jej inštanciu. Klikneme pravým tlačidlom myši na triedu **`Hrac`** a zvolíme príkaz **`new Hrac()`**. Vzniknutú inštanciu reprezentovanú obrázkom hráča presunieme na hraciu plochu, čím hráča umiestnime do sveta. Takto sme si vyskúšali vytvorenie prvej inštancie triedy **`Hrac`**.



Obrázok 1.8: Postup pri vytvorení inštancie triedy `Hrac`

Každý objekt je jednoznačne charakterizovaný svojim vnútorným stavom. Nástroj Greenfoot ponúka jednoduchý spôsob jeho prezretia. Kliknite pravým tlačidlom myši na inštanciu triedy **`Hrac`** a zvoľte voľbu **`Prehliadať`**. Greenfoot vytvorí okno so živým náhľadom vnútorného stavu inštancie, podobne, ako je zobrazené na nasledujúcom obrázku.



Obrázok 1.9: Postup pri preskúmaní stavu inštancie triedy `Hrac`

ÚLOHA 1.6

Chyťte vytvorenú inštanciu triedy **`Hrac`** pomocou myši a presuňte ju na inú pozíciu vo svete. Sledujte živý náhľad vnútorného stavu – čo pozorujete? Vytvorte ďalšiu inštanciu triedy **`Hrac`** a zobrazte aj jej vnútorný stav. Opäť presuňte pomocou myši jednu z dvoch inštancií – ktorý vnútorný stav sa zmenil?

Náhľady na vnútorný stav objektov sú živé – menia sa okamžite pri pohybe aktorom. Aktualizujú sa hodnoty atribútov tej inštancie triedy **`Hrac`**, ktorou sa aktuálne pohybuje.

Veľmi dôležitou vlastnosťou objektov je **identita** objektov. Je jedno, či objekty pochádzajú z tej istej triedy alebo či majú rozdielne alebo zhodné hodnoty niekoľkých alebo všetkých atribútov. Akonáhle objekt vznikne, je jedinečný a jednoznačne identifikovateľný. Identitu si môžeme všimnúť napríklad pri presúvaní objektu myšou, pri zmene polohy sa menia hodnoty atribútov iba presúvaného objektu.

1.5 Metódy objektov

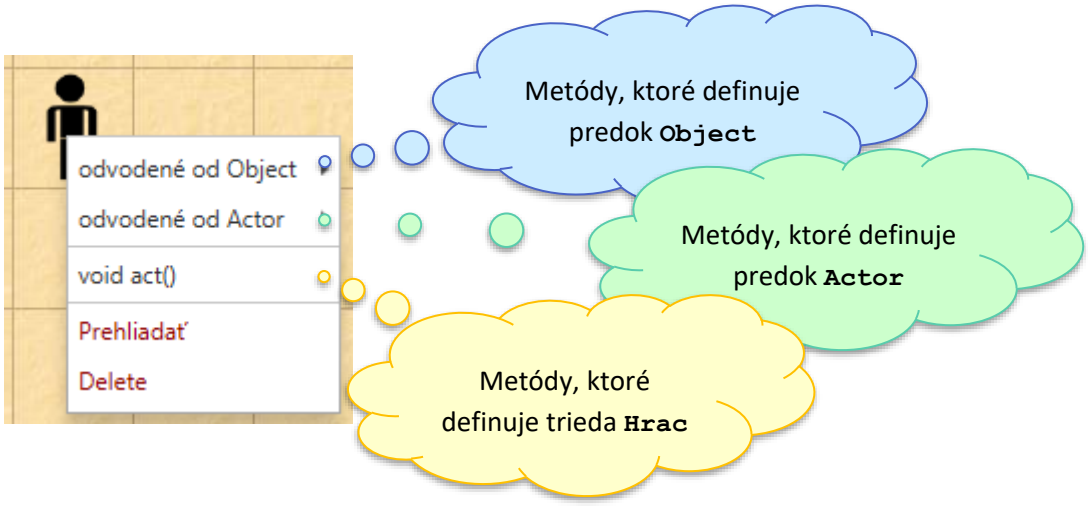
V úvode tejto kapitoly sme rozobrali základné vlastnosti objektov. Už vieme, že objekty sú charakterizované svojimi atribútmi a vieme tieto atribúty preskúmať. Ďalšou dôležitou súčasťou objektov je ich schopnosť vykonávať činnosť.

ZAPAMÄTAJTE SI!

Presne definovanej postupnosti činností objektu budeme hovoriť **metóda**. Metódy sa definujú v triedach. Inštancia danej triedy je schopná vykonať metódy, ktoré definovala jej rodičovská trieda alebo akýkoľvek predok jej rodičovskej triedy (podobne ako preberá atribúty, preberá aj metódy). Metóda sa skladá z troch častí:

1) Názov metódy rozlišuje metódy navzájom. Typicky sa ako názov volí slovo alebo slovné spojenie, ktoré charakterizuje jej činnosť.
2) Parametre metódy predstavujú rôzne vstupy do metódy, ktorými je možné upraviť alebo bližšie špecifikovať jej správanie.
3) Návratová hodnota metódy je voliteľná a umožňuje objektu informovať volajúceho o tom, ako vykonanie metódy skončilo.

Metódy, ktoré je možné vyvolať na inštancii triedy **`Hrac`**, môžeme preskúmať (a aj priamo vyvolať) pomocou kliknutia pravým tlačidlom myši na inštanciu triedy **`Hrac`**.

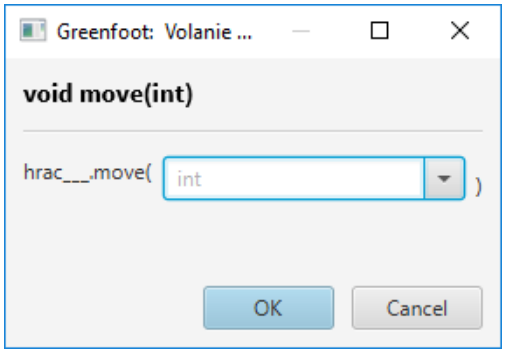


Obrázok 1.10: Preskúmanie metód inštancie triedy `Hrac`

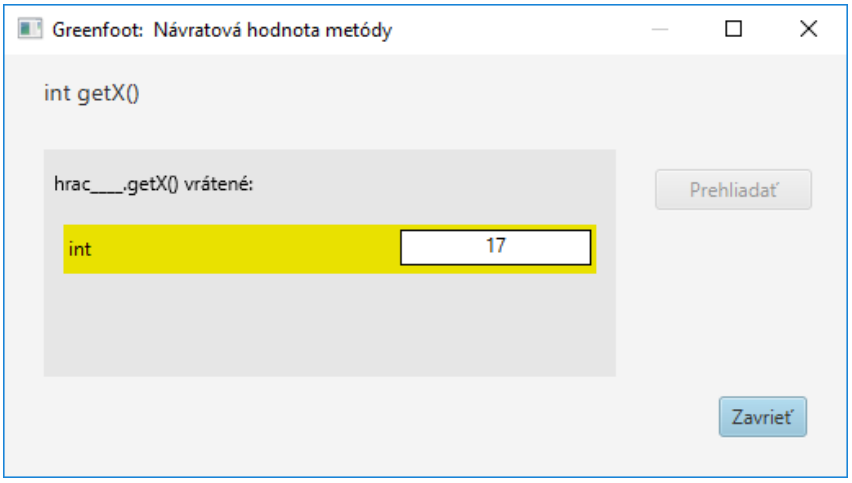
ÚLOHA 1.7

Vyvolajte nad rôznymi inštanciami triedy **`Hrac`** metódy, ktoré ponúka nástroj Greenfoot. Sledujte, ako sa mení vnútorný stav inštancie.

Ak vyvoláte metódu, ktorá požaduje vstupné parametre, Greenfoot si ich vypýta vo forme dialógu, ako je zobrazené na nasledujúcom obrázku. Podobne Greenfoot použije dialóg na oznámenie návratovej hodnoty, ak metóda nejakú mala.



Obrázok 1.11: Dialógové okno pre získanie hodnoty parametra metódy



Obrázok 1.12: Dialógové okno pre oznámenie návratovej hodnoty metódy

V nasledujúcej tabuľke sú uvedené niektoré dôležité metódy triedy **`Actor`**.

Tabuľka 1.1: Prehľad vybraných metód triedy Actor

Metóda	Parametre	Čo metóda vráti	Čo metóda robí
`move`	vzdialenosť	-	pohne aktora v smere, ako je otočený, o zadanú vzdialenosť dopredu.
`getX`	-	číslo	získa X-ovú súradnicu aktora (horizontálny index bunky, v ktorej sa nachádza).
`getY`	-	číslo	získa Y-ovú súradnicu aktora (vertikálny index bunky, v ktorej sa nachádza).
`setLocation`	polohaX, polohaY	-	presunie aktora do bunky sveta s danými súradnicami.
`turn`	stupne	-	otočí aktora o zadaný počet stupňov, kladné číslo znamená otočenie aktora v smere hodinových ručičiek, záporné v protismere.
`turnTowards`	pozíciaX, pozíciaY	-	otočí aktora tak, aby smeroval presne k bunke na daných súradniciach.
`getRotation`	-	číslo	vráti otočenie aktora. Otočenie je číslo od 0 do 359 a vyjadruje otočenie v stupňoch v smere hodinových ručičiek. 0 znamená, že aktor sa pozerá priamo na východ (doprava).
`setRotation`	stupne	-	nastaví otočenie aktora.

ZHRNUTIE

Základným prvkom objektovo orientovaného programovania je objekt. Objekt reprezentuje akýkoľvek prvok sveta, reálny či nereálny, hmotný alebo nehmotný. Objekty sú definované pomocou tried objektov, ktoré

- definujú ich vlastnosti – atribúty,
- definujú ich činnosti – metódy,
- sa starajú o vznik inštancií.

Každá inštancia má atribúty, ktoré preberá zo svojej triedy. Aktuálne hodnoty atribútov definujú jej stav. Každá inštancia je jednoznačne identifikovateľná – má svoju identitu.

Medzi triedami môže vzniknúť vzťah potomok – predok, ktorý sa nazýva dedičnosť. Pomocou dedičnosti potomok rozširuje činnosti a atribúty, ktoré zdedil od predka. Niekedy takto môže vzniknúť predok, pri ktorom nemá zmysel vytvárať inštancie – takúto triedu voláme abstraktná.

Prostredie Greenfoot definuje dve abstraktné triedy – **`World`** (javisko) a **`Actor`** (herci, ktorí sa podľa pravidiel pohybujú po svete).

Pomocou kontextového menu (kliknutie pravým tlačidlom myši) triedy v prostredí Greenfoot je možné zmeniť jej grafickú reprezentáciu, vytvoriť jej inštanciu alebo otvoriť zdrojový kód.

Pomocou kontextového menu (kliknutie pravým tlačidlom myši) inštancie v prostredí Greenfoot je možné preskúmať jej vnútorný stav a vyvolať jej metódy.

ÚLOHY NA PRECVIČOVANIE

ÚLOHA 1.A

Popíšte, aké atribúty majú triedy Strom, Nábytok, Vyučovacia hodina.

ÚLOHA 1.B

Vytvorte hierarchiu tried reprezentujúcu hudobné nástroje. Ktoré triedy budú abstraktné?

ÚLOHA 1.C

Vytvorte svet, ktorého rozmery budú 10x10 buniek a každá bude veľká 50 pixelov.

ÚLOHA 1.D

Zmeňte grafickú reprezentáciu sveta tak, aby vytvoril šachovnicu.

ÚLOHA 1.E

Zmeňte grafickú reprezentáciu objektov triedy **`Hrac`**.

ÚLOHA 1.F

Aké metódy a s akými parametrami musíte zavolať, aby inštancia triedy hráč postupne navštívila všetky 4 rohy sveta?

ÚLOHA 1.G

Aké metódy ponúka trieda **`World`** a jej potomok **`MyWorld`**?

2 ALGORITMUS, OVLÁDANIE APLIKÁCIE, TVORBA METÓD

KĽÚČOVÉ SLOVÁ

Algoritmus. Vlastnosti algoritmu. Algoritmizácia. Syntax kódu metódy. Metóda `act()`. Poznámky v kóde.

CIELE

Cieľom druhej kapitoly je naučiť sa, čo znamenajú pojmy algoritmus, algoritmizácia, aké sú vlastnosti algoritmu. Naučíme sa ovládať aplikáciu prostredníctvom predvolených príkazov v prostredí Greenfoot – `Run`, `Pause`, `Reset` a `Act`, ako aj správne syntakticky napísať jednoduchú sekvenciu kódu do predvolenej metódy `act()`. Kapitola predstaví ako je možné vytvoriť vlastnú metódu – naučíme sa syntax jej zápisu. Ukážeme si čo znamená `this.metoda()` a ako sa zapisujú poznámky do kódu.

OBSAH

2.1 Algoritmus, jeho vlastnosti a algoritmizácia

Každý z nás sa už v živote stretol s nejakou úlohou, ktorej riešenie je možné vyjadriť postupnosťou logicky nadväzujúcich krokov. Uvažujme napríklad aké kroky musíme vykonať pre prípravu čaju:

1) Najskôr je potrebné prevariť vodu.
2) Potom si je potrebné pripraviť pohár.
3) Do pohára vložíme vrecúško čaju.
4) Pohár zalejeme prevarenou vodou a počkáme, kým sa čaj vylúhuje.
5) Vyberieme vrecúško.
6) Čaj dochutíme.

Podobnú logickú postupnosť krokov je možné aplikovať aj na iné oblasti bežného života.

ÚLOHA 2.1

Zapíšte postup, ako je možné pripraviť kávu, dopraviť sa do školy, uvariť obed.

Príprava kávy:

1) Najskôr prevarím vodu.
2) Potom pripravím pohár.
3) Do pohára nasypem kávu.
4) Kávu zalejem prevarenou vodou.
5) Dochutím kávu.

Doprava do školy:

1) Ak je pekne
    a. tak pôjdem na bicykli,
2) inak:
    a. Prejdem na autobusovú zastávku.
    b. Počkám na autobus.
    c. Nastúpim do autobusu.
    d. Počkám, kým autobus príde na konečnú zastávku.
    e. Vystúpim z autobusu.
    f. Prejdem do školy.

Varenie obeda:

1) Nakrájam mäso.
2) Dám mäso dusiť.
3) Pokiaľ nie je mäso udusené, tak ho pravidelne kontrolujem.
4) Zároveň s kontrolou mäsa:
    a. Očistím zemiaky.
    b. Nakrájam zemiaky.
    c. Dám variť zemiaky.
5) Pokiaľ nie sú zemiaky uvarené, tak ich kontrolujem.
6) Keď je mäso udusené a zemiaky uvarené, obed je hotový.

Podobne, ako v predchádzajúcich príkladoch, aj pri tvorbe programu budeme musieť riešiť úlohy, z ktorých sa každý počítačový program skladá. Pre riešenie konkrétnej úlohy teda hľadáme vhodnú postupnosť krokov, ktorej budeme hovoriť **algoritmus**. Keďže programovanie a algoritmizácia (tvorba algoritmov) sú úzko späté, na tomto mieste vysvetlíme, čo to algoritmus je a aké vlastnosti by mal spĺňať.

ZAPAMÄTAJTE SI!

Algoritmus je konečná postupnosť/usporiadanie postupov aplikovaných na konečný počet dát, ktorý dovoľuje riešiť približne rovnaké triedy problému. [3]
Slovníková definícia uvádza, že algoritmus je súhrn matematických úkonov slúžiacich k účelnému vykonaniu určitého výpočtu platného pre všetky podobné úlohy.

Algoritmus môžeme zapísať rôznymi spôsobmi, či už slovne, algoritmickým jazykom alebo grafickým vyjadrením (vývojovým diagramom, diagramom aktivít alebo štruktúrogramom).

Zamyslime sa nad dvomi algoritmami, ktoré sme už vytvorili – algoritmus na prípravu čaju a algoritmus na prípravu kávy. Vidíme, že sú si dosť podobné. Uvažujme, ako by sme ich zlúčili do jedného algoritmu tak, aby sme zachovali jednu z vlastností algoritmu – hromadnosť.

ÚLOHA 2.2

Zostavte všeobecný algoritmus pre prípravu horúceho nápoja. Zamyslite sa, aké vstupy bude takýto algoritmus potrebovať, aby bol hromadný?

Príprava horúceho nápoja:

Vstupy: tekutina, teplota, zmes, doba lúhovania, dochucovadlo.

1) Zohrejem *tekutinu* na danú *teplotu.*
2) Potom pripravím pohár.
3) Do pohára vložím *zmes*.
4) Zmes zalejem zahriatou tekutinou.
5) Ak je doba lúhovania väčšia ako 0, tak počkám a potom vyberiem zmes.
6) Dochutím nápoj *dochucovadlami*.

Príprava čaju je potom príprava horúceho nápoja so vstupmi:
voda, 100°C, čajové vrecúško, 5 minút, med a citrón.
Príprava kávy je potom príprava horúceho nápoja so vstupmi:
voda, 100°C, káva, 0, cukor a mlieko.
Príprava kakaa je potom príprava horúceho nápoja so vstupmi:
mlieko, 60°C, kakao, 0, nič

Pri zostavovaní všeobecného algoritmu na prípravu horúceho nápoja sme museli urobiť viac krokov. Kávu a vrecúško čaju sme nahradili všeobecnou zmesou, namiesto vody a mlieka sme použili pojem tekutina atď. Taktiež sme sa zamysleli nad poradím krokov.

ZAPAMÄTAJTE SI!

Algoritmizácia je proces hľadania algoritmu pre danú úlohu. Správne zostavenie algoritmu by sa malo skladať z nasledovných krokov:

1) Formulovanie zadania úlohy, vstupov a výstupov.
2) Analýza riešenia a zovšeobecnenie úlohy.
3) Nájdenie vzťahu, ako sa z daných vstupov vytvoria požadované výstupy.
4) Zostavenie algoritmu definovaním presnej následnosti krokov.

2.2 Tvorba metódy

Algoritmus vyjadruje postupnosť krokov vedúcich k riešeniu nejakej úlohy. Spomeňme si, že pojmom metóda je určená presná postupnosť činnosti objektu. Do metód objektov teda budeme písať algoritmy.

Otvorme editor zdrojového kódu triedy **`Hrac`**. V tele triedy vidíme predpripravenú metódu **`act()`**, ktorá má nasledovný tvar (všimnite si veľkú podobnosť s definíciou konštruktora):

```
/**
* Act - do whatever the Hrac wants to do. This method is called
* whenever the 'Act' or 'Run' button gets pressed in the
* environment.
*/
public void act()
{
    // Add your action code here.
}
```

Metóda **`act()`** má v prostredí Greenfoot špeciálny význam (preto ju prostredie automaticky pripraví), nateraz nám poslúži na pochopenie zdrojového kódu metódy. Kód akejkoľvek metódy môžeme rozdeliť na dve časti:

1. Hlavička
2. Telo

Hlavička metódy začína kľúčovým slovom **`public`**. Tým špecifikujeme, že sa jedná o verejnú metódu (existujú aj iné ako verejné metódy, tými sa však budeme zaoberať až neskôr). Nasleduje typ návratovej hodnoty metódy (**`void`**, tento typ použijeme, keď budeme chcieť vyjadriť, že metóda nič nevracia, hlbšie sa typom návratových hodnôt budeme venovať neskôr), identifikátor metódy (jej názov) a povinné zátvorky s parametrami (metóda **`act()`** nemá žiadne parametre, zátvorky sú ale povinné). Identifikátor metódy sa snažíme tvoriť tak, aby vyjadroval, čo metóda robí. Ak potrebujeme na vyjadrenie tejto činnosti viac slov, využijeme konvenciu camelCase. Všetky slová píšeme dokopy, začíname malým písmenom a každé ďalšie slovo začneme veľkým písmenom. Uveďme niekoľko príkladov identifikátorov pomocou konvencie camelCase: **`urobKrok`**, **`otocSa`**, **`skacATlieskajRukami`**.

Telo metódy sa píše do zložených zátvoriek. Obsahuje algoritmus, ktorý sa má vykonať. Telo metódy bude môcť v budúcnosti obsahovať definíciu lokálnych premenných, rôzne príkazy (priraďovacie, príkazy vetvenia a cyklu) a volania iných metód objektov, vrátane volania vlastných metód.

Pripravme metódu (zatiaľ s prázdnym telom), ktorú použijeme na urobenie dlhého kroku. Metóda nebude mať návratovú hodnotu a nebude preberať žiadne parametre. Do tela triedy pod metódu **`act()`** dopíšeme:

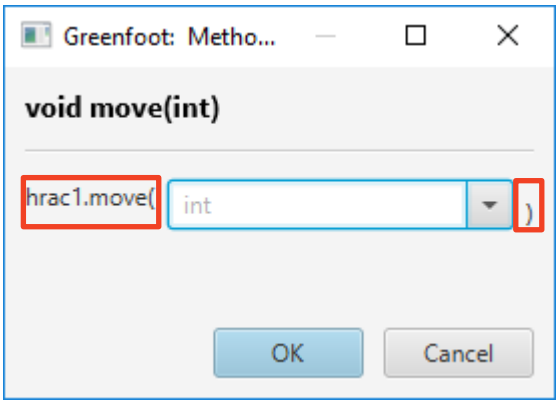
```
public void urobDlhyKrok()
{
}
```

ÚLOHA 2.3

Preskúmajte metódy inštancie triedy **Hrac**. Čo pozorujete?

Vytvoríme inštanciu triedy **Hrac** a umiestnime ju do sveta. Potom klikneme pravým tlačidlom myši na túto inštanciu. V menu by sme mali vidieť nami vytvorenú metódu.

Vyvolajme teraz metódu **move()**. Greenfoot nás pomocu dialógu vyzve, aby sme zadali parameter, ktorý znamená počet buniek, o ktoré sa chceme pohnúť. Všimnite si text okolo textového poľa (zvýraznený na nasledujúcom obrázku):



Obrázok 2.1: Dialógové okno pre zadanie hodnoty parametra metódy move()

Text tvorí nasledujúci kód, ktorý bližšie popíšeme:

```
hrac1.move(tu zadáme parameter);
```

- **hrac1** určuje, ktorému objektu bude vyvolaná metóda,
- **bodka** oddeľuje objekt od názvu metódy, ktorú chceme vyvolať,
- **move** je identifikátor metódy, ktorý chceme vyvolať,
- v povinných **zátvorkách** sa nachádzajú čiarkou oddelené parametre (tam vpíšeme hodnotu parametra, napr. 2),
- povinná **bodkočiarka** za príkazom (dialógové okno túto bodkočiarku nemá).

Ak teda chceme, aby sa hráč pohol o dve bunky v jeho aktuálnom smere, tak by príkaz vyzeral takto: **hrac1.move(2);**. Pripomeňme, že všetky inštancie tej istej triedy reagujú na volanie metódy rovnako. Zamyslime sa, aký kód budeme musieť napísať do tela metódy **urobDlhyKrok()** tak, aby sa vpred pohla iba tá inštancia triedy **Hrac**, ktorej metóda bola vyvolaná. Vieme, že použijeme metódu **move(2)** (táto metóda je dostupná, pretože je to metóda, ktorú definoval predok), avšak musíme špecifikovať hráča, ktorý to má urobiť – v našom prípade to bude ten a len ten hráč, ktorému bola vyvolaná metóda. Na toto využijeme kľúčové slovo **this**.

ZAPAMÄTAJTE SI!

Ak chceme vyvolať v inštancii triedy jej vlastnú metódu, použijeme na označenie cieľového objektu kľúčové slovo **`this`**. Vyvolať tak môžeme akúkoľvek metódu, ktorú sme definovali v rámci triedy alebo akúkoľvek verejnú metódu, ktorú definoval jej predok.

ÚLOHA 2.4

Doplňte do tela metódy **`urobDlhyKrok()`** taký príkaz, aby sa inštancia triedy **`Hrac`** pohla o dve bunky v aktuálnom smere. Následne vytvorte viac inštancií triedy **`Hrac`** a túto metódu vyvolajte na jednotlivých inštanciách. Je správanie očakávané?

Do tela metódy **`urobDlhyKrok()`** dopíšeme: **`this.move(2);`** Následne vytvoríme inštancie a z menu, vyvolaného po kliknutím pravého tlačidla myši na jednotlivé inštancie, vyvolávame metódu **`urobDlhyKrok()`**. Mali by sme pozorovať očakávané správanie, teda pohybovať sa bude iba tá inštancia, ktorej metódu sme vyvolali.

2.3 Písanie dokumentácie

Vráťme sa k metóde **`act()`**. Všimnime si, že nad hlavičkou tejto metódy sa nachádza anglický text, ktorý popisuje jej správanie. Tento text je ohraničený v bloku:

```
/**
 *
 *
 */
```

Textu v takomto bloku budeme hovoriť dokumentačný komentár. Prostredie Greenfoot umožňuje vygenerovať dokumentáciu pre jednotlivé časti kódu. Dobre vypracovaná dokumentácia umožňuje lepšie pochopenie kódu. Pre programátora, ktorý používa knižnice naprogramované niekým iným, tiež uľahčujú pochopiť význam jednotlivých metód a tried.

Všimnite si, že podobný dokumentačný komentár sa nachádza aj nad hlavičkou triedy. V nej sú naviac použité značky. Ak softvér pri tvorbe dokumentácie narazí na značku, vhodne daný text naformátuje a prepojí s inými relevantnými časťami kódu. Použitie značiek v dokumentačných komentároch síce nie je povinné, avšak prispieva k úplnosti a zrozumiteľnosti dokumentácie.

Úplný zoznam značiek nájdete v dokumentácii k Jave, napríklad na stránkach firmy Oracle. Prehľad dôležitých značiek sumarizuje nasledujúca tabuľka.

Tabuľka 2.1: Prehľad vybraných dokumentačných značiek

Značka	Typické použitie	Čo znamená
@author	trieda, metóda	Kto je autorom triedy alebo metódy
@param p	metóda	Vysvetlí význam parametra p v danej metóde
@return	metóda	Popíše, čo metóda vracia
@version	trieda, metóda	Označí verziu triedy alebo metódy.

Po dopísaní dokumentačných komentárov si môžeme pozrieť dokumentáciu. V okne so zdrojovým kódom triedy **`Hrac`** sa pomocou rozbaľovacieho menu v pravej hornej časti prepneme z položky **`Zdrojový kód`** do položky **`Dokumentácia`** (alebo zvolíme menu **`Nástroje`** a položku **`Prepnúť pohľad`** prípadne využijeme klávesovú skratku **`CTRL+J`**). Späť do zdrojového kódu sa dostaneme rovnakou cestou. V okne s dokumentáciou vidíme prehľadne zhrnuté všetky metódy spolu s patričnými komentármi.

Okrem dokumentačných komentárov je možné písať aj poznámky v tele metód. Tie sa píšu typicky priamo ku kódu, ktorý popisujú. Prekladač text poznámky ignoruje, musíme ju však vhodne označiť. Môžeme zvoliť:

- dve verzie jednoriadkového komentára:

```
// toto je poznamka, az do konca riadku prekladac text ignoruje
/* toto je poznamka, prekladac ignoruje text po ukoncenie poznamky */
```

- viacriadkový komentár:

```
/*
toto je poznámka na viac riadkov.
dokumentacny komentar na rozdiel od poznamky v texte zacina /**
*/
```

Písanie poznámok do textu je, hlavne v prípade komplikovaných metód, veľmi vhodné. Negeneruje sa z nich dokumentácia, avšak programátorovi výrazne uľahčujú orientáciu a pochopenie činnosti danej metódy.

ÚLOHA 2.5

Doplňte dokumentačný komentár pre metódu **`urobDlhyKrok()`**.

Nad hlavičku metódy **`urobDlhyKrok()`** dopíšeme napríklad:

```
/**
 * Hráč urobí krok o dve bunky v aktuálnom smere.
 */
```

ÚLOHA 2.6

Upravte dokumentačný komentár triedy **Hrac**. Vypíšte verziu triedy a jej autora.

Upravený dokumentačný komentár triedy **Hrac** môže vyzerať napríklad takto:

```
/**
 * Trieda predstavuje hráča.
 *
 * @author Peter
 * @version 1.0
 */
```

ÚLOHA 2.7

Preskúmajte dokumentačné okno.

Všimnite si hlavne previazanie metód pomocou živých odkazov (po kliknutí na identifikátor metódy v časti **Method Summary** sa zobrazenie presunie do časti **Method Detail**).

2.4 Ovládanie aplikácie z prostredia Greenfoot

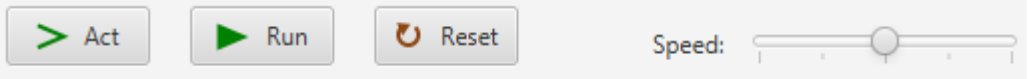
Aby sme mohli tvoriť a spúšťať algoritmy v prostredí Greenfoot, tak si musíme najskôr vysvetliť, ako sa aplikácia v prostredí Greenfoot ovláda. Každá trieda, ktorá je potomkom triedy **Actor**, obsahuje metódu **act()** (po slovensky rob, alebo konaj), ktorá je prostredím Greenfoot neustále opakovane vyvolávaná. Metóda **act()** teda bude obsahovať algoritmus, ktorý bude inštancia triedy **Actor** periodicky vykonávať. Máme pripravenú a zdokumentovanú metódu **urobDlhyKrok()**. Využime ju teraz na to, aby sme automaticky rozhýbali inštanciu triedy **Hrac**.

ÚLOHA 2.8

Upravte telo metódy **act()** triedy **Hrac** tak, aby sa zavolala metóda **urobDlhyKrok()**.

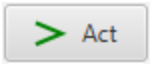
```
public void act()
{
     this.urobDlhyKrok();
}
```

Pripomeňme, že v prostredí Greenfoot sa metóda **act()** nachádza v každom potomkovi triedy **Actor**. To umožňuje elegantné ovládanie aktorov a jednoduchú tvorbu aplikácie. Na ovládanie metódy **act()** slúžia tlačidlá pod hlavnou plochou zobrazené na nasledujúcom obrázku.

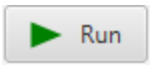


Obrázok 2.2: Ovládacie prvky aplikácie v prostredí Greenfoot

Význam jednotlivých tlačidiel je nasledujúci:



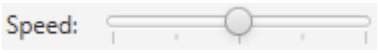
vyvolá každej inštancii triedy **`Actor`** (a **`World`**) práve jedenkrát metódu **`act()`**.



opakovane vyvoláva každej inštancii triedy **`Actor`** a **`World`** metódu **`act()`** až kým nie je vykonávanie prerušené tlačidlom 

.

vráti aplikáciu do pôvodného stavu (zruší aktuálny svet a vytvorí inštanciu naposledy vytvoreného sveta).



zrýchli alebo spomalí vyvolávanie metódy **`act()`**.

ÚLOHA 2.9

Vyskúšajte tlačidlá na ovládanie aplikácie. Vytvorte si viac inštancií triedy **`Hrac`**. Stlačte tlačidlo **`Act`** – čo sa stane? Stlačte tlačidlo **`Run`** – čo sa stane? Po prvom posune tlačidla **`Run`** stlačíme tlačidlo **`Pause`**, čo sa stane? Aký vplyv má posuvník **`Speed`** na volanie metódy **`act()`** po stlačení tlačidla **`Run`**?

Všetky inštancie by sa mali hýbať naraz o dve bunky smerom doprava. Pri stlačení tlačidla **`Act`** práve raz. Pri stlačení **`Run`** dovtedy, kým sa nestlačí **`Pause`** alebo kým nenarazia na koniec plochy. Posuvník **`Speed`** by mal viditeľne ovplyvniť rýchlosť inštancií triedy **`Hrac`** na hracej ploche.

ÚLOHA 2.10

Pridajte do triedy **`Hrac`** metódu, pomocou ktorej bude inštancia triedy **`Hrac`** chodiť do zvoleného štvorca. Zdokumentujte svoju metódu. Na pohyb a otáčanie využite vhodné metódy z predka **`Actor`**. Upravte metódu **`act()`** tak, aby inštancia triedy **`Hrac`** chodila po jej vyvolaní do štvorca. Potom svoje riešenie overte spustením aplikácie.

Vytvoríme novú metódu:

```
/**
 * Prejde štvorec 5x5 buniek
 */
public void chodDoStvorca()
{
     this.move(5);
     this.turn(90);
     Greenfoot.delay(10);
```

```
        this.move(5);
        this.turn(90);
        Greenfoot.delay(10);
        this.move(5);
        this.turn(90);
        Greenfoot.delay(10);
        this.move(5);
        this.turn(90);
        Greenfoot.delay(10);
    }
```

Následne upravíme telo metódy **`act()`**:

```
public void act()
{
    // hráč pôjde do štvorca
    this.chodDoStvorca();
}
```

ZHRNUTIE

Pred samotným písaním kódu je potrebné nájsť algoritmus na riešenie danej úlohy. Dobrý algoritmus by mal mať vlastnosti ako sú elementárnosť, determinovanosť, hromadnosť, rezultatívnosť, konečnosť a efektívnosť. Proces tvorby algoritmov nazývame algoritmizácia.

Algoritmy píšeme do metód objektov. Každá metóda sa skladá z dvoch častí – z hlavičky a tela. Hlavička začína kľúčovým slovom **`public`**, nasleduje typ návratovej hodnoty, identifikátor a parametre v zátvorkách. Telo metódy píšeme do zložených zátvoriek.

Pre lepšiu orientáciu v kóde sa používajú komentáre. Dokumentačné komentáre umožňujú na to určeným programom vygenerovať prehľadnú programátorskú dokumentáciu.

Prostredie Greenfoot sa ovláda pomocou tlačidiel (**`Act`**, **`Run`**/**`Pause`** a **`Reset`**) a posuvníka **`Speed`**. Každá trieda **`Actor`** a **`World`** definuje metódu **`act()`**, ktorá je vyvolávaná raz (pomocou tlačidla **`Act`**) alebo opakovane (pomocou tlačidla **`Run`**).

ÚLOHY NA PRECVIČOVANIE

ÚLOHA 2.A

Popíšte algoritmus, ktorým by ste navigovali cudzinca z vašej školy na vlakovú stanicu, do reštaurácie, ku vám domov.

ÚLOHA 2.B

Preskúmajte dokumentáciu triedy **`Actor`**.

ÚLOHA 2.C

Napíšte a zdokumentuje metódu **`chodDoSestuholnika()`**, pomocou ktorej prejde inštancia triedy **`Hrac`** po dráhe v tvare šesťuholníka. Na pohyb využite metódu **`urobDlhyKrok()`**.

ÚLOHA 2.D

Vytvorte metódy **`dopravaDole()`**, **`dopravaHore()`**, **`dolavaDole()`** a **`dolavaHore()`**, ktoré posunú inštanciu triedy **`Hrac`** v danom smere. Využite metódu **`setRotation()`**, preskúmajte jej činnosť v dokumentácii.

3 VETVENIE A OVLÁDANIE HRÁČA

KĽÚČOVÉ SLOVÁ

Príkaz **`if`**. Príkaz **`switch`**. Trieda **`Mur`** a **`Stena`**.

CIELE

V ďalšej časti projektu sa naučíme vetviť svoj kód pomocou príkazov neúplného vetvenia **`if`**, úplného vetvenia **`if-else`** a prostredníctvom príkazu viacnásobného vetvenia **`switch`**. Tieto príkazy využijeme na to, aby sme v projekte vedeli rozhýbať hráča prostredníctvom klávesnice. Naučíme sa používať jednoduchý logický výraz typu **`boolean`**. Tiež vytvoríme nové triedy **`Stena`** a **`Mur`** ako základ pre ďalšie rozvíjanie projektu, ktoré nám zatiaľ budú slúžiť na navigáciu hráča vo svete.

OBSAH

3.1 Neúplné vetvenie kódu

Inštancie triedy **`Hrac`** dokážu zatiaľ vykonávať v metóde **`act()`** jednoduché pohyby. Keď však narazia na koniec sveta, tak sa zastavia a nevedia ďalej pokračovať. Skúsme sa zamyslieť, ako by vyzeral algoritmus, ktorý by spôsobil, že hráč bude chodiť zľava doprava a keď narazí na stenu, tak sa otočí a pôjde sprava doľava, keď narazí na ľavú stenu tak sa otočí a bude pokračovať doprava, a tak ďalej. Vieme, že môžeme využiť metódu **`move()`**, pomocou ktorej urobí každá inštancia triedy **`Actor`** krok v smere, v ktorom je otočená. Potrebujeme teda, aby sa po dosiahnutí okraja sveta otočila o 180°. To vieme zabezpečiť metódou **`turn()`**. Kód metódy **`act()`** by teda mohol vyzerať takto:

```
public void act()
{
     this.move(1);
     this.turn(180); // toto chceme vykonať iba na konci sveta
}
```

Narazili sme na situáciu, v ktorej potrebujeme určitú časť príkazov vykonať iba pri splnení určitých **podmienok**. V našom prípade je to splnenie podmienky „inštancia triedy **`Hrac`** je na kraji sveta“.

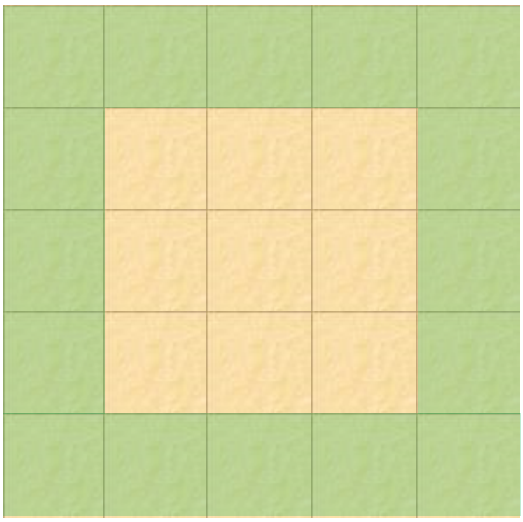
ZAPAMÄTAJTE SI!

Ak je potrebné vykonať nejaké príkazy iba v nejakom prípade, je možné využiť **neúplné vetvenie**:

```
if (podmienka) {
     // príkazy, ktoré sa vykonajú, ak je podmienka splnená
}
```

Za kľúčovým slovom **`if`** nasleduje **v povinnej zátvorke** podmienka. Podmienka musí byť vo forme **logického výrazu**. Logický výraz nadobúda iba hodnoty **`true`** (pravda) alebo **`false`** (nepravda). Telo neúplného vetvenia sa píše do zložených zátvoriek. **Príkazy uvedené v tele sa vykonajú iba vtedy, ak podmienka platí** (teda jej hodnota je **`true`**). Ak podmienka neplatí (teda jej hodnota je **`false`**), telo sa preskočí a program pokračuje vykonávaním nasledujúceho príkazu pod príkazom **`if`**.

Môžeme teda využiť neúplné vetvenie a príkaz **`turn()`** vložiť do tela príkazu **`if`**. Musíme však správne zostaviť podmienku vetvenia. Zo slovnej analýzy vyššie vyplýva, že inštancia triedy **`Hrac`** sa musí otočiť iba vtedy, ak dosiahne kraj sveta. Za kraj sveta môžeme považovať všetky bunky, ktoré sú na jeho obvode (pozri nasledujúci obrázok). Trieda **`Actor`** obsahuje metódu **`isAtEdge()`**, ktorá vracia pravdivostnú hodnotu (v dokumentácii vidíme hodnotu typu **`boolean`**). Nasledujúci obrázok zobrazuje bunky sveta s veľkosťou 5x5 buniek, v ktorých metóda **`isAtEdge()`** triedy **`Actor`** vráti hodnotu **`true`**.



Obrázok 3.1: Okrajové bunky sveta

Vytvorme inštanciu triedy **`Hrac`** a umiestnime ju do stredu sveta. Po kliknutí na inštanciu pravým tlačidlom myši vyvoláme z menu **`inherited from Actor`** metódu **`isAtEdge()`**. Pozorujme, čo sa stane.

Potom inštanciu presunieme na kraj a opäť vyvoláme túto metódu. Vidíme, že ak sme inštanciu triedy **`Hrac`** posunuli do krajných buniek, návratová hodnota metódy **`isAtEdge()`** bola **`true`**. To je presne to, čo potrebujeme. Podmienku vo vetvení teda môžeme formulovať ako volanie metódy **`this.isAtEdge()`**. Celá metóda **`act()`** bude vyzerať takto:

```
public void act()
{
     this.move(1);
     // ak som na kraji sveta, tak sa otočím
     if (this.isAtEdge()) {
          this.turn(180);
     }
}
```

S využitím neúplného vetvenia dokážeme jednoducho implementovať ovládanie hráča. Hráč sa môže pohnúť iba vtedy, ak je stlačený príslušný kláves (napr. šípka). Teda, ak je stlačený správny kláves, tak sa pohni v danom smere. Potrebujeme však zistiť, či je stlačený príslušný kláves.

Pre zistenie toho, či je stlačený daný kláves, ponúka nástroj Greenfoot metódu **`Greenfoot.isKeyDown(kláves)`**. Parameter kláves vyjadruje kláves, stlačenie ktorého chceme otestovať. Návratová hodnota je **`true`**, ak je daný kláves stlačený, **`false`** v opačnom prípade. Parameter kláves musí byť zadaný ako **reťazec**. Hodnota reťazca sa v jazyku Java uvádza do úvodzoviek **`""`**. Teda, ak chceme zistiť, či je stlačený kláves A, potom napíšeme **`Greenfoot.isKeyDown("a")`**. Vo všeobecnosti ako parameter kláves pre otestovanie stlačenia klávesu asociovaného s:

- písmenom uvedieme: **`"a"`**, **`"b"`**, **`"c"`**, **`"d"`**, **`"e"`**, **`"f"`**, **`"g"`**, **`"h"`**, **`"i"`**, **`"j"`**, **`"k"`**, **`"l"`**, **`"m"`**, **`"n"`**, **`"o"`**, **`"p"`**, **`"q"`**, **`"r"`**, **`"s"`**, **`"r"`**, **`"u"`**, **`"v"`**, **`"w"`**, **`"x"`**, **`"y"`**, **`"z"`**.
- číslom uvedieme: **`"0"`**, **`"1"`**, **`"2"`**, **`"3"`**, **`"4"`**, **`"5"`**, **`"6"`**, **`"7"`**, **`"8"`**, **`"9"`**.
- funkčným klávesom uvedieme: **`"F1"`**, **`"F2"`**, **`"F3"`**, **`"F4"`**, **`"F5"`**, **`"F6"`**, **`"F7"`**, **`"F8"`**, **`"F9"`**, **`"F10"`**, **`"F11"`** , **`"F12"`**.
- šípkou uvedieme: **`"up"`** pre šípku hore, **`"down"`** pre šípku dole, **`"left"`** pre šípku doľava, **`"right"`** pre šípku doprava.
- iným klávesom uvedieme: **`"space"`** pre medzerník, **`"tab"`** pre tabulátor a **`"backspace"`**, **`"enter"`**, **`"escape"`**, **`"shift"`** a **`"control"`** pre príslušné klávesy.

ÚLOHA 3.1

Upravte kód metódy **`act()`** v triede **`Hrac`** tak, aby sa hráč hýbal iba vtedy, keď je stlačený kláves P (ako pohyb). Zachovajte kód zodpovedný za otočenie hráča, keď dôjde na kraj sveta, ale zamyslite sa na jeho umiestnení. Kedy sa môže vykonať otočenie hráča?

Otočenie hráča môžeme vykonať iba vtedy, keď sa hýbe. Preto neúplné vetvenie kontrolujúce to, či je inštancia na okraji sveta vložíme do tela neúplného vetvenia s podmienkou testujúcou stlačenie klávesu.

```
public void act()
{
     if (Greenfoot.isKeyDown("p")) {
           this.move(1);
           // ak som na kraji sveta, tak sa otočím
           if (this.isAtEdge()) {
                 this.turn(180);
           }
     }
}
```

ÚLOHA 3.2

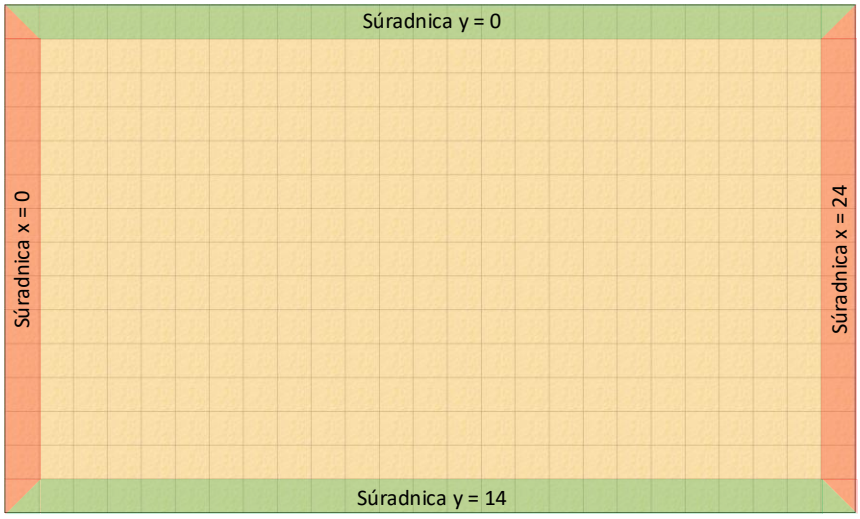
Vytvorte inštanciu triedy **`Hrac`** a umiestnite ju do stredu hracej plochy. Otvorte okno s vnútorným stavom inštancie a umiestnite ho tak, aby bolo viditeľné počas behu aplikácie. Potom spustite aplikáciu a pozorujte, ako sa menia hodnoty atribútov **`x`** a **`y`** v triede **`Hrac`**. Ako sa menia tieto hodnoty pri pohybe nahor, nadol, doľava a doprava?

Pri pohybe doprava rastie hodnota atribútu **`x`**, pri pohybe doľava hodnota tohto atribútu klesá. Hodnota atribútu **`y`** je v týchto prípadoch nemenná. Pri pohybe nahor klesá hodnota atribútu **`y`**, pri pohybe nadol hodnota tohto atribútu rastie. Hodnota atribútu **`x`** je v týchto prípadoch nemenná.

Umiestnime teraz inštanciu triedy **`Hrac`** na spodný okraj sveta. Po spustení aplikácie zistíme, že náš hráč sa pohybuje stále tam a späť. Problémom je, že hráč sa stále nachádza v bunke na okraji sveta, a teda výraz **`this.isAtEdge()`** vráti zakaždým **`true`**, čo spôsobí otáčanie dookola. Pre správne fungovanie potrebujeme rozlíšiť, na ktorom okraji sa nachádzame a podľa toho nastaviť správne otočenie hráča.

Trieda **`Actor`** definuje metódy **`getX()`** a **`getY()`**, ktoré vrátia aktuálnu pozíciu inštancie tejto triedy vo svete. Tieto atribúty reprezentujú hodnoty súradníc bunky v rámci sveta. Bunka na súradniciach [0; 0] sa nachádza v ľavom hornom rohu. Ak máme svet o veľkosti 25x15 buniek, potom vieme formulovať nasledujúce tvrdenia (pozri obrázok 3.2):

- Ak je súradnica x inštancie triedy **`Actor`** rovná 0, potom sa nachádza v bunke v stĺpci najviac vľavo.
- Ak je súradnica x inštancie triedy **`Actor`** rovná 24, potom sa nachádza v bunke v stĺpci najviac vpravo.
- Ak je súradnica y inštancie triedy **`Actor`** rovná 0, potom sa nachádza v bunke v riadku najviac hore.
- Ak je súradnica y inštancie triedy **`Actor`** rovná 14, potom sa nachádza na bunke v riadku najviac dole.



Obrázok 3.2: Vyznačenie okrajových súradníc vo svete s rozmermi 25x14 buniek

Pre správne určenie otočenia nám teda stačí zostaviť sústavu podmienok tak, aby sme skontrolovali aktuálnu hodnotu súradníc inštancie triedy **`Actor`** s hranicami nášho sveta. Na to však potrebujeme vedieť porovnať hodnoty.

ZAPAMÄTAJTE SI!

Pre porovnanie dvoch hodnôt A a B, ktoré sú rovnakého typu, je možné použiť relačné operátory:

- **`A == B`** (A je rovnaké ako B),
- **`A != B`** (A je rôzne od B),
- **`A > B`** (A je väčšie ako B),
- **`A < B`** (A je menšie ako B),
- **`A >= B`** (A je väčšie alebo rovné B),
- **`A <= B`** (A je menšie alebo rovné B) a

vytvoriť tak **logický výraz**. Návratová hodnota logického výrazu je typu **`boolean`** (teda **`true`**, ak je výsledok logického výrazu pravdivý, inak je rovná **`false`**).

Nahraďme teda nevyhovujúce vetvenie s podmienkou **`this.isAtEdge()`** štyrmi vetveniami, v ktorých nastavíme správne natočenie hráča. Začnime s horným a pravým okrajom. Do tela metódy **`act()`** doplníme:

```
// ak je hráč na hornom okraji sveta
if (this.getY() == 0) {
     // tak nastavím otočenie nadol
     this.setRotation(90);
}

// ak je hráč na pravom okraji sveta
if (this.getX() == 24) {
     // tak nastavím otočenie doľava
     this.setRotation(180);
}
```

ÚLOHA 3.3

Doplňte do tela metódy **`act()`** kód na správne natočenie hráča po dosiahnutí dolného a ľavého okraja sveta.

```
// ak je hráč na dolnom okraji sveta
if (this.getY() == 14) {
     // tak nastavím otočenie nahor
     this.setRotation(270);
}
// ak je hráč na ľavom okraji sveta
if (this.getX() == 0) {
     // tak nastavím otočenie doprava
     this.setRotation(0);
}
```

3.2 Úplné vetvenie kódu

Hra Bomberman obsahuje okrem hráčov aj steny a múry. Zaveďme ich preto aj do našej aplikácie.

ÚLOHA 3.4

Vytvorte dve nové triedy, potomkov triedy **`Actor`**. Prvá bude trieda **`Mur`** a druhá trieda bude **`Stena`**. Pripravte si v grafickom editore vhodné obrázky s veľkosťou 60x60 pixelov. Tieto obrázky potom priraďte novovytvoreným triedam.

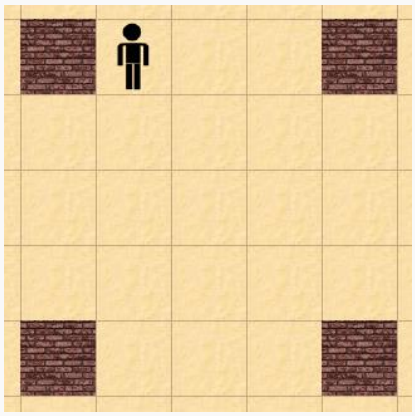
Do múru ani do steny nie je možné vojsť. Pre automatické zabránenie vstupu do bunky s inštanciou triedy **`Stena`** alebo **`Mur`** zatiaľ nemáme potrebné vedomosti, môžeme sa však naučiť inštancie týchto tried detegovať a zmeniť podľa ich prítomnosti činnosť metódy **`act()`** v triede **`Hrac`**. Budeme ich zatiaľ používať ako určité značky, na základe ktorých sa bude hráč orientovať. Upravme teda aplikáciu tak, aby hráči automaticky zatočili, ak vstúpia na bunku, ktorá obsahuje inštanciu triedy **`Mur`** alebo **`Stena`**. Na zistenie, či sa inštancia triedy **`Actor`** dotýka nejakej inštancie inej triedy je možné použiť metódu **`isTouching()`**, ktorá ako parameter preberá **triedu** inštancie, ktorej dotyk testuje. Napríklad, ak chceme zistiť, či sa hráč dotýka inštancie triedy **`Mur`**, potom napíšeme **`this.isTouching(Mur.class)`**. Všimnite si, že parameter je vždy identifikátor triedy nasledovaný **`.class`**. Metóda vráti **`true`** vtedy, keď sa grafické reprezentácie dvoch inštancií dotýkajú, inak vráti **`false`**. Pozor, nestačí, aby boli dve inštancie v susedných bunkách.

Zmeňme teda činnosť metódy **`act()`** v triede **`Hrac`** tak, aby po vstupe na políčko, na ktorom sa nachádza inštancia triedy Stena, hráč zatočil o 90° v smere hodinových ručičiek. Za kontroly dosiahnutia okraja sveta dopíšeme príkazy:

```
// ak sa hráč dotýka múru, otočí sa o 90° doprava
if (this.isTouching(Mur.class)) {
     this.turn(90);
}
```

ÚLOHA 3.5

Vytvorte štyri inštancie triedy **Mur** a jednu inštanciu triedy **Hrac** tak, ako je to zobrazené na obrázku nižšie. Odhadnite, ako sa bude hráč pohybovať? Spustite aplikáciu. Zhoduje sa Vaša predpoveď s tým, čo pozorujete?



Obrázok 3.3: Rozostavenie múrov a hráča

Hráč sa bude pohybovať do štvorca v smere hodinových ručičiek. Rohy štvorca sú určené inštanciou triedy **Mur**.

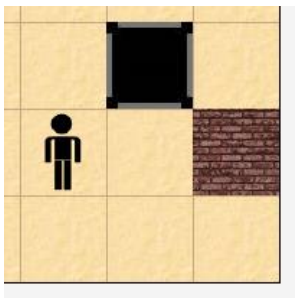
ÚLOHA 3.6

Pridajte do metódy **act()** triedy **Hrac** kód, ktorý zabezpečí, aby sa hráč otočil o 90° proti smeru hodinových ručičiek, keď vojde na políčko, kde sa nachádza inštancia triedy **Stena**.

Za kontrolu dotyku s múrom dopíšeme:

```
// ak sa hráč dotýka steny, otočí sa o 90° doľava
if (this.isTouching(Stena.class)) {
     this.turn(-90);
}
```

Umiestnime teraz inštancie tried **Hrac**, **Mur** a **Stena** v pravom dolnom rohu sveta tak, ako je to zobrazené na nasledujúcom obrázku.



Obrázok 3.4: Testovacie rozostavenie múru, steny a hráča v rohu sveta

ÚLOHA 3.7

Predpovedzte, ako sa bude pohybovať inštancia triedy **`Hrac`**. Zhoduje sa výsledok s predpoveďou?

Po dosiahnutí konca sveta sa nastaví otočenie hráča na 180°. Keďže sa však nachádza na políčku s inštanciou triedy **`Mur`** tak sa k otočeniu pridá ešte ďalších 90° v smere hodinových ručičiek. Splnené sú totiž dve podmienky.

ÚLOHA 3.8

Postupne umiestnite jednu inštanciu triedy **`Hrac`** do rohov sveta. Predpovedzte, ako sa bude táto inštancia pohybovať po spustení aplikácie. Zhoduje sa výsledok s predpoveďou?

Podobne, ako v predchádzajúcom prípade, v každom rohu sú splnené podmienky v dvoch neúplných vetveniach (kontrolujúce dosiahnutie príslušného okraja sveta). Hráč sa otočí do toho smeru, ktorý uvádza posledná podmienka, ktorá bola splnená.

Zamyslime sa, ako musíme upraviť kód, ak by sme chceli, aby sa hráč po dosiahnutí konca sveta vždy otočil nezávisle na tom, či je v príslušnej bunke na hranici sveta inštancia triedy **`Mur`** alebo **`Stena`**. Taktiež sa zamyslime, ako musíme upraviť náš kód, ak by sme chceli niektoré smery uprednostniť. Môžeme sa zamerať aj na efektívnosť algoritmu a položiť si otázku – môžu byť v jednej bunke naraz inštancia triedy **`Stena`** a **`Mur`** a je teda potrebné kontrolovať obe podmienky? Všetky tieto problémy dokážeme odstrániť s využitím úplného vetvenia.

ZAPAMÄTAJTE SI!

Ak je potrebné vykonať určité príkazy iba v nejakom prípade a v opačnom prípade vykonať iné príkazy, je možné využiť **úplné vetvenie**:

```
if (podmienka) {
      // príkazy, ktoré sa vykonajú keď podmienka platí
}
else {
      // príkazy, ktoré sa vykonajú keď podmienka neplatí
}
```

Za kľúčovým slovom **`if`** nasleduje v povinnej zátvorke podmienka vo forme logického výrazu. Telo úplnej podmienky sa píše do zložených zátvoriek. Príkazy uvedené v tele sa vykonajú iba vtedy, ak podmienka platí. Ak podmienka neplatí (teda hodnota logického výrazu je **`false`**), vykonajú sa príkazy umiestnené v tele za kľúčovým slovom **`else`**.

Kód môžeme napísať vylučovacím spôsobom. Začnime s vyriešením situácie na krajoch sveta. Aby sme jasne vyriešili situáciu v rohoch, tak povedzme, že prioritne sa hráč bude odrážať od

horizontálnych okrajov sveta (teda po dosiahnutí prvého a posledného riadka). Preto môžeme príslušný kód tela metódy **`act()`** zapísať takto:

```
// ak je hráč na hornom okraji sveta
if (this.getY() == 0) {
     // tak nastavím otočenie nadol
     this.setRotation(90);
}
// inak
else {
     // ak je hráč na pravom okraji sveta
     if (this.getX() == 24) {
          // tak nastavim otocenie doľava
          this.setRotation(180);
     }
     // inak
     else {
          // ak je hráč na dolnom okraji sveta
          if (this.getY() == 14) {
               // tak nastavím otočenie nahor
               this.setRotation(270);
          }
          // inak
          else {
               // ak je hráč na ľavom okraji sveta
               if (this.getX() == 0) {
                    // tak nastavím otočenie doprava
                    this.setRotation(0);
               }
          } // test na dolný okraj sveta
     } // test na pravý okraj sveta
} // test na horný okraj sveta
```

Vidíme, že sme zapísali akúsi „kaskádu“. Vykoná sa iba telo toho vetvenia, ktorého podmienka bude platná ako prvá. Takto sme definovali presné poradie toho, čo sa má kontrolovať najskôr. Navyše je takýto spôsob aj efektívny (spomeňme si, že na to, aby bol algoritmus dobrý, musí byť napísaný efektívne, ak je to možné). Nie je predsa potrebné testovať, či som na pravom okraji sveta, ak som na ľavom okraji sveta (ak teda neuvažujeme extrémny svet so šírkou jednej bunky). Tým, že sú testy postupne umiestňované vo vetve **`else`**, tak sa určite nebudú kontrolovať, ak niektorý z vyšších testov bude splnený. Vyhnutie sa nepotrebnému testovaniu teda zvýšilo efektívnosť algoritmu.

ÚLOHA 3.9

Doplňte kaskádu podmienok tak, aby sa kontroloval dotyk s inštanciou triedy **`Mur`** a triedy **`Stena`** iba vtedy, ak sa hráč nenachádza na okrajoch sveta. Najskôr kontrolujte inštanciu triedy **`Mur`**.

Neúplné vetvenie s podmienkou **`this.getX() == 0`** zmeníme na úplné vetvenie. Do tela **`else`** vetvy tejto podmienky dopíšeme:

```
// ak sa hráč dotýka múru, otočí sa o 90° doprava
if (this.isTouching(Mur.class)) {
     this.turn(90);
}
// inak
else {
     // ak sa hráč dotýka steny, otočí sa o 90° doľava
     if (this.isTouching(Stena.class)) {
          this.turn(-90);
     }
} // test na dotyk s múrom
```

Hráč sa pomocou naprogramovaného kódu dokáže hýbať iba dopredu (po stlačení klávesu P) a otáča sa po vojdení na bunku s inštanciou triedy **`Mur`** alebo **`Stena`**. Takéto správanie, samozrejme, nie je postačujúce, avšak pomohlo nám pochopiť princíp vetvenia a určiť prioritu istých častí kódu. Upravme preto teraz ovládanie nášho hráča tak, aby rešpektovalo šípky. Chceme, aby sa hráč pohyboval do tej strany, do ktorej smeruje stlačená šípka. Naprogramované správanie však nezahadzujme.

ÚLOHA 3.10

Vytvorte metódu pre automatický pohyb inštancie triedy **`Hrac`**. Presuňte do nej všetok kód z metódy **`act()`**. Identifikátor metódy môže byť napríklad **`pohybujSaAutomaticky`**.

Do tela nasledujúcej metódy jednoducho presunieme všetok existujúci kód z metódy **`act()`**.

```
public void pohybujSaAutomaticky()
{
}
```

S vedomosťami, ktoré máme dokážeme naprogramovať správne reakcie na stlačenie šípok tak, aby sa inštancia triedy **`Hrac`** správne pohybovala.

ÚLOHA 3.11

Vytvorte v triede **`Hrac`** metódu **`pohybujSaSipkami()`**. Naprogramujte túto metódu tak, aby sa hráč hýbal iba vtedy, keď je stlačená šípka. Pohybovať sa bude v smere stlačenej šípky. Dbajte na efektívnosť kódu. V metóde **`act()`** vyvolajte túto metódu.

```
public void pohybujSaSipkami() {
     if (Greenfoot.isKeyDown("left")) {
           this.setRotation(180);
           this.move(1);
     }
     else {
           if (Greenfoot.isKeyDown("right")) {
                 this.setRotation(0);
                 this.move(1);
           }
           else {
                 if (Greenfoot.isKeyDown("up")) {
                       this.setRotation(270);
                       this.move(1);
                 }
                 else {
                       if (Greenfoot.isKeyDown("down")) {
                             this.setRotation(90);
                             this.move(1);
                       }
                 } // else "up"
           } // else "right"
     } // else "left"
}
```

Metódu **`act()`** upravíme takto:

```
public void act()
{
     this.pohybujSaSipkami();
}
```

3.3 Viacnásobné vetvenie kódu

Počas tvorby programu sme si všimli, že sa otáča aj obrázok hráča. To však v našom prípade nevyzerá prirodzene. Lepšie by bolo, aby sme pri pohybe nahor videli jeho chrbát, pri pohybe doprava resp. doľava správny profil hráča. Poďme toto správanie naprogramovať.

ÚLOHA 3.12

Pripravte si štyri obrázky pre hráča pre pohyb smerom hore, dole, doľava a doprava. **Rozmery obrázkov nesmú presiahnuť rozmery bunky**, v našom prípade 60x60 pixelov. Pripravené obrázky vložte do projektového adresára **`images`**.

vpravo.png vlavo.png hore.png dole.png

Obrázok 3.5: Obrázky hráča smerujúceho do rôznych smerov [4]

Trieda Actor obsahuje metódu **setImage(obrazok)**, ktorá ako parameter **obrazok** preberá reťazec s názvom súboru. Tento súbor by mal byť uložený v adresári **images** a obsahovať obrázok, ktorý sa má inštancií priradiť po volaní tejto metódy. Napríklad, ak máme uložený obrázok v súbore s názvom **vpravo.png**, potom by sme pre zmenu obrázku triedy **Hrac** na tento obrázok vyvolali **this.setImage("vpravo.png");** .

Priradenie správneho obrázka vieme vykonať na základe otočenia inštancie. Jeho aktuálnu hodnotu získame pomocou metódy **getRotation()**. Hráča otáčame iba v pravých uhloch, preto bude táto metóda vracať iba hodnoty 0, 90, 180 a 270.

ÚLOHA 3.13

Vytvorte metódu **aktualizujObrazok()**, ktorá zmení obrázok hráča podľa jeho aktuálneho otočenia. Doplňte volanie tejto metódy do tela metódy **act()**.

```
public void aktualizujObrazok()
{
     if (this.getRotation() == 0) {
           this.setImage("vpravo.png");
     }
     else {
           if (this.getRotation() == 90) {
                 this.setImage("dole.png");
           }
           else {
                 if (this.getRotation() == 180) {
                       this.setImage("vlavo.png");
                 }
                 else {
                       if (this.getRotation() == 270) {
                             this.setImage("hore.png");
                       }
                 } // vľavo
           } // dole
     } // vpravo
}
```

Do tela metódy **act()** doplníme volanie:

```
this.aktualizujObrazok();
```

Pri implementácií predchádzajúcej metódy sme často opakovali rovnakú kontrolu – porovnávali sme výraz **this.getRotation()** s nejakou konkrétnou hodnotou a pri splnení podmienky sme vykonali určité príkazy. Takáto programátorská konštrukcia je bežná a je na ňu možné efektívne využiť príkaz viacnásobného vetvenia:

ZAPAMÄTAJTE SI!

Ak chcete na základe hodnoty nejakého výrazu vykonať rôzny kód, je možné využiť **viacnásobné vetvenie**:

```
switch (vyraz) {
     case hodnotaA:
          // príkazy, ktoré sa vykonajú, keď výraz == hodnotaA
          break;
     case hodnotaB:
          // príkazy, ktoré sa vykonajú, keď výraz == hodnotaB
          break;
     case hodnotaC:
          // príkazy, ktoré sa vykonajú, keď výraz == hodnotaC
          break;
     default:
          /* príkazy, ktoré sa vykonajú, keď sa výraz
             nerovná žiadnej z vyššie uvedených hodnôt */
}
```

Za kľúčovým slovom **`switch`** nasleduje v povinnej zátvorke výraz, ktorého výsledkom je celé číslo, pravdivostná hodnota alebo reťazec. Telo viacnásobného vetvenia sa píše do zložených zátvoriek. Jednotlivé vetvy sú ohraničené príkazmi **`case`** a **`break`**. Za kľúčovým slovom **`case`** nasleduje hodnota, ktorú môže výraz nadobudnúť a potom povinná dvojbodka. Ak sa hodnota testovaného výrazu zhoduje s hodnotou uvedenou za kľúčovým slovom **`case`**, vykonajú sa všetky príkazy medzi týmto **`case`** a príkazom **`break`**. Počet vetiev **`case`** nie je obmedzený, každá takáto vetva však musí v rámci toho istého príkazu **`switch`** testovať unikátnu hodnotu výrazu (žiadne dve vetvy nemôžu testovať takú istú hodnotu výrazu).
Príkaz **`switch`** môže obsahovať nepovinnú vetvu **`default`**, ktorá sa vykoná vtedy, keď ani jedna z vetiev **`case`** nebola úspešne otestovaná.

S využitím viacnásobného vetvenia teda môžeme prepísať telo metódy **`aktualizujObrazok()`** nasledovne:

```
switch (this.getRotation()) {
     case 0:
          this.setImage("vpravo.png");
          break;
     case 90:
          this.setImage("dole.png");
          break;
     case 180:
          this.setImage("vlavo.png");
          break;
     case 270:
          this.setImage("hore.png");
          break;
     default:
          // môžeme zmeniť obrázok na otáznik
}
```

ÚLOHA 3.14

Spustite aplikáciu. Vidíme, že obrázky sa nastavujú, ale otáčajú sa podľa toho, ako je otočený hráč. Vyriešte tento problém.

Každý obrázok je potrebné v grafickom editore otočiť tak, aby kompenzoval otočenie aktora v programe Greenfoot. Teda, ak ideme doľava, prevrátime príslušný obrázok dole hlavou, atď.

ÚLOHA 3.15

Vyskúšajte vytvoriť viacerých hráčov a ovládajte ich pomocou klávesnice. Čo pozorujete?

Takto implementované ovládanie nie je najlepšie navrhnuté, pretože sa všetky inštancie triedy **`Hrac`** ovládajú rovnako. Na zabezpečenie nezávislého ovládania inštancií budeme potrebovať premenné – atribúty, ktorým bude venovaná ďalšia kapitola.

ZHRNUTIE

Tok kódu je potrebné niekedy vetviť na základe podmienok. Poznáme neúplné vetvenie, úplné vetvenie a viacnásobné vetvenie.

Neúplné vetvenie začína kľúčovým slovom **`if`**, za ktorým je v povinnej zátvorke uvedený logický výraz. Ak logický výraz nadobudne hodnotu **`true`**, vykoná sa telo neúplného vetvenia uvedené v zložených zátvorkách **`{}`**.

Úplné vetvenie pridáva k neúplnému vetveniu vetvu, ktorá sa vykoná, ak logický výraz nadobudne hodnotu **`false`**. Táto vetva sa označuje kľúčovým slovom **`else`**, za ktorým nasleduje telo vetvy. Kľúčové slovo **`else`** píšeme za telo príkazu **`if`**.

Viacnásobné vetvenie začína kľúčovým slovom **`switch`**, za ktorým v povinnej zátvorke nasleduje výraz (jeho výsledkom je celočíselná alebo pravdivostná hodnota alebo reťazec). Telo príkazu **`switch`** píšeme do zložených zátvoriek **`{}`** a skladá sa z jednotlivých vetiev. Každá vetva je ohraničená kľúčovým slovom **`case`**, za ktorým nasleduje možná hodnota testovaného výrazu a dvojbodka. Vetva končí príkazom **`break;`** . Keď výraz nadobudne hodnotu, akú uvádza vetva, vykonajú sa všetky jej príkazy. Ak príkaz **`switch`** definuje vetvu **`default`**, táto sa vykoná v prípade, ak testovaný výraz nenadobudne ani jednu z hodnôt uvedených za kľúčovými slovami **`case`**.

ÚLOHY NA PRECVIČOVANIE

ÚLOHA 3.A

Vytvorte metódu, ktorá po spustení tlačidlom nechá hráča kráčať po jednej bunke až k okraju sveta. Po jeho dosiahnutí sa zmení obrázok hráča na iný obrázok.

ÚLOHA 3.B

Vytvorte metódu **`skusZmenitObrazok()`**, tak aby sa po stlačení klávesu "z" zobrazila žena, po stlačení klávesu "m" zobrazil muž, a po stlačení klávesu "d" zobrazilo dieťa. Nezabudnite si uložiť potrebné obrázky do priečinka **`images`**.

ÚLOHA 3.C

Nechajte hráča otáčať o určitý uhol po stlačení klávesu "p" smerom doprava a po stlačení klávesu "l" smerom doľava. Kód doplňte na vhodné miesto.

ÚLOHA 3.D

Upravte kód v metóde **`pohybujSaAutomaticky()`** tak, že ak hráč dosiahne okraj sveta v strede (teda súradnice [0; 7], [24; 7], [12, 0] alebo [12, 15]) bude pomocou metódy **`setLocation()`** premiestnený do stredu sveta (na súradnice 12, 7).

4 PREMENNÉ, VÝRAZY A POKROČILÉ OVLÁDANIE HRÁČA

KĽÚČOVÉ SLOVÁ

Premenné. Typy premenných. Výrazy. Aritmetické výrazy. Logické výrazy. Parametre metód. Atribúty. Lokálne premenné. Referenčná premenná. Pokročilé ovládanie hráča. Konštruktor.

CIELE

Vysvetlíme si pojem premenná, ako aj jednotlivé typy premenných. Následne sa budeme zaoberať výrazmi a operátormi – aritmetickými a logickými. Naučíme sa čo je to atribút, lokálna premenná i parameter metódy. Zadefinujeme a vytvoríme si konštruktor a vysvetlíme si čo znamená, že konštruktor alebo metóda je prekrytá. V hre Bomberman doplníme nezávislé ovládanie pre každého hráča s doplnením ďalších možností.

OBSAH

4.1 Premenné

Program spracováva dáta, resp. údaje a tieto údaje je potrebné pri spracovaní uložiť alebo presúvať. Na ukladanie dát slúžia v programe premenné. Premenné môžu byť rôzne, doteraz sme sa s nimi už stretli vo forme atribútov objektov alebo parametrov metód.

Premenné sú časti operačnej pamäte s určenou adresou v pamäti a veľkosťou pamäťového miesta. Pri premenných sa hodnota uložená na pamäťovom mieste danej premennej môže meniť, je teda premenná. Na identifikáciu, deklaráciu alebo inicializáciu premenných má Java, podobne ako každý programovací jazyk, svoje pravidlá.

- **Identifikácia premennej** znamená dať premennej identifikátor, teda meno. Identifikátor premennej typicky tvoríme podľa konvencie camelCase.
- **Deklarácia premennej** je zjednodušene povedané vytvorenie premennej v operačnej pamäti počítača vrátane určenia jej typu. Typ premennej určuje aké hodnoty môže táto premenná nadobúdať a zároveň stanovuje aké veľké miesto v pamäti premenná zaberie.
- **Inicializácia premennej** je prvé nastavenie konkrétnej hodnoty premennej (často býva súčasťou deklarácie). Ďalšie nastavenie hodnoty danej premennej označujeme ako **priradenie hodnoty premennej**.

4.2 Identifikácia premenných

Meno premennej, nazývame ho aj identifikátor, je sekvencia znakov pozostávajúca z:

- písmen,
- čísiel (nesmie ním však začínať),
- _ (znak podčiarkovník),
- $ (znak dolár).

Názov premennej musí byť jednoznačný (rozlišujú sa malé a veľké písmená) a nesmie sa zhodovať so žiadnym kľúčovým slovom jazyka Java (kľúčové slová sú napríklad **`true`**, **`class`**, **`int`**, **`if`**, **`else`** a mnohé ďalšie).

4.3 Deklarácia premenných

Premenné, ktoré sú deklarované v rámci definície triedy označujeme ako **atribúty**. Premenné deklarované v hlavičke metódy voláme **parametre** a premenné, ktoré deklarujeme vo vnútri metód sa označujú ako **lokálne premenné**. Lokálna premenná je použiteľná iba v rámci bloku (v programe je to miesto medzi dvomi zloženými zátvorkami **`{}`**), v ktorom je deklarovaná.

Deklarácia v Jave v plnej forme vyzerá nasledovne (výrazy uvedené v **`[]`** nie sú povinné):

```
[modif. prístupu] [static, final] typ nazovPremennej [= hodnota];
```

`modifikátor prístupu`	Používa sa iba pri deklarácii atribútu, určuje možnosť prístupu k atribútu pre iné triedy, napr. môže byť atribút deklarovaný ako súkromný (**`private`**), teda viditeľný iba v rámci danej triedy alebo verejný (**`public`**), teda dostupný aj ďalším triedam.
`static`	Modifikátor určujúci, že atribút patrí triede a je spoločný pre všetky jej inštancie.
`final`	Modifikátor určujúci, že premennú je možné inicializovať iba raz a jej hodnotu nie je možné ďalej meniť.
`typ`	Určuje typ premennej, napr. **`int`**, **`double`**.
`nazovPremennej`	Identifikátor atribútu. Typicky začína malým písmenom, každé ďalšie slovo začína veľkým písmenom (camelCase).

4.4 Inicializácia a priradenie hodnôt do premenných

Skôr ako je premennú môžné použiť, je potrebné inicializovať ju priradením hodnoty. Atribúty objektov sú (ak sa explicitne neinicializujú pri deklarácii) automaticky inicializované na určenú inicializačnú hodnotu. Parametre metód inicializujeme priamo pri volaní danej metódy. Lokálne premenné je potrebné inicializovať v programe. Inicializáciu premennej je možné vykonať už v rámci jej deklarácie, prípadne kedykoľvek neskôr, no vždy pred jej prvým použitím.

Do premennej môžeme ukladať iba hodnoty toho typu, s ktorým bola daná premenná deklarovaná (napr. ak je premenná deklarovaná ako celočíselná, tak do premennej ukladáme len celočíselného hodnoty a nie je do nej možné uložiť napríklad reťazec znakov).

Priradenie hodnoty do premennej vykonáme pomocou operátora **`=`** , napríklad **`a = 10`** alebo **`x = a + b`**.

4.5 Dátové typy premenných a príklady deklarácií

Každá premenná má určený svoj typ. Základné dátové typy jazyka Java sú uvedené v nasledujúcej tabuľke.

Tabuľka 4.1: Základné typy jazyka Java

Typ	MINIMÁLNA HODNOTA	MAXIMÁLNA HODNOTA	Inicial. HODNOTA
byte	-128	127	0
short	-32768	32767	0
int	-2^{31}	$2^{31} - 1$	0
long	-2^{63}	$2^{63} - 1$	0L
float	$-(2-2^{-23})\cdot 2^{127}$	$(2-2^{-23})\cdot 2^{127}$	0.0f
double	$(2-2^{-52})\cdot 2^{1023}$	$(2-2^{-52})\cdot 2^{1023}$	0.0d
char	‘\u0000‘	‘\uffff‘	‘\u0000‘
boolean	true	false	false
string (objekt)			null

Príklady deklarácie premenných:

```
int i;             //celočíselná premenná i
int c = 128;       //celočíselná premenná s inicializáciou
long x, y, z;      //niekoľko premenných rovnakého typu
float r;           //premenná r nadobúdajúca reálne hodnoty
char ch = 'A';     //znaková premenná s inicializáciou
boolean b;         //premenná nadobúdajúca hodnoty true alebo false
final int M = 10; //celočíselná konštanta s hodnotou 10
```

4.6 Výrazy a operátory

Výrazy poznáme z hlavne z matematiky alebo iných vedných disciplín, ktoré matematiku aplikujú. Aj v programovaní je niekedy potrebné zapísať zložitejšie výpočty alebo vzťahy medzi premennými zapísať prostredníctvom výrazov. **Výrazy tvoria operandy a operátory. Operátory** vykonávajú určitú operáciu a podľa toho ich členíme na:

- Aritmetické operátory, prostredníctvom ktorých sa tvoria aritmetické výrazy.
- Relačné operátory, prostredníctvom nich sa tvoria relačné výrazy.
- Logické operátory, prostredníctvom nich sa tvoria logické výrazy.

Vo výrazoch majú jednotlivé operátory resp. operácie, ktoré reprezentujú rôznu prioritu v závislosti od toho, ktorý operátor použijeme. Poradie vykonávania operácií je možné zmeniť pomocou zátvoriek (), je to rovnaké ako v matematike. Prehľad vybraných operátorov v Jave usporiadaných podľa priority (od najvyššej po najnižšiu) je uvedený v nasledujúcej tabuľke.

Tabuľka 4.2: Vybrané operátory a ich priority

Operátor	Popis
++ --	postfix inkrementácia a dekrementácia
++ -- + - ! ~ (TYP) VAL NEW	prefix inkrementácia a dekrementácia unárne plus a mínus logická negácia a bitová negácia pretypovanie vytvorenie inštancie triedy alebo poľa
* / %	násobenie, delenie, modulo
+ - +	súčet, rozdiel spájanie reťazcov
>> << >>>	bitový posun doľava, doprava, neznamienkový posun doprava
< <= > >= INSTANCEOF	relačný operátor menší a menší rovný relačný operátor väčší a väčší rovný porovnanie typov
== !=	relačný operátor rovnosti a nerovnosti
&	bitové „a“
^	bitové „xor“
|	bitové „alebo“
&&	logický operátor „a“
||	logický operátor „alebo“
=	príkaz priradenia

4.6.2 Aritmetické operátory a výrazy

Aritmetické operátoryTabuľka 4.3: Aritmetické operátory sú operátory, ktoré slúžia na aritmetické operácie a vzťahujú sa k nim konkrétne aritmetické výrazy. Aj pri aritmetických operátoroch majú jednotlivé operátory rozdielnu prioritu.

Tabuľka 4.3: Aritmetické operátory

operátor	príklad	popis
+	op1 + op2	sčítanie dvoch čísel
-	op1 - op2	odčítanie dvoch čísel
*	op1 * op2	násobenie dvoch čísel
/	op1 / op2	delenie
%	op1 % op2	modulo
-	- op	aritmetická negácia znamienka

Príklady aritmetických výrazov s deklaráciami:

```
int a, b, c;
float a, b;
a = c * (a + b);
```

```
c = a + b * c;
int c;
c = (a / b) % c;
```

Ak máme v aritmetickom výraze premenné viacerých typov, tak je výsledok výrazu nasledovný:

Tabuľka 4.4: Výsledky aritmetických výrazov

Typy hodnôt vo výraze	Typ výslednej hodnoty
`int ⊗ int`	**`int`**
`int ⊗ double`	**`double`**
`double ⊗ int`	**`double`**
`double ⊗ double`	**`double`**

Znak ⊗ reprezentuje binárny operátor +, -, * alebo /.

ZAPAMÄTAJTE SI!

Pri delení v jazyku Java je spôsob delenia daný typom operandov. Toto delenie je teda odlišné od delenia, na ktoré ste zvyknutí z matematiky. Ak v Jave delíme dve celé čísla, výsledok bude opäť celé číslo, napr. 10/8 bude rovné 1. Ak sú však jeden alebo oba operandy neceločíselné, bude výsledok tiež neceločíselný, teda 10/8.0 bude 1.25.

4.6.3 Logické operátory

Logické operátory vyjadrujú logické operácie AND, OR, NOT, XOR, ktoré sa používajú v matematike a iných vedných odboroch. Výsledok logických operácií, resp. výrazov je typu **`boolean`**. Premenná typu **`boolean`** môže nadobúdať iba hodnoty **`true`** alebo **`false`**. Význam logických operátorov a ich syntax v jazyku Java je uvedená v nasledujúcej tabuľke.

Tabuľka 4.5: Logické operátory

Operácia	syntax	význam
AND	**`x && y`**	x a súčasne y
OR	**`x || y`**	x alebo y
NOT	**`!x`**	negácia x
XOR	**`x ^ y`**	nonekvivalencia x a y

Operácie AND, OR, NOT a XOR a výsledky po jednotlivých operáciách sú nasledovné (poznáme ich z matematiky a v tabuľke sú zapísané do syntaxe jazyka Java):

Tabuľka 4.6: Výsledky operácií AND, OR, NOT, XOR

Hodnoty operandov		Výsledok operácie			
`p`	**`q`**	**`p && q`**	**`p || q`**	**`p ^ q`**	**`!p`**
false	**false**	false	false	false	true
false	**true**	false	true	true	true
true	**false**	false	true	true	false
true	**true**	true	true	false	false

4.6.4 Relačné operátory

Tieto operátory slúžia na porovnanie premenných. Často sa používajú napr. v podmienkach **if**.

Tabuľka 4.7: Operátory relačné:

operátor	význam
`x == y`	x sa rovná y
`x != y`	x sa nerovná y
`x > y`	x je väčšie ako y
`x >= y`	x je väčšie alebo rovné y
`x < y`	x je menšie ako y
`x <= y`	x je menšie alebo rovné y

Výsledok operácie s relačnými operátormi je typu **boolean** – pravda alebo nepravda (**true**, **false**).

Pri týchto výrazoch je potrebné si uvedomiť, že znak = je znak priradenia výsledku výpočtu na pravej strane do premennej na ľavej strane. Neznamená teda rovnosť. Na rovnosť sa v jazyku Java používajú dve rovná sa za sebou ==.

Príklady s relačnými operátormi:

```
int a, b;
a = 10; b = 20;
if (a < b){b = b - a;}
if (a != b){a = a - b;}

float a, b, c
a = 10.1; b = 20.5;
if (a > b){c = a;}
else c = b;

int a = 10;
boolean c;
if (a > 0){c = true;}
```

ÚLOHA 4.1

Aký je rozdiel medzi nasledovnými algoritmami?

```
1.   int a;
     boolean c;
     ...
     if(a > 0){c = true;}
     if(a < 0){c = false;}

2.   int a;
     boolean c;
     ...
     if(a > 0){c = true;}
     else {c = false;}
```

Ak sa premenná a rovná 0 nebude v prvom prípade do premennej c priradená žiadna hodnota.

4.6.5 Logické výrazy

Logický výraz je často používaným pojmom hlavne v matematike a všetkých vedných disciplínach, ktoré matematiku používajú vrátane informatiky. Využíva sa často v podmienkach. Je tvorený matematickými, ale aj logickými operáciami – AND, OR, NOT, XOR alebo operáciami porovnávania – menší, väčší, rovný, menší alebo rovný a pod. Logický výraz môže byť:

- **Jednoduchý**, napríklad: `(a && b)`, `(a || b)`, `! a`, `(a > b)`, `(a > 0)`.
- **Zložený**, ktorý je zložený z viacerých jednoduchých výrazov, napríklad: `(a > b) || (c < b)`, `(a > b) && (b > c)`, `(a + b) == 6`.

Pri vyhodnocovaní nezabudnime, že relačné operátory majú vyššiu prioritu ako logické.

ÚLOHA 4.2

Napíšte prostredníctvom logických a relačných operátorov výraz vyjadrujúci, že premenná `int a` má hodnotu patriacu do nasledovných intervalov: <-10,10), (5,142), (-11,-3) OR (1,25>.

```
int a
(a >= -10) && (a < 10)
(a > 5) && (a < 142)
((a > -11) && (a < -3)) || ((a > 1) && (a <= 25))
```

ÚLOHA 4.3

Zvoľte si hodnoty premenných x a y a dosaďte ich do výrazov. Aké budú hodnoty premenných b1 a b2 v jednotlivých prípadoch (1 a 2)?

```
1.   int x, y;
     ...
     boolean b1 = x > 0 && y == 1;
     boolean b2 = x <= 0 || y <= 0;

2.   int x, y;
     ...
     boolean b1 = (x > 0) && (y == 1);
     boolean b2 = (x <= 0) || (y <= 0);
```

Hodnoty premenných budú rovnaké v oboch prípadoch.

4.7 Pokročilé ovládanie hráča

V hre *Bomberman* sme zrealizovali ovládanie hráča pomocou šípok na klávesnici. Problém takéhoto ovládania je, že rovnako boli ovládané všetky inštancie triedy **`Hrac`**. Je teda vhodné upraviť program tak, aby bolo možné jednotlivých hráčov (inštancie tej istej triedy) ovládať nezávisle, teda pomocou iných klávesov.

Pripomeňme si, že inštanciu triedy (objekt) vytvorí konštruktor.

ZAPAMÄTAJTE SI!

Konštruktor triedy je špeciálna metóda, ktorá inicializuje inštanciu danej triedy.
Konštruktor má rovnaký názov ako trieda, nemá návratovú hodnotu a musí byť verejný, teda deklarovaný ako **`public`**. Trieda môže mať aj viac konštruktorov. Novú inštanciu triedy vytvoríme pomocou operátora **`new`**, za ktorý napíšeme volanie konštruktora triedy, napr. **`new Hrac()`**.

Doteraz sa pre triedu **`Hrac`** automaticky používal prázdny konštruktor bez kódu. Ak chceme nastaviť ovládanie hráča nezávislé pre každú inštanciu, musíme upraviť konštruktor tak, aby sme mohli klávesy zmeniť. Konštruktor môže byť

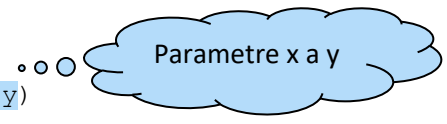
- **bezparametrický** – v zátvorke za názvom nemá žiaden parameter,
- **parametrický** – v zátvorke za názvom má vymenované parametre. Typy a názvy parametrov sa píšu za sebou a sú oddelené čiarkou. Každý parameter musí mať presne určený dátový typ.

ZAPAMÄTAJTE SI!

Parametre metódy alebo konštruktora sú v metódach alebo v konštruktore zapísané v hlavičke v zátvorkách.
`nazovMetody([typPar1 menoPar1, ..., typParN menoParN]);`
Každý parameter musí mať uvedený typ a identifikátor. Parametre slúžia na odovzdanie hodnôt do metódy či konštruktora, napríklad na inicializáciu atribútov alebo na ďalšie spracovanie v algoritme metódy. Vo vnútri metódy k parametrom pristupujeme podľa ich mena.

Napríklad hlavička metódy, ktorá má dva parametre typu **`float`** môže vyzerať nasledovne:

```
public void nastavSuradnice(float x, float y)
```



Konštruktor sa nachádza v tele triedy, pre prehľadnosť ho zvyčajne umiestňujeme pred ostatné metódy triedy. Okrem toho, že konštruktor vytvorí inštanciu triedy, môže napríklad aj priradiť hodnoty atribútom objektu.

ZAPAMÄTAJTE SI!

Atribúty – sú premenné, ktoré definujú vlastnosti inštancie, objektu. Atribúty sú deklarované v tele triedy, zvyčajne ich umiestňujeme pred metódy.
Atribúty by mali byť vždy neverejné, teda deklarované ako **`private`**.

Ako naprogramujeme konštruktor triedy **`Hrac`**? Pripomeňme si, že našim cieľom je umožniť nezávislé ovládanie jednotlivých inštancií triedy pomocou rozdielnych klávesov. Bude teda potrebné, aby každá inštancia triedy mala atribúty, v ktorých si uloží, na aké klávesy bude reagovať. Hodnoty týchto atribútov je výhodné nastaviť už pri vytváraní inštancie, teda pri volaní konštruktora. Aby konštruktor vedel, aké klávesy chceme pre danú inštanciu použiť, odovzdáme mu ich pomocou parametrov konštruktora – náš konštruktor teda bude parametrický. Na uloženie klávesov musíme ešte zadeklarovať atribúty v triede **`Hrac`**. Klávesy sú v prostredí Greenfoot označované prostredníctvom reťazcov, takže atribúty aj parametre reprezentujúce klávesy budú mať typ **`String`**.

Pridajme teda do triedy **`Hrac`** štyri atribúty (na štyri klávesy) typu **`String`**, nazvime ich podľa smeru pohybu. Vytvorme konštruktor, ktorý bude mať štyri parametre typu **`String`** reprezentujúce štyri klávesy na ovládanie pohybu daného hráča. V tele konštruktora napíšeme kód na priradenie hodnôt parametrov konštruktora do zodpovedajúcich atribútov objektu.

```
/**
 * Vytvorí inštanciu triedy Hrac.
 * Ovládanie nastaví podľa odovzdaných parametrov.
 * @param klavesHore kláves na ovládanie hráča smerom hore
 * @param klavesDole kláves na ovládanie hráča smerom dole
 * @param klavesDoprava kláves na ovládanie hráča smerom doprava
 * @param klavesDolava kláves na ovládanie hráča smerom doľava
 */
public class Hrac extends Actor {
  private String klavesHore;
  private String klavesDole;
  private String klavesDoprava;
  private String klavesDolava;

  public Hrac(String klavesHore, String klavesDole,
              String klavesDoprava, String klavesDolava)
  {
    this.klavesHore = klavesHore;
    this.klavesDole = klavesDole;
    this.klavesDoprava = klavesDoprava;
    this.klavesDolava = klavesDolava;
  }
}
```

Atribúty objektu

Parametrický konštruktor

Priradenie hodnôt parametrov do atribútov objektu

ZAPAMÄTAJTE SI!

V konštruktore využívame na prístup k atribútom daného objektu kľúčové slovo **`this`**. Kľúčové slovo **`this`** nám umožňuje získať prístup k inštancii objektu – môžeme ho využiť na volanie metód objektu, prípadne na prístup k jeho atribútom ako v tomto prípade. Využitie **`this`** nám v konštruktore umožňuje odlíšiť parametre metódy (napr. **`klavesDolava`**) od atribútov objektu (napr. **`this.klavesDolava`**) – parametre konštruktora majú totiž, podľa konvencie v jazyku Java, rovnaké meno ako zodpovedajúce atribúty objektu.

Upravme teraz metódu na pohyb hráča **`pohybujSaSipkami()`** tak, aby využívala klávesy uložené v atribútoch objektu, čím zabezpečíme nezávislé ovládanie pohybu jednotlivých inštancií triedy **`Hrac`**. Názov metódy môžeme upraviť, aby lepšie zodpovedal jej činnosti – teda **`pohybujSaKlavesmi`** (nezabudnite upraviť aj jej volanie v metóde **`act()`**).

```
public void pohybujSaKlavesmi() {
    if (Greenfoot.isKeyDown(this.klavesDolava)) {
        this.setRotation(180);
        this.move(1);
    }
    else {
        if (Greenfoot.isKeyDown(this.klavesDoprava)) {
            this.setRotation(0);
            this.move(1);
        }
        else {
            if (Greenfoot.isKeyDown(this.klavesHore)) {
                this.setRotation(270);
                this.move(1);
            }
            else {
                if (Greenfoot.isKeyDown(this.klavesDole)) {
                    this.setRotation(90);
                    this.move(1);
                }
            } // else "up"
        } // else "right"
    } // else "left"
}
```

ÚLOHA 4.4

Otestujte si vytvorený konštruktor a upravenú metódu pre ovládanie pohybu hráča vložením dvoch inštancií triedy **`Hrac`** do sveta. Pre každú inštanciu v dialógu nastavte iné klávesy pre ovládanie jej pohybu. Vyskúšajte, či je možné vložených hráčov ovládať nezávisle.

Pripomíname, že inštanciu vložíte tak, že kliknete pravým tlačidlom myši na danú triedu a z menu zvolíte položku začínajúcu slovom **new**, za ktorým nasleduje konštruktor, ktorý chcete použiť pre vytvorenie inštancie triedy. Ak zvolíte parametrický konštruktor (iný v triede **Hrac** nemáte na výber) otvorí sa dialógové okno, v ktorom zadáte hodnoty jednotlivých parametrov konštruktora. Nezabudnite, že reťazce sa v jazyku Java uvádzajú v úvodzovkách (napr. **"left"**).

4.8 Vytvorenie nového hráča vo svete a referenčná premenná

V hre Bomberman sa hráč pohybuje v takzvanej aréne. Premenujme si preto náš svet, triedu **MyWorld**, na triedu s priliehavejším názvom **Arena**. Nezabudnime pritom premenovať aj jej konštruktor.

ÚLOHA 4.5

Premenujte triedu **MyWorld** na triedu **Arena**.

```
public class Arena extends World {
    public Arena() {
        super(25, 15, 60);
    }
}
```

V hre budeme mať dvoch hráčov, ktorých je potrebné umiestniť do sveta, pričom chceme, aby ich ovládanie bolo nezávislé. Prirodzeným miestom na uloženie inštancií (objektov) hráčov je práve aréna, v ktorej sa hra odohráva. Vytvorme si preto v tejto triede dva atribúty, ktoré budú reprezentovať jednotlivých hráčov, nazvime si ich **hrac1** a **hrac2**. Do týchto atribútov si budeme ukladať inštancie triedy **Hrac** (túto triedu už máme vytvorenú), budú teda typu **Hrac** a ich deklarácia bude vyzerať nasledovne:

```
private Hrac hrac1;
private Hrac hrac2;
```

V konštruktore arény teraz môžeme vytvoriť dve inštancie triedy **Hrac** a priradiť ich do pripravených atribútov **hrac1** a **hrac2**. Novú inštanciu vytvoríme pomocou operátora **new**, za ktorým nasleduje volanie konštruktora, v našom prípade teda nasledovne:

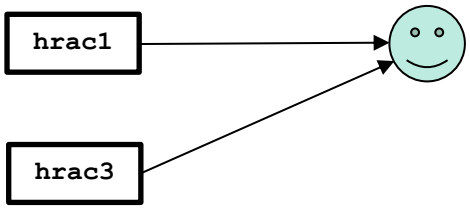
```
this.hrac1 = new Hrac("up", "down", "right", "left");
```

Atribút **hrac1** teraz obsahuje referenciu (odkaz) na inštanciu triedy **Hrac**, preto takýto atribút označujeme ako **referenčný atribút**. Referenciu môžeme chápať ako ukazovateľ na miesto v pamäti, kde sa nachádza daná inštancia objektu.

Je samozrejme možné, aby na rovnaké miesto ukazovalo aj viac premenných. Môžeme napríklad v konštruktore deklarovať novú lokálnu referenčnú premennú **hrac3** a priradiť jej rovnakú hodnotu, ako má atribút **hrac1**.

```
Hrac hrac3 = this.hrac1;
```

Je dôležité uvedomiť si, že aj lokálna premenná **hrac3** aj atribút **hrac1** teraz ukazujú na ten istý objekt. V pamäti sa teda nachádza iba jedna inštancia triedy **Hrac**.



Pomocou referenčného atribútu, či premennej sa môžeme odkazovať na objekt, na ktorý atribút, či premenná ukazuje. Môžeme takto teda priamo volať jeho metódy, napríklad **this.hrac1.move(2);** alebo využiť referenčný atribút ako hodnotu parametra metódy.

Vytvorenú inštanciu triedy **Hrac** je ešte potrebné vložiť do sveta. Trieda **World** pre vloženie objektu do sveta (seba) obsahuje metódu

```
void addObject(Actor object, int x, int y);
```

Táto metóda má tri parametre: prvým je ukazovateľ na objekt, ktorý chceme vložiť, ďalšie dva reprezentujú súradnice bunky, na ktorú bude objekt vložený. Pre vloženie hráča, ktorého máme uloženého v atribúte **hrac1** na pozíciu 0, 0 teda v konštruktore arény môžeme použiť nasledovný príkaz:

```
this.addObject(this.hrac1, 0, 0);
```

ÚLOHA 4.6

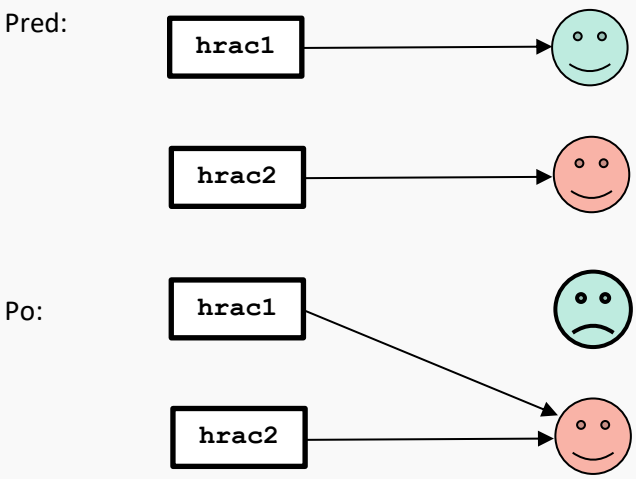
Pridajte do sveta ďalšieho hráča, napríklad pomocou referenčného atribútu **hrac2**, ktorý bude mať ovládanie pomocou kláves w - hore, s - dole, d - vpravo a a - vľavo. Hráča vložte na súradnice [24,14]. Po pridaní vyskúšajte ovládanie.

```
this.hrac2 = new Hrac("w", "s", "d", "a");
this.addObject(this.hrac2, 24, 14);
```

ÚLOHA 4.7

Čo by sa stalo, keby sme po vytvorení oboch hráčov vykonali priradenie **this.hrac1 = this.hrac2;**? Bol by to problém?

Áno, je to problém, pretože sme navždy stratili referenciu na objekt, ktorá bola pôvodne uložená v atribúte **hrac1**. Atribút **hrac1** teraz ukazuje na rovnaký objekt ako atribút **hrac2**. Pozrime si to:



4.9 Rozšírenie vlastností hráča a preťaženie konštruktorov

V aktuálnom stave našej hry sme s využitím atribútov objektov a parametrov konštruktora naprogramovali nezávislé ovládanie pohybu jednotlivých hráčov. Čo ak by sme však chceli rozšíriť schopnosti hráčov o možnosť pohybovať sa o niekoľko buniek naraz, používať teda krok s inou dĺžkou? Čo budeme musieť v hre upraviť? Jednoduchým riešením sa javí doplnenie ďalšieho atribútu, ktorý bude reprezentovať veľkosť kroku hráča a rozšírenie zoznamu parametrov existujúceho konštruktora o ďalší parameter pre zadanie veľkosti kroku pri pohybe hráča. Tento postup je funkčný, no nie veľmi flexibilný. Nie vždy totiž musíme poznať alebo chcieť zadať vlastnú hodnotu pre veľkosť kroku hráča. V takomto prípade by sme očakávali, že hráča budeme môcť vytvoriť bez zadania tejto hodnoty a ten sa bude pohybovať so štandardnou veľkosťou kroku, tak ako ostatní hráči v aréne (teda vždy o 1 bunku). Túto situáciu by bolo možné riešiť aj neskorším priradením hodnoty nového atribútu (až po vytvorení objektu), no je potrebné dodržiavať princíp, že objekt by mal byť konštruktorom vytvorený úplne a vo funkčnom stave. Z uvedeného teda vyplýva, že bude potrebné doplniť ďalší konštruktor triedy **Hrac** – taký, ktorý bude obsahovať dodatočný parameter reprezentujúci veľkosť kroku pri pohybe hráča.

Vieme však, že meno konštruktora musí byť rovnaké ako meno triedy, dostávame sa teda do situácie, že trieda bude obsahovať dve metódy (v tomto prípade konštruktory) s rovnakým názvom. V takomto prípade hovoríme o **preťažovaní metód alebo konštruktorov**. Dôležité je aby tieto metódy nemali vo svojej hlavičke rovnaké parametre (musia sa líšiť počtom a/alebo typom). Takto bude prekladač, práve na základe parametrov a ich typov, jednoznačne vedieť, ktorý konštruktor sa snažíme vyvolať. V našom prípade teda budeme mať dva konštruktory triedy **Hrac** s nasledovnými hlavičkami:

```
public Hrac(String klavesHore, String klavesDole,
            String klavesDoprava, String klavesDolava)
public Hrac(String klavesHore, String klavesDole,
            String klavesDoprava, String klavesDolava,
            int velkostKroku)
```

ÚLOHA 4.8

Rozšírte triedu **`Hrac`** o ďalší atribút typu **`int`** reprezentujúci veľkosť kroku hráča.

Za existujúce atribúty triedy pridáme riadok

```
int velkostKroku;
```

Nový konštruktor bude vo svojom vnútri obsahovať nastavenie všetkých atribútov na hodnoty odovzdaných parametrov:

```
public Hrac(String klavesHore, String klavesDole,
            String klavesDoprava, String klavesDolava,
            int velkostKroku) {
    this.klavesHore = klavesHore;
    this.klavesDole = klavesDole;
    this.klavesDoprava = klavesDoprava;
    this.klavesDolava = klavesDolava;
    this.velkostKroku = velkostKroku;
}
```

Rovnaký kód ako v už existujúcom konštruktore

Pre nastavenie prvých štyroch parametrov bude teda obsahovať presne rovnaký kód ako už doteraz existujúci konštruktor. Keďže konštruktor musí vytvoriť objekt vo funkčnom stave, musíme inicializáciu atribútu **`velkostKroku`** pridať aj do nášho prvého konštruktora a inicializovať ho na základnú hodnotu (keďže táto hodnota nebola zadaná vo forme parametra). Všimnite si, že takéto konštruktory by vyzerali takmer identicky. Je však zbytočné opakovať už existujúci programový kód a preto využijeme možnosť vyvolania vlastného konštruktora z iného konštruktora. Typicky postupujeme tak, že konštruktory s menším počtom parametrov volajú konštruktory s väčším počtom parametrov. V Jave vyvolanie konštruktora z iného konštruktora vykonáme prostredníctvom príkazu **`this()`**, v ktorom do zátvoriek uvedieme parametre konštruktora (pripomíname, že na základe ich počtu a parametrov bude počítač vedieť, ktorý konštruktor sa má zavolať). Implementácia nášho pôvodného konštruktora bude teda vyzerať nasledovne:

Volanie preťaženého konštruktora

```
public Hrac(String klavesHore, String klavesDole,
            String klavesDoprava, String klavesDolava) {
    this(klavesHore, klavesDole, klavesDoprava, klavesDolava, 1);
}
```

ÚLOHA 4.9

Upravte metódu **`pohybujSaKlavesmi()`** tak, aby rešpektovala zvýšenú veľkosť kroku. Vytvorte novú inštanciu triedy **`Hrac`** a otestujte funkčnosť programu.

V metóde **`pohybujSaKlavesmi()`** nahradíme všetky volania **`this.move(1);`** za **`this.move(this.velkostKroku);`**

ÚLOHA 4.10

Vašou úlohou je zabezpečiť, aby sa hráč pohyboval vždy po jednej bunke, no s rôznou rýchlosťou. Každý hráč môže mať inú rýchlosť. Ako pomôcku uvádzame, že rôznu rýchlosť pohybu je možné naprogramovať napríklad tak, že hráča neposuniete pri každom zistení stlačenia klávesu (zisťovanie robíme v metóde act(), takže ak držíte kláves stlačený, jeho stlačenie zistíte pri každom jej vykonaní), ale až pri každom N-tom. Čím bude N väčšie, tým bude rýchlosť hráča nižšia. Ako zadáte N? Kde si ho uložíte? Ako budete vedieť, koľko stlačení klávesu už ubehlo? (Pomôcka: je potrebné aj počítadlo).

Je potrebné vytvoriť nové atribúty **`rychlost`** a **`počítadlo`**. Atribút **`rychlost`** obsahuje hodnotu N (N stlačení). Počítadlo bude svoju hodnotu meniť tak, že na začiatku je inicializované na nulu a vždy pri volaní metódy **`act()`** sa jeho hodnota zvýši o 1. V tejto metóde budeme zároveň testovať splnenie podmienky **`this.pocitadlo == this.rychlost`**. Ak je podmienka splnená môžeme pohnúť hráčom a vynulovať hodnotu počítadla, v opačnom prípade hráčom nehýbeme.
Úplný kód programu je možné nájsť v priložených zdrojových kódoch.

ZHRNUTIE

Spoznali sme čo je premenná, základne typy premenných, aritmetické, logické a relačné operátory a výrazy. Spoznali sme rozdiel medzi atribútom, parametrom metódy a lokálnou premennou. Naučili sme sa vytvárať konštruktor a zistili sme čo to znamená preťažiť ho. Vo svete sme vytvorili dva objekty hráčov a naučili sme sa používať referenčnú premennú. V hre Bomberman sme doplnili nezávislé ovládanie hráčov a pridali sme nové atribúty pre hráča.

ÚLOHY NA PRECVIČOVANIE

ÚLOHA 4.A

Doplňte ďalšie vlastnosti hráča podľa vlastného uváženia. Čím by sa hráči mohli v hre odlišovať? Aké ďalšie vlastnosti či činnosti bude hráč potrebovať?

5 SPOLUPRÁCA OBJEKTOV TRIED

KĽÚČOVÉ SLOVÁ

Metódy s návratovou hodnotou. Lokálne premenné. Trieda **`List`**. Spolupráca objektov viacerých tried. Trieda **`Bomba`**.

CIELE

S využitím metódy s návratovou hodnotou, lokálnych premenných a triedy **`List`** (zoznam) rozšírime program tak, aby hráč nemohol prechádzať cez steny. Vytvoríme novú triedu **`Bomba`**, ktorá bude spolupracovať s triedou reprezentujúcou hráča. Inštanciu triedy **`Bomba`** vytvorí hráč. Zabezpečíme, aby bomba po vybuchnutí zmizla (zatiaľ nebude nič ničiť).

OBSAH

5.1 Zabezpečenie nepriechodnosti stien

Hra Bomberman sa odohráva v aréne kde sa nachádzajú steny, múry a iné prekážky. Zamerajme sa najprv na steny. Ak má byť hra realistická, je potrebné zabezpečiť, aby hráč cez steny nemohol prejsť. V aktuálnom stave hry to však neplatí, čo si môžeme veľmi jednoducho overiť – vytvorme inštanciu triedy **`Stena`** pred hráčom a skúsme cez ňu prejsť – zistíme, že hráč cez ňu prejde.

Na zabezpečenie nepriechodnosti steny, ako i ďalších objektov pridaných do hry v budúcnosti, je vhodné upraviť metódu realizujúcu pohyb hráča. Pred vykonaním pohybu najprv overíme, či je možné na cieľovú bunku vstúpiť. Na to budeme potrebovať zistiť súradnice cieľovej bunky.

5.1.1 Zistenie súradníc a využitie lokálnych premenných

Upravme metódu realizujúcu pohyb hráča tak, aby sme si do lokálnych premenných uložili súradnice cieľovej bunky. Pripomíname, že lokálne premenné sú premenné deklarované v tele metódy. Nezabúdajte, že lokálne premenné sú dostupné iba v rámci bloku, v ktorom sú deklarované.

Metódu **`pohybujSaKlavesmi()`** upravíme tak, že si najprv deklarujeme dve lokálne premenné **`x`** a **`y`** na uloženie súradníc, tieto premenné budú typu **`int`**. Na ich inicializáciu využijeme funkcie **`getX()`** a **`getY()`**, ktoré sú vytvorené v triede **`Actor`**, ktorá je predkom našej triedy **`Hrac`**.

```
public void pohybujSaKlavesmi() {
     //deklarujeme lokálne premené a inicializujeme ich
     int x = this.getX();
     int y = this.getY();
```

Po vykonaní týchto príkazov obsahujú premenné **`x`** a **`y`** aktuálne súradnice hráča. Cieľové súradnice zistíme na základe stlačenej klávesy a zodpovedajúceho smeru hráča. Pre smery doľava a doprava to bude takto:

```
if (Greenfoot.isKeyDown(this.klavesDolava)) {
     this.setRotation(180);
     x = x - 1;  //nastavenie súradnice x cieľovej bunky
     this.move(1);
}
else {
     if (Greenfoot.isKeyDown(this.klavesDoprava)) {
          this.setRotation(0);
          x = x + 1;  //nastavenie súradnice x cieľovej bunky
          this.move(1);
     }
...
}
```

ÚLOHA 5.1

Analogicky doplňte kód pre smery hore a dole (budete teda meniť hodnotu lokálnej premennej **`y`**).

```
if (Greenfoot.isKeyDown(this.klavesHore)) {
     this.setRotation(270);
     y = y - 1;  //nastavenie súradnice y cieľovej bunky
     this.move(1);
}
else {
     if (Greenfoot.isKeyDown(this.klavesDole)) {
          this.setRotation(90);
          y = y + 1;  //nastavenie súradnice y cieľovej bunky
          this.move(1);
     }
```

5.1.2 Metóda s návratovou hodnotou

Po získaní súradníc cieľovej bunky môžeme pristúpiť k overeniu toho, či je na danú bunku možné hráča presunúť. Vytvoríme si novú metódu, nazvime ju napríklad **`mozeVstupit()`**, ktorej úlohou bude zistiť, či hráč na danú bunku môže vstúpiť alebo nie. Metóda teda bude potrebovať vstupné parametre, ktoré budú obsahovať súradnice bunky, ktorú chceme otestovať na možnosť vstupu. Metóda vráti logickú hodnotu **`true`** alebo **`false`** (teda typ **`boolean`**), ktorá bude indikovať možnosť alebo nemožnosť vstupu do bunky. Nateraz predpokladajme, že do každej bunky je možné vstúpiť, nech teda metóda vždy vráti hodnotu **`true`**.

ZAPAMÄTAJTE SI!

Návratovú hodnotu metódy priraďujeme prostredníctvom kľúčového slova **`return`**, za ktorým nasleduje hodnota, ktorú daná metóda vráti. Po vykonaní tohto príkazu sa vykonávanie metódy končí.

Metódu **mozeVstupit()** teda môžeme implementovať takto:

```
public boolean mozeVstupit(int x, int y)
{
     return true;  // zatiaľ vždy vrátime true a umožníme vstup
                   // príkazom return ukončíme vykonávanie metódy
}
```

ÚLOHA 5.2

Upravte metódu zabezpečujúcu pohyb hráča tak, aby ste pred zmenou jeho polohy overili možnosť vstupu do cieľovej bunky.

V metóde sa aktuálne nachádzajú štyri vetvy zodpovedajúce štyrom smerom pohybu. V predošlej úlohe sme zaviedli lokálne premenné, ktoré obsahujú súradnicu ďalšej pozície hráča. Pohyb hráča nebudeme vykonávať v rámci každej vetvy, no presunieme ho až na koniec metódy. Presun hráča však vyvoláme iba v prípade, že nám metóda **mozeVstupit()** pre súradnice novej polohy vráti hodnotu **true**:

```
// Po spracovaní klávesov máme v lokálnych premenných x a y uložené
// súradnice novej polohy hráča.
// Otestujeme možnosť vstupu do určenej bunky
if (this.mozeVstupit(x, y)) {
     this.setLocation(x, y); //presun do bunky na súradniciach x, y
}
```

Všimnite si, že sme namiesto metódy **move()** použili metódu **setLocation()** triedy **Actor**, ktorá umožňuje umiestniť hráča na určené súradnice.

Pokračujme ďalej implementáciou správnej funkčnosti metódy **mozeVstupit()**. Preskúmajme triedu **World** a pokúsme sa nájsť atribút či metódu, ktorú by sme na tento účel mohli použiť. Vidíme, že trieda **World** neobsahuje žiadny test na to, či sa v danej bunke nachádza nejaký iný aktor. Pri dôkladnejšom preskúmaní však zistíme, že svet dokáže poskytnúť zoznam (inštanciu triedy **List**) aktorov, ktorí sa nachádzajú v bunke na zadaných súradniciach (vo svete totiž môže byť na tej istej bunke súčasne aj viac aktorov).

5.1.3 Použitie zoznamu objektov – trieda List

S triedou **List** sme sa ešte nestretli, táto trieda predstavuje zoznam prvkov. Jej úloha je podobná, ako úloha nákupného zoznamu alebo zoznamu študentov v triednej knihe – dokáže evidovať inštancie určenej triedy (nákupný zoznam eviduje párky, tresku, rožky, kofolu; zoznam študentov eviduje Adama, Borisa, Cecíliu, Dušana). Zoznamom sa budeme podrobnejšie venovať v nasledujúcich kapitolách. Teraz si vystačíme s tromi poznatkami:

- Trieda **List** je už naprogramovaná v inom súbore. Ak chceme túto triedu použiť, musíme v našom programe zapísať, kde sa táto trieda nachádza a oznámiť, že ju budeme využívať. Zapíšeme to tak, že na začiatok súboru, v ktorom chceme zoznam použiť napíšeme **import java.util.List**.

- Zoznam je objekt ako každý iný. Obsahuje metódy, ktoré môžeme vyvolávať. V našom prípade nás zaujíma, či sa na bunke nič nenachádza – na to môžeme použiť metódu zoznamu **`isEmpty()`**, ktorá vráti **`true`**, ak je zoznam prázdny.
- Aby sme mohli vytvoriť premennú typu zoznam, tak za názov triedy List musíme do ostrých zátvoriek **`< >`** uviesť, aké objekty sa budú v zozname nachádzať, teda inštancie akej triedy zoznam eviduje. Keďže chceme pracovať so zoznamom stien, bude naša premenná typu **`List<Stena>`**.

S týmito poznatkami implementujme metódu **`mozeVstupit()`** v triede **`Hrac`**.

```
public boolean mozeVstupit(int x, int y) {
    // Najskôr získame svet a uložíme si ho do lokálnej premennej
    World svet = this.getWorld();
    // Svet požiadame, aby nám vrátil zoznam stien,
    // ktoré sa nachádzajú na daných súradniciach
    List<Stena> steny = svet.getObjectsAt(x, y, Stena.class);
    // Nakoniec sa zoznamu môžeme opýtať, či je prázdny.
    // Ak áno, môžeme vstúpiť na danú bunku, návratová hodnota
    // metódy je teda rovná výsledku volania steny.isEmpty()
    return steny.isEmpty();
}
```

ÚLOHA 5.3

Upravte metódu **`mozeVstupit()`** tak, aby hráč reagoval aj na múry a nemohol cez ne prejsť.

Budeme postupovať analogicky ako pri stenách. Vytvoríme a inicializujeme lokálnu premennú obsahujúcu zoznam múrov nachádzajúcich sa v danej bunke. Do bunky je možné vstúpiť vtedy, ak sa v nej nenachádza žiadna stena a zároveň žiadny múr.

```
public boolean mozeVstupit(int x, int y) {
     // Najskôr získame svet a uložíme si ho do lokálnej premennej
     World svet = this.getWorld();
     // Svet požiadame, aby nám vrátil zoznamy stien a múrov,
     // ktoré sa nachádzajú na daných súradniciach
     List<Stena> steny = svet.getObjectsAt(x, y, Stena.class);
     List<Mur> mury = svet.getObjectsAt(x, y, Mur.class);
     // Ak bunka neobsahuje ani jeden múr a zároveň ani jednu stenu,
     // môžeme na ňu vstúpiť
     return steny.isEmpty() && mury.isEmpty();
}
```

5.2 Vytvorenie triedy Bomba a spolupráca s triedou Hrac

V objektovo orientovanom programovaní sa vytvárajú aplikácie z tried a výsledná aplikácia je založená na vzájomnej spolupráci inštancií rôznych tried. V praxi sa na začiatku tvorby programu urobí analýza celého projektu a určia sa vzájomné interakcie inštancií jednotlivých tried.

5.2.1 Trieda reprezentujúca bombu

Hra Bomberman by nebola dobrou hrou ak by neobsahovala to podstatné, čo ju vystihuje – bombu. Bomberman predsa hlavne kladie bomby. Je vhodné si najprv rozmyslieť ako bude bomba fungovať a aké bude mať vlastnosti, napr. kedy vznikne, kedy zanikne (vybuchne), akú bude mať silu a podobne. Následne bude potrebné naprogramovať hráča tak, aby vedel bomby umiestňovať v aréne.

ÚLOHA 5.4

Vytvorte novú triedu **Bomba**, navrhnite jej atribút, ktorý bude reprezentovať silu výbuchu. Vytvorte parametrický konštruktor a inicializujte atribúty objektu.

Vytvoríme novú triedu **Bomba** s atribútom **sila** a parametrickým konštruktorom.

```
public class Bomba extends Actor {

     private int sila;

    /**
     * Vytvorí bombu s danou silou.
     */
     public Bomba(int sila) {
           this.sila = sila;
    }
}
```

Pri voľbe obrázku, ktorý bude reprezentovať bombu nezabudnite, že jeho rozmery nemôžu byť väčšie, ako sú rozmery bunky, teda 60x60 pixelov.

5.2.2 Položenie bomby hráčom

Aby sme hráča naučili klásť bomby je potrebné doplniť triedu **Hrac** o atribúty **String klavesBomba** (nastaví kláves na položenie bomby) a **int silaBomb** (určí silu bomby). Obidva atribúty inicializujeme vo vhodnom konštruktore (nezabudnite upraviť vytváranie inštancií triedy **Hrac** v konštruktore triedy **Arena**).

Na položenie bomby musia byť splnené určité pravidlá. Musí byť stlačený príslušný kláves a na políčku, na ktoré chceme položiť bombu sa nesmie nachádzať žiadna iná bomba. Hráč môže byť limitovaný aj obmedzeným počtom bômb, ktorý sa bude v priebehu hry meniť. Je vhodné, aby sme dodržiavanie týchto pravidiel overovali spoločne v jednej metóde **mozePolozitBombu()** s návratovou hodnotou typu **boolean**.

ÚLOHA 5.5

Doplňte do triedy **Hrac** bezparametrickú metódu **mozePolozitBombu()** s návratovou hodnotou typu **boolean**, ktorá vráti príznak, či je možné položiť bombu na políčko, kde aktuálne stojí hráč. Bombu je možné položiť vtedy, keď je stlačený príslušný kláves a keď sa na políčku nenachádza iná bomba.

```
public boolean mozePolozitBombu()
{
     if (!Greenfoot.isKeyDown(this.klavesBomba)) {
          // ak nie je stlačený kláves, metóda ihneď skončí
          return false;
     }

     // je stlačený kláves, pole musí byť prázdne
     int x = this.getX();
     int y = this.getY();
     World svet = this.getWorld();
     List<Bomba> bomby = svet.getObjectsAt(x, y, Bomba.class);
     return bomby.isEmpty();
}
```

V metóde **act()** pridáme novú podmienku využívajúcu metódu **mozePolozitBombu()**. Ak podmienka platí vytvoríme nový objekt bomba a vložíme ho do sveta na súradnice, na ktorých sa aktuálne nachádza hráč:

```
Bomba bomba = new Bomba(this.silaBomb); // vytvoríme objekt bomba
World svet = this.getWorld(); // získame svet
svet.addObject(bomba, this.getX(), this.getY()); // vložíme do sveta
```

5.2.3 Vybuchnutie bomby po určitom čase

Aktuálne už teda vieme do sveta umiestniť bombu, no bomba zatiaľ nevybuchuje. Na zabezpečenie vybuchnutia bomby po uplynutí určitého času budeme potrebovať časovač. Po vyčerpaní času bomba vybuchne a odstráni sa zo sveta. V triede **Bomba** teda doplníme ďalší atribút **casovac** a inicializujeme ho v konštruktore bomby (nezabudnite upraviť vytváranie bomby pri stlačení príslušného klávesu v triede **Hrac**). V metóde **act()** v triede **Bomba** budeme upravovať hodnotu časovača tak, že pri každom volaní tejto metódy znížime jeho hodnotu o 1. Následne s využitím neúplného vetvenia vyhodnotíme, či časovač nedosiahol hodnotu 0 a v kladnom prípade necháme bombu vybuchnúť a zmiznúť.

Na to, aby bomba zmizla, musíme ju odstrániť z nášho sveta. Na odstránenie objektov ponúka svet metódu **removeObject(Actor object)**. Parametrom tejto metódy je objekt, ktorý bude zo sveta odstránený, v našom prípade to bude bomba sama, takže využijeme kľúčové slovo **this**. Kód metódy **act()** v triede **Bomba** bude teda vyzerať takto:

```
this.casovac = this.casovac - 1;
if (this.casovac == 0) {
     // bomba vybuchla, odstránime ju zo sveta
     World svet = this.getWorld();
     svet.removeObject(this);
}
```

5.2.4 Obmedzenie počtu aktívnych bômb

Rozšírme teraz hru o možnosť obmedzenia počtu bômb, ktoré môže každý hráč použiť. Riešenie tejto úlohy môžeme rozdeliť medzi spolupracujúce objekty typu **Hrac** a **Bomba**. Hráč bude evidovať počídalo, ktoré pri položení bomby skontroluje (musí byť väčšie ako nula) a následne zníži jeho hodnotu. Po vyčerpaní daného počtu už nebude hráč môcť položiť ďalšiu bombu. Aby hráč mohol zvýšiť hodnotu tohto počítadla (a teda mohol položiť ďalšiu bombu), musí sa dozvedieť o výbuchu bomby. Samotné zistenie, či bomba vybuchla nemusí byť v triede **Hrac** naprogramované. Toto zisťovanie by totižto z tejto triedy bolo pomerne pracné (hráč môže mať aj viac bômb, každá môže vybuchnúť inokedy – hráč by teda musel mať pomerne náročnú evidenciu bômb). Inštancie triedy **Bomba** však vedia veľmi jednoducho zistiť kedy vybuchnú. Stačí, aby inštancia triedy **Bomba** pri svojom výbuchu notifikovala inštanciu triedy **Hrac** . V triede **Hrac** teda naprogramujeme reakciu na výbuch bomby ako metódu, ktorú pri svojom výbuchu vyvolá práve vybuchujúca inštancia triedy **Bomba**.

Implementujme vyššie popísanú spoluprácu objektov. V triede **Hrac** doplníme ďalší atribút **pocetBomb** a doplníme ho aj do konštruktora, v ktorom inicializujeme jeho hodnotu. Metóda **mozePolozitBombu()** musí byť rozšírená o kontrolu, ktorá vyhodnotí, či má hráč dostatočný počet bômb. Podmienka bude teda zložená – ak nie je stlačený kláves alebo hráč nemá dostatočný počet bômb, potom hráč nemôže položiť bombu. Ak je možné položiť bombu, potom nesmieme zabudnúť na upravenie hodnoty atribútu **pocetBomb** – jeho hodnotu znížime o 1.

```
if (this.mozePolozitBombu()) {
     Bomba bomba = new Bomba(this.silaBomb, 30);
     World svet = this.getWorld();
     svet.addObject(bomba, this.getX(), this. getY());
     this.pocetBomb = this.pocetBomb - 1;
}
```

Pokračujme tým, že hru upravíme tak, aby obmedzenie na počet bômb platilo iba pre súčasne existujúce bomby, ktoré hráč položil. Ak niektorá z bômb vybuchne môže hráč opäť položiť ďalšiu bombu, no ich počet nikdy nemôže prekročiť určený počet bômb.

Aby bolo možné danú funkciu implementovať, je potrebné vedieť určiť, ktorý hráč danú bombu položil. V triede **Bomba** si preto doplníme referenčný atribút s názvom **vlastnik**, tento atribút bude typu **Hrac**. Nezabudneme upraviť aj konštruktor triedy **Bomba**, v ktorom tento atribút inicializujeme. Pri vytvorení bomby teda hráč poskytne (ako parameter konštruktora triedy Bomba) seba ako vlastníka práve vytváranej bomby.

```
Bomba bomba = new Bomba(this, this.silaBomb, 30);
```

Po vybuchnutí bomby je potrebné upozorniť hráča, že ním položená bomba už vybuchla a bola zo sveta odstránená. Hráč si teda môže upraviť svoje počítadlo bômb. Do triedy **Hrac** doplníme metódu **vybuchlaBomba()**, v tele ktorej zvýšime hodnotu atribútu **pocetBomb** o 1.

```
public void vybuchlaBomba(Bomba bomba) {
     this.pocetBomb = this.pocetBomb + 1;
}
```

Po vybuchnutí bomby (v metóde **act()** triedy **Bomba**, po splnení podmienky, že časovač má hodnotu nula) zistíme jej vlastníka (atribút **vlastnik**) a ešte predtým, ako ju odstránime zo sveta, vyvoláme vlastníkovi metódu **vybuchlaBomba()**.

```
this.vlastnik.vybuchlaBomba(this);
```

ZAPAMÄTAJTE SI!

Všimnite si, že neprirađujeme hodnotu počítadla bômb priamo z objektu typu **Bomba**, no využívame volanie metódy objektu typu **Hrac**, ktorý si následne sám upraví hodnotu svojho vlastného atribútu.
Priame priradenie hodnoty do atribútu **pocetBomb** objektu **Hrac** ani nie je možné, pretože tento atribút je správne deklarovaný ako **private** – teda prístupný iba z triedy **Hrac**. Deklarovanie atribútu ako **public** a následný priamy prístup k atribútu danej triedy z inej triedy by bol v príkrom rozpore so základnými princípmi objektovo orientovaného programovania a nikdy sa nepoužíva!
Ďalším dôvodom využitia metód triedy **Hrac** je fakt, že trieda **Hrac** môže na vybuchnutie bomby reagovať rôznym spôsobom, nie iba jednoduchou zmenou hodnoty atribútu **pocetBomb** – môže napríklad upraviť svoje správanie pri ukladaní bômb a podobne. Je preto správne reakciu na vzniknutú situáciu ponechať v pôsobnosti tejto triedy a reakciu na vybuchnutie bomby realizovať prostredníctvom vyvolania metódy **vybuchlaBomba()**, ktorú implementuje trieda **Hrac**.

ÚLOHA 5.6

Rozšírte hru tak, aby bol výbuch bomby sprevádzaný zvukovým efektom. Zvuk si môžete nahrať alebo stiahnuť z internetu. Nájdite príkaz na prehratie zvuku v dokumentácií triedy **Greenfoot**.

Zvuk je možné prehrať pomocou metódy **Greenfoot.playSound("zvukovySubor.wav")**. Parametrom metódy je názov zvukového súboru vo formáte **wav**, ktorý by mal byť uložený v priečinku **sounds**.

ZHRNUTIE

Naučili sme sa používať lokálne premenné a definovať metódu s návratovou hodnotou. Prostredníctvom triedy **`List`** sme zabezpečili nepriechodnosť stien a múrov pre hráča. Vytvorili sme novú triedu **`Bomba`**, ktorej vlastníkom je hráč, ktorý bombu vytvára. Zabezpečili sme, aby bomba po určitom čase vybuchla. Spoluprácou objektov typu **`Bomba`** a **`Hrac`** sme zabezpečili, aby hráč mohol položiť iba obmedzený počet bômb.

ÚLOHY NA PRECVIČOVANIE

ÚLOHA 5.A

Skúste si nahrať vlastný zvuk a použite ho napríklad pri položení bomby. Zvuky môžete nahrávať aj v prostredí Greenfoot. (**`Nástroje | Show sound recorder)`**.

ÚLOHA 5.B

Skúste porozmýšľať ako by bomba zničila aj samotného hráča. Čo je potrebné zmeniť, dopracovať.

6 DEDIČNOSŤ A CYKLUS FOR

KĽÚČOVÉ SLOVÁ

Dedičnosť. super. Pretypovanie. Cyklus s pevným počtom opakovaní.

CIELE

V tejto kapitole sa budeme zaoberať dedičnosťou. Vytvoríme predka pre triedy **`Stena`** a **`Mur`**. Potom vytvoríme testovaciu arénu ako potomka triedy **`Arena`**. Ďalej sa v rámci tejto časti budeme venovať cyklu, ktorý sa vykoná stanovený počet krát a naučíme sa vnárať takéto cykly do seba.

OBSAH

6.1 Dedičnosť

Spomeňme si na úplný úvod, kde sme sa zoznamovali s prostredím Greenfoot. Jeden z pojmov, ktoré sme spomenuli, bola dedičnosť. Hovorili sme, že potomok preberá všetky vlastnosti svojho predka – stáva sa ním (štvorec je geometrickým útvarom, pes je zviera, saxofonista je hudobník). Poďme sa pokúsiť nájsť spoločné vlastnosti triedam, ktoré sa v našom svete už nachádzajú: **`Hrac`**, **`Bomba`**, **`Mur`** a **`Stena`**. Hráč je na rozdiel od ostatných tried ovládaný človekom pomocou klávesnice a dokáže klásť bomby – takéto vlastnosti nemajú inštancie žiadnej inej triedy. Inštancie triedy **`Bomba`** vybuchujú a ničia svoje okolie – to nevedia ani múry ani steny. Múry a steny zatiaľ nič nevedia, avšak vieme nájsť ich spoločnú vlastnosť – hráč do nich nemôže vstúpiť. Múr aj stenu môžeme považovať za prekážku – je preto vhodné zaviesť spoločného predka.

ZAPAMÄTAJTE SI:

Ak chceme definovať vzťah dedičnosti medzi dvomi triedami, tak v hlavičke triedy potomka použijeme za jeho identifikátorom kľúčové slovo **`extends`**, za ktorým nasleduje identifikátor triedy, ktorú potomok dedí:

```
public class Stena extends Prekazka { ...
public class Mur extends Prekazka { ...
```

Predok môže mať akýkoľvek počet potomkov, avšak potomok má práve jedného predka. Predok všetkých tried v jazyku Java je trieda **`Object`**. Graficky vzťah dedičnosti medzi triedami vyjadrujeme pomocou šípky  , ktorá smeruje od potomka k predkovi. Potomok preberá všetky vlastnosti a metódy svojho predka. Predok však nemôže využívať vlastnosti potomka (predok nevie o svojich potomkoch).

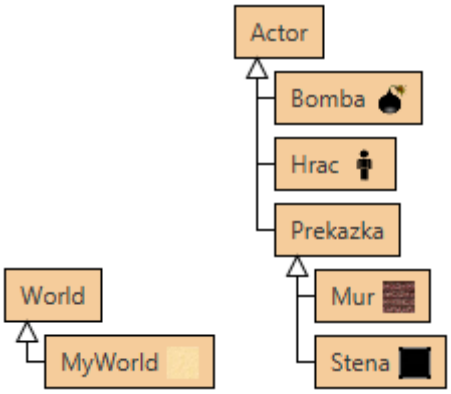
ÚLOHA 6.1

Vytvorte triedu **`Prekazka`**. Aká trieda je predkom triedy **`Prekazka`**? Upravte hlavičky tried **`Stena`** a **`Mur`** tak, aby boli potomkami triedy **`Prekazka`**.

Predok triedy **`Prekazka`** je trieda **`Actor`**.

```
public class Stena extends Prekazka { ...
public class Mur extends Prekazka { ...
```

Všimnime si, ako sa zmenila hierarchia tried v nástroji Greenfoot.



Obrázok 6.1: Hierarchia tried projektu Bomberman po zavedení predka `Prekazka`

Po zavedení spoločného predka je možné správať sa k inštanciám triedy **`Stena`** aj inštanciám triedy **`Mur`** jednotne. Zatiaľ s nimi pracujeme napr. v triede **`Hrac`** v metóde **`mozeVstupit()`**, kde osobitne testujeme prítomnosť stien a múrov na danej bunke.

ÚLOHA 6.2

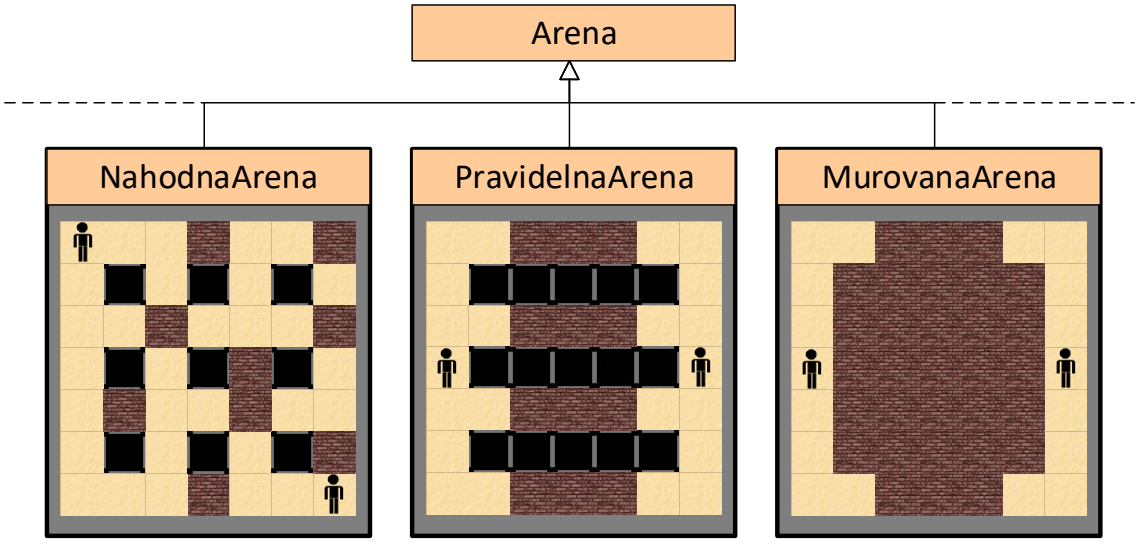
Po pridaní predka **`Prekazka`** je možné jednoduchšie otestovať, či môže hráč vstúpiť na danú bunku. Svet vo svojej metóde **`getObjectsAt()`** požaduje ako tretí parameter triedu, ktorú má na danej bunke hľadať. Keďže sú aj steny aj múry prekážkami, je možné sa k nim správať jednotne. Upravte metódu **`mozeVstupit()`** v triede **`Hrac`** tak, aby ste využili iba jediný zoznam prekážok.

V metóde nahradíme dva zoznamy jediným zoznamom prekážok, ktorý Greenfoot naplní aj inštanciami triedy **`Stena`** aj inštanciami triedy **`Mur`**, nakoľko sú to potomkovia triedy **`Prekazka`**. Potom stačí testovať, či je prázdny tento jediný zoznam. Dotknuté riadky by mohli vyzerať takto:

```
List<Prekazka> prekazky = svet.getObjectsAt(x, y, Prekazka.class);
return prekazky.isEmpty();
```

6.2 Vytvorenie potomka Arena

Poďme preskúmať arénu. Hru Bomberman je možné hrať na rôznych mapách – v rôznych arénach. Tie môžu mať rôzne rozloženie stien, múrov či štartovacie pozície hráčov a rôzne vylepšenia dostupné iba v konkrétnej aréne. Spomeňme si, že pre zastrešenie spoločnej funkčnosti je vhodné použiť dedičnosť tried a špecifiká nechať na potomkov triedy. Tento princíp využijeme aj tu. Predok **`Arena`** bude definovať všetku potrebnú funkčnosť a potomkovia sa budú starať o konkrétne rozloženie sveta. Hierarchiu tried si môžeme predstaviť takto:



Obrázok 6.2: Hierarchia tried tvoriacich arény

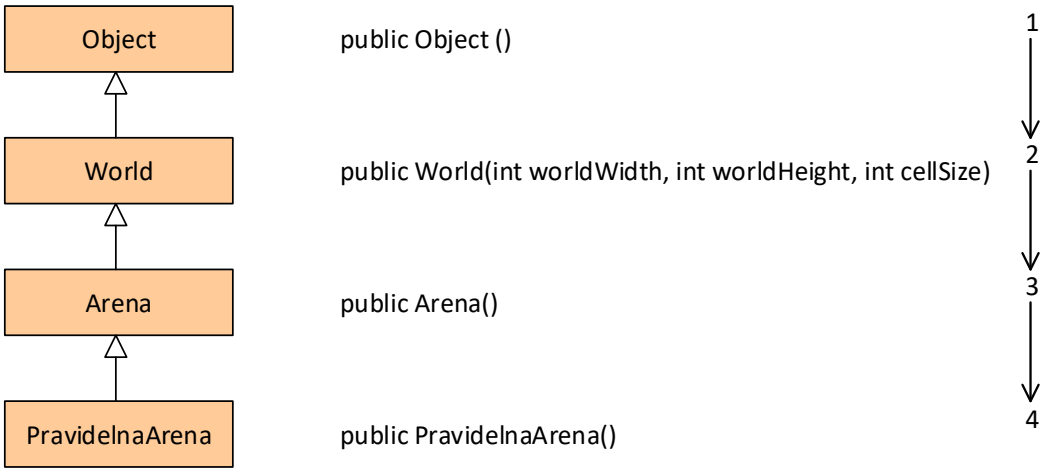
Vytvorenie sveta môžeme urobiť v niekoľkých krokoch:

1) Vytvorenie prázdneho sveta s určitým rozmerom. O toto sa postará predok **`Arena`** tak, ako doteraz.
2) Vygenerovanie stien. Steny určujú prechodnosť arénou. Typicky sa generujú v pravidelných útvaroch. Prípustné sú však aj arény, ktoré steny neobsahujú.
3) Vygenerovanie múrov. Múry sa nesmú prekrývať so stenami. Múry, podobne ako steny, môžu byť generované v pravidelných útvaroch alebo nemusia byť generované vôbec.
4) Vygenerovanie hráčov na správnych pozíciách. Hráči sa nesmú vygenerovať ani na stenách a ani na múroch.

Jednotlivé arény by mali svoje rozloženie stien, múrov a hráčov vytvoriť ihneď pri svojom vzniku. Vieme, že na inicializáciu stavu inštancie ihneď po vzniku slúži špeciálna metóda – konštruktor. Zamyslime sa, v akom poradí musia byť volané konštruktory, ak chceme vytvoriť napríklad pravidelnú arénu:

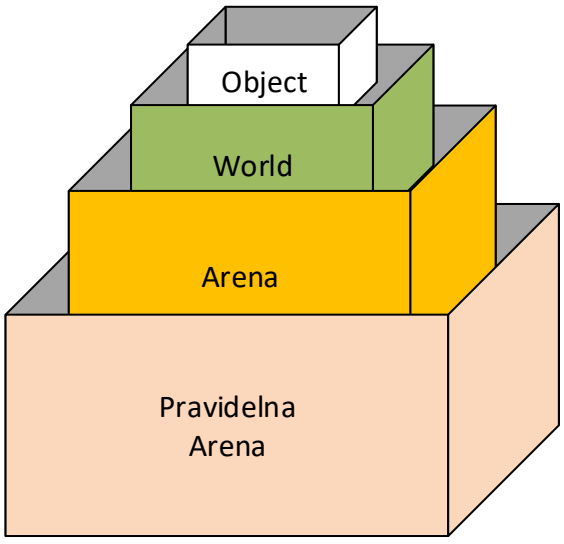
- Aby sme mohli do pravidelnej arény vložiť steny, múry a hráčov, musí už existovať aréna, ktorá bude mať dané rozmery – musíme inicializovať predka **`Arena`**.
- Aby mohla vzniknúť aréna, musí existovať svet. Vo svete sú rozložené bunky do daného počtu riadkov a stĺpcov a majú určenú veľkosť – musíme inicializovať predka **`World`**.
- Aby mohol vzniknúť svet, musí sa inicializovať ako objekt – inicializácia predka **`Object`**.

Pozrime sa na odrážky vyššie v opačnom poradí a sledujme nasledujúci obrázok: Inicializujeme objekt (1), ktorý sa skladá z daného počtu riadkov a stĺpcov buniek s určitým rozmerom (2), ktoré môžeme považovať za arénu (3), ktorá do týchto buniek pravidelne rozmiestni steny, múry a hráčov podľa daného vzoru (4).



Obrázok 6.3: Poradie volania konštruktorov triedy `PravidelnaArena`

Na akúkoľvek hierarchiu tried sa môžeme pozerať cez jednotlivé vrstvy (podobne, ako na cibuľu, matriošku alebo na krabice). Na to, aby bolo možné inicializovať danú vrstvu (krabicu), musia byť inicializované všetky vrstvy (krabice), ktoré sú v nej.



Obrázok 6.4: Grafické znázornenie triedy `PravidelnaArena` a jej zdedených častí

Keď vytvárame inštanciu triedy, vieme, že sa volá jej konštruktor. Ak teda vytvoríme inštanciu triedy **`PravidelnaArena`**, bude sa volať jej konštruktor (vytvára sa jej krabica). My však musíme zabezpečiť, aby sa vytvorili aj všetky jej súčasti (krabice predkov). Navyše, toto musíme zabezpečiť prednostne.

ZAPAMÄTAJTE SI!

Konštruktor každej triedy musí ako svoj prvý príkaz volať konštruktor predka. Volanie konštruktora predka sa robí pomocou kľúčového slova **`super`** a v zátvorkách uvedieme parametre, ktoré budeme posielať do predkovho konštruktora. Za volaním predkovho konštruktora nasledujú príkazy, ktoré uvedú novo vznikajúcu inštanciu do počiatočného stavu.
Ak má predok bezparametrický konštruktor, nemusíme písať riadok **`super();`** Bezparametrický konštruktor predka je vždy vyvolaný automaticky (teda aj ak takýto príkaz nenapíšeme, tak sa automaticky vykoná). Ak má ale predok parametrický konštruktor, vždy takýto konštruktor musíme zavolať so správnymi hodnotami parametrov.

Vytvorme potomka triedy **`Arena`** s názvom **`TestovaciaArena`**. Preskúmajme konštruktor takto vytvorenej triedy. Vidíme, že neobsahuje žiadne príkazy, ale vieme, že sa aj tak automaticky vyvolá konštruktor predka **`Arena`**. Predok **`Arena`** vždy vytvorí svet (inicializuje predka **`World`**) tak, aby obsahoval 25 x 15 buniek o veľkosti 60x60 pixelov. Takýto rozmer sveta by však nemusel byť vždy vhodný. Ak by sme napríklad pridali potomka triedy **`Arena`** – **`MalaArena`**, tak by presne definovaný rozmer bol na škodu. Čo nám v tomto prípade neprekáža je rozmer bunky – ten by mal byť v každej aréne rovnaký, aby hra pôsobila jednotne.

Vieme, že na špecifikovanie správania metód môžeme použiť parametre. Ak pridáme parametre reprezentujúce šírku a výšku do konštruktora triedy **`Arena`** (ktoré následne aréna odovzdá svetu), potom bude môcť akýkoľvek jej potomok špecifikovať rozmer, aký požaduje.

ÚLOHA 6.3

Upravte triedu **`Arena`** tak, aby mal jej konštruktor dva parametre reprezentujúce šírku a výšku. Upravte volanie predkovho konštruktora v triede **`Arena`** tak, aby prebral tieto parametre. Odstráňte z tohto konštruktora kód zodpovedný za tvorbu a umiestnenie hráčov do arény, budú to vykonávať potomkovia. Takisto môžete odstrániť deklaráciu atribútov typu **`Hrac`** z triedy **`Arena`**.

```
public Arena(int sirka, int vyska) {
     super(sirka, vyska, 60);
}
```

S takto upraveným konštruktorom triedy **`Arena`** však nie je možné spustiť projekt. Problém spôsobuje prázdny konštruktor triedy **`TestovaciaArena`**. Spomeňme si, že ak má predok bezparametrický konštruktor, tak ho potomok vo svojom konštruktore nemusí uvádzať a napriek tomu sa automaticky zavolá. My sme však z pôvodne bezparametrického konštruktora triedy **`Arena`** urobili parametrický konštruktor. V konštruktore potomka **`TestovaciaArena`** preto musíme zavolať konštruktor predka so správne vyplnenými parametrami. Za volaním predkovho konštruktora potom môžu nasledovať akékoľvek príkazy, ktoré uvedú novo vznikajúcu inštanciu do správneho počiatočného stavu.

ÚLOHA 6.4

Upravte konštruktor triedy **TestovaciaArena** tak, aby vytvoril prázdnu arénu s veľkosťou 7x7 buniek. Na overenie vytvorte inštanciu triedy **TestovaciaArena**.

```
public TestovaciaArena() {
     super(7, 7);
}
```

Inštanciu triedy **TestovaciaArena** vytvoríme tak, že klikneme pravým tlačidlom myši na triedu **TestovaciaArena** a z menu vyberieme **new TestovaciaArena()**. Tým zabezpečíme, že pri obnovení sveta bude nástroj Greenfoot vytvárať inštanciu tejto triedy.

6.3 Pretypovanie

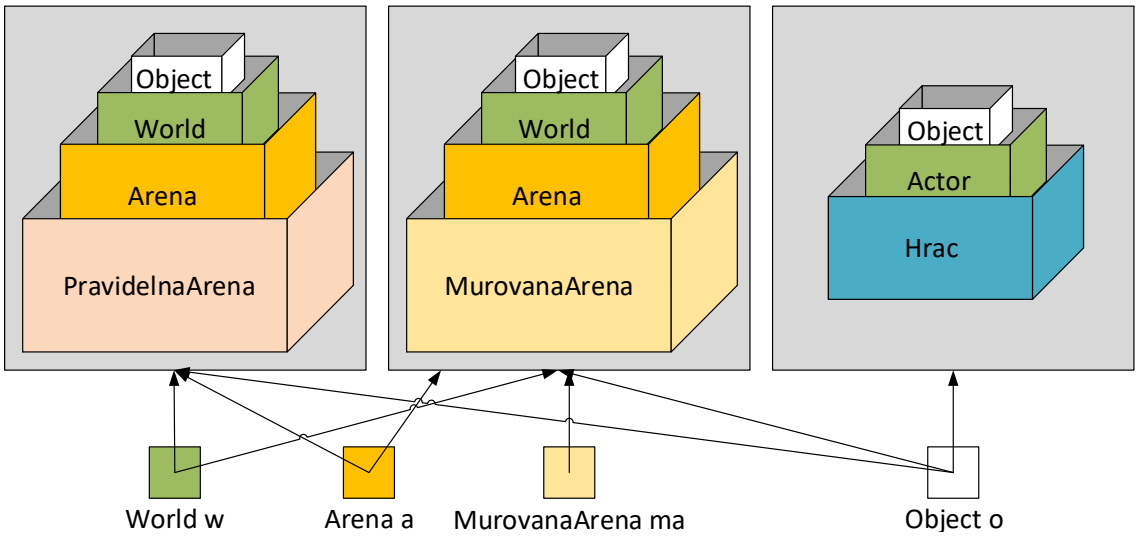
Prezrime si ešte raz triedy sveta, ktoré sú zobrazené na obrázku 6.2. Ak deklarujeme premennú **World w**, tak do nej môžeme vložiť akýkoľvek svet. Je jedno, či to bude **World** alebo **Arena** alebo **PravidelnaArena** – všetko sú to potomkovia sveta, triedy **World** (ako sme uviedli v deklarácií premennej **w**). Ak chceme byť konkrétnejší, tak môžeme deklarovať premennú **Arena a** – do nej môžeme uložiť akúkoľvek arénu. Nemôžeme tam však uložiť napríklad hráča, pretože hráč nie je aréna.

ZAPAMÄTAJTE SI!

Do referenčnej premennej môžete uložiť akýkoľvek objekt, ktorého trieda je zhodná s triedou deklarovanou pri premennej alebo taký, ktorého trieda je potomkom triedy deklarovanej pri premennej. Hovoríme, že **potomok môže nahradiť predka**. Tomuto pravidlu sa hovorí Liskovej **substitučný princíp**.
Akákoľvek premenná typu **Object** teda môže v jazyku Java ukazovať na ľubovoľný objekt.

Sledujme nasledujúci obrázok. Ak chceme pracovať s arénou (oranžová krabica), potom môžeme ako typ objektu použiť akúkoľvek triedu, ktorú obsahuje vo svojej hierarchií (napríklad **PravidelnaArena**). Keďže ale chceme pracovať iba s arénou, môžeme volať iba jej metódy alebo akékoľvek iné metódy z predkov, čo sú v nej – teda **Arena** a **Object** (vidíme oba oranžovú krabicu a jej vnútro). Na triedu **PravidelnaArena** ale z premennej **a** nemáme dosah (nevieme, čo je okolo nás), pretože oranžovú krabicu **Arena** má ako predka napríklad aj trieda **MurovanaArena**.

Premenná **w** môže volať metódy definované v triede **World** a aj vo všetkých triedach predkov, a môže v skutočnosti ukazovať na pravidelnú alebo murovanú arénu. Premenná **a** môže ukazovať tiež na obe arény, ale môže volať aj metódy triedy **Arena**. Premenná typu **Object** môže v jazyku Java ukazovať na akýkoľvek objekt.



Obrázok 6.5: Ilustrácia premenných ukazujúcich na objekty v hierarchii

Objektové programovanie ponúka prostriedky, ako sa spýtať, či je okolo modrej krabice ešte iná krabica testovaného typu a potom ďalej pracovať s premennou, ako s novým typom.

ZAPAMÄTAJTE SI!

Každú referenčnú premennú v jazyku Java je možné otestovať pomocou operátora **`instanceof`**, ktorý na ľavej strane preberá kontrolovanú premennú a na pravej strane preberá triedu:

```
premenná instanceof Trieda
```

Volanie vráti **`true`**, ak premenná ukazuje na objekt, ktorý je danej triedy (obsahuje testovanú krabicu). Ak test vráti **`true`**, je ďalej možné pracovať s premennou tak, ako keby bola nového typu a to tak, že pred ňu do zátvoriek uvedieme novú triedu:

```
(NovaTrieda)premenna
```

Tomuto postupu hovoríme **pretypovanie**.

ÚLOHA 6.5

Pridajte do triedy **`TestovaciaArena`** metódu **`ukazRozmery()`**, ktorá vypíše na obrazovku rozmery arény.

```
public void ukazRozmery() {
     this.showText("Šírka: " + this.getWidth() +
                   " Výška: " + this.getHeight(),
                   this.getWidth() / 2, this.getHeight() / 2);
}
```

ÚLOHA 6.6

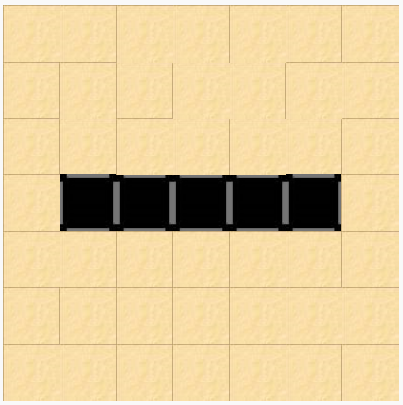
Pridajte do triedy **`Hrac`** metódu **`ukazRozmery()`**, ktorá zobrazí rozmery arény, ak sa nachádza v testovacej aréne. Ak áno, využite metódu **`ukazRozmery()`** triedy **`TestovaciaArena`**.

```
public void ukazRozmery() {
     // získame svet
     World svet = this.getWorld();
     // zistíme, či sa nachádzame v testovacej aréne
     if (svet instanceof TestovaciaArena) {
           // bezpečné pretypovanie
           TestovaciaArena arena = (TestovaciaArena)svet;
           // zavoláme správnu metódu
           arena.ukazRozmery();
     }
}
```

6.4 Rozloženie stien do riadku

ÚLOHA 6.7

Upravte konštruktor triedy **`TestovaciaArena`** tak, aby vytvoril arénu s rozmermi 7x7 buniek so stenami, ktoré budú rozložené takto:



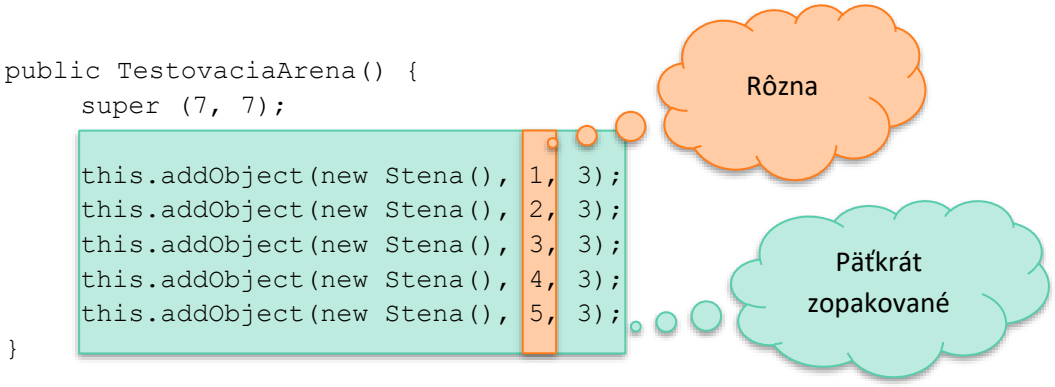
Pripomeňme si, že na vloženie inštancie triedy **`Actor`** do sveta (teda do potomkov triedy **`World`** a v našom prípade **`Arena`**) môžeme využiť metódu **`addObject()`** triedy **`World`**, ktorá má tri parametre:

- Aktor, ktorý má byť vložený
- x-ová súradnica bunky, na ktorú má byť vložený (teda index stĺpca číslovaný od 0)
- y-ová súradnica bunky, na ktorú má byť vložený (teda index riadku číslovaný od 0)

Pozícia [0;0] sa vo svete nachádza vľavo hore.

```
public TestovaciaArena() {
     super (7, 7);

     this.addObject(new Stena(), 1, 3);
     this.addObject(new Stena(), 2, 3);
     this.addObject(new Stena(), 3, 3);
     this.addObject(new Stena(), 4, 3);
     this.addObject(new Stena(), 5, 3);
}
```

Všimnime si, že kód na umiestnenie piatich stien sa opakoval. Jediné, čo sa zmenilo, bola x-ová súradnica polohy každej steny. Predstavme si, že by sme chceli rozložiť nie päť, ale päťdesiat stien. Takýto prístup by bol veľmi prácny a nie je veľmi vhodný.



ZAPAMÄTAJTE SI!

Ak vieme presný počet opakovaní, je možné využiť cyklus **`for`**:

```
for (inicializácia; podmienka; krok) {
}
```

Za kľúčovým slovom **`for`** nasledujú v zátvorke tri (nepovinné) časti oddelené (povinnými) bodkočiarkami.

Cyklus **`for`** funguje takto:

1) Vykoná sa **inicializácia**. V tejto časti sa typicky definuje a inicializuje premenná – počítadlo, ktorá sa volá aj riadiaca premenná cyklu a zvyčajne býva celočíselného typu (**`int`**). Je konvenciou nazývať túto premennú **`i`**. Premennú využívame na počítanie vykonaných opakovaní cyklu.
2) Skontroluje sa **podmienka**. Ak podmienka platí, vykonajú sa príkazy v tele (tie, ktoré sú uvedené medzi zátvorkami **`{}`**) cyklu. Ak podmienka neplatí, cyklus končí. Podmienka je typicky kontrola, či cyklus nepresiahol stanovený počet opakovaní (iterácií). Po skončení cyklu pokračuje vykonávanie programu príkazom, ktorý bezprostredne nasleduje za cyklom.
3) Ak sa vykonalo telo cyklu, tak sa vykoná **krok**. V kroku sa typicky zvýši počítadlo opakovaní cyklu (typicky premennú **`i`**) o jedna.
4) Pokračujeme bodom 2.

Skúsme teraz prepísať predchádzajúce riadky na vytvorenie testovacieho sveta pomocou cyklu **`for`**. Potrebujeme vytvoriť tri časti hlavičky:

1) **Inicializácia**: Riadiacou premennou nášho cyklu bude premenná, ktorá bude postupne nadobúdať hodnoty od 1 do 5. Bude preto celočíselného typu. Prvá hodnota, ktorú premenná nadobudne je 1, preto inicializáciu zapíšeme ako: **`int i = 1`**.
2) **Podmienka**: Náš cyklus musíme zopakovať päťkrát. Posledná platná hodnota riadiacej premennej cyklu je preto 5. Podmienkou teda vyjadríme, že chceme, aby cyklus prebiehal, pokiaľ je premenná **`i`** menšia alebo rovná 5: **`i <= 5`**.
3) **Krok**: Aby sa cyklus vykonal tak, ako sme si určili vyššie, musíme po každom opakovaní cyklu zvýšiť hodnotu v premennej **`i`** o 1. To spôsobí, že premenná **`i`** bude postupne nadobúdať hodnoty od 1 do 5: **`i = i + 1`**.

Hlavičku cyklu teda môžeme napísať takto:

```
for (int i = 1; i <= 5; i = i + 1)
```

Poslednou časťou je definovanie tela cyklu. To bude obsahovať jediný príkaz (ktorý sa vďaka cyklu niekoľkokrát zopakuje). Riadiacu premennú cyklu zároveň využijeme na zadanie x-ovej súradnice steny. Telo cyklu bude teda vyzerať takto:

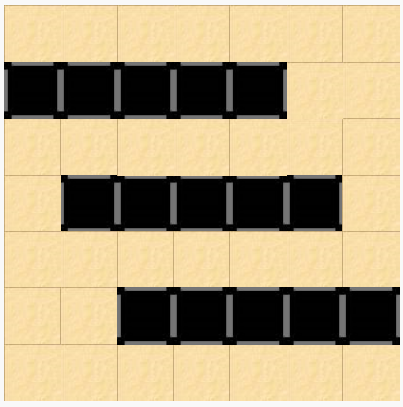
```
this.addObject(new Stena(), i, 3);
```

a celý cyklus **`for`**, ktorým môžeme vygenerovať päť stien vedľa seba teda bude:

```
for (int i = 1; i <= 5; i = i + 1) {
     this.addObject(new Stena(), i, 3);
}
```

ÚLOHA 6.8

Upravte konštruktor triedy **`TestovaciaArena`** tak, aby boli bunky rozložené tak, ako je to zobrazené na obrázku.



```
public TestovaciaArena() {
     super (7, 7);

     // vytvorí bunky na druhom riadku na pozíciách 0-4.
     for (int i = 0; i <= 4; i++) {
          this.addObject(new Stena(), i, 1);
     }

     // vytvorí bunky na štvrtom riadku na pozíciách 1-5.
     for (int i = 1; i <= 5; i++) {
          this.addObject(new Stena(), i, 3);
     }

     // vytvorí bunky na šiestom riadku na pozíciách 2-6.
     for (int i = 2; i <= 6; i++) {
          this.addObject(new Stena(), i, 5);
     }
}
```

Všimnime si, že pre rozloženie stien sme trikrát zopakovali veľmi podobný kód.

ÚLOHA 6.9

Zamyslime sa, koľko informácií potrebujeme na to, aby sme vedeli vytvoriť akýkoľvek rad stien idúcich za sebou?

Stačia nám tri informácie:

- Riadok, na ktoré majú byť umiestnené steny
- Stĺpec, na ktorý má byť umiestnená prvá stena.
- Počet stien, ktoré majú byť umiestnené za sebou.

Tvorba stien v riadku za sebou je bežná pre množstvo arén. Bolo by preto vhodné, aby túto schopnosť arény zdieľali. Spomeňme si, že potomok zdedí všetku funkčnosť predka. Takúto funkčnosť je preto možné dosiahnuť jednoducho tak, že v predkovi **`Arena`** vytvoríme metódu, ktorá sa postará o vytvorenie za sebou idúcich stien v riadku. Parametre metódy budú také, aké vzišli z predošlej úlohy.

ÚLOHA 6.10

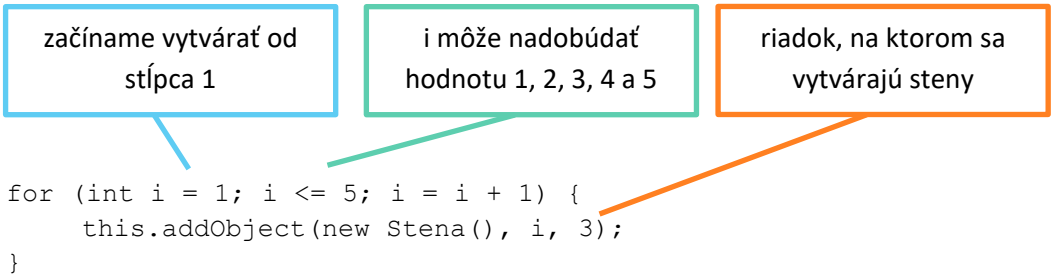
Deklarujte v predkovi **`Arena`** metódu **`vytvorRiadokStien()`**, ktorá bude mať tri parametre:

- Riadok (horný riadok má index 0), na ktorom sa majú začať vytvárať steny.
- Stĺpec (ľavý stĺpec má index 0), od ktorého sa majú začať vytvárať steny.
- Počet vyjadrujúci, koľko stien za sebou má byť vytvorených.

Metóda nemá návratovú hodnotu (použijeme teda kľúčové slovo **`void`**).

```
public void vytvorRiadokStien (int naKtoromRiadku,
     int odKtorehoStlpca, int kolko) {

}
```

Pre implementáciu tela metódy môžeme využiť príkaz **`for`**. Analyzujme druhý príkaz **`for`** z úlohy 6.8, v ktorom sme použili nasledujúce hodnoty:




```
for (int i = 1; i <= 5; i = i + 1) {
     this.addObject(new Stena(), i, 3);
}
```

Postupnou úpravou cyklu s pomocou parametrov metódy vytvoríme univerzálnu metódu, ktorá bude schopná vygenerovať rad stien.

- Začneme inicializáciou premennej. Musíme zmeniť jej význam. Už nebude určovať, v ktorom stĺpci sa bude vytvárať stena, ale bude iba počítať iterácie cyklu. Začneme počítať od 1.
- Cyklus musí vytvoriť toľko stien, koľko udáva parameter **`kolko`**. Podmienku konca cyklu teda musíme formulovať tak, aby zabezpečila vytvorenie všetkých stien. Koľkú stenu vytvárame evidujeme v premennej **`i`**.
- Metóda v riadku, v ktorom sa steny vkladajú má niekoľko parametrov. Prvý určuje aký objekt bude vložený, druhý a tretí definujú polohu (riadok a stĺpec) vloženého objektu. Keďže v našej metóde budeme vkladať na parametrami určené a meniace sa pozície, musíme polohu (teda stĺpec a riadok) vyjadriť výrazmi:
  - Stĺpec musíme vypočítať – jeho hodnota bude vychádzať z hodnoty parametra **`odKtorehoStlpca`**. K tejto hodnote stačí pripočítať hodnotu riadiacej premennej cyklu zmenšenú o jedna. Zistite, prečo je od tohto výrazu ešte potrebné odpočítať 1?
  - Riadok je určený parametrom **`odKtorehoStlpca`** metódy **`vytvorRiadokStien()`**, môžeme teda hodnotu 3 nahradiť priamo ním.

Výsledná verzia metódy **`vytvorRiadokStien()`** v triede **`Arena`** by teda mohla vyzerať takto:

```
public void vytvorRiadokStien (int naKtoromRiadku,
     int odKtorehoStlpca, int kolko) {

     for (int i = 1; i <= kolko; i = i + 1) {
          this.addObject(new Stena(),
               odKtorehoStlpca + i - 1, naKtoromRiadku);
     }
}
```

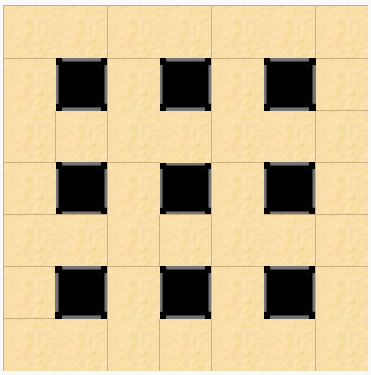
ÚLOHA 6.11

Upravte kód v konštruktore potomka triedy **Arena** tak, aby využíval metódu **vytvorRiadokStien()**.

```
public TestovaciaArena() {
     super(7, 7);

     this.vytvorRiadokStien(1, 0, 5);
     this.vytvorRiadokStien(3, 1, 5);
     this.vytvorRiadokStien(5, 2, 5);
}
```

ÚLOHA 6.12

Upravte konštruktor triedy **TestovaciaArena** tak, aby vytvoril arénu zobrazenú na obrázku nižšie. Upravte na to metódu **vytvorRiadokStien()** tak, aby mala štvrtý parameter definujúci medzery medzi stenami.



Upravíme metódu **vytvorRiadokStien()** tak, že pridáme štvrtý parameter, ktorý bude určovať počet voľných polí medzi stenami (napr. **int medzery**). V metóde je potom potrebné zmeniť iba výraz pre výpočet stĺpca:

```
odKtorehoStlpca + (i - 1) *  (medzery + 1)
```

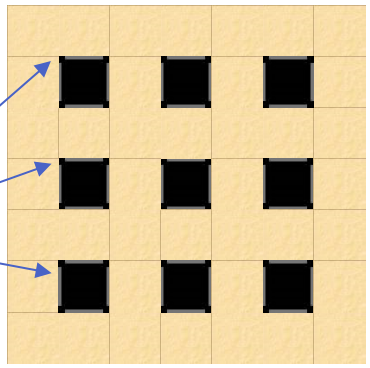
Upravenú metódu **vytvorRiadokStien()** použijeme v konštruktore triedy **TestovaciaArena**.

```
public TestovaciaArena() {
     super (7, 7);

     this.vytvorRiadokStien (1, 1, 3, 1);
     this.vytvorRiadokStien (3, 1, 3, 1);
     this.vytvorRiadokStien (5, 1, 3, 1);
}
```

6.5 Vytvorenie obdĺžnika stien

Všimnime si, že podobne, ako v úlohe 6.7, aj tu sa opakujú príkazy, ktoré sa odlišujú iba v hodnotách niektorých parametrov. Postupnosť metód **`vytvorRiadokStien()`**, ktoré majú zhodné súradnice počiatočného stĺpca a líšia sa iba v riadku rozkladajú steny do obdĺžnika. Jeho dĺžku v stĺpcoch určuje parameter **`kolko`** v kombinácii s parametrom **`medzery`**, ktorý je posielaný metódam **`vytvorRiadokStien()`**. Výšku tohto obdĺžnika v riadkoch určuje počet volaní metódy a prípadné medzery medzi riadkami.

```
public TestovaciaArena() {
super (7, 7);

this.vytvorRiadokStien (1, 1, 3, 1);
this.vytvorRiadokStien (3, 1, 3, 1);
this.vytvorRiadokStien (5, 1, 3, 1);
}
```



Podobne, ako v predchádzajúcom prípade, môžeme rozšíriť predka triedy **`Arena`** o metódu, ktorá bude rozkladať steny do obdĺžnika.

ÚLOHA 6.13

Zamyslime sa, koľko a akých informácií potrebujeme na to, aby sme vedeli vytvoriť akýkoľvek rad stien v obdĺžniku, ktorého počiatočný bod bude možné zadať a ktorému bude možné nastaviť medzery medzi stenami v riadkoch aj v stĺpcoch?

Potrebujeme šesť informácií:

- **Riadok**, na ktorý má byť umiestnená prvá stena.
- **Stĺpec**, na ktorý má byť umiestnená prvá stena.
- **Počet riadkov**, koľko má byť vygenerovaných.
- **Počet stien v riadku**, ktoré majú byť umiestnené za sebou.
- **Medzera medzi riadkami**.
- **Medzera medzi stenami v riadku**.

ÚLOHA 6.14

Deklarujte v predkovi **`Arena`** metódu **`vytvorObdlznikStien()`**, ktorá bude mať nasledujúce parametre:

- Riadok (horný riadok má index 0), od ktorého má začať vytvárať steny.
- Stĺpec (ľavý stĺpec má index 0), od ktorého má začať vytvárať steny.
- Počet riadkov, koľko má byť vytvorených.
- Počet stien, koľko má byť za sebou v riadku vytvorených.
- Veľkosť medzery medzi riadkami.
- Veľkosť medzery medzi stenami v riadku.

Metóda nemá návratovú hodnotu.

```
public void vytvorObdlznikStien (int odKtorehoRiadku,
     int odKtorehoStlpca, int kolkoRiadkov,
     int pocetStienVRiadku, int medzeraRiadky, int medzeraStlpce) {
}
```

V tele metódy budeme tentokrát opakovať volanie metódy **`vytvorRiadokStien()`**, pričom hodnoty jej parametrov preberieme z parametrov metódy **`vytvorObdlznikStien()`**. Na opakovanie využijeme cyklus **`for`**, podobne ako sme to spravili v úlohe 6.10.

Hlavičku cyklu **`for`** môžeme napísať takto:

```
for (int i = 1; i <= kolkoRiadkov; i = i + 1)
```

Jediným príkazom v tele cyklu je volanie metódy **`vytvorRiadokStien()`**, ktorá má štyri parametre, ktoré okrem parametra **`naKtoromRiadku`** môžeme prevziať priamo z argumentov metódy **`vytvorObdlznikStien()`**:

```
public void vytvorObdlznikStien (
     int odKtorehoRiadku,
     int odKtorehoStlpca,
     int kolkoRiadkov,
     int pocetStienVRiadku,
     int medzeraRiadky,
     int medzeraStlpce) {

     for (int i = 1; i <= kolkoRiadkov; i = i + 1) {
          this.vytvorRiadokStien(
               [naKtoromRiadku],
               odKtorehoStlpca,
               pocetStienVRiadku,
               medzeraStlpce);
     }
}
```

Hodnotu, ktorú zadáme do parametra **`naKtoromRiadku`** môžeme vypočítať obdobne, ako v úlohe 6.12:

```
odKtorehoRiadku + (i - 1) * (medzeraRiadky + 1)
```

ÚLOHA 6.15

Upravte konštruktor potomka triedy **`Arena`** tak, aby využil metódu **`vytvorObdlznikStien()`** a rozložil arénu tak, ako je zobrazená v úlohe 6.12.

```
public TestovaciaArena() {
     super(7, 7);

     this.vytvorObdlznikStien(1, 1, 3, 3, 1, 1);
}
```

Zatiaľ sme sa zaoberali vytvorením rôznych konfigurácií stien. Spomeňme si na analýzu tvorby arén a na posledné dva kroky:

- **Vygenerovanie múrov** – múry je možné rozložiť podobnými algoritmami, ako sme navrhli pre steny. Namiesto stien sa budú vytvárať inštancie triedy **`Mur`**.
- **Vygenerovanie hráčov na správnych pozíciách** – aby sme mohli hru hrať, je potrebné do arény vložiť aj hráčov. V našom prípade tak môžeme urobiť veľmi jednoducho. Na konci tela konštruktora triedy **`TestovaciaArena`** vytvoríme dve inštancie triedy **`Hrac`**, ktoré umiestnime na správne pozície.

ÚLOHA 6.16

Vytvorte vo svojej aréne niekoľko múrov a pridajte dvoch hráčov. Otestujte funkčnosť hry, teda či hráči nevojdú ani do steny a ani do múru.

Múry, podobne ako aj hráči, môžu byť do sveta vkladaní priamo pomocou volania

```
this.addObject(new Mur(), X, Y);
this.addObject(new Hrac(...), X, Y);
```

Pozície X a Y je potrebné vhodne zadať. Taktiež je potrebné vhodne vypísať parametre konštruktora triedy **`Hrac`** (aby mali hráči rozdielne ovládanie).

ZHRNUTIE

Dedičnosť je vzťah medzi dvomi triedami, v ktorom potomok preberá všetky vlastnosti a schopnosti predka. Pre vytvorenie dedičnosti je potrebné v hlavičke triedy potomka použiť za jej identifikátorom (názvom) kľúčové slovo **`extends`** nasledované identifikátorom triedy predka. Pri dedičnosti je nutné vhodne inicializovať potomka v konštruktore. Prvý riadok

konštruktora musí byť vždy volanie konštruktora predka pomocou kľúčového slova **`super`** nasledovaného zátvorkami s parametrami predkovho konštruktora. Ak má predok bezparametrický konštruktor, potom potomok nemusí písať **`super();`** pretože je takýto predkov konštruktor volaný automaticky ako prvý príkaz konštruktora potomka.

Pretypovanie je zmena premennej z jedného typu na iný. Referenčné premenné je možné bezpečne pretypovať po kontrole operátorom **`instanceof`**, ktorý na ľavej strane preberá referenčnú premennú a na pravej strane triedu. Operátor vráti **`true`**, ak je skutočný objekt, na ktorý premenná ukazuje typu testovanej triedy (teda obsahuje túto triedu niekde v hierarchii svojich predkov). Po tejto kontrole je možné premennú pretypovať uvedením nového typu (testovanej triedy) do okrúhlych zátvoriek pred premennou.

Cyklus **`for`** je cyklus s pevným počtom opakovaní, skladá sa z hlavičky a tela cyklu. Hlavička sa skladá z kľúčového slova **`for`** a troch nepovinných častí, ktoré sú uvedené v zátvorke a oddelené bodkočiarkou. Prvá časť sa vykoná práve raz pred vykonaním cyklu a je určená na inicializáciu riadiacej premennej cyklu. Druhá časť obsahuje podmienku, ktorá určuje, dokedy sa cyklus opakuje (cyklus skončí, keď podmienka neplatí, nemusí sa teda vykonať ani raz – ak podmienka neplatí už na začiatku cyklu). Tretia časť je krok, ktorý sa vykoná vždy po skončení tela cyklu. Telo cyklu sa píše do zložených zátvoriek **`{}`** a obsahuje príkazy, ktoré sa vykonajú pri každom opakovaní cyklu.

ÚLOHY NA PRECVIČOVANIE

ÚLOHA 6.A

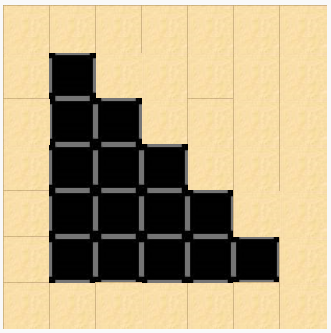
Pridajte do triedy **`Arena`** metódu, ktorá bude mať celočíselné parametre **`naKtoromStlpci`**, **`odKtorehoRiadku`**, **`kolko`** a **`medzery`**. Metóda nech vytvorí stĺpec stien, podľa zadaných parametrov. Medzera určuje, koľko polí je medzi stenami voľných.

ÚLOHA 6.B

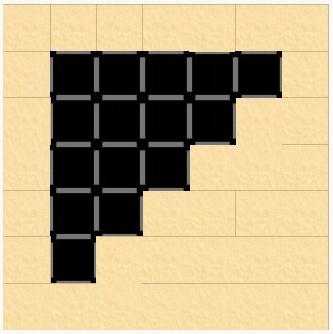
Pridajte do triedy **`Arena`** metódy, ktoré budú schopné vygenerovať a) riadok a b) obdĺžnik zložený z múrov. Parametre metód sú zhodné ako parametre metód generujúcich steny.

ÚLOHA 6.C

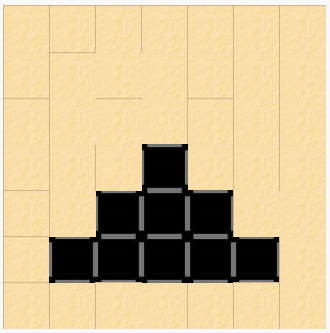
Pridajte do trieda **`Arena`** metódy, ktoré vygenerujú nasledujúce trojuholníky stien (medzi riadkami a stĺpcami bude možné zadať medzery). Parametre metód vhodne navrhnite.



a



b



c

ÚLOHA 6.D

Vytvorte triedy **`PravidelnaArena`** a **`MurovanaArena`** ako potomkov triedy **`Arena`**, pričom rozloženie stien, múrov a hráčov bude také, ako na obrázku 6.2.

7 ZOZNAM A CYKLUS FOREACH

KĽÚČOVÉ SLOVÁ

Zoznam. Cyklus foreach.

CIELE

V tejto kapitole sa podrobnejšie zoznámime so zoznamami – objektami, ktoré evidujú iné objekty. Naučíme sa základné metódy pre prácu so zoznamom (vytvorenie, pridanie prvku, odobratie prvku, sprístupnenie prvku) a naučíme sa používať cyklus **`foreach`** pre jednoduché sprístupnenie všetkých prvkov v zozname.

OBSAH

7.1 Evidencia hráčov v aréne

Uvažujme, ako aktuálne vytvárame hráčov v našich arénach. Každý potomok triedy **`Arena`** vytvorí hráčov, ktorých vloží na správne pozície. Zamyslime sa, ako budeme vedieť, že hra skončila? Odpoveď je jednoduchá – keď ostane jediný hrajúci hráč. Hra tiež môže skončiť aj nerozhodne a to tak, že všetci hráč skončia naraz (napr. obaja poslední hráči budú zničení tou istou bombou). Taktiež si môžeme položiť otázku, či je počet hráčov zakaždým rovnaký. Zas zistíme, že môžu byť arény, kde sa zmestia dvaja hráči, ale taktiež sa môžu vytvoriť aj väčšie arény, so štyrmi, či ôsmimi hráčmi. Ak máme teda jednoznačne odpovedať na to, či nastal koniec hry, musíme vedieť, koľkí hráči ešte hrajú hru.

Testovanie konca hry by mohli robiť všetky arény osobitne. Podľa toho, koľko hráčov v aréne začína, toľko atribútov typu **`Hrac`** by bolo v potomkovi triedy **`Arena`** deklarovaných. Takéto riešenie by však vyžadovalo duplikovanie kódu v arénach, ktoré majú rovnaký počet hráčov. Musíme teda vymyslieť lepšie riešenie.

Už vieme, že ak chceme zabezpečiť jednotnú funkčnosť pre všetkých potomkov, potom môžeme vytvoriť metódu u predka, v našom prípade v triede **`Arena`**.

ÚLOHA 7.1

Vytvorte v triede **`Arena`** bezparametrickú metódu **`koniecHry()`**, ktorá zistí, či nastal koniec hry (ostal iba jeden alebo žiadny hráč) a v návratovej hodnote typu **`boolean`** oznámi, či sa tak stalo. Zatiaľ predpokladajme, že koniec hry nenastane nikdy.

```
public boolean koniecHry() {
     return false; // zatiaľ nebudeme predpokladať koniec
}
```

Predtým, ako sa budeme venovať telu metódy **koniecHry()**, zabezpečme jej volanie na vhodnom mieste. Bolo by dobré, aby sa koniec hry testoval pravidelne. Podobne, ako tomu je v triede **Actor**, aj do potomkov triedy **World** môžeme napísať metódu **act()**, ktorú potom bude prostredie Greenfoot pravidelne vyvolávať.

ÚLOHA 7.2

Pridajte do triedy **Arena** metódu **act()**. V nej skontrolujete, či nastal koniec hry (pomocou metódy **koniecHry()**) a ak áno, zastavte hru. Pre zastavenie činnosti prostredia Greenfoot využite príkaz **Greenfoot.stop();** .

```
public void act() {
     if (this.koniecHry()) {
          Greenfoot.stop();
     }
}
```

Automatické volanie testu na koniec hry máme zabezpečené, vráťme sa teda k samotnej metóde **koniecHry()**. Ako sme spomenuli vyššie, každá aréna môže obsahovať rozdielny počet hráčov. Nakoľko testovanie musí prebehnúť v predkovi, nemôžu byť títo hráči evidovaní v atribútoch potomka (pretože na atribúty potomka nemá predok dosah). Musíme teda zabezpečiť, aby všetkých hráčov evidoval predok.

Predstavme si, že by sme vytvorili v triede **Arena** dva atribúty:

```
private Hrac hrac1;
private Hrac hrac2;
```

Vo väčšine arén by sme si vystačili. Ak by sme potrebovali viac hráčov, jednoducho ich pridáme podľa nasledujúcej schémy.

```
private Hrac hrac1;
private Hrac hrac2;
private Hrac hrac3; // null, ak hrajú dvaja hráči
private Hrac hrac4; // null, ak hrajú dvaja alebo traja hráči
```

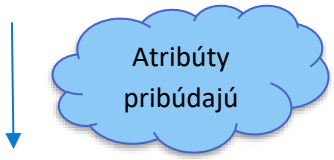
Pri takomto spôsobe by sme však čoskoro prišli na to, že by bolo veľmi nepraktické písať kód v tele metódy **koniecHry()**. Mohol by vyzerať takto:

```
public boolean koniecHry() {
     boolean jeHrac1 = hrac1 != null && hrac1.getWorld() != null;
     boolean jeHrac2 = hrac2 != null && hrac2.getWorld() != null;
     boolean jeHrac3 = hrac3 != null && hrac3.getWorld() != null;
     boolean jeHrac4 = hrac4 != null && hrac4.getWorld() != null;

     if (!jeHrac1 && ! jeHrac2 && ! jeHrac3 && ! jeHrac4) {
          this.showText("Remíza",
               this.getWidth() / 2, this.getHeight() / 2);
          return true;
     } else {
          if (jeHrac1 && ! jeHrac2 && ! jeHrac3 && ! jeHrac4) {
```

```
                this.showText("Vyhral hráč 1",
                        this.getWidth() / 2, this.getHeight() / 2);
                return true;
            } else
            if (!jeHrac1 && jeHrac2 && ! jeHrac3 && ! jeHrac4) {
                this.showText("Vyhral hráč 2",
                        this.getWidth() / 2, this.getHeight() / 2);
                return true;
            } else
            if (!jeHrac1 && ! jeHrac2 && jeHrac3 && ! jeHrac4) {
                        this.showText("Vyhral hráč 3",
                                this.getWidth() / 2, this.getHeight() / 2);
                        return true;
            } else
            if (!jeHrac1 && ! jeHrac2 && ! jeHrac3 && jeHrac4) {
            this.showText("Vyhral hráč 4",
                this.getWidth() / 2, this.getHeight() / 2);
                return true;
            } else {
                // v hre sú aspoň dvaja hráči
                return false;
            }
        } // else vetva podmienky testujúcej remízu
}
```

Všimnime si ale, ako sme pridávali atribúty typu **`Hrac`**. Na začiatku boli dva, ak by sme potrebovali, boli štyri a pokojne by sa mohli rozšíriť na osem, atď.

```
private Hrac hrac1;
private Hrac hrac2;
private Hrac hrac3;
private Hrac hrac4;
```



Ideálne by bolo, aby sme využili nejaký druh „košíka“, do ktorého by sme podľa potreby vložili dvoch, troch, štyroch, ôsmich (a tak ďalej) hráčov. Potom by sme sa jednoducho pozreli, koľko hráčov v košíku máme, a na základe toho jednoducho napísali všeobecný kód v tele metódy **`koniecHry()`**. Jazyk Java ponúka na uskladnenie vopred neznámeho počtu inštancií rovnakého typu špeciálne košíky – **kontajnery**.

ZAPAMÄTAJTE SI!

Jazyk Java ponúka pre ukladanie objektov rovnakého typu (rovnakej triedy) niekoľko druhov **kontajnerov**. Termínom kontajner označujeme každý objekt, ktorý eviduje iné objekty – dokáže ich napríklad do seba vložiť, sprístupniť a vymazať.

Medzi základné kontajnery patrí **zoznam** – **LinkedList**, ktorý sa nachádza v balíčku (súbor s triedou, ktorý ponúka jazyk Java) **`java.util.LinkedList`**. Ak chceme zoznam použiť, musíme najskôr v súbore uviesť balíček, v ktorom sa nachádza tento zoznam. Na to použijeme príkaz **`import`**, ktorý musíme napísať na úplný začiatok súboru so zdrojovým kódom triedy, teda v našom prípade:

```
import java.util.LinkedList;
```

Pre úplnú deklaráciu triedy **`LinkedList` musíme** v zátvorkách **< >** uviesť, čo v zozname chceme ukladať (uvedením triedy), napríklad:

```
LinkedList<Hrac> hraci;
```

Triede v zátvorkách < > hovoríme **generický parameter**. Na základe generického parametra sa odvíja typ parametrov a návratových hodnôt metód triedy **`LinkedList`** (ak uvedieme ako generický parameter triedu **`Hrac`**, potom relevantné vstupné parametre do metód triedy **`LinkedList`**, rovnako ako aj relevantné návratové hodnoty metód tejto triedy, budú typu **`Hrac`**). Triedu, ktorá obsahuje zátvorky **< >** , nazývame **generická trieda**.
Predok zoznamu **`LinkedList<E>`** sa nazýva **`List<E>`** (z balíčka **`java.util.List`**).

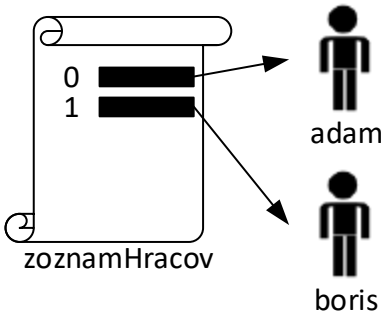
So zoznamami sme sa prvý krát stretli, keď sme kontrolovali, či môže hráč vstúpiť do danej bunky. Vtedy sme si ukázali jednu z množstva metód, ktoré zoznamy ponúkajú. Poďme preskúmať aj ďalšie metódy zoznamov, pomocou ktorých je možné do nich objekty pridávať, odoberať či sprístupňovať. Prehľad tých najdôležitejších uvádzame v nasledujúcej tabuľke:

Tabuľka 7.1: Prehľad vybraných metód triedy `List<E>`

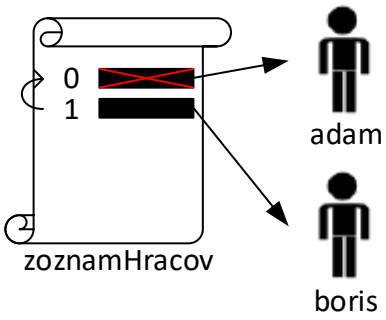
Metóda	Parametre	Čo metóda vráti	Čo metóda robí
`add`	objekt generického typu	-	Pridá objekt na koniec zoznamu.
`clear`	-	-	Zruší evidenciu všetkých objektov.
`contains`	objekt	boolean	Zistí, či sa objekt nachádza v zozname .
`get`	poradie (index). Objekt na prvej pozícií má index 0.	objekt generického typu	Sprístupní objekt na mieste indexu.
`indexOf`	objekt	int	Vráti poradie (index) objektu. Vráti -1, ak sa objekt v zozname nenachádza.
`isEmpty`	-	boolean	Vráti **`true`**, ak je zoznam prázdny, **`false`** inak.
`remove`	poradie (index) alebo objekt	objekt generického typu	Odstráni (zruší evidenciu) a vráti objekt alebo objekt na danom indexe. Ak sa objekt v zozname nenachádza, nespraví nič.
`set`	pozícia (index) a objekt generického typu	-	Nahradí evidovaný objekt na pozícií novým objektom.
`size`	-	int	Vráti počet prvkov evidovaných v zozname.

Ak máme zoznam hráčov, je celkom jednoduché pracovať s ním. Predstavme si, že máme štyri inštancie triedy **`Hrac`**: **`adam`**, **`boris`**, **`cyril`** a **`dusan`** a jeden zoznam hráčov **`zoznamHracov`** typu **`LinkedList<Hrac>`**. Prácu so zoznamom ilustrujeme na nasledujúcich obrázkoch. Všimnite si, že zoznam je číslovaný (indexovaný) od 0, a teda posledný platný index hráča je vždy o jedna menší, ako je aktuálny počet prvkov v zozname.

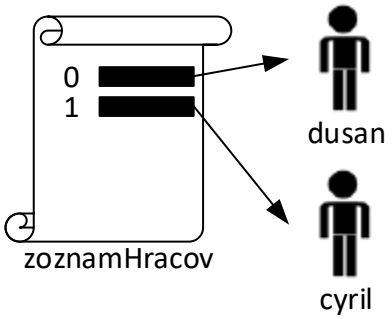
```
zoznamHracov.add(adam);
zoznamHracov.add(boris);

zoznam.size(); // 2
zoznam.indexOf(adam); // 0
zoznam.indexOf(boris); // 1
zoznam.contains(cyril); // false
```




```
zoznam.remove(adam);
// boris sa posunie na pozíciu 0

zoznam.get(0); // boris
zoznam.size(); // 1
```




```
zoznam.set(0, dusan);
zoznam.add(cyril);
```



Obrázok 7.1: Práca so zoznamom

Použime zoznam na evidenciu hráčov v triede **Arena**.

ÚLOHA 7.3

Pridajte do triedy **Arena** atribút **zoznamHracov** typu **LinkedList<Hrac>**. Nezabudnite, že musíte dopísať import balíčka s triedou **LinkedList**. Inicializujte inštanciu atribút **zoznamHracov** v konštruktore triedy **Arena**.

Do hornej časti súboru s triedou **Arena** doplníme import:

```
import java.util.LinkedList;
```

Následne deklarujeme atribút:

```
private LinkedList<Hrac> zoznamHracov;
```

Nakoniec upravíme konštruktor triedy **Arena**:

```
public Arena(int sirka, int vyska) {
     super (sirka, vyska, 60);
     this.zoznamHracov = new LinkedList<Hrac>();
}
```

Zamyslime sa teraz, ako sa hráči do zoznamu dostanú. Potomkovia triedy **Arena** naň nemajú dosah. Trieda **Arena** preto musí pripraviť metódu na registráciu, ktorú môžu potomkovia využiť. Pre pridanie hráča do zoznamu ho jednoducho pridáme na jeho koniec pomocou metódy **add()**.

ÚLOHA 7.4

Pridajte do triedy **Arena** metódu **zaregistrujHraca()**, ktorá preberie jediný parameter typu **Hrac** a pomocou metódy **add()** ho vloží na koniec zoznamu **zoznamHracov**.
Upravte potomkov triedy **Arena** tak, aby hráča po vložení hráča do sveta na správne miesto aj zaregistrovali u predka.

```
public void zaregistrujHraca(Hrac hrac) {
     // nebudeme registrovať už zaregistrovaného hráča
     if (!this.zoznamHracov.contains(hrac)) {
          this.zoznamHracov.add(hrac);
     }
}
```

Pre registráciu hráča musím upraviť kód v potomkoch triedy **Arena**. Vytvoreného hráča (s parametrami ovládania) si uložíme do pomocnej premennej. Hráča potom vložíme do sveta (na správne pozície X a Y) a zaregistrujeme pomocou metódy **zaregistrujHraca()**.

```
Hrac h = new Hrac(...); // vytvoríme hráča
this.addObject(h, X, Y); // vložíme ho do sveta
this.zaregistrujHraca(h); // zaregistrujeme ho u predka
```

ÚLOHA 7.5

Pridajte do triedy **Arena** metódu **odregistrujAOdstranHraca()**, ktorá má jediný parameter typu **Hrac**. Metóda odstráni hráča zo zoznamu hráčov a následne ho odstráni zo sveta.

```
public void odregistrujAOdstranHraca(Hrac hrac) {
     this.zoznamHracov.remove(hrac);
     this.removeObject(hrac);
}
```

S možnosťou registrácie a odregistrácie hráčov v triede **Arena** je teraz jednoduché napísať kód metódy **koniecHry()**. Hra skončí, ak zoznam obsahuje jediného (víťaz) alebo žiadneho hráča (remíza).

ÚLOHA 7.6

Implementujte telo metódy **koniecHry()** tak, aby metóda vrátila **true** vtedy, keď sa v hre nachádza jeden alebo žiadny hráč. Využite vhodné metódy zoznamu.

```
public boolean koniecHry() {
     // hra končí, keď má najviac jedného hráča
     return this.zoznamHracov.size() <= 1;
}
```

7.2 Identifikácia zasiahnutých hráčov

Hráč sa musí odregistrovať z arény vtedy, keď ho zasiahne bomba. Naše bomby sú zatiaľ takmer neškodné, spravme ich teda nebezpečné. Vieme, že bomba vybuchne, keď jej vyprší časovač. Potom bomba oznámi vlastníkovi, že vybuchla, prehrá zvuk výbuchu a odstráni sa zo sveta. Predtým však môže zničiť svoje okolie.

Trieda **Actor** definuje metódu **getObjectsInRange()**, ktorá vráti zoznam typu **List** objektov v okolí aktora podľa nasledujúcich kritérií:

- Vzdialenosť medzi stredom aktora, ktorý metódu **getObjectsInRange()** volá, a testovaným aktorom v okolí musí byť menšia nanajvýš rovná ako špecifikovaný dosah.
- Ak je testovaný aktor v dosahu potom sa skontroluje jeho trieda. Ak je zhodná s triedou, ktorú požadujeme, potom je takýto testovaný aktor zaradený do výsledného zoznamu.

ZAPAMÄTAJTE SI!

Pre poslanie triedy ako parametra, uvedieme jej názov a za ním napíšeme **.class**, teda napríklad **Hrac.class**, **Stena.class** alebo **Bomba.class**.

ÚLOHA 7.7

Upravte kód v metóde **act()** v triede **Bomba** tak, aby predtým, ako sa bomba odstráni zo sveta, získala všetkých hráčov, ktorí sú od nej vzdialení najviac o hodnotu jej sily. Metóda **getObjectsInRange()** vracia zoznam typu **List**. Jeho generický parameter je zhodný s triedou, ktorá je poslaná ako druhý parameter.

Aby sme v triede **Bomba** mohli použiť triedu **List**, musíme v zdrojovom súbore najskôr napísať import:

```
import java.util.List;
```

Predtým, ako bombu odstránime zo sveta, získame všetkých hráčov:

```
List<Hrac> zasiahnutiHraci = this.getObjectsInRange(this.sila,
```

```
Hrac.class);
```

7.3 Cyklus foreach

Hráči, ktorí sú zasiahnutí výbuchom bomby, musia byť z hry odregistrovaní. Pre odregistrovanie hráča môžeme využiť metódu **`odregistrujAOdstranHraca()`** v triede **`Arena`**. Odregistrovať musíme postupne všetkých hráčov. Využijeme metódy tried **`List`**: metódu **`size()`** na zistenie počtu hráčov a metódu **`get()`** na získanie hráča na danej pozícii. Pripomeňme ešte, že prvý prvok v zozname je na pozícii 0.

ÚLOHA 7.8

Pomocou cyklu **`for`** prejdite všetkých hráčov v zozname zasiahnutých hráčov. Každého hráča odregistrujte v triede **`Arena`**.

```
// získame arénu, lebo tá definuje metódu odregistrujAOdstranHraca()
Arena arena = (Arena)this.getWorld();

// musíme prejsť všetkých hráčov, teda od 0 po počet
for (int i = 0; i < zasiahnutiHraci.size(); i = i + 1) {
     Hrac hrac = zasiahnutiHraci.get(i);
     arena.odregistrujAOdstranHraca(hrac);
}
```

Vykonanie rovnakej činnosti so všetkými prvkami v zozname býva veľmi časté. Jazyk Java preto ponúka nový druh cyklu, ktorý podobné zápisy veľmi zjednodušuje. Je to cyklus **`foreach`**.

ZAPAMÄTAJTE SI!

Ak je potrebné vykonať rovnakú operáciu s každým prvkom v nejakom kontajneri, potom je možné použiť cyklus **`foreach`**:

```
for (Typ premenná : kontajner) {
}
```

Za kľúčovým slovom **`for`** nasleduje v zátvorke uvedená riadiaca premenná cyklu a za dvojbodkou kontajner. Riadiaca premenná cyklu musí byť rovnakého typu, ako sú prvky v kontajneri.

Cyklus **`foreach`** zabezpečí, aby riadiaca premenná cyklu v prvej iterácii ukazovala na prvý prvok v kontejneri a vykonala telo cyklu, v druhej iterácii cyklu na druhý prvok v kontajneri a vykonala telo cyklu, v poslednej iterácii cyklu na posledný prvok v kontajneri a vykonala telo cyklu.

Počas vykonávania cyklu `foreach` nesmieme modifikovať obsah (teda volať metódy ako je **`add()`** alebo **`remove()`**) **kontajnera, nad ktorým cyklus prebieha!**

Všimnite si rovnaké kľúčové slovo pre príkaz **`for`** aj pre príkaz **`foreach`** (v oboch prípadoch píšeme iba **`for`**). Rozdiel je v zátvorke za týmto kľúčovým slovom:

- Zátvorka v cykle for: (inicializácia; podmienka; krok).
- Zátvorka v cykle foreach: (Typ premenná : kontajner).

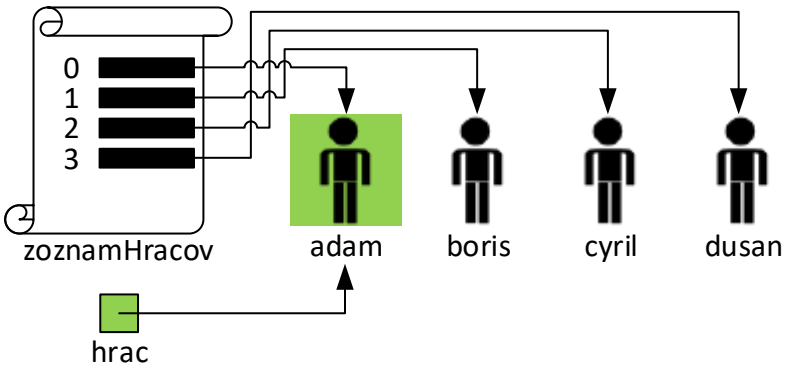
Prepíšme teraz horeuvedený cyklus **`for`** pomocou cyklu **`foreach`**. Musíme zistiť, čo je našim kontajnerom a čo riadiacou premennou.

- **Kontajner** v našom prípade predstavuje zoznam **`zasiahnutiHraci`**. V ňom sa nachádzajú prvky, ktoré chceme spracovať.
- **Riadiaca premenná** je **`hrac`** typu **`Hrac`**.

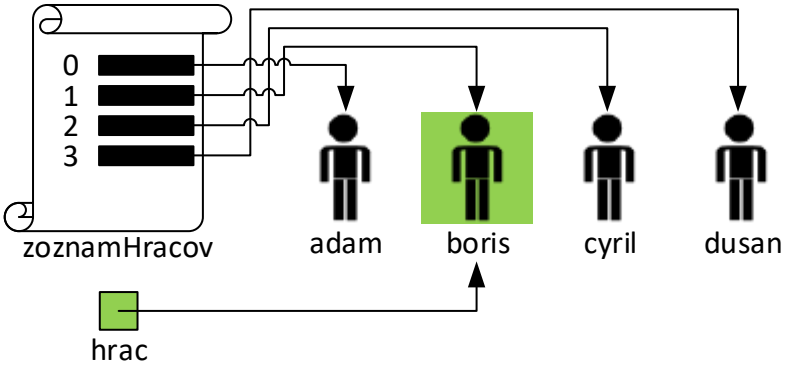
```
Arena arena = (Arena)this.getWorld();
for (int i = 0; i < zasiahnutiHraci.size(); i++) {
    Hrac hrac = zasiahnutiHraci.get(i);
    arena.odregistrujAOdstranHraca(hrac);
}

Arena arena = (Arena)this.getWorld();
for (Hrac hrac : zasiahnutiHraci) {
    arena.odregistrujAOdstranHraca(hrac);
}
```

Automatické v cykle **`foreach`**

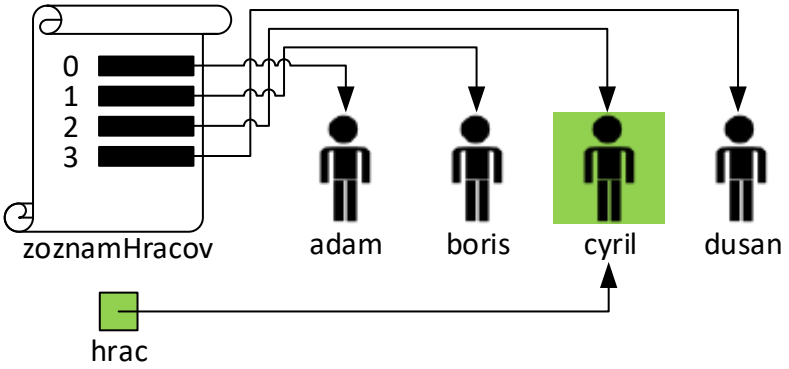
Princíp činnosti tohto cyklu ilustrujú aj nasledujúce obrázky.



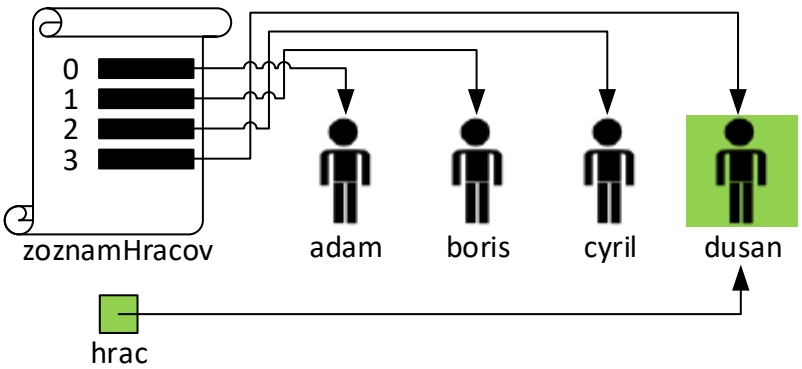
Obrázok 7.2: Prvá iterácia cyklu foreach



Obrázok 7.3: Druhá iterácia cyklu foreach



Obrázok 7.4: Tretia iterácia cyklu foreach



Obrázok 7.5: Štvrtá (v tomto prípade posledná) iterácia cyklu foreach

Zamyslime sa, či je správne, aby bol hráč odregistrovávaný bombou. Hráč nemusí ihneď skončiť hru – môže mať viac životov alebo štít, ktorý ho ochráni. Bude preto lepšie, ak hráč bude mať metódu, ktorou vhodne zareaguje na zásah, podobne, ako reaguje na to, keď vybuchne bomba.

ÚLOHA 7.9

Vytvorte v triede **`Hrac`** metódu **`zasah()`**, ktorá bude vyvolaná hráčovi bombou po tom, ako ho bomba zasiahne. Hráč sa v tejto metóde odregistruje zo sveta. Upravte kód v metóde **`act()`** triedy **`Bomba`** tak, aby odrážala novú funkčnosť.

Do triedy **Hrac** pridáme metódu:

```
public void zasah() {
     Arena arena = (Arena)this.getWorld();
     arena.odregistrujAOdstranHraca(this);
}
```

Upravená metóda **act** v triede **Bomba** bude obsahovať cyklus:

```
for (Hrac hrac : zasiahnutiHraci) {
     hrac.zasah(); // hráč bol zasiahnutý
}
```

Spomeňme si, že každá bomba pri svojom výbuchu notifikuje metódou **vybuchlaBomba()** svojho vlastníka. Takéto správanie však nebude vhodné, ak hráč (vlastník bomby) viac nebude vo svete existovať, pretože bol zasiahnutý. Zamyslime sa, ako zabezpečíme, aby bomby, ktoré ostali po hráčovi odstránenom zo sveta, viac takéhoto hráča nenotifikovali.

Bomba vo svojej metóde **act()** musí kontrolovať, či má vlastníka (bomba mohla byť napr. vytvorená arénou) a iba ak je vlastník nastavený, notifikuje ho metódou **vybuchlaBomba()**.

```
if (this.vlastnik != null) {
     this.vlastnik.vybuchlaBomba(this);
}
```

Ako jednoduché riešenie sa teda javí odstrániť vlastníka bomby, keď je hráč (vlastník) zasiahnutý.

ÚLOHA 7.10

Vytvorte v triede **Bomba** metódu **zrusVlastnika()**, ktorá nastaví jej atribút **vlastnik** na **null**.

```
public void zrusVlastnika() {
     this.vlastnik = null;
}
```

Teraz musíme zabezpečiť, aby všetky bomby toho istého vlastníka boli pri jeho prehre informované o zrušení vzťahu s ním. Môžeme použiť podobný princíp, aký sme využili pri registrácii hráčov v aréne. Každý hráč si môže evidovať všetky jeho aktívne bomby v atribúte **zoznamAktivnychBomb**, ktorý je typu **LinkedList<Bomba>**. So zoznamom môže pracovať podľa nasledujúcich pravidiel:

- Keď hráč vytvorí bombu (po stlačení správnej klávesy), novovytvorenú bombu pridá do zoznamu.
- Keď bomba vybuchne, hráč ju zo svojho zoznamu odstráni.
- Ak hráča nejaká bomba zasiahne, tak všetkým bombám, ktoré vytvoril, zruší vlastníka.

ÚLOHA 7.11

Vytvorte v triede **`Hrac`** atribút **`zoznamAktivnychBomb`** typu **`LinkedList<Bomba>`**. Inicializujte ho v správnom konštruktore. Upravte telá metód podľa nasledujúcich pravidiel:

- V metóde **`act()`** zaregistrujte novo vytvorenú bombu do zoznamu **`zoznamAktivnychBomb`.**
- V metóde **`vybuchlaBomba()`** odstráňte zo zoznamu **`zoznamAktivnychBomb`** bombu, ktorá prišla ako parameter (tá vybuchla).
- V metóde **`zasah()`** pomocou cyklu **`foreach`** zrušte vlastníka všetkým bombám zo zoznamu **`zoznamAktivnychBomb`**.

Najskôr musíme importovať balíček so zoznamom:

```
import java.util.LinkedList;
```

Deklarácia atribútu:

```
private LinkedList<Bomba> zoznamAktivnychBomb;
```

Atribút je inicializovaný v parametrickom konštruktore, ktorý je volaný zo všetkých ostatných konštruktorov:

```
this.zoznamAktivnychBomb = new LinkedList<Bomba>();
```

Relevantný kód v metóde **`act`**:

```
Bomba bomba = new Bomba(this, this.silaBomb, 30);
World svet = this.getWorld();
svet.addObject(bomba, this.getX(), this. getY());
this.pocetBomb = this.pocetBomb - 1;
this.zoznamAktivnychBomb.add(bomba);
```

Kód metódy **`vybuchlaBomba`**:

```
public void vybuchlaBomba(Bomba bomba) {
     this.pocetBomb = this.pocetBomb + 1;
     this.zoznamAktivnychBomb.remove(bomba);
}
```

Kód metódy **`zasah`**:

```
public void zasah() {
     Arena arena = (Arena)this.getWorld();
     arena.odregistrujAOdstranHraca(this);
     for (Bomba bomba : this.zoznamAktivnychBomb) {
           bomba.zrusVlastnika();
     }
}
```

ZHRNUTIE

Jazyk Java ponúka pre ukladanie objektov rovnakého typu kontajnery. Základným typom kontajnerov je zoznam. Poznáme viac druhov zoznamov, medzi najpoužívanejšie patrí **`LinkedList<E>`**, ktorý je definovaný v balíčku **`java.util.LinkedList`**. Predkom zoznamu **`LinkedList<E>`** je **`List<E>`**, ktorý definuje funkčnosť pre zoznamy – pridávanie, odoberanie a sprístupňovanie prvkov. K prvkom v zozname je možné pristupovať aj na základe indexu, najnižší index má hodnotu 0, posledný **`size - 1`** (**`size`** je počet prvkov v zozname).

Triedy, ktoré obsahujú generické zátvorky **`< >`** nazývame generické triedy. Do generických zátvoriek sa píšu generické parametre (triedy). Tieto typy sú potom vstupným a výstupným typom vhodných metód generickej triedy. Ako generický parameter sa v jazyku Java uvádza vždy trieda.

Každý kontajner je možné prejsť efektívne cyklom **`foreach`**. Za kľúčovým slovom **`for`** nasleduje v zátvorke riadiaca premenná cyklu a za dvojbodkou kontajner, ktorý chceme prejsť. Cyklus **`foreach`** zabezpečí, aby v každej iterácii cyklu bola hodnota v riadiacej premennej aktualizovaná na ďalšiu hodnotu prvku v kontajneri. Musí platiť, že typ riadiacej premennej je zhodný s typom objektov v kontajneri. Cyklus sa opakuje, kým nespracuje všetky prvky v kontajneri. Počas cyklu **`foreach`** nesmieme modifikovať kontajner (teda volať metódy ako je **`add()`** alebo **`remove()`**). Telo cyklu sa píše do zložených zátvoriek **`{ }`** a obsahuje príkazy, ktoré sa vykonajú pri každom opakovaní cyklu.

ÚLOHY NA PRECVIČOVANIE

ÚLOHA 7.A

Upravte výbuch bomby tak, aby zo sveta odstránila aj všetky inštancie triedy **`Mur`** v jej dosahu.

ÚLOHA 7.B

Pridajte hráčovi životy. Počiatočný počet životov môže byť pevne stanovený alebo ho môžete zadať v parametri konštruktora. Hráč je odregistrovaný zo sveta až po tom, ako bol zasiahnutý určený počet krát. Upravte metódu **`zasah()`** v triede **`Hrac`**.

ÚLOHA 7.C

Rozšírte hráča o možnosť okamžitej jednorazovej detonácie bômb. Definujte klávesu, ktorou odpálite všetky aktívne hráčove bomby. Umožnite hráčovi využiť túto schopnosť iba raz.

ÚLOHA 7.D

Upravte hru tak, aby aréna oživila vypadnutého hráča (pri hre viac ako dvoch hráčov), ak hra trvá posledným dvom hráčom pridlho. Hráča oživte na pozícií podľa svojho výberu.

8 CYKLUS WHILE A SÚKROMNÉ METÓDY

KĽÚČOVÉ SLOVÁ

Cyklus s podmienkou na začiatku. Súkromná metóda.

CIELE

V rámci tejto časti sa budeme venovať cyklu, ktorý sa vykonáva pokiaľ platí určitá podmienka, definovaná na začiatku cyklu. Naučíme sa tiež vytvárať súkromné metódy – metódy, ktoré môže vyvolávať iba inštancia tej triedy, v ktorej je metóda definovaná.

OBSAH

8.1 Analýza výbuchu bomby

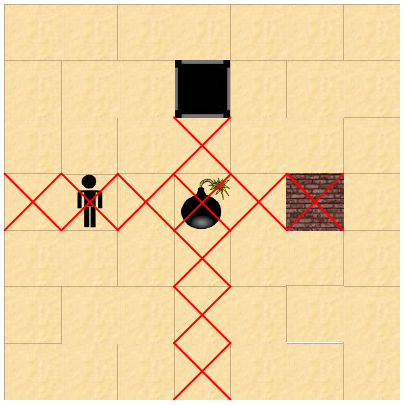
V predchádzajúcej časti naše bomby vybuchovali do vzdialenosti, ktorú určoval atribút `sila` bomby. Takýto výbuch nebol uspokojivý z viacerých dôvodov:

- Zasiahnutý je aj hráč, ktorý sa nachádza za stenou alebo prekážkou .
- Výbuch sme graficky nevideli.
- Bomby v hre Bomberman vybuchujú inak – iba v riadku a stĺpci, v ktorom sa nachádzajú.

Implementujme teda výbuch bomby správne. Na začiatok definujme charakteristiku výbuchu tak, ako by mal v hre správne prebiehať:

- Každá bomba má svoju silu. Sila určuje, ako ďaleko v každom smere bomba vybuchne.
- Výbuch bomby zastavujú steny a múry. Stenu výbuch nijako nepoškodí a nešíri sa za ňu. Naopak, múr sa následkom výbuchu zničí, ale výbuch zaň nepokračuje.
- Ak výbuch zasiahne hráča, tak môže nastať koniec hry alebo remíza (ak vypadli naraz obaja hráči). Výbuch sa na políčku s hráčom nezastaví.

Pre lepšie pochopenie priebehu výbuchu sa pozrime na nasledujúci obrázok:



Obrázok 8.1: Výbuch bomby so silou tri. Červené krížiky znázorňujú zasiahnuté bunky.

Pripomeňme si ešte, ako v súčasnosti bomba vzniká, kedy bomba vybuchne a ako spolupracujú bomba a hráč. Hráč bombu vytvára ak sa stlačí príslušný kláves a navyše iba ak má dostupné bomby (jeho atribút **pocetBomb** je väčší ako 0). Pri vytvorení bomby jej hráč nastaví časovač a vlastníka (na seba). Hráč bombu zaradí do svojho zoznamu aktívnych bômb. Bomba následne v metóde **act()** „tiká" (znižuje hodnotu svojho atribútu **casovac**), až kým tento nedosiahne hodnotu 0 – vtedy bomba vybuchne. Pri výbuchu bomba najskôr svojmu vlastníkovi oznámi, že vybuchla vyvolaním hráčovej metódy **vybuchlaBomba()** (ten si v tejto metóde zvýši počet bômb a odstráni vybuchnutú bombu zo zoznamu aktívnych bômb) a všetkým hráčom, ktorí sú v jej dosahu (do vzdialenosti sila) vyvolá metódu **zasah()** (v nej sa hráči odregistrujú z arény a všetkým svojim aktívnym bombám vyvolajú metódu **zrusVlastnika()**).

8.2 Výbuch bomby

Začnime jednoduchým výbuchom bomby. Aby bol výbuch viditeľný, vytvoríme v centre výbuchu oheň.

ÚLOHA 8.1

Vytvorte triedu **Ohen**. Zvoľte vhodnú grafickú reprezentáciu. Konštruktor tejto triedy má jeden parameter, ktorý určuje, ako dlho oheň na svojom mieste horí. Zabezpečte, aby oheň po prejdení daného času zmizol zo sveta.

```
public class Ohen extends Actor {

    private int casovac;

    public Ohen(int casovac) {
        this.casovac = casovac;
    }

    public void act() {
        this.casovac = this.casovac - 1;
        if (this.casovac == 0) {
            World svet = this.getWorld();
            svet.removeObject(this);
        }
    }

}
```

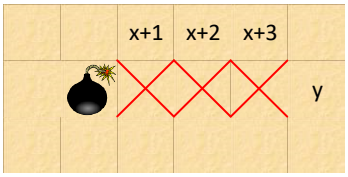
ÚLOHA 8.2

Upravte existujúci kód výbuchu bomby tak, aby na mieste bomby zanechal inštanciu triedy **Ohen**. Otestujte svoje riešenie.

Do metódy **act()** v triede Bomba doplníme:

```
// na svojom mieste vytvorí bomba oheň ešte predtým,
// ako sa odstráni zo sveta (aby boli dostupné jej súradnice)
svet.addObject(new Ohen(5), this.getX(), this.getY());
```

Poďme teraz napísať kód, ktorý rozšíri výbuch doprava od bomby. Na obrázku 8.2 môžeme vidieť bunky, ktoré budú výbuchom zasiahnuté. Pre zasiahnuté bunky platí, že sa nachádzajú na rovnakom riadku (majú rovnakú y-ovú súradnicu, ako má bunka, na ktorej sa nachádza bomba) a rôznu x-ovú súradnicu, ktorá určuje stĺpec. Výbuch sa rozšíri v danom smere do takého počtu buniek, ktorý zodpovedá hodnote atribútu **sila** bomby. Poznáme preto presný počet buniek, a teda môžeme výbuch rozšíriť pomocou cyklu s pevným počtom opakovaní.



Obrázok 8.2: Výbuch bomby so silou 3 smerom doprava

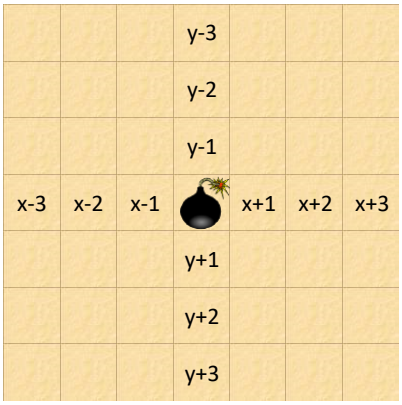
ÚLOHA 8.3

Pomocou cyklu **for** rozšírte výbuch bomby (vytvorte inštancie triedy **Ohen**) v smere doprava od bomby. Výbuch rozšírte na toľko buniek, koľko udáva atribút **sila** bomby.

Do metódy **act()** v triede **Bomba** pridáme cyklus **for** v kóde za miesto, kde sme vytvorili inštanciu triedy Ohen.

```
for (int i = 1; i <= this.sila; i = i + 1) {
     svet.addObject(new Ohen(5), this.getX() + i, this.getY());
}
```

Nakreslime, ako sa menia súradnice do všetkých strán od centra výbuchu. Dostaneme nasledujúci obrázok:



Obrázok 8.3: Zmena súradníc v rôznych smeroch od bomby

ÚLOHA 8.4

Upravte výbuch bomby tak, aby generoval ohne do všetkých strán. Pomôžte si zmenou súradníc zobrazenou na predchádzajúcom obrázku.

```
// rozšírime ohne v smere doprava
for (int i = 1; i <= this.sila; i = i + 1) {
     svet.addObject(new Ohen(5), this.getX() + i, this.getY());
}
// rozšírime ohne v smere doľava
for (int i = 1; i <= this.sila; i = i + 1) {
     svet.addObject(new Ohen(5), this.getX() - i, this.getY());
}
// rozšírime ohne v smere nahor
for (int i = 1; i <= this.sila; i = i + 1) {
     svet.addObject(new Ohen(5), this.getX(), this.getY() - i);
}
// rozšírime ohne v smere nadol
for (int i = 1; i <= this.sila; i = i + 1) {
     svet.addObject(new Ohen(5), this.getX(), this.getY() + i);
}
```

8.3 Cyklus while

Zamyslime sa nad cyklami, ktoré sme napísali. Tieto cykly vždy rozšíria presný počet ohňov. Čo však, ak sa bomba nachádza blízko okraja, pri stene, či múre? Vtedy sa v danom smere nerozšíri toľko ohňov, ako je sila výbuchu, ale iba zodpovedajúci počet (príklady sú na obrázku 8.1). Musíme preto upraviť cyklus a to tak, že sa nebudeme spoliehať iba na silu bomby, ale budeme brať do úvahy aj jej okolie. Slovne by sme mohli výbuch v smere doprava popísať nasledovne (oranžovou farbou je vyznačený text, ktorý pochádza z cyklu **`for`**):

1) Poznač si, ktorý oheň umiestňuješ. Na začiatku je to prvý oheň. (inicializácia **`int i = 1`**).
2) Ak ešte neboli položené všetky ohne (podmienka **`i <= this.sila`**) a zároveň je možné do nasledujúcej bunky položiť oheň, tak
    a. Vlož do bunky v aktuálnom stĺpci oheň.
    b. Poznač si, že umiestňuješ ďalší oheň (krok **`i = i + 1`**).
3) Opakuj krok 2.

Všimnime si, že v kroku 2 vykonáme príkazy a. a b. iba vtedy, ak platí podmienka: *Ak ešte neboli položené všetky ohne a zároveň je možné do nasledujúcej bunky položiť oheň*. To by naznačovalo použitie príkazu **`if`**. Avšak po skončení týchto dvoch krokov sa chceme opäť vrátiť a skontrolovať, či je možné výbuch ešte rozšíriť. Už vieme, že ak sa niečo opakuje, použijeme cyklus – v pôvodnom kóde sme na to využívali cyklus **`for`**. Keďže však už **nevieme** presný počet opakovaní, môžeme použiť iný druh cyklu – **`while`** – a tým nahradiť v tejto situácii nevyhovujúci cyklus **`for`**.

ZAPAMÄTAJTE SI:

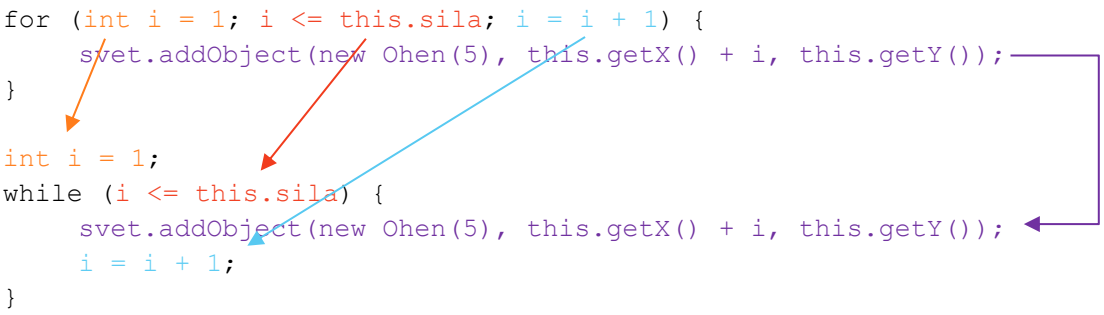
Cyklus **while** je cyklus s podmienkou na začiatku:

```
while (podmienka) {
}
```

Skladá sa z kľúčového slova **`while`**, za ktorým nasleduje podmienka (môže to byť akýkoľvek výraz typu **`boolean`**) v povinnej zátvorke. Cyklus **`while`** funguje takto:

1) Najskôr skontroluje, či podmienka platí. Ak áno, tak sa vykonajú príkazy v tele cyklu (tie, ktoré sú uvedené medzi zátvorkami **`{ }`**). Ak podmienka neplatí, cyklus končí.
2) Po vykonaní tela cyklu sa pokračuje bodom 1.

Akýkoľvek cyklus **`for`** je možné prepísať s pomocou cyklu **`while`**:



V cykle **`for`** sa inicializácia vykoná práce raz (teda v cykle **`while`** pred samotným cyklom), podmienka sa testuje pred každou iteráciou cyklu (rovnako, ako v cykle **`while`**) a ak podmienka platí, tak sa vykoná telo (rovnako, ako v cykle **`while`**) a vykoná sa krok (posledný príkaz v tele cyklu **`while`**).

ÚLOHA 8.5

Prepíšte všetky cykly **`for`** pre šírenie ohňa s využitím cyklu **`while`**. Druhú časť podmienky (je možné do nasledujúcej bunky položiť oheň) zatiaľ vynechajte.

```
int i = 1;
// rozšírime ohne v smere doprava
while (i <= this.sila) {
     svet.addObject(new Ohen(5), this.getX() + i, this.getY());
     i = i + 1;
}

i = 1;
// rozšírime ohne v smere doľava
while (i <= this.sila) {
     svet.addObject(new Ohen(5), this.getX() - i, this.getY());
     i = i + 1;
}
```

```
i = 1;
// rozšírime ohne v smere nahor
while (i <= this.sila) {
     svet.addObject(new Ohen(5), this.getX(), this.getY() - i);
     i = i + 1;
}

i = 1;
// rozšírime ohne v smere nadol
while (i <= this.sila) {
     svet.addObject(new Ohen(5), this.getX(), this.getY() + i);
     i = i + 1;
}
```

8.4 Súkromná metóda

Všimnime si, že cykly, ktoré generujú riadky a stĺpce výbuchov sú si veľmi podobné. Ak by sme teraz chceli pridať do podmienky cyklu **`while`** časť, ktorá zastaví výbuch, museli by sme to spraviť štyrikrát. Zamyslime sa, či to nedokážeme urobiť efektívnejšie.

Vieme, že každý objekt môže mať svoje metódy. Metóda obsahuje postupnosť príkazov. Každému objektu môžeme jeho metódy vyvolať. Niekedy sa však hodí, aby sme definovali metódu, ktorú môže využívať iba objekt samotný, pretože zastrešuje činnosť, ktorá nemusí byť mimo objektu samotného dostupná. V takejto situácii sa nachádzame aj teraz – bolo by vhodné vytvoriť metódu, ktorá rozšíri výbuch do správnej strany. Takúto funkčnosť však objekt nemusí poskytnúť navonok, nakoľko nemá samostatne veľký význam (pri výbuchu bomby sa toho musí udiať viac, ako iba rozšíriť ohne).

ZAPAMÄTAJTE SI!

Každej metóde objektu môžeme pomocou kľúčových slov **`public`** a **`private`** modifikovať jej **viditeľnosť** pred ostatnými objektmi.
Metódy, ktorých hlavička začína kľúčovým slovom **`public`** môžu byť volané kýmkoľvek a sú teda dostupné aj pre potomkov tej triedy, v ktorej je metóda definovaná. Takéto metódy označujeme ako **verejné metódy**.
Metódy, ktorých hlavička začína kľúčovým slovom **`private`** patria výsostne triede, v ktorej sú definované. Nie sú dostupné pre nikoho, ani pre potomkov triedy, kde je takáto metóda definovaná. Takéto metódy označujeme ako **súkromné metódy**.

Pre rozšírenie ohňa teda vytvorme súkromnú metódu **`rozsirOhen()`**, ktorá nemá žiadnu návratovú hodnotu. Metóda bude vedieť rozšíriť oheň v smere, ktorý určíme parametrami. Pred určením parametrov tejto metódy je vhodné urobiť analýzu. Pozrime sa čo sa mení v štyroch cykloch **`while`** vyššie? Vidíme, že všetko ostáva rovnaké okrem súradníc x a y, na ktoré vkladáme oheň. Naviac, tieto sa v každom cykle menia inak, ako to sumarizuje nasledujúca tabuľka.

Tabuľka 8.1: Zmeny súradníc v smeroch od stredu bomby

Smer	Zmena x-ovej súradnice	Zmena y-ovej súradnice
Doprava	Pripočítava 1	Bez zmeny
Doľava	Odpočítava 1	Bez zmeny
Nahor	Bez zmeny	Odpočítava 1
Nadol	Bez zmeny	Pripočítava 1

Parametre metódy teda budeme potrebovať dva – jeden pre určenie zmeny x-ovej a druhý pre určenie zmeny y-ovej súradnice. Hodnoty týchto parametrov získame jednoducho: Ak sa v danom smere nemá vykonať žiadna zmena, do parametra zadáme 0. Ak má v danom smere súradnica rásť, zadáme +1 (pripočítavame), ak má klesať, parameter bude mať hodnotu -1 (odpočítavame). Môžeme teda napísať hlavičku našej súkromnej metódy a prázdne telo:

```
private void rozsirOhen(int zmenaX, int zmenaY) {
}
```

Telo metódy tvorí cyklus **`while`**, ktorý rozširuje nové inštancie triedy **`Ohen`**. Musíme však upraviť vzťah pre získanie súradníc, na ktoré má byť táto inštancia vložená do sveta. Všimnime si, že vzdialenosť od bomby sme vyjadrili premennou **`i`**. Naše parametre nadobúdajú iba hodnoty -1, 0, +1. Stačilo by, aby sme v každej iterácií cyklu vynásobili hodnotu v premennej i príslušným parametrom a získali by sme dané posunutie od súradníc bomby.

Zmenu súradníc vieme získať aj iným spôsobom. Ak v každej iterácií cyklu zmeníme aktuálnu polohu tak, že k nej pripočítame zmenu a to budeme považovať za aktuálnu polohu, tak dostaneme rovnaké správanie. Celá metóda s rešpektovaním zmeny súradníc teda môže vyzerať takto:

```
private void rozsirOhen(int zmenaX, int zmenaY) {
     // inicializujeme počítadlo ohňov
     int i = 1;
     // vypočítame súradnice, na ktoré sa bude klásť oheň
     int suradnicaStlpec = this.getX() + zmenaX;
     int suradnicaRiadok = this.getY() + zmenaY;

     World svet = this.getWorld();
     while (i <= this.sila) {
          svet.addObject(new Ohen(5),
                         suradnicaStlpec,
                         suradnicaRiadok);

          // aktualizujeme počítadlo ohňov
          i = i + 1;
          // prepočítame súradnice, na ktoré budeme klásť ďalší oheň
          suradnicaStlpec = suradnicaStlpec + zmenaX;
          suradnicaRiadok = suradnicaRiadok + zmenaY;
      }

 }
```

ÚLOHA 8.6

Využite súkromnú metódu **`rozsirOhen()`** na rozšírenie ohňa po výbuchu bomby.

Namiesto štyroch cyklov **`while`** v tele metódy **`act()`** v triede **`Bomba`** môžeme napísať:

```
this.rozsirOhen(+1, 0); // rozšírime ohne v smere doprava
this.rozsirOhen(-1, 0); // rozšírime ohne v smere doľava
this.rozsirOhen(0, -1); // rozšírime ohne v smere nahor
this.rozsirOhen(0, +1); // rozšírime ohne v smere nadol
```

Po vytvorení jedinej metódy na rozšírenie výbuchu do zadanej strany môžeme jednoducho pridať podmienku, či môže výbuch pokračovať. Výbuch zastavia tri situácie:

1) Oheň dosiahol okraj arény.
2) V bunke sa nachádza stena.
3) V bunke sa nachádza múr.

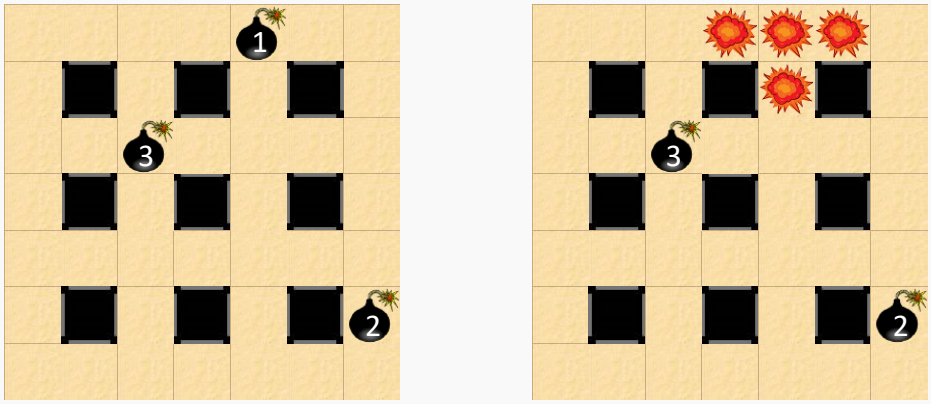
ÚLOHA 8.7

Vytvorte v triede **`Bomba`** súkromnú metódu **`mozeBunkaVybuchut()`**, ktorá v parametroch preberie súradnice riadku a stĺpca a vráti **`true`**, ak na danej bunke môže výbuch nastať podľa prvých dvoch podmienok vyššie. Ak to nie je možné, metóda vráti **`false`**.

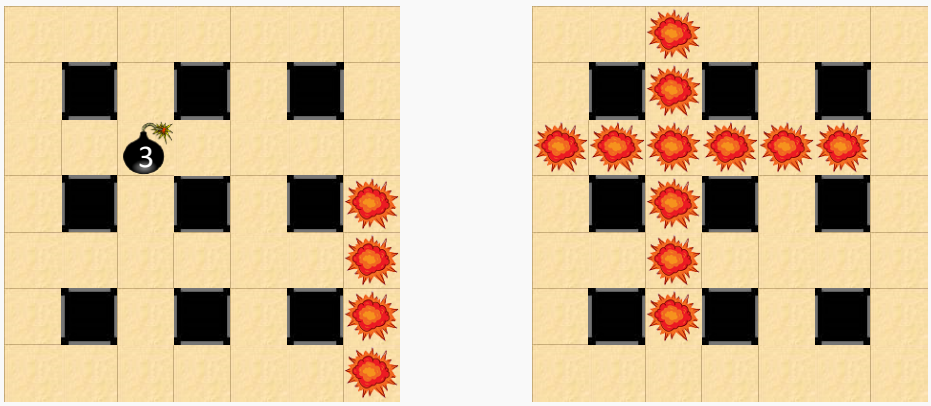
```
private boolean mozeBunkaVybuchnut(int x, int y) {
     World svet = this.getWorld();
     if (x >= 0 && x < svet.getWidth() &&
         y >= 0 && y < svet.getHeight()) {
           List<Stena> steny = svet.getObjectsAt(x, y, Stena.class);
          return steny.isEmpty();
     }
     else {
           // súradnice nepatria do sveta
           return false;
     }
}
```

ÚLOHA 8.8

S využitím metódy **`mozeBunkaVybuchnut()`** môžete teraz upraviť podmienku v cykle **`while`** v metóde **`rozsirOhen`** v triede **`Bomba`**. Upravte podmienku v cykle tak, aby sa rešpektoval výsledok kontroly z metódy **`mozeBombaVybuchnut()`**. Otestujte funkčnosť riešenia s bombami s rôznou silou medzi stenami. Testy rôznych výbuchov sú zobrazené na nasledujúcich obrázkoch:



a b



c d

V časti (a) vidíme pôvodné rozloženie, v časti (b) vybuchla bomba so silou 1, v časti (c) vybuchla bomba so silou 2 a v časti (d) bomba so silou 3.

Podmienku v cykle **`while`** v metóde **`rozsirOhen()`** upravíme takto:

```
while (i <= this.sila &&
       this.mozeBunkaVybuchnut(suradnicaStlpec, suradnicaRiadok)){
```

Z analýzy na obrázku 8.1 sme zistili, že výbuch bomby zastaví aj inštancia triedy **`Mur`**. Na rozdiel od steny však na jej mieste vznikne inštancia triedy **`Ohen`**. Zamyslime sa teda, ako je potrebné upraviť metódu na šírenie ohňa.

Keďže sa na miesto múru môže vložiť oheň, daná bunka môže vybuchnúť. Nie je preto potrebný žiadny zásah do metódy **`mozeBunkaVybuchnut()`**. Rozdiel však nastáva po tom, ako bol oheň umiestnený. Ak sa tak stalo na políčku, kde je stena, tak výbuch ďalej nemôže pokračovať. Pre zastavenie výbuchu stačí jednoducho navýšiť hodnotu v premennej **`i`** na číslo väčšie, ako je sila bomby. Toto spôsobí, že prestane platiť prvá časť podmienky v cykle **`while`**, a teda sa nevykoná ďalšia iteráciu cyklu. Pre zistenie, či môže výbuch ďalej pokračovať, môžeme zaviesť ďalšiu súkromnú metódu v triede **`Bomba`**, v ktorej tento fakt zistíme.

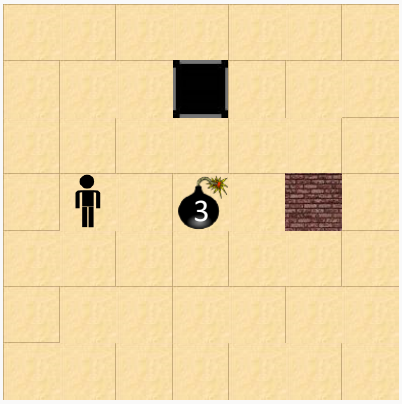
ÚLOHA 8.9

Pridajte do triedy **Bomba** súkromnú metódu **mozeVybuchPokracovat()**, ktorá v parametroch preberie súradnice riadku a stĺpca a vráti **true**, ak bunka na daných súradniciach nezastavila výbuch. Ak bunka výbuch zastavila, tak metóda vráti **false**. Výbuch nemôže pokračovať, ak narazil na múr.

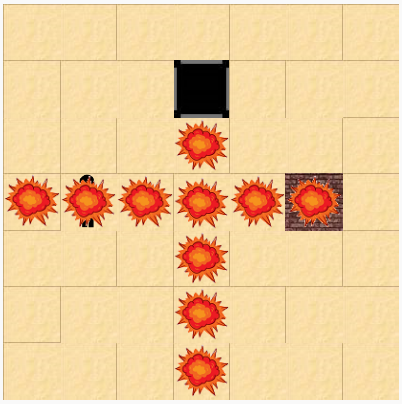
```
private boolean mozeVybuchPokracovat(int x, int y) {
     World svet = this.getWorld();
     if (x >= 0 && x < svet.getWidth() &&
         y >= 0 && y < svet.getHeight()) {
           List<Mur> mury = svet.getObjectsAt(x, y, Mur.class);
          return mury.isEmpty();
     }
     else {
           // súradnice nepatria do sveta
           return false;
     }
}
```

ÚLOHA 8.10

S využitím metódy **mozeVybuchPokracovat()** upravte metódu **rozsirOhen()** v triede **Bomba**. Ak výbuch môže ďalej z danej bunky pokračovať, zvýšime hodnotu premennej **i** o 1 a prepočítame súradnice nového riadku a stĺpca výbuchu. Inak umelo zvýšime hodnotu premennej **i** na hodnotu vyššiu ako je sila bomby, čím sa zastaví cyklus. Otestujte svoje riešenie na situácií z nasledovného obrázku:



a



b

Navýšenie premennej **i** v metóde **mozeVybuchPokracovat()** upravíme takto:

```
if (this.mozeVybuchPokracovat(suradnicaStlpec, suradnicaRiadok)) {
     i = i + 1;
     suradnicaStlpec = suradnicaStlpec + zmenaX;
     suradnicaRiadok = suradnicaRiadok + zmenaY;
}
else {
     // ak výbuch nemôže pokračovať, zvýšime hodnotu v premennej i,
     // čo zabezpečí, že podmienka v cykle while nebude platiť
     i = this.sila + 1;
}
```

8.5 Reakcia rôznych prvkov na oheň

Poslednou časťou je odstránenie zasiahnutých prvkov z arény. Zamyslime sa, aký vplyv má oheň na potomkov triedy **Actor**:

- **Stena**: na bunky, v ktorých je umiestnená stena sa oheň nikdy nedostane.
- **Mur**: oheň spôsobí okamžité odstránenie dotknutého múru zo sveta.
- **Hrac**: oheň spôsobí okamžité odstránenie hráča zo sveta. Bomba už teda nebude volať metódu **zasah()** triedy **Hrac** sama.
- **Bomba**: oheň spôsobí detonáciu bomby.
- **Ohen**: oheň ohňu nevadí, nestane sa nič.

Jednoduchý spôsob, ako dosiahnuť správnu reakciu dotknutých tried je umiestniť ju do ich metódy **act()**. Ak na začiatku metódy **act()** aktor zistí, že stojí na rovnakej bunke s ohňom, musí na to vhodne zareagovať.

Začnime triedou **Mur**. Do jej metódy **act()** pridajme test, či sa prekrýva s nejakou inštanciou triedy **Ohen**. Ak áno, odstránime múr zo sveta. Využiť môžeme metódu **isTouching()**, ktorá má jeden parameter – triedu (potomka triedy **Actor**). Metóda vráti **true**, ak sa objekt prekrýva s inštanciou triedy odovzdanej ako parameter.

```
public void act() {
     if (this.isTouching(Ohen.class)) {
          World svet = this.getWorld();
          svet.removeObject(this);
     }
}
```

ÚLOHA 8.11

Upravte správanie inštancie triedy **Bomba** tak, aby nevolala metódu **zasah()** hráčov v jej okolí. Namiesto toho bude hráč sám kontrolovať, či sa neprekrýva s ohňom. Upravte správanie hráča v jeho metóde **act()** tak, aby najskôr zistil, či sa neprekrýva s inštanciou triedy **Ohen**. Ak áno, sám vyvolá vlastnú metódu **zasah()**. Takto zabezpečíme, aby bol hráč zasiahnutý aj ohňom, ktorý horí po výbuchu bomby.

Začneme odstránením nasledujúcich riadkov z metódy **act()** v triede **Bomba** (stačí, aby bomba iba vytvorila ohne a odstránila sa zo sveta):

```
List<Hrac> zasiahnutiHraci = this.getObjectsInRange(this.sila,
                                                    Hrac.class);
for (Hrac hrac : zasiahnutiHraci) {
     hrac.zasah();
}
```

Metódu **act()** v triede hráč upravíme tak, aby hráč sám skontroloval zásah. Ak bol zasiahnutý, potom už nemôže reagovať na ovládanie – preto ďalšie príkazy vložíme do vetvy **else**.

```
if (this.isTouching(Ohen.class)) {
     this.zasah();
}
else { ...
```

ÚLOHA 8.12

Upravte metódu **act()** v triede **Bomba** tak, aby bomba vybuchla aj keď je na rovnakej bunke s ohňom. Riešenie overte tak, že spravíte reťazovú reakciu niekoľkých bômb.

Upravíme podmienku výbuchu v metóde **act()**:

```
if (this.casovac == 0 || this.isTouching(Ohen.class))
```

ZHRNUTIE

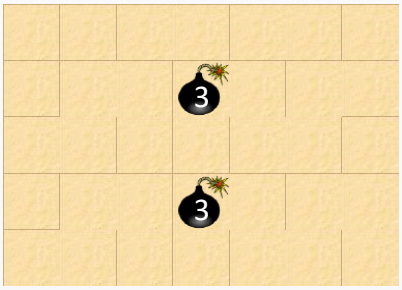
Cyklus **while** je cyklus s podmienkou na začiatku. Začína kľúčovým slovom **while**, za ktorým v zátvorke nasleduje podmienka (logický výraz typu **boolean**). Ak podmienka platí, začnú sa postupne vykonávať príkazy v tele cyklu. Po dokončení posledného príkazu v tele cyklu sa opätovne skontroluje podmienka. Cyklus končí, keď podmienka prestane platiť a vykonávanie pokračuje ďalším príkazom za telom cyklu.

Všetkým metódam v jazyku Java je možné nastaviť ich viditeľnosť. Hlavička verejných metód začína kľúčovým slovom **public**. Verejné metódy sú dostupné pre všetky objekty bez obmedzení. Hlavičky súkromných metód začínajú kľúčovým slovom **private** a je ich možné použiť iba v inštanciách tých tried, v ktorých sú definované (nemôžu ich využívať ani potomkovia týchto tried). Súkromné metódy zväčša zastrešujú čiastkové kroky inak zložitých metód alebo činnosti, ktoré objekt nechce poskytovať navonok.

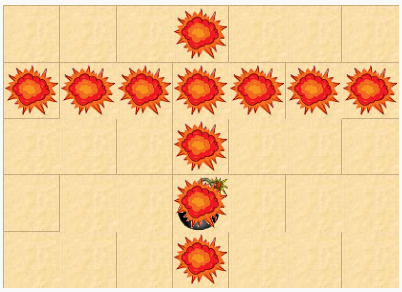
ÚLOHY NA PRECVIČOVANIE

ÚLOHA 8.A

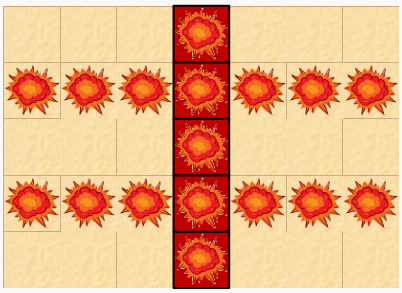
Upravte metódu **`rozsirOhen()`** tak, aby negenerovala ohne v bunkách, v ktorých sa už oheň nachádza (napr. v dôsledku reťazovej reakcie). Šírenie výbuchu sa týmto ale nezastaví. Nasledovný obrázok ukazuje príklad prekrytia viacerých inštancií triedy Ohen. Časť (a) ukazuje postavenie dvoch bômb. V časti (b) vybuchla horná bomba, čo spôsobilo umiestnenie ohňov a zároveň reťazovú reakciu dolnej bomby. Tá opäť umiestni ohne (časť (c)), ale niektoré z nich by mali byť vkladané do buniek, kde sa už ohne nachádzajú v dôsledku výbuchu prvej bomby (červené bunky). Do týchto buniek teda druhá bomba ohne negeneruje.



a



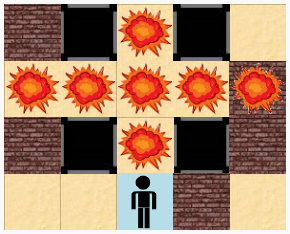
b



c

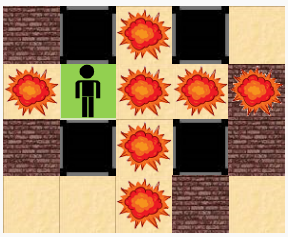
ÚLOHA 8.B

Vytvorte v aréne nový druh aktora – silové pole. Toto pole chráni hráča pred účinkami ohňa. Bunka so silovým poľom nemôže vybuchnúť ani sa cez takúto bunku nemôže šíriť oheň. Silové pole môže zaniknúť iba tak, že sa priamo naň umiestni bomba.

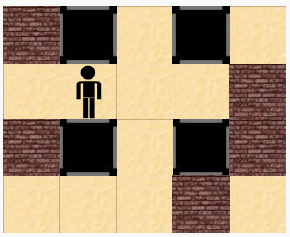


ÚLOHA 8.C

Vytvorte v aréne nový druh aktora – silový štít. Toto pole chráni hráča pred účinkami ohňa, ale iba jednorázovo. Bunka so silovým štítom nemôže vybuchnúť, ale ak bol štít zasiahnutý, odstráni sa. Oheň sa cez silový štít môže šíriť ďalej. Obrázok zobrazuje silový štít pred výbuchom (a) a po výbuchu (b) bomby so silou tri.



a



b

9 POLYMORFIZMUS

KĽÚČOVÉ SLOVÁ

Virtuálna metóda. Prekrytie metódy. Polymorfizmus. Protected. Pretypovanie.

CIELE

Cieľom tejto časti je naučiť sa tvoriť virtuálne metódy a prekrývať ich podľa potreby v potomkoch triedy. Taktiež zavedieme novú viditeľnosť atribútov a metód – chránená (`protected`).

OBSAH

9.1 Pridanie míny

V tejto kapitole sa naučíme ako je možné efektívne definovať funkčnosť podobných objektov tak, aby kostra zostala rovnaká, ale líšili sa prípadné detaily. Už vieme, že na definovanie spoločnej funkčnosti je možné využiť dedičnosť. Pri nej predok definuje spoločné metódy, ktoré potomkovia môžu využiť. Niekedy sa však môže stať, že potomkovia potrebujú v tej istej metóde vykonať odlišnú činnosť. Spomeňme si na hudobníkov v orchestri. Môžeme definovať triedu predka `Hudobnik` a potom niekoľko jej potomkov. Každý hudobník vie hrať, takže trieda `Hudobnik` by mohla definovať metódu `hraj()`. Je však rozdiel, či hrá bubeník alebo gitarista – metódy `hraj()` v potomkoch `Bubenik` a `Gitarista` preto budú vyzerať rozdielne. Tento rozdiel vieme zabezpečiť využitím ďalšej z vlastností objektovo orientovaného programovania – **polymorfizmu**.

Uvažujme, že náš hráč bude vedieť klásť nielen bomby, ale aj iný druh výbušnín – míny. Mína na rozdiel od bomby nevybuchne vtedy, keď vyprší časovač, ale vtedy, ak na ňu niekto stúpi a zanechá po sebe oheň (iba na jedinom poli, kde sa mína nachádzala).

ÚLOHA 9.1

Začnime pridaním míny. Mína vybuchne práve vtedy, keď na ňu hráč stúpi. Mína tiež vždy vybuchne, keď ju zasiahne oheň (napr. z bomby, ktorá vybuchla blízko). Na svojom mieste (a iba tam) zanechá oheň. Pridajte hráčovi možnosť klásť míny (podobne, ako bomby) po stlačení klávesy (napr. `control` alebo `shift`). Podobne, ako v prípade bômb, má hráč aj obmedzený počet mín (teda ak položí všetky míny, tak ďalšiu mínu môže položiť až vtedy, ak niektorá z predtým položených mín vybuchne). Počiatočný počet mín získa hráč z parametra konštruktora. Pre evidovanie mín a reakciu na ich výbuch v triede `Hrac` postupujte rovnako, ako pri bombách (vytvorte zoznam mín, pridajte metódy `vybuchlaMina()`, `mozePolozitMinu()`, atď.).

```
public class Mina extends Actor {

    private Hrac vlastnik;
    private int casovac;

    public Mina(Hrac vlastnik)
    {
          this.vlastnik = vlastnik;
          this.casovac = 5; // prvých 5 tikov nie je mína aktívna
    }

    public void act()
    {
          boolean maVybuchnut = false;

          // mína má určite vybuchnúť, ak sa jej dotýka oheň
          if (this.isTouching(Ohen.class)) {
                maVybuchnut = true;
          }
          else {
                // oheň sa jej nedotýka
                if (this.casovac > 0) {
                      // ak ešte nevypršal časovač,
                      // tak mína nie je aktívna
                      // iba znížime časovač
                      this.casovac = this.casovac - 1;
                }
                else {
                      // časovač vypršal, mína je aktívna.
                      // skontrolujeme, či na nej niekto stojí
                      if (this.isTouching(Hrac.class)) {
                            maVybuchnut = true;
                      }
                }
          } // ak sa jej dotýka oheň

          if (maVybuchnut) {
                // dáme hráčovi vedieť, že táto mína vybuchuje
                if (this.vlastnik != null) {
                      this.vlastnik.vybuchlaMina(this);
                }

                // na svojom mieste vytvorí mína oheň ešte predtým,
                // ako sa odstráni zo sveta
                // (aby boli dostupné jej súradnice)
                World svet = this.getWorld();
                svet.addObject(new Ohen(5),
                               this.getX(),
                               this.getY());
                // mína sa odstráni zo sveta
                svet.removeObject(this);
          } // ak má vybuchnúť
    } // act

    public void zrusVlastnika()
```

```
    {
    this.vlastnik = null;
    }

} // trieda
```

Triedu **Hrac** upravíme pre míny analogicky k bombám.

Trieda **Mina** je veľmi podobná triede **Bomba**. Oba prvky môžeme považovať za výbušniny. Vieme, že je vhodné vytvoriť spoločného predka tým triedam, ktoré zdieľajú spoločné vlastnosti alebo spoločnú funkčnosť.

ÚLOHA 9.2

Vytvorte spoločného predka pre triedy **Bomba** a **Mina** – triedu **Vybusnina**. Ktoré atribúty a metódy je vhodné presunúť do predka a ktoré by mali ostať v potomkoch? Upravte podľa vášho návrhu triedy.

```
public class Vybusnina extends Actor {
    private Hrac vlastnik;

    public Vybusnina(Hrac vlastnik) {
        this.vlastnik = vlastnik;
    }

    public void act() {
    }

    public void zrusVlastnika() {
        this.vlastnik = null;
    }
}
```

Z triedy **Bomba** aj z triedy **Mina** odstránime atribút **vlastnik**. Upravíme hlavičky tried tak, aby boli triedy odvodené z triedy **Vybusnina**.

```
public class Bomba extends Vybusnina

public class Mina extendy Vybusnina
```

Keďže konštruktor predka je parametrický, je potrebné ho v potomkoch uviesť. Prvý riadok konštruktora potomkov preto bude:

```
super(vlastnik);
```

Do predka mohla byť presunutá aj metóda **zrusVlastnika()**.

Všimnite si, že aj trieda **Bomba** aj trieda **Mina** obsahujú celočíselný atribút **casovac**. Mohlo by sa zdať, že je dobré takýto atribút premiestniť do spoločného predka. Problém je v rôznej úlohe tohto atribútu:

- V triede **Bomba** vyjadruje, kedy musí vybuchnúť.
- V triede **Mina** vyjadruje, kedy sa stáva aktívna.

Atribút má síce rovnaký názov, ale jeho hodnoty majú iné dôsledky, preto ho musíme ponechať v potomkoch.

Jediný vhodný atribút na presun do predka je **vlastnik** (všetky výbušniny môžu mať svojho vlastníka). Problém je ale tentokrát s prístupom k nemu. Vieme, že atribúty definujú vnútorný stav každého objektu a je dobré, aby boli výlučne v správe objektu samotného. Preto vždy používame kľúčové slovo **private**, ktorým určujeme súkromnú viditeľnosť atribútu. Viditeľnosť **private** ale zároveň spôsobí, že atribút môže využívať iba trieda samotná, ani jej potomkovia k nemu nemajú prístup (rovnako, ako pri súkromných metódach). Potrebujeme teda využiť nový druh viditeľnosti, ktorý ukryje atribút pred okolitými objektami, ale umožní jeho využitie v potomkoch triedy, v ktorej je definovaný.

ZAPAMÄTAJTE SI!

Každému atribútu, metóde a konštruktoru objektu je možné definovať rozdielnu viditeľnosť: Pomocou kľúčového slova **public** (verejný) vyjadrujeme, že k danému atribútu, metóde alebo konštruktoru má prístup akýkoľvek iný objekt alebo potomok triedy a trieda samotná, kde sú daný atribút, metóda alebo konštruktor definované. Viditeľnosť **public** sa neodporúča používať na atribúty.
Pomocou kľúčového slova **private** (súkromný) vyjadrujeme, že k danému atribútu, metóde alebo konštruktoru nemá prístupný nik, okrem triedy samotnej, v ktorej je atribút, metóda alebo konštruktor definovaný. Viditeľnosť **private** sa odporúča používať na atribúty.
Pomocou kľúčového slova **protected** (chránený) vyjadrujeme, že k danému atribútu, metóde alebo konštruktoru má prístup **trieda**, v ktorej je atribút, metóda alebo konštruktor definovaný a všetci jej potomkovia.

ÚLOHA 9.3

Upravte viditeľnosť atribútu **vlastnik** v predkovi **Vybusnina** na **protected**.

```
protected Hrac vlastnik;
```

9.2 Definovanie spoločnej činnosti tried Bomba a Mina

Analyzujme metódy **`act()`** triedy **`Bomba`** a triedy **`Mina`**. Slovne by sa dali popísať nasledovne:

Tabuľka 9.1: Analýza metódy act triedy Bomba a triedy Mina

Krok	Bomba	Mina
1	Zistí, či sa jej dotýka oheň alebo uplynul čas.	Zistí, či sa jej dotýka oheň alebo na ňu niekto stúpil po čase aktivácie.
2	Ak áno, tak:	Ak áno, tak:
a	Oznámi vlastníkovi, že vybuchuje bomba.	Oznámi vlastníkovi, že vybuchuje mína.
b	Vytvorí pod sebou oheň a rozšíri ohne v priamom smere od centra výbuchu.	Vytvorí pod sebou oheň.
c	Odstráni sa zo sveta.	Odstráni sa zo sveta.

Vidíme, že obe metódy **`act`** sú veľmi podobné. Navrhnime jednu metódu tak, aby obsahovala činnosť triedy **`Mina`** aj triedy **`Bomba`**. Činnosť takejto metódy môžeme považovať za správanie sa výbušniny.

Tabuľka 9.2: Návrh metódy act() triedy Vybusnina

Krok	Vybusnina
1	Zistí, či má vybuchnúť (napr. sa jej dotýka oheň).
2	Ak áno, tak:
a	Oznámi vlastníkovi, že vybuchuje výbušnina.
b	Vytvorí ohne.
c	Odstráni sa zo sveta.

To, čo odlišuje mínu od bomby sú iba tri riadky:

1) Zistenie, ako a kedy má vybuchnúť (riadok 1)
2) Oznámenie o výbuchu (riadok 2a)
3) Rozšírenie ohňa (riadok 2b)

Vidíme, že okrem týchto troch riadkov je možné správanie oboch tried zapísať jednotne už v predkovej metóde **`act()`**. Ako však zabezpečíme rozdielne správanie pri zisťovaní, či má výbušnina vybuchnúť, pri navýšení počtu výbušniny a pri samotnom výbuchu?

Pre pochopenie rôznych reakcií inštancií rôznych tried na tú istú metódu si vyskúšajme jednoduchý test.

ÚLOHA 9.4

Vytvorte bezparametrickú metódu **vypisKtoSi()** v triede **Vybusnina**, ktorá nemá návratovú hodnotu. Metóda vypíše na obrazovku text VYBUSNINA tam, kde sa aktuálne výbušnina nachádza. Vytvorte inštanciu triedy **Mina** a vyvolajte metódu **vypisKtoSi()**. Čo sa stane? Vytvorte inštanciu triedy **Bomba** a vyvolajte metódu **vypisKtoSi()**. Čo sa stane v tomto prípade?

Do triedy **Vybusnina** pridáme metódu:

```
public void vypisKtoSi() {
     World svet = this.getWorld();
     svet.showText("VYBUSNINA", this.getX(), this.getY());
}
```

Obrázok 9.1: Reakcie inštancií tried Mina a Bomba na metódu vypisKtoSi().

ÚLOHA 9.5

Vytvorte v triede **Mina** metódu **vypisKtoSi()** s rovnakou hlavičkou ako v triede **Vybusnina** (teda metóda bude mať rovnaký názov, rovnaké parametre a rovnaký typ návratovej hodnoty). Metóda vypíše na obrazovku text MINA. Opäť vytvorte inštanciu triedy **Mina** a triedy **Bomba**. Odhadnite, čo sa stane, keď vyvoláte metódu test v inštancii triedy **Mina** a v inštancii triedy **Bomba**. Potom naozaj vyvolajte metódy. Zhoduje sa predpoveď s výsledkom?

Do triedy **Mina** pridáme metódu:

```
public void vypisKtoSi() {
     World svet = this.getWorld();
     svet.showText("MINA", this.getX(), this.getY());
}
```

Obrázok 9.2: Reakcie inštancií tried Mina a Bomba na metódu vypisKtoSi().

ZAPAMÄTAJTE SI!

Ak predok aj potomok definuje metódu s rovnakou hlavičkou (teda s rovnakým názvom, typom návratovej hodnoty a s rovnakými parametrami), potom hovoríme, že potomok metódu **prekrýva**. Kedykoľvek bude vyvolaná metóda, ktorá je v inštancii prekrytá, bude vyvolaná prekrytá metóda inštancie. Metódy, ktoré sú prekrývané voláme **virtuálne metódy**. Reakcia na tú istú metódu rôznym spôsobom je vlastnosť objektov, ktorá sa nazýva **polymorfizmus**.

ÚLOHA 9.6

Prekryte metódu **`vypisKtoSi()`** v triede **`Bomba`** tak, aby vypísala na obrazovku text BOMBA. Overte funkčnosť svojho riešenia.

Do triedy **`Bomba`** pridáme metódu:

```
public void vypisKtoSi() {
     World svet = this.getWorld();
     svet.showText("BOMBA", this.getX(), this.getY());
}
```

Vráťme sa naspäť k metóde **`act()`** v triede **`Vybusnina`** a napíšme ju s využitím polymorfizmu. Z predchádzajúcej analýzy vyplýva, že sa potomkovia správajú rozdielne iba na troch miestach. Pre tieto situácie musíme vytvoriť v predkovi metódy, ktoré potomkovia následne prekryjú.

1) Pre zistenie, či má výbušnina vybuchnúť môžeme definovať novú virtuálnu metódu **`maVybuchnut()`**, ktorá vráti **`true`**, ak má vybuchnúť a **`false`** inak. Parametre pre túto metódu nie sú potrebné. Výbušninu môžeme považovať za stabilnú, teda nevybuchne, kým to nešpecifikuje potomok.
2) Pre oznámenie o výbuchu môžeme nahradiť pôvodne dve metódy **`vybuchlaBomba()`** a **`vybuchlaMina()`** jedinou metódou, ktorá ako parameter preberie výbušninu (mína aj bomba sú výbušninou – sú potomkami tejto triedy). Hráč na základe typu výbušniny zistí, o akú výbušninu sa jedná a podľa toho zareaguje (zvýši svoje počítadlo bômb alebo mín).
3) Pre vytvorenie ohňov v okolí výbušniny môžeme definovať bezparametrickú virtuálnu metódu **`vybuch()`**, ktorá nebude mať návratovú hodnotu.

Metódy **`maVybuchnut()`** a **`vybuch()`** potrebuje výbušnina na to, aby vedela napísať všeobecnú verziu správania sa výbušniny. Metódy však nebude volať iný objekt. Je preto vhodné, aby ich viditeľnosť bola **`protected`**. Metódy takto nebudú dostupné nikomu inému iba predkovi **`Vybusnina`** na ich vyvolanie a potomkom **`Bomba`** a **`Mina`** na úpravu ich správania. Metódu **`vybuchlaVybusnina()`** má hráč. Výbušnina vyvolá túto metódu vtedy, keď vybuchne. Na to, aby mohla výbušnina volať metódu hráča, musí byť táto metóda verejná (**`public`**).

ÚLOHA 9.7

Vytvorte metódy **maVybuchnut()** a **vybuch()** v triede **Vybusnina** a metódu **vybuchlaVybusnina()** v triede **Hrac** podľa popisu vyššie. Telá metód zatiaľ neimplementujte. Ak je potrebná návratová hodnota, vráťte **false**.

Do triedy **Vybusnina** pridáme dve metódy:

```
protected boolean maVybuchnut() {
     // predok je stabilný – nikdy nevybuchne
     return false;
}

protected void vybuch() {
}
```

Do triedy **Hrac** pridáme jednu metódu:

```
public void vybuchlaVybusnina(Vybusnina vybusnina) {
}
```

S využitím pripravených metód je možné napísať telo metódy **act()** v triede **Vybusnina**.

ÚLOHA 9.8

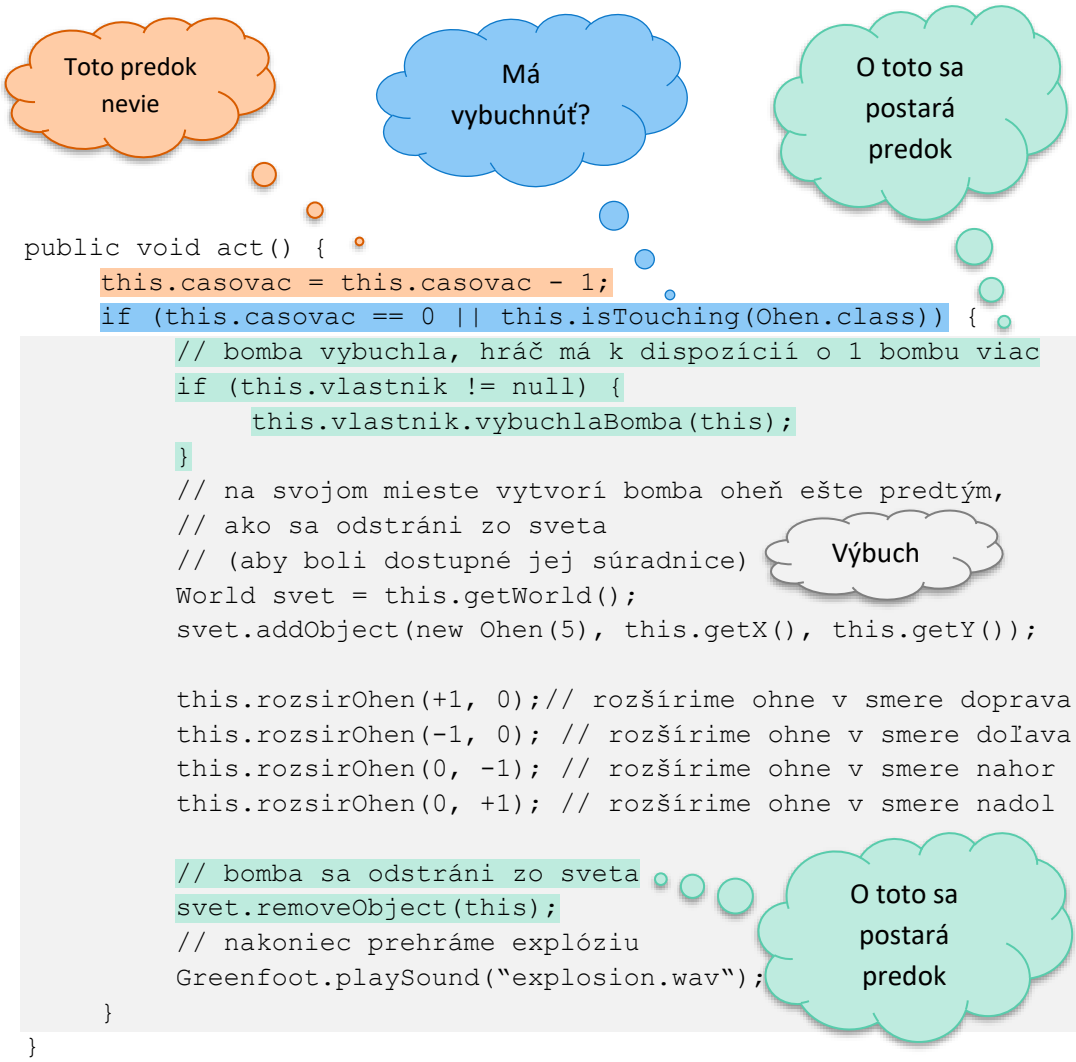
Napíšte telo metódy **act()** v triede **Vybusnina**.

```
public void act() {
     // zistíme, či má vybuchnúť (zistí to potomok)
     if (this.maVybuchnut()) {
           // výbušnina má vybuchnúť, tak:

           // oznámime vlastníkovi, že mu táto výbušnina vybuchla
           if (this.vlastnik != null) {
                 this.vlastnik.vybuchlaVybusnina(this);
           }

           // necháme potomka zareagovať (vytvoriť ohne)
           this.vybuch();

           // nakoniec sa výbušnina odstráni zo sveta
           World svet = this.getWorld();
           svet.removeObject(this);
     }
}
```

Práve definované telo metódy **act()** v triede **Vybusnina** (teda v predkovi) by sa však nikdy nevykonalo, keďže potomkovia túto metódu prekrývajú. Aktuálne zadané príkazy v metóde **act()** v triedach **Bomba** a **Mina** je teraz potrebné rozdeliť medzi dve metódy – **maVybuchnut()**

a **vybuch()**, ktoré prekryjú správanie predkových metód. V potomkovi stačí potom iba zabezpečiť, aby sa vyvolala metóda **act()** predka (napríklad tak, že danú metódu v potomkovi neprekryjeme). Predkova metóda **act()** vyvoláva dve spomenuté metódy, ktoré potomkovia prekrývajú, čím sa zabezpečí požadované správanie.

Začnime s úpravou metódy **act()** v triede **Bomba**.




```
public void act() {
     this.casovac = this.casovac - 1;
     if (this.casovac == 0 || this.isTouching(Ohen.class)) {
          // bomba vybuchla, hráč má k dispozícii o 1 bombu viac
          if (this.vlastnik != null) {
               this.vlastnik.vybuchlaBomba(this);
          }
          // na svojom mieste vytvorí bomba oheň ešte predtým,
          // ako sa odstráni zo sveta
          // (aby boli dostupné jej súradnice)
          World svet = this.getWorld();
          svet.addObject(new Ohen(5), this.getX(), this.getY());

          this.rozsirOhen(+1, 0);// rozšírime ohne v smere doprava
          this.rozsirOhen(-1, 0); // rozšírime ohne v smere doľava
          this.rozsirOhen(0, -1); // rozšírime ohne v smere nahor
          this.rozsirOhen(0, +1); // rozšírime ohne v smere nadol

          // bomba sa odstráni zo sveta
          svet.removeObject(this);
          // nakoniec prehráme explóziu
          Greenfoot.playSound("explosion.wav");
     }
}
```

ÚLOHA 9.9

Prekryte metódy **maVybuchnut()** a **vybuch()** v triede **Bomba**. Využite vhodný kód z jej metódy **act()**. Všimnite si, že je možné jednoducho napísať telá metód, nakoľko nie je potrebné uvažovať nad podmienkami (to spravil predok).

```
protected boolean maVybuchnut() {
     return this.casovac == 0 || this.isTouching(Ohen.class);
}

protected void vybuch() {
     World svet = this.getWorld();
     // na mieste bomby vytvoríme oheň
     svet.addObject(new Ohen(5), this.getX(), this.getY());

     this.rozsirOhen(+1 ,0); // rozšírime ohne v smere doprava
     this.rozsirOhen(-1 ,0); // rozšírime ohne v smere doľava
     this.rozsirOhen(0, -1); // rozšírime ohne v smere nahor
     this.rozsirOhen(0, +1); // rozšírime ohne v smere nadol

     // nakoniec prehráme zvuk explózie
     Greenfoot.playSound("explosion.wav");
}
```

Všimnime si, že trieda **Bomba** vo svojej metóde **act()** vykonávala aj zníženie hodnoty svojho časovača výbuchu a tiež prehrá zvuk explózie. Túto funkcionalitu ale predok nemá. Ak by sme v triede **Bomba** napísali metódu **act()**, v ktorej by sme iba znížili hodnotu časovača, tak by táto metóda prekryla predkovu metódu **act()** a nevolala by sa kontrola výbuchu, neposlalo by sa oznámenie hráčovi a ani by sa nevykonal výbuch samotný (toto všetko je umiestnené v metóde predka). Ak metódu neprekryjeme, tak sa časovač nebude znižovať. Dostali sme sa do situácie, v ktorej požadujeme aj funkčnosť definovanú predkom a aj rozšírenie funkčnosti metódy. Pre vyvolanie predkovej verzie metódy **act()** je potrebné zavolať **super.act();**. Toto (nepovinné) volanie je možné spraviť kedykoľvek v tele prekrytej metódy. Na tomto mieste bude vyvolaná metóda predka a potom bude prekrytá metóda pokračovať vo svojej činnosti.

ZAPAMÄTAJTE SI!

Každá prekrytá metóda môže kedykoľvek vyvolať predkovu metódu. Stačí, aby použila kľúčové slovo **super** namiesto **this** a ako názov metódy uviedla svoj názov (prekryté metódy majú rovnaké názvy). Všimnite si podobnosť s volaním predkovho konštruktora. Kľúčové slovo **super** sprístupňuje položku (atribút, metódu, konštruktor) predka.

Metóda **act()** v triede **Bomba** by teda mohla vyzerať takto:

```
public void act() {
     this.casovac = this.casovac - 1;
     super.act();
}
```

ÚLOHA 9.10

Definujte metódy **maVybuchnut()** a **vybuch()** v triede **Mina**. Využite vhodný kód z jej metódy **act()**. Prečo je nakoniec potrebné odstrániť metódu **act()**?

```
protected boolean maVybuchnut() {
     // mína má určite vybuchnúť, ak sa jej dotýka oheň
     if (this.isTouching(Ohen.class)) {
          return true;
     }

     if (this.casovac > 0) {
          // ak ešte nevypršal časovač, tak mína nie je aktívna
          // iba znížime časovač
          this.casovac = this.casovac - 1;
          // mína nemôže vybuchnúť
          return false;
     }
     else {
          // časovač vypršal, mína je aktívna.
          // vybuchne, ak sa jej dotýka hráč alebo oheň
          return this.isTouching(Hrac.class);
     }
}

protected void vybuch() {
     World svet = this.getWorld();
     // na svojom mieste vytvorí bomba oheň
     svet.addObject(new Ohen(5), this.getX(), this.getY());
}
```

Metódu **`act()`** musíme odstrániť, aby nebola v potomkovi prekrytá. Ak by tam ostala s prázdnym telom, vyvolala by sa táto (nič nerobiaca) verzia metódy.

Za pozornosť stojí kontrola bomby aj míny v metóde **`maVybuchnut()`**. Obe triedy najskôr kontrolujú, či sa nedotýkajú ohňa. Jedná sa o spoločnú funkčnosť, ktorú je vhodné mať na jedinom mieste. Spomeňme si, že predok definuje metódu **`maVybuchnut()`** ako virtuálnu a vždy vráti **`false`**. Túto metódu preto môžeme upraviť tak, že budeme predpokladať, že akákoľvek výbušnina vybuchne po dotyku s ohňom. Potomkovia potom namiesto vlastného testu, či sa nedotýkajú ohňa môžu vyvolať predkovu verziu metódy, ktorá im na to odpovie.

ÚLOHA 9.11

Upravte telo metódy **`maVybuchnut()`** v triede **`Vybusnina`** tak, aby metóda vrátila **`true`**, ak sa dotýka inštancie triedy **`Ohen`**. Upravte prekryté metódy **`maVybuchnut()`** v triede **`Bomba`** a triede **`Mina`** tak, aby využívali funkčnosť predkovej metódy.

Predkova verzia metódy **`maVybuchnut()`** bude vyzerať takto:

```
protected boolean maVybuchnut() {
     // predok vybuchne, keď sa ho dotkne oheň
     return this.isTouching(Ohen.class);
}
```

Potomok **Bomba** potom upraví telo v prekrytej metóde na:

```
return this.casovac == 0 || super.maVybuchnut();
```

Potomok **Mina** upraví prvú podmienku v prekrytej metóde na:

```
if (super.maVybuchnut()) {
      return true;
}
```

9.3 Reakcia hráča na výbuch výbušniny

Ostáva ešte vyriešiť reakciu hráča na výbuch výbušniny. Pri pridaní míny do hry sme postupovali veľmi podobne ako v prípade bomby:

1) Vytvorili sme zoznam a počítadlo mín (bômb).
2) Pri položení míny (bomby) sme pridali novo vytvorenú mínu (bombu) do príslušného zoznamu a znížili počítadlo mín (bômb).
3) Pri výbuchu míny (bomby) sme odstránili vybuchnutú mínu (bombu) zo zoznamu a zvýšili počítadlo mín (bômb).

Vidíme, že duplikovaný kód by sa veľmi zjednodušil, ak by sme sa správali k míne aj k bombe jednotne. Stačil by nám jediný zoznam (zoznam výbušnín), do ktorého by sme vzniknuté výbušniny vkladali, a z ktorého by sme ich po výbuchu vyberali.

Do triedy reprezentujúcej hráča sme navyše počas návrhu pridali metódu **vybuchlaVybusnina()**, ktorá má parameter typu **Vybusnina**. Túto metódu vyvoláva výbušnina vo svojej metóde **act()**. Spomeňme si na dedičnosť a Liskovej substitučný princíp. **Bomba** aj **Mina** sú potomkami triedy **Vybusnina**, teda sa k nim môžeme správať jednotne – ako k výbušninám. Nemusíme teda vytvárať zoznamy osobitne pre aktívne bomby a osobitne pre aktívne míny – stačí nám jediný zoznam aktívnych výbušnín.

ÚLOHA 9.12

Odstráňte z triedy **Hrac** atribúty **zoznamAktivnychBomb** a **zoznamAktivnychMin**. Pridajte do triedy **Hrac** jediný atribút **zoznamAktivnychVybusnin** typu **LinkedList<Vybusnina>**. Inicializujte ho v konštruktore a odstráňte z konštruktora inicializáciu pôvodných atribútov.

Deklarácia atribútu:

```
private LinkedList<Vybusnina> zoznamAktivnychVybusnin;
```

Inicializácia vo vhodnom konštruktore:

```
this.zoznamAktivnychVybusnin = new LinkedList<Vybusnina>();
```

Po odstránení pôvodných atribútov trieda hlási chyby na troch miestach:

1) Pridanie bomby alebo míny do zoznamu pri jej vytvorení v metóde **act()**.
2) Cykly **foreach**, ktoré rušia vlastníka bombe alebo míne v metóde **zasah()**.
3) Odstránenie bomby zo zoznamu v metóde **vybuchlaBomba()** a odstránenie míny zo zoznamu v metóde **vybuchlaMina()**.

Začnime postupne opravovať náš kód. Vieme, že k bombe aj k míne sa môžeme správať rovnako – ako k výbušnine. Preto môžeme vytvorenú inštanciu typu **Bomba** alebo typu **Mina** vložiť priamo do zoznamu **zoznamAktivnychVybusnin**. Dotknuté riadky v metóde **act()** teda môžeme upraviť nasledovne:

```
// premenná bomba je typu Bomba a obsahuje vytvorenú bombu
this.zoznamAktivnychVybusnin.add(bomba);

// premenná mina je typu Mina a obsahuje vytvorenú minu
this.zoznamAktivnychVybusnin.add(mina);
```

Opravme teraz cyklus **foreach** v metóde **zasah()**. Ten môžeme vďaka Liskovej substitučnému princípu upraviť jednoducho tak, aby bola riadiaca premenná typu **Vybusnina** a cyklus prechádzal cez zoznam **zoznamAktivnychVybusnin**. Všimnime si, že si vystačíme s jediným cyklom.

```
for (Vybusnina vybusnina : this.zoznamAktivnychVybusnin) {
     vybusnina.zrusVlastnika();
}
```

Nakoniec musíme upraviť metódy vyvolávané po zásahu. Tu je potrebné uvedomiť si, že metódy **vybuchlaBomba()** a ani **vybuchlaMina()** už nikto nevolá. Pôvodne boli tieto metódy volané v triedach **Bomba** a **Mina**. My sme však zaviedli spoločné správanie v predkovi **Vybusnina** a ten volá iba metódu **vybuchlaVybusnina()**. **Bomba** aj **Mina** sú výbušniny, my ich však v metóde **vybuchlaVybusnina()** musíme rozlíšiť kvôli tomu, aby sme vedeli, ktoré počítadlo dostupných výbušnín navýšiť. Na to môžeme využiť operátor **instanceof**.

ÚLOHA 9.13

Implementujte telo metódy **vybuchlaVybusnina()**. Pomocou operátora **instanceof** zistite, či je výbušnina **Bomba** alebo je výbušnina **Mina**. Na základe jej skutočného typu zvýšte počítadlo dostupných bômb alebo počítadlo dostupných mín. Nezabudnite výbušninu odstrániť zo zoznamu aktívnych výbušnín. Nakoniec odstráňte nepotrebne metódy **vybuchlaBomba()** a **vybuchlaMina()**.

```
public void vybuchlaVybusnina(Vybusnina vybusnina) {
     if (vybusnina instanceof Bomba) {
           this.pocetBomb = this.pocetBomb + 1;
     }
     else if (vybusnina instanceof Mina) {
           this.pocetMin = this.pocetMin + 1;
     }
     this.zoznamAktivnychVybusnin.remove(vybusnina);
}
```

ZHRNUTIE

Pri objektovo orientovanom programovaní sa často stáva, že triedy majú spoločnú funkčnosť, ktorá sa líši iba v niekoľkých detailoch. V takomto prípade je možné spoločnú funkčnosť zastrešiť v spoločnom predkovi a rozdiely nechať potomkov špecifikovať pomocou virtuálnych metód. Virtuálna metóda je metóda, ktorá má rovnakú hlavičku, ako má daná metóda v predkovi. Ak potomok takúto metódu definuje, potom hovoríme, že potomok prekryl predkovu metódu a takúto metódu voláme virtuálna metóda. Pri vyvolaní virtuálnej metódy bude vždy vyvolaná tá prekrytá metóda, ktorá patrí skutočnej inštancií (**`Vybusnina`** síce vyvolala metódu **`this.maVybuchnut`**, ale výbušninou (**`this`**) bola v skutočnosti **`Mina`** alebo **`Bomba`** – zavolala sa teda prekrytá metóda na základe triedy inštancie). Takejto vlastnosti objektov hovoríme polymorfizmus. Z prekrytej metódy je možné kedykoľvek zavolať predkovu verziu prekrytej metódy s využitím kľúčového slova **`super`**.

Jazyk Java poskytuje viditeľnosť **`protected`**. Atribúty, metódy a konštruktory označené ako **`protected`** sú dostupné v triede, v ktorej sú definované a vo všetkých jej potomkoch. Objekty, ktoré nepatria do hierarchie triedy, v ktorej je prvok definovaný k nemu nemajú prístup.

ÚLOHY NA PRECVIČOVANIE

ÚLOHA 9.A ČASOVANÉ VÝBUŠNINY

V súčasnosti má bomba atribút reprezentujúci časovač výbuchu. Upravte hierarchiu tried **`Vybusnina`** tak, aby ste vytvorili potomka **`CasovanaVybusnina`**, ktorý bude automaticky odpočítavať čas, po ktorom vybuchne.

ÚLOHA 9.B

Porovnajte metódy **`vybuch()`** v triedach **`Bomba`** a **`Mina`**. Identifikujte rovnaký príkaz. Upravte metódu **`vybuch()`** predka **`Vybusnina`** tak, aby túto spoločnú funkčnosť poskytoval on. Potom upravte metódy potomkov. Ktorá metóda sa stane zbytočná?

ÚLOHA 9.C

Pridajte možnosť, aby hráči mohli klásť dynamit. Všetky inštancie triedy **`Dynamit`** vybuchnú naraz – keď to určí hráč stlačením klávesu. Kláves, ktorým sa ovláda výbuch dynamitov, je parametrom konštruktora triedy **`Hrac`**. Ak hráč zomrie, všetky jeho dynamity okamžite vybuchnú.

10 NÁHODNÉ ČÍSLA

KĽÚČOVÉ SLOVÁ

Trieda Random.

CIELE

V tejto časti sa naučíme pracovať s náhodou. Zoznámime sa s triedou `Random`. S pomocou jej inštancií budeme generovať náhodné čísla. Ukážeme si aj spôsob generovania náhodných čísel bez využitia triedy `Random`, priamo s využitím nástroja Greenfoot. Náhodné čísla využijeme pri náhodnom rozložení sveta a ďalej pridáme do sveta bonusy – špeciálne prvky, ktoré vzniknú po výbuchu múru, a ktoré zlepšujú vybrané vlastnosti hráča.

OBSAH

10.1 Náhodné rozloženie arény

Doteraz mali arény, ktoré sme generovali pevne dané rozloženie. Pre rozdielny zážitok z každej hry je ale dobré, aby sa menili aj arény. Steny typicky ostávajú na svojich miestach a bývajú pravidelne rozdelené, mení sa rozloženie múrov, ktoré býva náhodné.

ÚLOHA 10.1

Zamyslite sa, čo je to náhoda, ako vieme získať nejaký náhodný výsledok pokusu, aké náhodné javy pozorujeme vo svete okolo nás.

ZAPAMÄTAJTE SI!

Objekty, ktoré poskytujú **náhodné (zvyčajne číselné) výsledky** (napríklad minca, kocka) budeme označovať za **generátory náhodných čísel**.

ÚLOHA 10.2

Vezmime si klasickú hraciu kocku so šiestimi stranami. Vedeli by sme pomocou nej vygenerovať náhodnú pozíciu na šachovnici s rozmermi 6x6 políčok? A čo šachovnica s rozmermi 3x3 políčka? Ako by sa zmenil spôsob generovania, ak by sme použili mincu? Navrhnite takéto algoritmy generovania polohy.

Na šachovnici 6x6 je riešenie jednoduché – jednoducho hodíme dva krát hracou kockou a získame súradnice.

Na šachovnici 3x3 si tiež vieme ľahko pomôcť – ak hodíme 1 a 4, znamená to súradnicu 1, ak 2 a 5, znamená to 2, ak 3 a 6, znamená to 3. Kockou hodíme dva krát a súradnice získame horeuvedeným spôsobom.

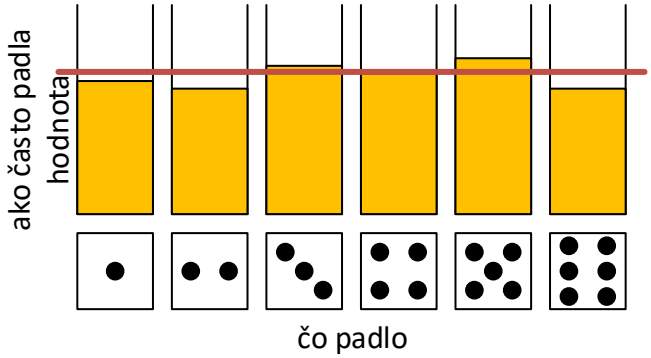
ÚLOHA 10.3

Pomocou algoritmu z predošlej úlohy a s pomocou kocky generujte náhodné pozície na šachovnici. Svoje výsledky si do šachovnice zaznačte. Dá sa pozorovať nejaká pravidelnosť výsledkov?

Mali by sme pozorovať rovnomerné zapĺňanie šachovnice, teda značky v šachovnici by nemali byť sústredené na jedinom mieste.

Základnou vlastnosťou náhodného pokusu je, že nevieme dopredu predpovedať, ako každý jednotlivý pokus dopadne. Čo ale niekedy dokážeme, je odhadnúť súhrnné vlastnosti mnohých pokusov. Ak veľakrát hodíme hracou kockou, tak všetky možnosti budú padať rovnako často. Nedokážeme vopred predpovedať výsledok nasledujúceho náhodného pokusu, avšak dokážeme odhadnúť, ako často by mal daný jav nastať pri dostatočne veľkom počte opakovaní pokusu za rovnakých podmienok. V prípade kocky má každý výsledok rovnakú šancu výskytu.

Ak má každý výsledok náhodného pokusu rovnakú pravdepodobnosť výskytu, tak hovoríme o **rovnomernom rozdelení pravdepodobnosti**.



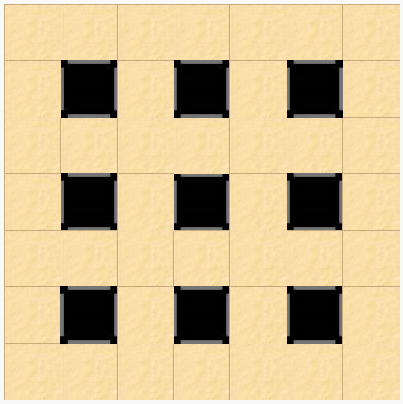
Obrázok 10.1: Rozdelenie výsledkov hodu kockou.

Problém nastáva, ak chceme aby náhodu, resp. náhodné čísla generoval počítač. Už vieme, že počítač vykonáva algoritmus, ktorý obsahuje presnú postupnosť krokov. Ak je postupnosť pevne daná, tak nie je možné hovoriť o náhode. Počítače však napriek tomu dokážu generovať čísla, ktoré sa nám javia ako náhodné. Existujú matematické vzorce, ktoré počítač používa na generovanie čísel. Tieto vzorce produkujú pre nás zdanlivo náhodné čísla (tieto označujeme ako pseudonáhodné). Ak by sme teda s takto vygenerovanými číslami z počítača urobili podobný test, ako s kockami vyššie, zistili by sme, že jednotlivé čísla sa vyskytujú s rovnakou pravdepodobnosťou a ich výskyt nie je možné jednoducho predpokladať – a to je pre nás postačujúce.

Všetko v programovacom jazyku Java je objekt, preto aj generátor náhodných čísel je objekt. Inštancie generátorov poskytuje trieda **`Random`**, ktorá sa nachádza v balíčku **`java.utils.Random`**. Inštanciu triedy si môžeme predstaviť ako hraciu kocku, ktorou dokážeme hodiť, a ktorá odpovie náhodným číslom. Trieda obsahuje metódu **`nextInt(int n)`** pomocou ktorej „hádžeme kockou“. Inštancia triedy vygeneruje náhodné číslo. Najmenšie vygenerované číslo je 0, najväčšie vygenerované číslo je n – 1 (teda vždy o 1 menšie, ako hodnota n). Aké číslo to bude, dopredu nevieme, ale všetky čísla medzi 0 a n-1 majú rovnakú šancu, že sa vyskytnú (teda každé celé číslo z daného intervalu bude vygenerované s pravdepodobnosťou $\frac{1}{n}$).

ÚLOHA 10.4

Pripravte si arénu. Vytvorte potomka triedy **`Arena`**, ktorý nazvete napr. **`NahodnaArena`**. V konštruktore nastavte vhodnú veľkosť sveta. Odporúčame pravidelné rozloženie stien s jedným voľným poľom medzi stenami tak, ako je ukázané na nasledujúcom obrázku:



```
public class NahodnaArena extends Arena {
     super(7, 7);

     this.vytvorObdlznikStien(1, 1, 3, 3, 1, 1);
}
```

Poďme teraz v našej aréne vytvoriť bezparametrickú metódu **`vytvorNahodnyMur()`**, ktorá nemá návratovú hodnotu a ktorá v aréne vytvorí múr na náhodných súradniciach.

```
public void vytvorNahodnyMur () {
}
```

Na vytvorenie múru budeme potrebovať „kocku“ (náhodný generátor), ktorá bude generovať polohu. Algoritmus metódy by mohol vyzerať takto:

1) Hoď dvakrát kockou a získaj tak súradnicu riadku a stĺpca.
2) Zisti, či je dané pole voľné.
3) Ak je voľné, vytvor tam múr, inak vygeneruj nové súradnice.

Všimnime si, že kocku budeme potrebovať vždy pri generovaní. Zdalo by sa, že môžeme zakaždým vytvárať novú inštanciu triedy **`Random`**, avšak takýto prístup nie je správny a vedie k zlým výsledkom. Spomeňte si, že trieda **`Random`** generuje čísla podľa istého matematického

vzorca. Tento vzorec dokáže zaručiť, že pri veľkom počte vygenerovaných čísiel v danej inštancii generátora, budú tieto čísla mať požadované vlastnosti. Dve rôzne inštancie triedy **Random** však nezdieľajú žiadne informácie a vzorec preto nedokáže zaručiť dodržanie vlastností generovaných čísiel, ktoré by pochádzali z rôznych inštancií. Ak by sme teda v metóde zakaždým vytvorili novú kocku, generovanie by prestalo vykazovať potrebné matematické vlastnosti. Ani v skutočnosti by ste predsa po jednom použití hraciu kocku nezahodili a nešli si zaobstarať novú. Vytvorme si teda kocku iba raz – generátor sa stane atribútom triedy **NahodnaArena**.

ÚLOHA 10.5

Pridajte do triedy **NahodnaArena** referenčný atribút triedy **Random**. Nezabudnite, že trieda **Random** sa nachádza v balíčku **java.util.Random**. Kocku inicializujte v konštruktore.

Import:

```
import java.util.Random;
```

Deklarácia atribútu:

```
private Random kocka;
```

Inicializácia atribútu v konštruktore:

```
this.kocka = new Random();
```

Poďme implementovať našu metódu **vytvorNahodnyMur()**. Najskôr musíme vygenerovať náhodné súradnice. To bude robiť **kocka**. Ako parameter svojej metódy požaduje celé číslo, po ktoré má generovať. Zamyslime sa, ako vygenerujeme náhodný index stĺpca (riadok vygenerujeme obdobne). Vieme, že prvý stĺpec (aj riadok) má index 0. Ak máme v aréne 7 stĺpcov, potom sú platné indexy 0, 1, 2, 3, 4, 5 a 6. Šírka takejto arény je 7 (stĺpcov). My chceme vygenerovať náhodný index stĺpca – ak teda ako parameter metódy **nextInt** objektu kocka použijeme šírku arény, potom určite vygeneruje platný index. Pripomeňme si ešte raz, že všetky čísla od 0 po šírku arény – 1 budú mať rovnakú šancu na vygenerovanie – a to je presne to, čo požadujeme.

```
int nahodnyStlpec = this.kocka.nextInt(this.getWidth());
int nahodnyRiadok = this.kocka.nextInt(this.getHeight());
```

Pokračujme v metóde ďalej. Musíme zistiť, či je pole voľné. Ak áno, tak vytvoríme múr a vložíme ho na vygenerované súradnice. Na zistenie voľnosti môžeme vytvoriť súkromnú metódu **jeBunkaVolna()**, ktorá bude v parametroch preberať súradnice bunky.

ÚLOHA 10.6

Pridajte do triedy **NahodnaArena** súkromnú metódu **jeBunkaVolna()**, ktorá preberie dva parametre – stĺpec a riadok. Metóda vráti **true**, ak je bunka vo svete voľná (neobsahuje žiadneho aktora), inak vráti **false**.

Doplníme import pre zoznam:

```
import java.util.List;
```

Implementujeme metódu:

```
private boolean jeBunkaVolna(int stlpec, int riadok) {
     List<Actor> zoznam = this.getObjectsAt(stlpec, riadok,
                                            Actor.class);
     return zoznam.isEmpty();
}
```

S využitím metódy **`jeBunkaVolna()`** by teda generovanie múru po získaní náhodných súradníc mohlo vyzerať takto:

```
if (this.jeBunkaVolna(nahodnyStlpec, nahodnyRiadok)) {
     this.addObject(new Mur(), nahodnyStlpec, nahodnyRiadok);
}
```

Čo však v prípade, ak voľno na danej pozícii nie je? Musíme vygenerovať nové súradnice. Čo sa stane, ak znovu nebude voľno? Musíme opäť vygenerovať nové súradnice, a tak ďalej. Takýto prístup by viedol ku nekonečnej kaskáde podmienok, ktorú môžeme v jednoduchosti zapísať takto:

```
Vygeneruj súradnice;
Ak (je voľno) {
     Vytvor stenu;
} inak {
     Vygeneruj súradnice;
     Ak (je voľno) {
          Vytvor stenu;
     } inak {
          Vygeneruj súradnice;
          Ak (je voľno) {
               Vytvor stenu;
          } inak {
               Vygeneruj súradnice;
               Ak (je voľno)...
     }
}
```

Vidíme, že sa opakuje kód, ktorý generuje náhodné súradnice. Opakuje sa dovtedy, pokiaľ nevygeneruje náhodné súradnice, ktoré sú voľné. Až potom je možné vytvoriť a umiestniť múr. Preto môžeme kód upraviť s využitím cyklu **`while`** takto:

```
while (!this.jeBunkaVolna(nahodnyStlpec, nahodnyRiadok)) {
     // Vygenerujeme nové súradnice
     nahodnyStlpec = this.kocka.nextInt(this.getWidth());
     nahodnyRiadok = this.kocka.nextInt(this.getHeight());
}
```

Celá metóda by teda mohla vyzerať takto:

```
public void vytvorNahodnyMur() {
     // Vygenerujeme prvé náhodné súradnice
     int nahodnyStlpec = this.kocka.nextInt(this.getWidth());
     int nahodnyRiadok = this.kocka.nextInt(this.getHeight());

     // Skontrolujeme, či sú súradnice voľné.
     // Ak nie, musíme generovať nové.
     while (!this.jeBunkaVolna (nahodnyStlpec, nahodnyRiadok)) {
          // Vygenerujeme súradnice nanovo
          nahodnyStlpec = this.kocka.nextInt(this.getWidth());
          nahodnyRiadok = this.kocka.nextInt(this.getHeight());
     }

     // Náhodné súradnice sú voľné, vytvoríme a vložíme múr
     this.addObject(new Mur(), nahodnyStlpec, nahodnyRiadok);
}
```

ÚLOHA 10.7

Upravte konštruktor triedy **`NahodnaArena`** tak, aby náhodne vygeneroval múry do tretiny všetkých buniek v aréne.

```
public NahodnaArena() {
     super(7, 7);

     this.kocka = new Random();

     this.vytvorObdlznikStien(1, 1, 3, 3, 1, 1);

     int pocetMurov = this.getWidth() * this.getHeight() / 3;
     for (int i = 0; i < pocetMurov; i = i + 1) {
          this.vytvorNahodnyMur();
     }
}
```

Generovanie múrov na náhodných pozíciách nám funguje. Rozloženie akýchkoľvek prvkov, teda nielen múrov, na náhodné súradnice však môže byť užitočná funkčnosť, ktorú by mohli zdieľať všetky arény. Upravme teda metódu **`vytvorNahodnyMur()`** a vytvorme všeobecnejšiu metódu, ktorá bude schopná na náhodné súradnice vložiť akýkoľvek objekt.

ÚLOHA 10.8

Presuňte metódy **`vytvorNahodnyMur()`**, **`jeBunkaVolna()`** a atribút (vrátane jeho inicializácie v konštruktore) **`kocka`** do predka **`Arena`**. Nezabudnite presunúť aj riadky **`import`**.

Stačí jednoducho kódy presunúť.

Po presunutí metódy **vytvorNahodnyMur()** môžeme začať s jej úpravami. Metóda sa skladá z dvoch častí:

- Vygenerovanie náhodnej pozície.
- Vytvorenie múru na danej pozícii.

Ak chceme vytvoriť všeobecnú metódu, táto nemusí vkladať na náhodné pozície do sveta iba inštancie triedy **Mur**, ale v princípe čokoľvek. Pripomeňme si, že čokoľvek, čo by sme chceli vložiť do sveta, musí byť aktor (potomok triedy **Actor**). Ak teda upravíme metódu tak, aby bola schopná vložiť akéhokoľvek aktora na náhodné súradnice, získame rozsiahlejšie možnosti využitia tejto funkcie.

ÚLOHA 10.9

Pridajte do metódy **vytvorNahodnyMur()** parameter typu **Actor** – toto bude aktor, ktorého budeme vkladať na náhodné súradnice. Upravte názov metódy (napr. **vlozNaNahodneSuradnice()**) a upravte volanie metódy z triedy **NahodnaArena**.

Úpravou pôvodnej metódy **vytvorNahodnyMur()** dostaneme nasledujúcu metódu:

```
public void vlozNaNahodneSuradnice(Actor vkladanyAktor) {
     ***
     pôvodné telo metódy, kde sme získali hodnoty
     premenných náhodnySlpec a nahodnyRiadok.
     ***

     // Náhodné súradnice sú voľné, tak vložíme aktora
     this.addObject(vkladanyAktor, nahodnyStlpec, nahodnyRiadok);
}
```

Úprava volania z konštruktora triedy **NahodnaArena**:

```
this.vlozNaNahodneSuradnice(new Mur());
```

Poslednou úpravou, ktorú by bolo vhodné v našej metóde urobiť je myslieť aj na aktora, ktorý sa už vo svete nachádza. Niektoré arény môžu obsahovať schopnosti, ktoré presunú aktora z jedného miesta na iné miesto. Ak sa aktor vo svete už nachádza, potom metóda **addObject()** triedy **World** nespraví nič. Pred vložením aktora do sveta ho preto najskôr zo sveta odstránime. Toto spôsobí, že pomocou metódy **vlozNaNahodneSuradnice()** bude možné premiestňovať akéhokoľvek aktora na náhodné voľné súradnice, nezávisle od toho, či sa aktor vo svete už nachádza alebo nie.

```
public void vlozNaNahodneSuradnice(Actor vkladanyAktor) {
     ***
     pôvodné telo metódy, kde sme získali hodnoty
     premenných náhodnySlpec a nahodnyRiadok.
     ***

     // Náhodné súradnice už sú voľné, tak odstránim aktora zo sveta
     // a potom ho vložím na nové súradnice.
     this.removeObject(vkladanyAktor);
     this.addObject(vkladanyAktor, nahodnyStlpec, nahodnyRiadok);
}
```

10.2 Bonusy

Bonusy tvoria neodmysliteľnú súčasť hry Bomberman. Pomocou bonusov dokážu hráči vylepšiť svoje schopnosti (napr. dosah bomby, počet bômb, rýchlosť). Bonusy sa odkrývajú po tom, ako vybuchne múr, v ktorom sa ukrývali. Základné bonusy, ktoré použijeme v našej hre sú tieto:

Tabuľka 10.1: Bonusy použité v hre Bomberman

Obrázok	Názov bonusu	Dôsledok
	Bonus oheň	Všetky bomby hráča budú mať odteraz zvýšenú silu o jedna.
	Bonus bomba	Hráč má odteraz k dispozícii o jednu bombu viac.

Bonus oheň aj bonus bomba sú bonusmi, preto je výhodné vytvoriť im spoločného predka **`Bonus`**, ktorý zabezpečí, aby sme sa ku všetkým bonusom vedeli v budúcnosti správať rovnako.

ÚLOHA 10.10

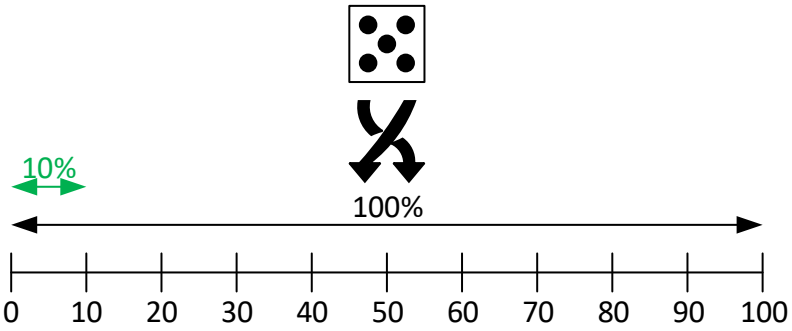
Vytvorte triedu **`Bonus`** ako potomka triedy **`Actor`**. Vytvorte dvoch potomkov triedy **`Bonus`** – triedu **`BonusOhen`** a triedu **`BonusBomba`**. Nastavte triedam vhodné obrázky.

Bonusy vznikajú na mieste zničeného múru. Múr sa zničí vo svojej metóde **`act()`** po tom, ako sa dotkne ohňa. V tejto metóde teda môže múr na svojom mieste vytvoriť inštanciu triedy **`Bonus`**.

Keby sa v každom múre skrývali bonusy, hráči by sa rýchlo stali príliš silnými. Na generovanie bonusu preto opäť využijeme náhodu a bonusy budeme generovať s rôznou pravdepodobnosťou. Pre jednoduchosť môžeme stanoviť, že sa všetky bonusy generujú s rovnakou pravdepodobnosťou 10 %. Ako takúto pravdepodobnosť dosiahneme v kóde?

Predstavme si, že generujeme čísla s hornou hranicou 100 (teda generujeme čísla od 0 do 99). S pravdepodobnosťou 100 % teda vygenerujeme číslo z tohto intervalu (inak povedané, určite vygenerujeme číslo z intervalu $\langle 0, 100)$). Vieme, že rozdelenie pravdepodobnosti je rovnomerné, teda každé číslo samostatne má rovnakú šancu na vygenerovanie rovnú $\frac{1}{100}$, teda

1 %. Akýchkoľvek 10 rôznych čísiel má spolu teda 10 % šancu ($\frac{10}{100}$), že sa vyskytne jedno z nich. Tento fakt môžeme jednoducho využiť na získanie pravdepodobnosti 10 %. Pre jednoduchosť je vhodné uvažovať 10 po sebe nasledujúcich čísiel, napríklad interval $\langle 0, 10)$. Potom platí, že ak vygenerujeme číslo z intervalu $\langle 0, 100)$, máme práve 10 % šancu, že to bude číslo z intervalu $\langle 0, 10)$. Ilustrácia tejto myšlienky je na nasledujúcom obrázku.



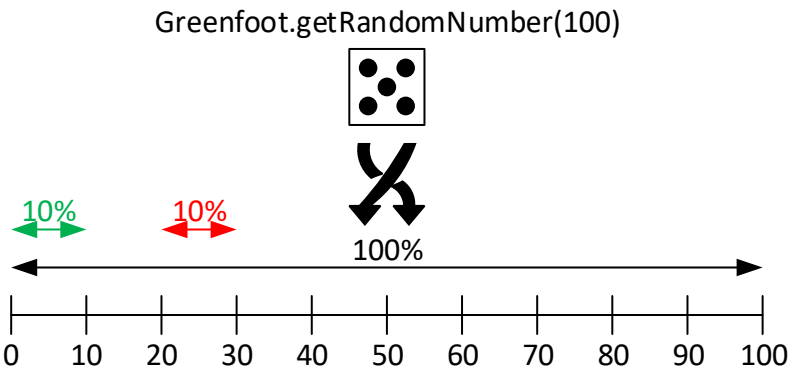
Obrázok 10.2: Pravdepodobnosť výskytu náhodnej udalosti zobrazená na číselnej osi.

Vieme, že na generovanie náhodných čísel budeme potrebovať kocku (generátor, inštanciu triedy **`Random`**), ktorá bude generovať náhodné čísla. Už sme sa naučili, že vytvárať kocku iba kvôli jednému hodu nie je vhodné riešenie. Nástroj Greenfoot ponúka už vytvorenú jednu kocku, ktorá je dostupná pre všetkých. Greenfoot obsahuje metódu **`getRandomNumber()`**, ktorá má jeden celočíselný parameter a funguje rovnako, ako metóda **`nextInt()`** triedy **`Random`**. Pre účely generovania bonusov budeme používať tento generátor náhodných čísel.

Ako teda zistíme, čí nastala udalosť s pravdepodobnosťou 10%?

```
int cislo = Greenfoot.getRandomNumber(100);
// padlo číslo od 0 do 9?
if (cislo < 10) {
     // toto nastane s pravdepodobnosťou 10 %
}
```

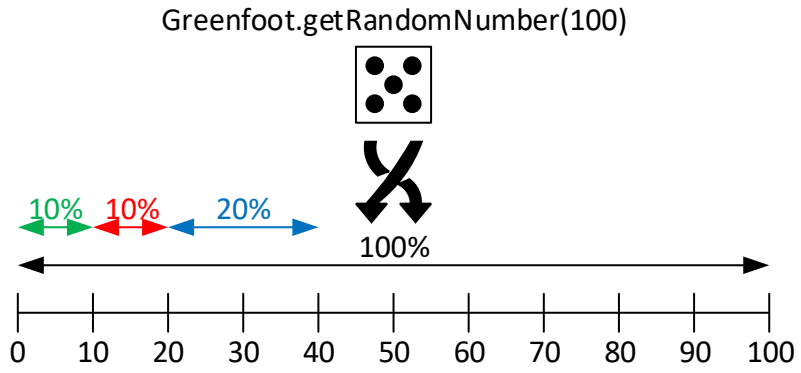
Predstavme si, že chceme generovať dve udalosti, pričom každá má pravdepodobnosť výskytu rovnú 10 % a zároveň nemôžu nastať súčasne. Môžeme využiť úplne rovnaký princíp, ako v predchádzajúcom prípade, pričom si môžeme vybrať, ktoré čísla budeme vyberať pre pravdepodobnosť 10 %.



Obrázok 10.3: Pravdepodobnosť výskytu dvoch náhodných udalostí zobrazená na číselnej osi.

```
// vygenerujeme číslo od 0 - 99
int cislo = Greenfoot.getRandomNumber(100);
// padlo číslo od 0 do 9?
if (cislo < 10) {
     // zelená udalosť nastala s pravdepodobnosťou 10 %.
// ak nepadlo číslo od 0 do 9, mohlo padnúť číslo od 20 do 29
} else if (cislo >= 20 && cislo < 30) {
     // červená udalosť nastala s pravdepodobnosťou 10 %.
}
```

Je však dobrou praxou „umiestňovať javy na číselnú os“ za seba. Tým budú podmienky jednoduchšie, pre tri udalosti s pravdepodobnosťou výskytu 10 %, 10 % a 20 % by sme mohli použiť nasledujúci kód:



Obrázok 10.4: Pravdepodobnosti výskytu rôznych náhodných udalostí zobrazené na číselnej osi.

```
// vygenerujeme číslo od 0 - 99
int cislo = Greenfoot.getRandomNumber(100);
// padlo číslo od 0 do 9?
if (cislo < 10) {
  // zelená udalosť nastala s pravdepodobnosťou 10 %.
// určite nepadlo číslo od 0 do 9, skontrolujme ďalších 10 hodnôt do 19
} else if (cislo < 20) {
     // červená udalosť nastala s pravdepodobnosťou 10 %
     // (čísla 10 až 19).
  // určite nepadlo číslo do 19, skontrolujme ďalších 20 hodnôt do 39
  } else if (cislo < 40) {
       // modrá udalosť nastala s pravdepodobnosťou 20 %
}
```

ÚLOHA 10.11

Upravte kód v metóde **`act()`** triedy **`Mur`**. S pravdepodobnosťou 10 % vygenerujte na jej mieste po zničení bonus oheň, s pravdepodobnosťou 10 % vygenerujte na jej mieste bonus bomba. V 80 % prípadoch sa po zničení nevygeneruje nič.

```
public void act() {
     if (this.isTouching(Ohen.class)) {
           // vytvoríme pomocnú premennú,
           // do ktorej uložíme vytvorený bonus
           Bonus bonus = null;

           // vygenerujeme číslo od 0 do 99
           int cislo = Greenfoot.getRandomNumber(100);
           if (cislo < 10) {
                 // s pravdepodobnosťou 10% vytvoríme ohen
                 bonus = new BonusOhen();
           }
           else if (cislo < 20) {
                 // s pravdepodobnosťou 10% vytvoríme bombu
                 bonus = new BonusBomba();
           }
           // ak nebola splnená ani jedna z predchádzajúcich
           // podmienok, potom sme s pravdepodobnosťou 80%
           // nič nevygenerovali a v premennej bonus je stále
           // hodnota null.

           // ak sme bonus vygenerovali,
           // vložíme ho do sveta na rovnaké miesto, kde bol múr
           World svet = this.getWorld();
           if (bonus != null) {
                 svet.addObject(bonus, this.getX(), this.getY());
           }
           // nakoniec odstránime múr zo sveta
           svet.removeObject(this);
     } // ak je múr zasiahnutý ohňom
}
```

Pre aplikovanie bonusu na hráča môžeme postupovať dvoma spôsobmi. Prvý spôsob, ktorý nám môže napadnúť je, aby hráč pri svojom pohybe otestoval, či sa nedostal na bunku s bonusom. Ak áno, tak ho na seba aplikuje a odstráni bonus zo sveta. Takýto spôsob môžeme urobiť efektívne, ak do triedy Bonus zavedieme virtuálnu metódu **`aplikujSa()`** s parametrom **`Hrac`**, ktorú hráč bude automaticky vyvolávať a potomkovia triedy **`Bonus`** vhodne prekrývať.

Druhý spôsob, ktorý môžeme zvoliť využíva opačný pohľad na problém. Bonus sám zistí, či na neho vstúpil hráč a ak áno, tak sa aplikuje jeho účinok na hráča a bonus sa odstráni zo sveta. Kontrolu na vstup hráča môže robiť bonus bezpečne v metóde **`act()`** – nikdy inokedy sa predsa hráč nemôže hýbať. Opäť však budeme potrebovať virtuálnu metódu **`aplikujSa()`** s parametrom typu **`Hrac`**. Metóda **`act()`** bonusu by teda mohla fungovať takto:

1) Zisti, či na bonus stúpil hráč.
2) Ak áno, tak:
    a. Aplikuj sa na hráča.
    b. Odstráň sa zo sveta.

To, ako sa konkrétny bonus aplikuje na hráča, závisí v oboch prípadoch od typu bonusu. Preto zavedieme virtuálnu metódu **aplikujSa()** s parametrom typu **Hrac**, ktorú predok automaticky vyvolá a potomkovia v nej aplikujú svoj efekt na hráča odovzdaného v parametri.

Rozdiel medzi obomi prístupmi je iba v tom, kto spustí aplikovanie bonusu – či to bude sám hráč alebo bonus. Ak vezmeme do úvahy aj potenciálne negatívne účinky bonusov, bude lepšie, aby o aplikovaní na hráča rozhodol bonus a nie hráč, ktorý by to mohol odmietnuť.

ÚLOHA 10.12

Pripravte predka **Bonus**. Vytvorte v ňom chránenú virtuálnu metódu bez návratovej hodnoty **aplikujSa()**, ktorá preberie jediný parameter typu **Hrac**. Predok túto metódu môže nechať prázdnu. Následne definujte činnosť v metóde **act()** tak, aby najskôr zistila, či na bonus stúpil hráč (metóda **(Hrac)this.getOneIntersectingObject(Hrac.class)**) a ak áno, tak sa naňho aplikuje (vyvolaním virtuálnej metódy) a nakoniec sa bonus odstráni zo sveta.

```
protected void aplikujSa(Hrac hrac) {
}
public void act(){
     Hrac hrac = (Hrac)this.getOneIntersectingObject(Hrac.class);
     if (hrac != null) {
           this.aplikujSa(hrac);
           World svet = this.getWorld();
           svet.removeObject(this);
     }
}
```

Začnime s aplikovaním bonusu bomba. Jeho úlohou je zvýšiť počet bômb, ktoré hráč môže klásť. Pripomeňme si, že hráč má počítadlo, ktoré určuje, koľko bômb môže položiť. Toto počítadlo sa zníži vždy, keď bombu položí a zvýši sa vždy, keď bomba vybuchne. Bonus bomba teda ako svoje aplikovanie sa iba navýši počítadlo dostupných bômb v hráčovi. Keďže hráč má toto počítadlo ako svoj súkromný atribút, ktorý nie je prístupný iným objektom, hráč musí pripraviť metódu, pomocou ktorej umožní zvýšenie počtu bômb o jedna.

ÚLOHA 10.13

Pridajte do triedy **Hrac** verejnú bezparametrickú metódu **zvysPocetBomb()**, ktorá nemá návratovú hodnotu, a ktorá zvýši počet bômb, ktoré môže hráč položiť o jedna. Potom prekryte metódu **aplikujSa()** v triede **BonusBomba**. Aplikácia bomby znamená navýšenie počtu bômb hráčovi o jedna (vyvolanie metódy **zvysPocetBomb()**).

Najskôr vytvoríme verejnú metódu **zvysPocetBomb()** v triede **Hrac**.

```
public void zvysPocetBomb() {
     this.pocetBomb = this.pocetBomb + 1;
}
```

Potom prekryjeme metódu **aplikujSa()** v triede **BonusBomba**. Metódu **act()** z tejto triedy musíme odstrániť.

```
protected void aplikujSa(Hrac hrac) {
     hrac.zvysPocetBomb();
}
```

Vidíme, že aplikácia bonusu je veľmi jednoduchá, nie sú potrebné žiadne podmienky, iba musíme pripraviť hráča na účinky bonusu (musí poskytovať vhodné metódy). Poďme teraz implementovať bonus oheň. Jeho úlohou je posilniť silu bômb, ktoré kladie hráč. Ten zatiaľ vytvára bomby, ktorých sila je vždy rovnaká (určená hodnotou jeho atribútu **silaBomb**, ktorý je inicializovaný raz, v jeho konštruktore). Na to, aby bolo možné ich silu meniť, musíme urobiť v triede **Hrac** podobné zmeny, ako pri aplikovaní bonusu bomba.

ÚLOHA 10.14

Pridajte do triedy **Hrac** verejnú bezparametrickú metódu **zvysSiluBomb()**, ktorá nemá návratovú hodnotu, a ktorá zvýši hodnotu atribútu **silaBomb** o jedna. Potom prekryte metódu **aplikujSa()** v triede **BonusOhen**, a zvýšte v nej silu bômb hráčovi o jedna.

Najskôr vytvoríme verejnú metódu **zvysSiluBomb()** v triede **Hrac**.

```
public void zvysSiluBomb() {
     this.silaBomb = this.silaBomb + 1;
}
```

Potom prekryjeme metódu **aplikujSa()** v triede **BonusOhen**. Metódu **act()** z tejto triedy musíme odstrániť.

```
protected void aplikujSa(Hrac hrac) {
     hrac.zvysSiluBomb();
}
```

ZHRNUTIE

Pre generovanie náhodných čísiel v jazyku Java sa používa trieda **Random** z balíčka **java.util.Random**. Táto trieda funguje ako „kocka“, ktorá pomocou metódy **nextInt(int n)** generuje náhodné číslo z intervalu $\langle 0, n)$. Každé číslo má rovnakú pravdepodobnosť $\frac{1}{n}$, že bude vygenerované. Objektom, ktoré vracajú náhodné čísla hovoríme generátory náhodných čísel. Pre dodržanie matematických vlastností generátora (teda aby každé číslo bolo generované s rovnakou pravdepodobnosťou) je nevyhnutné generátor vytvoriť raz a uložiť si ho.

Ďalší spôsob generovania náhodných čísel ponúka nástroj Greenfoot. Ten poskytuje metódu **`getRandomNumber(int n)`**, ktorá funguje principiálne rovnako, ako metóda **`nextInt()`** v triede **`Random`**. Pre jej použitie však nie je potrebné vytvárať žiadne inštancie triedy **`Random`**.

ÚLOHY NA PRECVIČOVANIE

ÚLOHA 10.A

Vytvorte v triede **`Arena`** metódu **`vytvorNahodneSteny()`**, ktorá v parametri preberie počet stien a takýto počet stien vygeneruje a vloží na náhodné voľné miesta do sveta.

ÚLOHA 10.B

Upravte konštruktor triedy **`Ohen`** tak, aby si pri vzniku inštancia zvolila náhodný obrázok ohňa. Všetky ohne majú rovnakú pravdepodobnosť výskytu. Pre zmenu obrázku použite metódu **`setImage()`** triedy **`Actor`**, ktorá v parametri preberá reťazec s názvom súboru obrázka (obrázky musia byť umiestnené v priečinku **`images`** v koreňovom priečinku projektu).

ÚLOHA 10.C

Pridajte bonusy, ktoré ovplyvnia hráča negatívne, teda znížia mu počet bômb a znížia mu dosah bômb.

ÚLOHA 10.D

Pridajte bonus teleport. Nech sa tento bonus vytvorí s pravdepodobnosťou 5 %. Teleport presunie hráča na náhodné pole v aréne.

INDEX OBRÁZKOV, GRAFOV A TABULIEK

BIBLIOGRAFIA

[1] School of Computing, University of Kent in Canterbury, UK, „Greenfoot,“ [Online]. Available: www.greenfoot.org.

[2] SLEX - Slovník slovenského jazyka. SLEX - Slovník slovenského jazyka. [Online] 2019. http://www.slex.sk.

[3] Wróblewski, Piotr. Algoritmy Datové struktury a programovací techniky. [prekl.] Bogdan Kiszka, Marek Michalek. Brno : Computer Press, 2004. s. 351. 80-251-0343-9.

[4] Bomberman sprite png » PNG Image. PNG Image. [Online] 2019. https://pngimage.net/bomberman-sprite-png/.

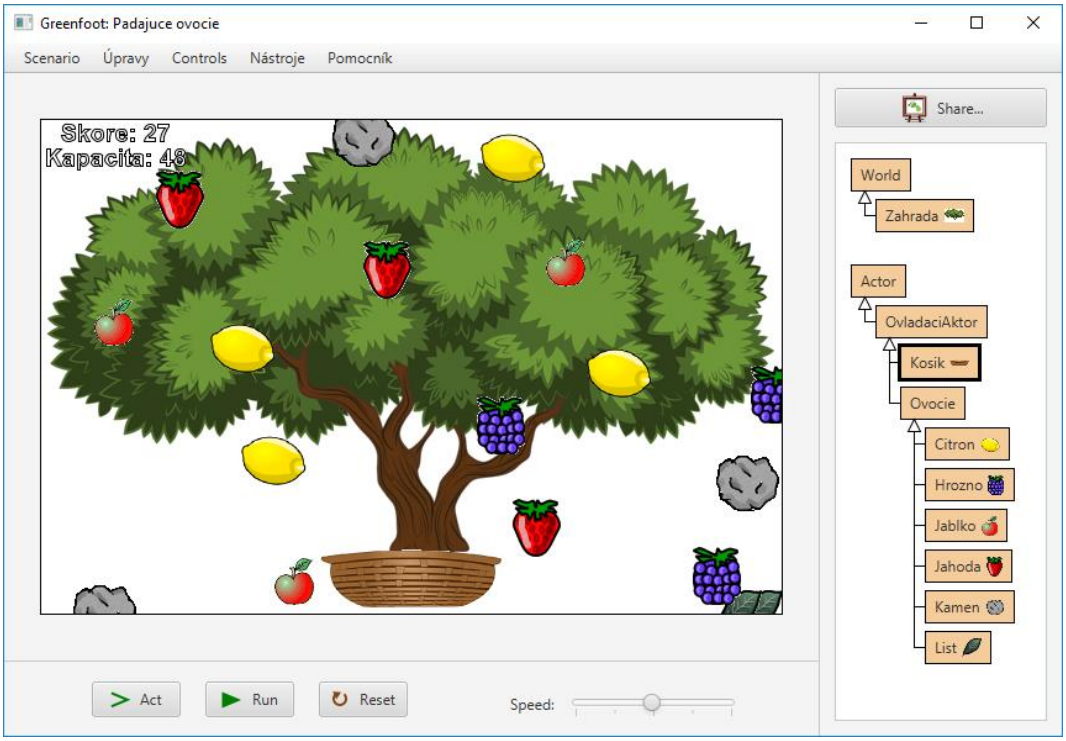
[5] Mine Icon #357734 - Free Icons Library. Free Icons Library. [Online] 2019. https://icon-library.net/icon/mine-icon-7.html">Mine Icon #357734.

PRÍLOHA – STRUČNÝ PREHĽAD ĎALŠÍCH PROJEKTOV

PADAJÚCE OVOCIE

POPIS A PRAVIDLÁ HRY

Padajúce ovocie je hra pre jedného hráča. Hráč pomocou myšky ovláda košík, do ktorého zachytáva ovocie a iné predmety padajúce rôznou rýchlosťou zo stromu. Každé ovocie a predmet má inú hodnotu, ktorá sa pripočíta do skóre, keď sú ovocie alebo predmet zachytené alebo spadnú na zem. Košík má určenú kapacitu. Hra končí, keď sa kapacita košíka naplní.



ZAMERANIE PROJEKTU

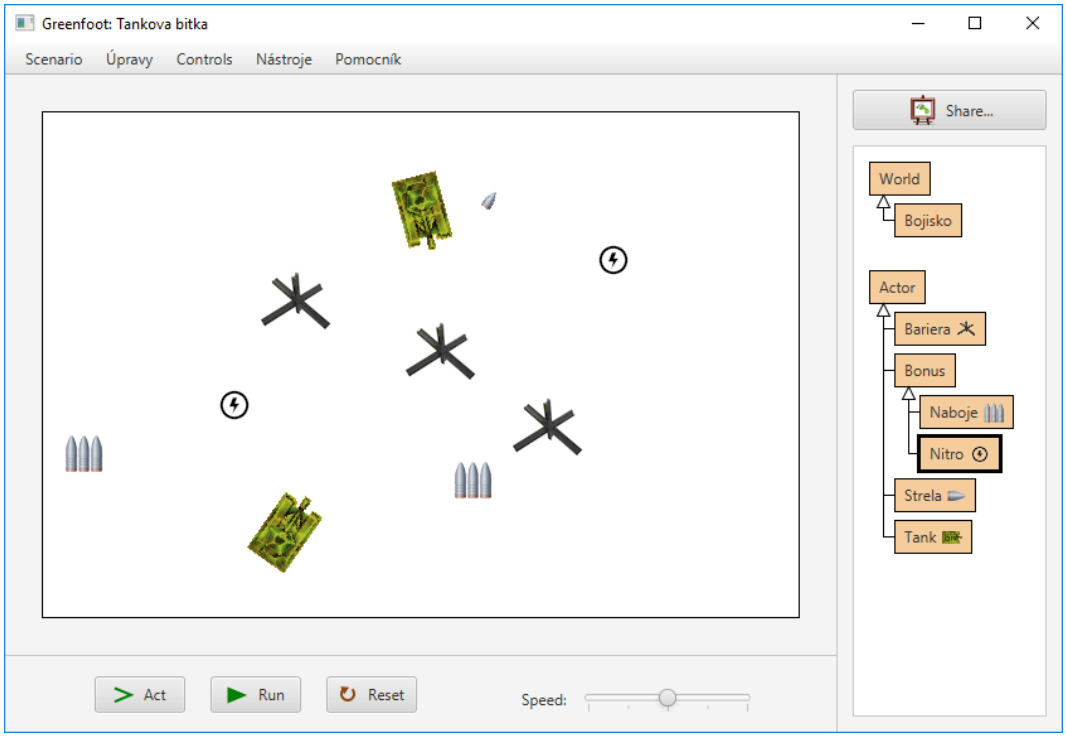
Projekt je zamýšľaný ako veľmi jednoduchá hra, s pomocou ktorej je možné rýchlo ukázať prostredie Greenfoot (ovládanie prostredia, tvorbu a interakciu inštancií tried). Voliteľne je možné projekt rozšíriť o dedičnosť a polymorfizmus (zavedenie predka **`Ovocie`** a prekrytie príslušnej metódy v potomkovi), avšak je možné sa tomu vyhnúť pomocou úplného vetvenia a polymorfizmus do projektu pridať neskôr (alebo nepridávať vôbec). Projekt taktiež obsahuje triedu **`OvladaciAktor`**, ktorá rozširuje funkcionalitu triedy **`Actor`**, čím uľahčuje tvorbu jednoduchých aplikácií.

- Zoznámenie sa s prostredím Greenfoot.
- Jednoduchá práca s potomkami pripravenej triedy **`OvladaciAktor`**.
- Neúplné, úplné a viacnásobné vetvenie kódu (**`if`**, **`if-else`**, **`switch`**).
- Práca s náhodnými číslami (**`Greenfoot.getRandomNumber()`**, trieda **`Random`**).
- Polymorfizmus (trieda **`Ovocie`**).

TANKOVÁ BITKA

POPIS A PRAVIDLÁ HRY

Tanková bitka je hra pre dvoch hráčov. Každý hráč ovláda pohyb a strieľanie svojho tanku na bojisku. Tank má obmedzený počet nábojov a danú rýchlosť. Na bojisku sa nachádzajú zábrany a rôzne bonusy (doplnenie nábojov, zvýšenie rýchlosti pohybu tanku a pod.). Cieľom hry je strelou zneškodniť protivníka.



ZAMERANIE PROJEKTU

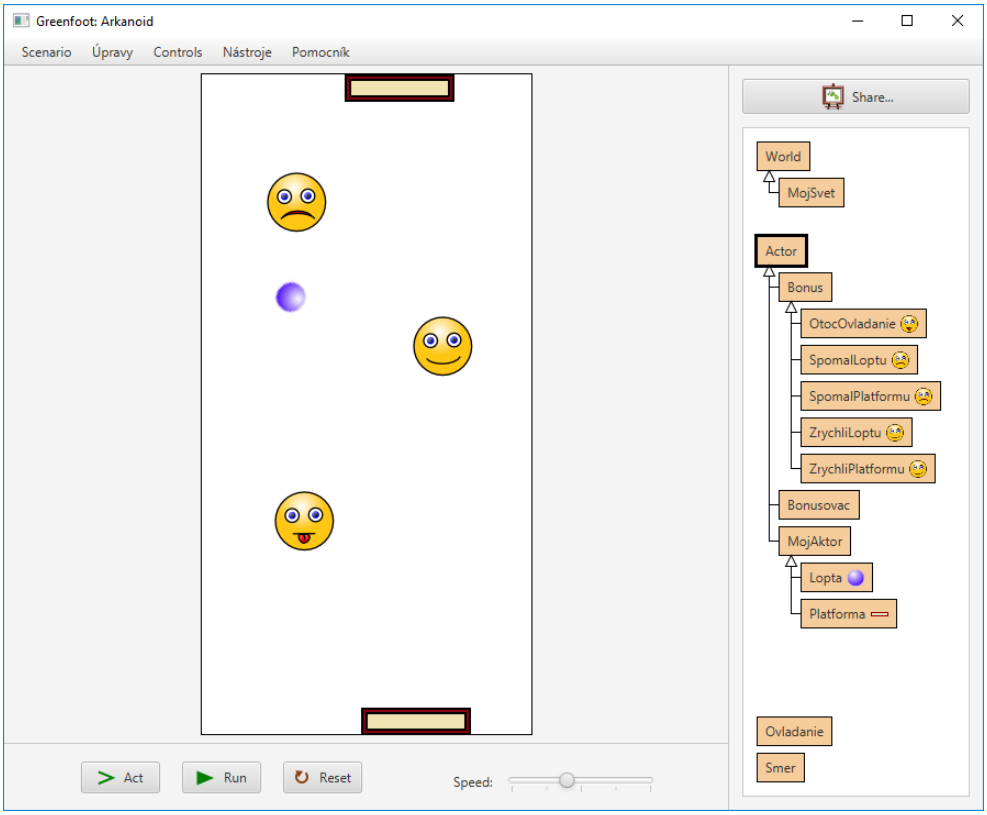
Projekt je zamýšľaný ako jednoduchá hra, s pomocou ktorej je možné rýchlo ukázať prostredie Greenfoot (ovládanie prostredia, tvorbu a interakciu inštancií tried). Voliteľne je možné projekt rozšíriť o dedičnosť a polymorfizmus (zavedenie predka **`Bonus`** a prekrytie príslušnej metódy v potomkovi), avšak je možné sa tomu vyhnúť pomocou úplného vetvenia a polymorfizmus do projektu pridať neskôr (alebo nepridávať vôbec). Projekt viac využíva spoluprácu objektov (vytvorenie inštancie triedy **`Strela`** z triedy **`Tank`**) a dokáže jednoduchým spôsobom zaviesť prácu s náhodou (napr. náhodné rozloženie inštancií triedy **`Bariera`** pri vzniku sveta).

- Zoznámenie sa s prostredím Greenfoot.
- Zapúzdrenie objektov.
- Spolupráca objektov.
- Neúplné, úplné a viacnásobné vetvenie kódu (**`if`**, **`if-else`**, **`switch`**).
- Polymorfizmus (trieda **`Bonus`**).

ARKANOID

POPIS A PRAVIDLÁ HRY

Arkanoid je známa hra odrážania lopty pre dvoch hráčov, ktorí ovládajú horizontálne sa pohybujúce pálky. Lopta zmení smer ak sa odrazí od pálky alebo od ľavej resp. pravej steny. Prehrá ten hráč, ktorému sa nepodarí včas odraziť loptičku svojou pálkou, a teda loptička sa dotkne hornej resp. dolnej steny.



ZAMERANIE PROJEKTU

Hra využíva jednoduchú vektorovú algebru. Môžeme vytvoriť triedu **`Smer`** (analogicky k vektoru v matematike), ktorá bude nezávislá na triede **`World`** a **`Actor`**. S využitím tejto triedy sa pred každým pohybom prepočítavajú súradnice a uhol loptičky.

Hra tiež uvádza triedu **`Ovladanie`**, ktorú je možné využiť v iných projektoch.

V hre môžu byť doplnené aj bonusy (napr. spomalenie loptičky alebo pálky, zrýchlenie loptičky alebo pálky, otočenie ovládania, atď.) a využiť tak polymorfizmus.

- Zoznámenie sa s prostredím Greenfoot.
- Zapúzdrenie objektov.
- Spolupráca objektov.
- Neúplné, úplné a viacnásobné vetvenie kódu (**`if`**, **`if-else`**, **`switch`**).
- Polymorfizmus (trieda **`Bonus`**).
- Tvorba vlastných tried nezávislých na predkoch **`Actor`** a **`World`** (**`Ovladanie`**, **`Smer`**).

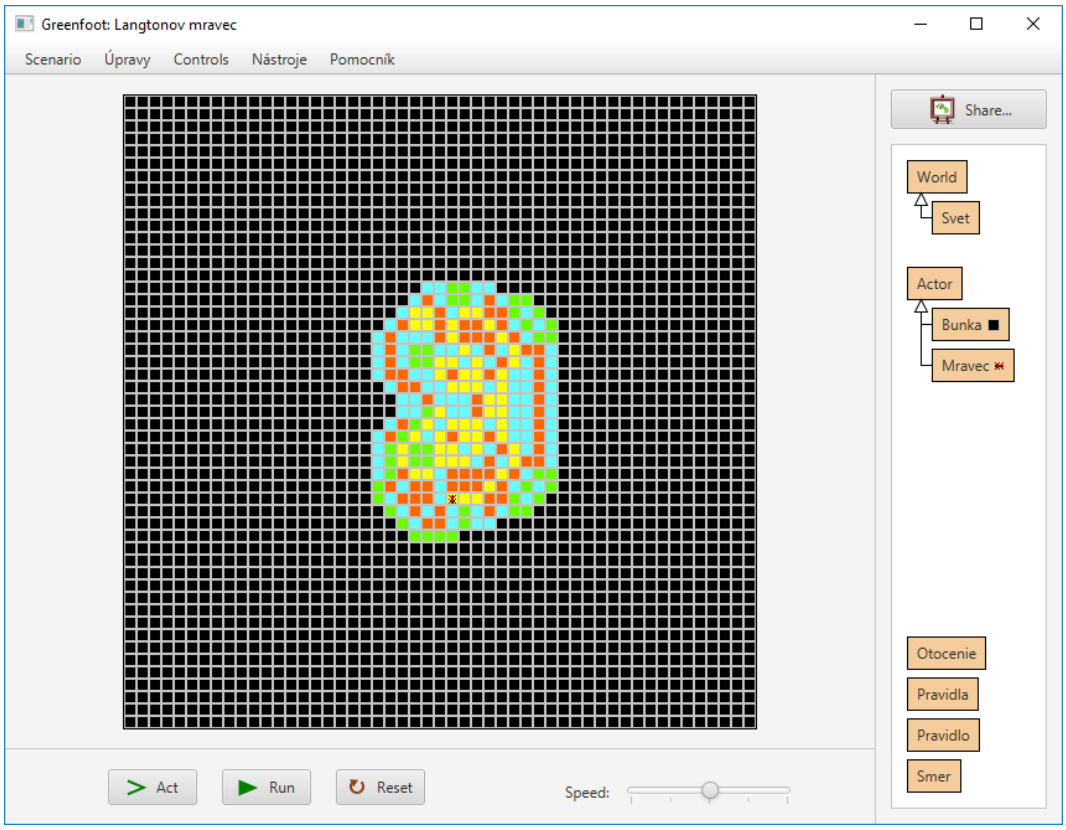
LANGTONOV MRAVEC

POPIS PROJEKTU

Langtonov mravec je dvojrozmerný univerzálny Turingov stroj, ktorý má veľmi jednoduché pravidlá, avšak komplexné správanie. Po štvorcovej mriežke buniek sa pohybuje mravec. Každá bunka môže byť v dvoch stavoch (živá alebo mŕtva). Pre pohyb mravca platí, že:

- ak je na mŕtvej bunke, otočí sa o 90° doprava, zmení bunku na živú a pohne sa o 1 bunku vpred,
- ak je na živej bunke, otočí sa o 90° doľava, zmení bunku na mŕtvu a pohne sa o 1 bunku vpred.

Pravidlá môžeme rôzne rozširovať, napr. bunková sieť môže mať počiatočné nastavenie, bunky môžu mať viac stavov, atď. Pre viac informácií navštívte napr. tento odkaz.



ZAMERANIE PROJEKTU

Tento projekt nie je interaktívna hra, ale simulácia jednoduchého systému. Kladie väčší dôraz na skúmanie systému (môžete skúsiť predpovedať pohyb mravca; vložiť do sveta viac mravcov; diskutovať, či sa dá zrekonštruovať spätný pohyb mravca atď.). Základná verzia Langtonovho mravca je veľmi jednoduchá na implementáciu (stačia iba triedy **`Bunka`** a **`Mravec`**).

Do projektu je však možné pridať univerzálne pravidlá a definovať si vlastné správanie bez nutnosti modifikácie triedy **`Mravec`**. Projekt dokáže tiež veľmi jednoduchou formou uviesť aj

prácu s triedou s konštantou extenziou (teda s pevným počtom inštancií) – **`enum`** (v jazyku Java je typ **`enum`** implementovaný inak ako v iných programovacích jazykoch).

- Zapúzdrenie objektov.
- Vetvenie kódu (v jednoduchej verzii aplikácie využijete iba jediné úplné vetvenie, v komplexnej verzií sa pridá viacnásobné vetvenie).
- Tvorba vlastných tried nezávislých na predkoch **`Actor`** a **`World`** (**`Pravidla`**, **`Pravidlo`**).
- Práca so zoznamom.
- Typ enum (**`Otocenie`** , **`Smer`**).

SPACE INVADERS

POPIS PROJEKTU

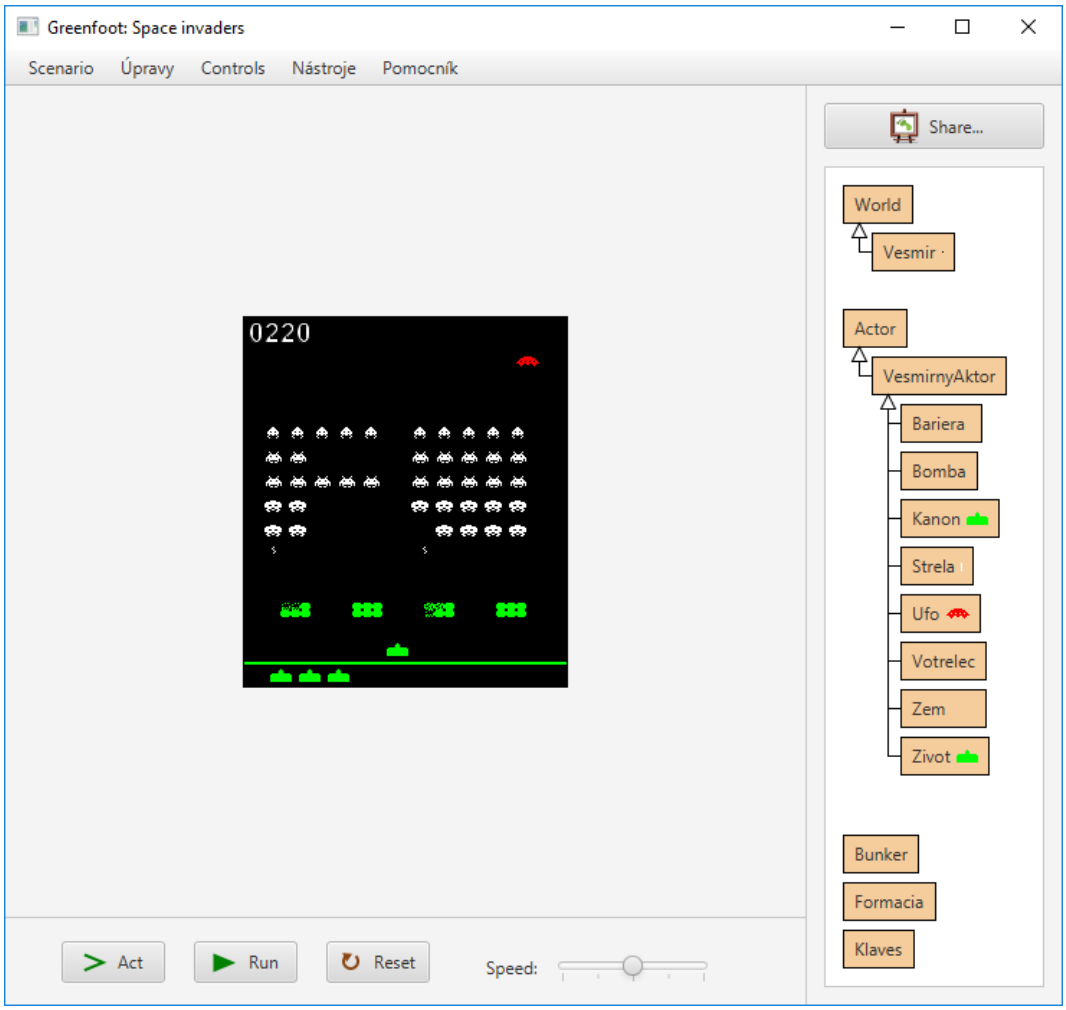
Space invaders je klasická arkádová hra z roku 1978. Hráč ovláda kanón v dolnej časti hracej plochy. Kanón je schopný vystreliť proti votrelcom, ktorí sa pohybujú vo formácií nad ním. Aktívna je súčasne vždy len jedna strela.

Votrelci sa hýbu zľava doprava a naopak. Pomaly sa približujú k zemi. Existujú tri typy votrelcov (vojak, kapitán a generál), ktorí sú typicky zoradení v piatich riadkoch a jedenástich stĺpcoch. Občas sa objaví ufo, ktoré po zásahu náhodne zvýši skóre. Votrelci bombardujú zem bombami.

Hráč má na začiatku tri životy. Za každých 1000 bodov získa hráč život navyše. Zásah bombou odoberie hráčovi život. Hráča chránia štyri bunkre. Bunker poškodí bomba od votrelcov, ale aj strela od hráča.

Hra končí ak:

- hráč zničí všetkých votrelcov (zjednodušenie),
- hráč stratí všetky životy,
- votrelec dosadne na zem alebo na kanón.



ZAMERANIE PROJEKTU

Ide o pomerne komplexný projekt, ktorý je vhodný pre študentov, ktorí už zvládli základné princípy objektovo orientovaného programovania. Projekt využíva koncept polí a matíc, a teda využíva vo väčšej miere cykly. Rovnako využíva triedy mimo hierarchie tried **`World`** a **`Actor`**.

Zložitosť projektu umožňuje precvičiť do hlbšej miery mnoho konceptov. Je možné tvoriť iné počiatočné formácie votrelcov, pridávať ďalšie schopnosti kanónu, robiť iné typy bunkrov, atď. Projekt je mimoriadne vhodný hlavne na prácu s cyklami (tvorba počiatočnej formácie zloženej z votrelcov rôznych typov; tvorba jednej bariéry zloženej z malých prvkov, tvorba celého bunkru z bariér, rozloženie bunkrov).

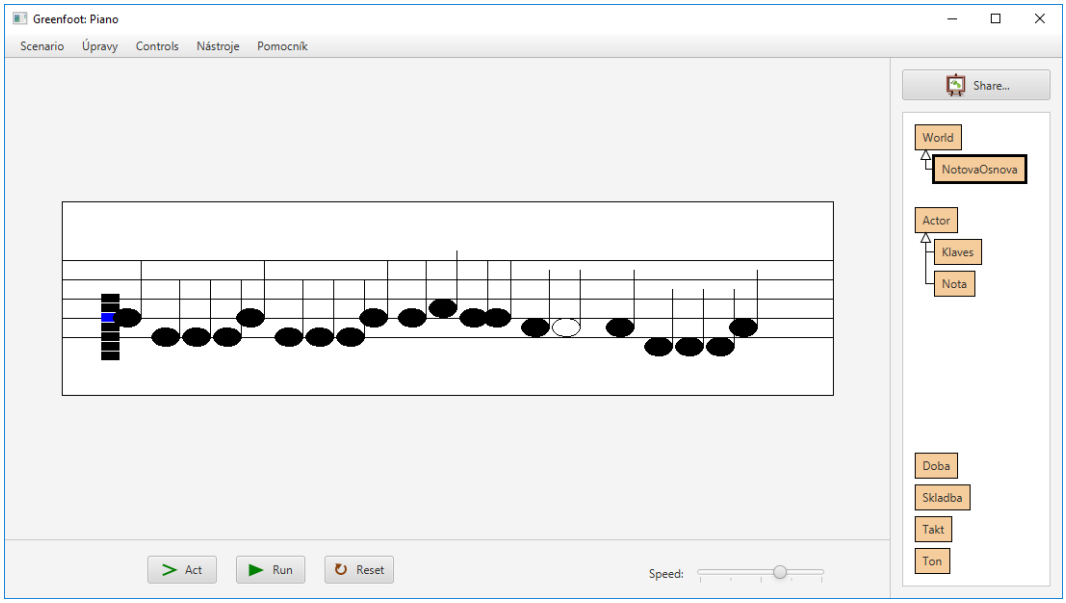
- Zapúzdrenie objektov.
- Vetvenie kódu.
- Tvorba vlastných tried nezávislých na predkoch **`Actor`** a **`World`** (**`Bunker`**, **`Formacia`** a **`Klaves`**).
- Práca s konštantami (trieda **`Klaves`**).
- Práca s poľami a maticami.
- Cykly (**`while`**).

PIANO

POPIS PROJEKTU

Hra je inšpirovaná známu hrou Guitar Hero. Používateľ hrá na piane prostredníctvom stláčania jednotlivých kláves. Po notovej osnove sa pohybujú noty oproti klávesom, ktoré sú v ľavej časti hracej plochy. Keď sa nota prekrýva s klávesom a hráč stlačí správne tlačidlo (napr. funkčné klávesy), zaznie tón.

Projekt umožňuje tvoriť vlastné skladby a tie následne prehrať na piane.



ZAMERANIE PROJEKTU

Projekt je vo svojej podstate jednoduchý a základnú verziu bez tried definujúcich skladbu je možné naprogramovať za krátku dobu. Projekt však ukazuje, ako je možné programovo vytvoriť vlastný obrázok. Navyše uplatňuje príkaz viacnásobného vetvenia bez príkazu **`break`**. Taktiež je v ňom možné vhodne uplatniť typ **`enum`**.

- Spolupráca objektov.
- Práca s triedou **`GreenfootImage`**.
- Tvorba vlastných tried nezávislých na predkoch **`Actor`** a **`World`** (**`Skladba`**, **`Takt`**).
- Práca so zoznamom.
- Práca s poľami.
- Cykly (**`for`**, **`for each`**).

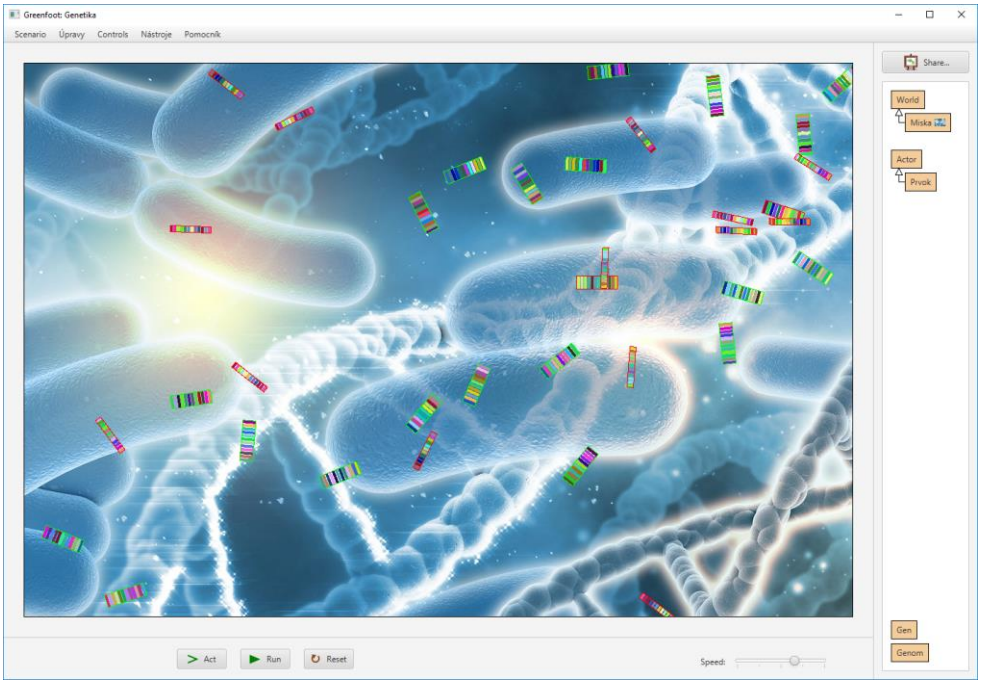
GENETIKA

POPIS PROJEKTU

Projekt nie je hra, ale simulácia pohybu jednoduchých organizmov – prvokov. Každý prvok vlastní náhodný genóm zložený z presne určeného počtu génov. Každý gén obsahuje červenú (R), zelenú (G) a modrú (B) zložku. Cieľom simulácie je vytvoriť prvok s krásnym genómom.

Počas svojho života sa prvoky náhodne pohybujú. Keď prvok nájde vhodného jedinca na párenie (dospelý jedinec, s ktorým sa nepáril naposledy), tak sa s ním spári. Párenie je proces výmeny génov tak, aby sa viac podobali krásnemu genómu. Keď sa prvok spári dostatočný počet krát, nasleduje jeho rozdelenie na dvoch geneticky rovnakých jedincov.

Simuláciou môžeme skúmať rôzne vlastnosti systému, napr. aký počet jedincov je hraničný, aby populácia v miske prežila, ako je možné upraviť intervaly párenia, dospelosti, atď.



ZAMERANIE PROJEKTU

Jedná sa o relatívne zložitý projekt, hoci pozostáva z malého množstva tried. Vo veľkej miere využíva spoluprácu objektov a prácu s náhodou. Pre jeho zvládnutie je potrebná znalosť vetvenia, cyklov a polí. Projekt taktiež vytvára vlastné obrázky pre jednotlivé prvoky na základe ich genómu – teda vizualizuje dáta uložené v poli.

- Spolupráca objektov.
- Práca s triedou **`GreenfootImage`**.
- Tvorba vlastných tried nezávislých na predkoch **`Actor`** a **`World`** (**`Gen`**, **`Genom`**).
- Práca s náhodnými číslami (**`Greenfoot.getRandomNumber()`**).
- Práca s poľami a zoznamom.
- Cykly (**`for`**, **`for each`**).

PRÍLOHA - PUBLIKOVANIE PROJEKTU

Každý projekt, ktorý je vytvorený v prostredí Greenfoot, je možné viacerými spôsobmi publikovať a zdieľať:

- projekt je možné zverejniť na webovej stránke www.greenfoot.org v sekcií **Greenfoot Library**.
- je možné vytvoriť spustiteľný **.jar** súbor.
- je možné vytvoriť archív s príponou **.gfar**, ktorý obsahuje všetky dôležité časti projektu.

Pre všetky horeuvedené spôsoby je potrebné vyvolať dialóg **Share**: z hlavného menu **Scenario** zvolíme položku **Share...** alebo použijeme klávesovú skratku **CTRL+E**.

ZVEREJNENIE PROJEKTU NA WEBOVEJ STRÁNKE WWW.GREENFOOT.ORG

Na zverejnenie projektu na stránke www.greenfoot.org je potrebná malá príprava pozostávajúca z dvoch krokov:

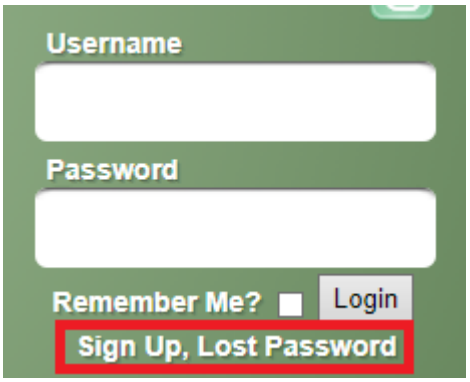
1) Vytvorenie konta na stránke www.greenfoot.org. Ak máte konto vytvorené, je možné tento krok preskočiť.
2) Príprava náhľadu pre projekt.

Vytvorenie konta

Pre vytvorenie konta navštívte webovú lokalitu www.greenfoot.org. a pokračujte ďalej pomocou tlačidla **Enter Greenfoot site** (pozri obrázok 5).

Obrázok 5: Pokračovanie do sekcie home z úvodnej stránky www.greenfoot.org

V pravej hornej časti ďalšej stránky je možné sa prihlásiť s využitím existujúceho konta alebo si vytvoriť nové konto. Pre vytvorenie nového konta klikneme na **Sign Up, Lost Password** (pozri obrázok 6).



Obrázok 6: Pokračovanie na stránku s vytvorením nového konta

V pravej časti stránky je potrebné v časti **Sign Up** (zobrazenej na obrázku 7) zadať platný e-mail a prepísať overovací kód. Po kliknutí na tlačidlo **Sign Up** Vám bude poslaný e-mail na uvedenú adresu. Pre pokračovanie v registračnom procese je potrebné validovať e-mailovú adresu kliknutím na odkaz v doručenom e-maile.

Obrázok 7: Registrácia e-mailovej adresy

Po validovaní projektu je potrebné ukončiť registračný proces vytvorením užívateľského konta. Na túto stránku budete automaticky presmerovaní po kliknutí na odkaz v overovacom e-maile z predchádzajúceho kroku. Na formulári (zobrazenom na obrázku 8) je potrebné vyplniť dve povinné polia:

- Používateľské meno (**Username**) – toto meno je verejné a nebude ho možné už viac meniť.
- Heslo (**Password**) a potvrdenie hesla (**Password Confirmation**).

Profil je možné ďalej doplniť o ďalšie voliteľné údaje:

- Mesto (**Your hometown**).
- Škola (**Your school/university**).
- Rok (**Your year of birth**) a dátum (**Month**) narodenia.
- Ďalšie informácie o Vás (**About you**).
- Fotka (**A photo of you**) vo formáte PNG alebo JPEG s veľkosťou maximálne 1MB a s rozmermi 200x200 pixelov (fotka bude automaticky upravená na túto veľkosť). Fotku nahráte tlačidlom Vybrať súbor.
- Interval, ako často budete dostávať relevantné notifikácie (**Receive email digests**).

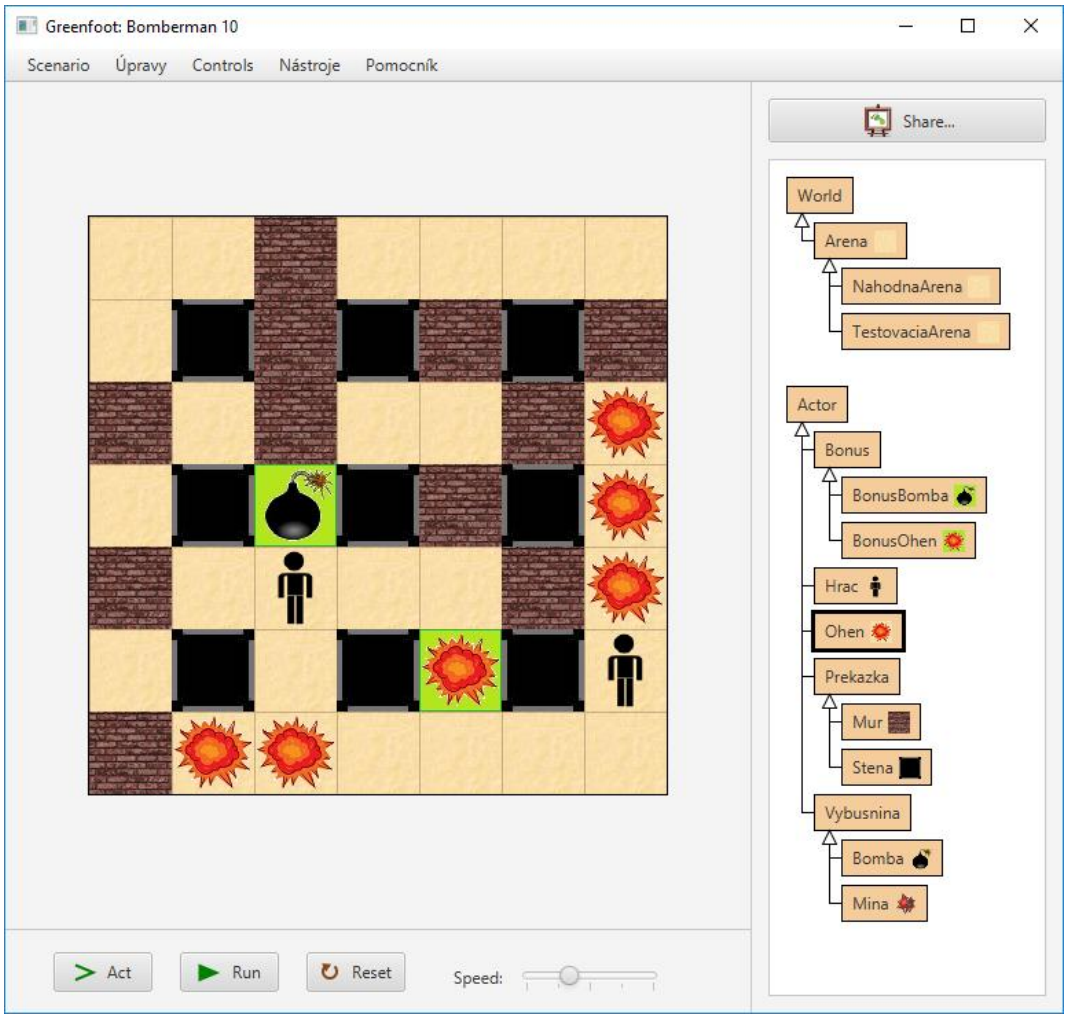
Registračný proces dokončíte tlačidlom **`New Account`** v dolnej časti formulára. V prípade, že ste zvolili používateľské meno, ktoré nie je dostupné, budete opätovne vyzvaní, aby ste zadali nové používateľské meno a heslo (môže samozrejme ostať pôvodne zadané).

Obrázok 8: Registrácia nového používateľa

Používateľské meno a heslo si uschovajte. Využijete ho na prístup do Greenfoot Library, do diskusného fóra, ale aj na publikovanie projektov z prostredia Greenfoot.

Príprava náhľadu sveta

Uverejneným projektom sa vždy automaticky vytvorí náhľad. Tento náhľad je skonštruovaný z aktuálneho rozloženia sveta. Je preto dobré, aby sme svet upravili tak, aby čo najlepšie vystihoval aplikáciu. To nemusí byť nutne počiatočné rozloženie sveta, kde aplikácia pôsobí „nudne“. Dobré je preto aplikáciu spustiť, a **zastaviť** vtedy, keď bude vyzerať pútavo, prípadne ručne vytvoriť inštancie objektov tak, aby vytvorili zaujímavú scénu. Samozrejme, pútavosť je veľmi subjektívna a je možné, že sa Vám páči aj základné rozloženie sveta. Príklad rozohranej hry Bomberman, ktorá poslúži ako základ pre vytvorenie náhľadu aplikácie, je zobrazený na obrázku 9.



Obrázok 9: Rozohraná hra Bomberman

Zverejnenie projektu

Zverejnenie projektu je jednoduché. Z dialógového okna **`Share`** vyberieme prvú možnosť **`Publish the scenario to: Greenfoot Gallery`** (pozri obrázok 10). Následne vyplníme položky.

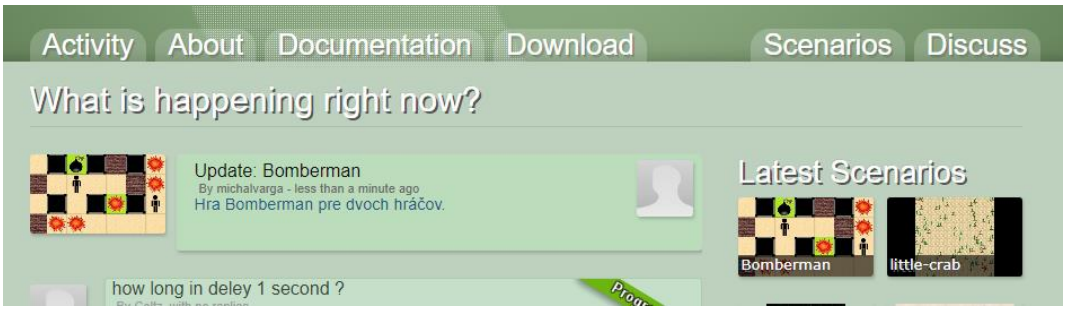
- Vytvorte ikonku (**`Scenario icon`**). Vytvorený obrázok ukazuje náhľad na aktuálne rozloženie sveta. Pomocou vertikálneho posuvníka je možné obrázok priblížiť. Priblížený obrázok je možné pomocou ľavého tlačidla myši chytiť a posunúť, a teda upraviť tak jeho náhľad, kým nebudeme s ikonou spokojní.
- Názov (**`Title`**).
- Krátky popis (**`One-line desciption`**).
- Podrobnejší popis (**`Longer description`**).
- Voliteľne zadajte odkaz na vlastnú stránku (**`Your own page (URL)`**).
- Pre jednoduchšie vyhľadávanie vášho projektu ostatnými používateľmi zaškrtnite značky (**`Popular tags`**). Ak ponúkané značky nestačia, môžete pridať ďalšie (**`Additional tags`**), každú novú značku jednoducho uveďte na nový riadok v textovom poli.

- Rozhodnite sa, či chcete zverejniť zdrojové kódy (Publish source code) a či chcete uzamknúť projekt (**`Lock scenario`**). Uzamknutému projektu nie je možné vo webovom prehliadači meniť rýchlosť pomocou posuvníka Speed.
- Zadajte prihlasovacie údaje (používateľské meno – **`Username`**, heslo - **`Password`**) do Vášho konta do Greenfoot Gallery.

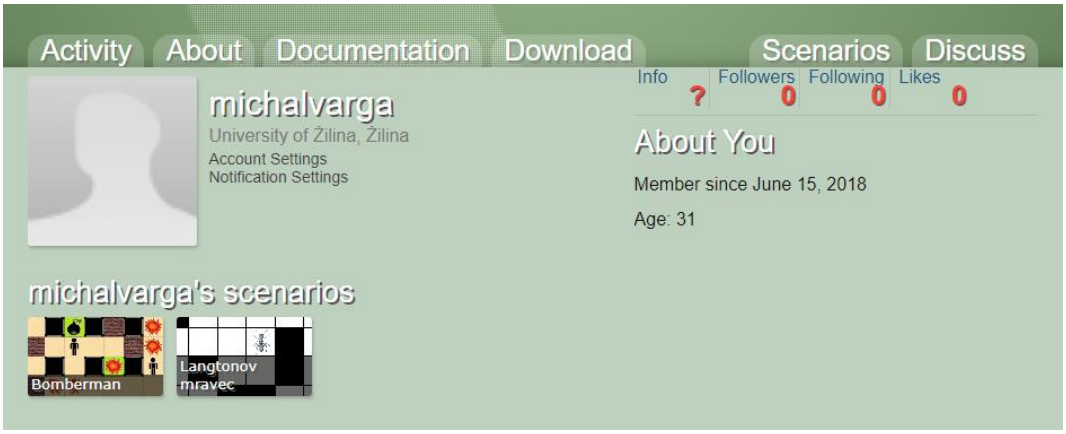
Projekt zverejníte tlačidlom **`Export`**. Po zverejnení projektu budete presmerovaní na webovú stránku **`Greenfoot Gallery`**, na ktorej bude automaticky vytvorená novinka o pridaní vašej hry (pozri obrázok 11). Po prihlásení sa do svojho konta si budete môcť prezrieť všetky vami zverejnené scenáre (pozri obrázok 12). Po kliknutí na konkrétny scenár sa zobrazia detaily tak, ako boli zadané v dialógovom okne pre publikovanie (pozri obrázok 13).

Takýmto spôsobom je možné publikovať všetky projekty, jednoducho a efektívne zdieľať svoje zdrojové kódy, komentovať projekty a čerpať ďalšie inšpirácie. Projekt sa stáva dostupný odkiaľkoľvek. Naviac je ho možné priamo v prehliadači spustiť a pochváliť sa tak komukoľvek s vytvorenou aplikáciou.

Obrázok 10: Dialógové okno pre zverejnenie projektu na webovej stránke www.greenfoot.org



Obrázok 11: Novinka na hlavnej stránke Greenfoot Gallery - pridanie projektu Bomberman



Obrázok 12: Prehľad scenárov používateľa

Obrázok 13: Zverejnená hra Bomberman

VYTVORENIE SPUSTITEĽNÉHO SÚBORU .JAR

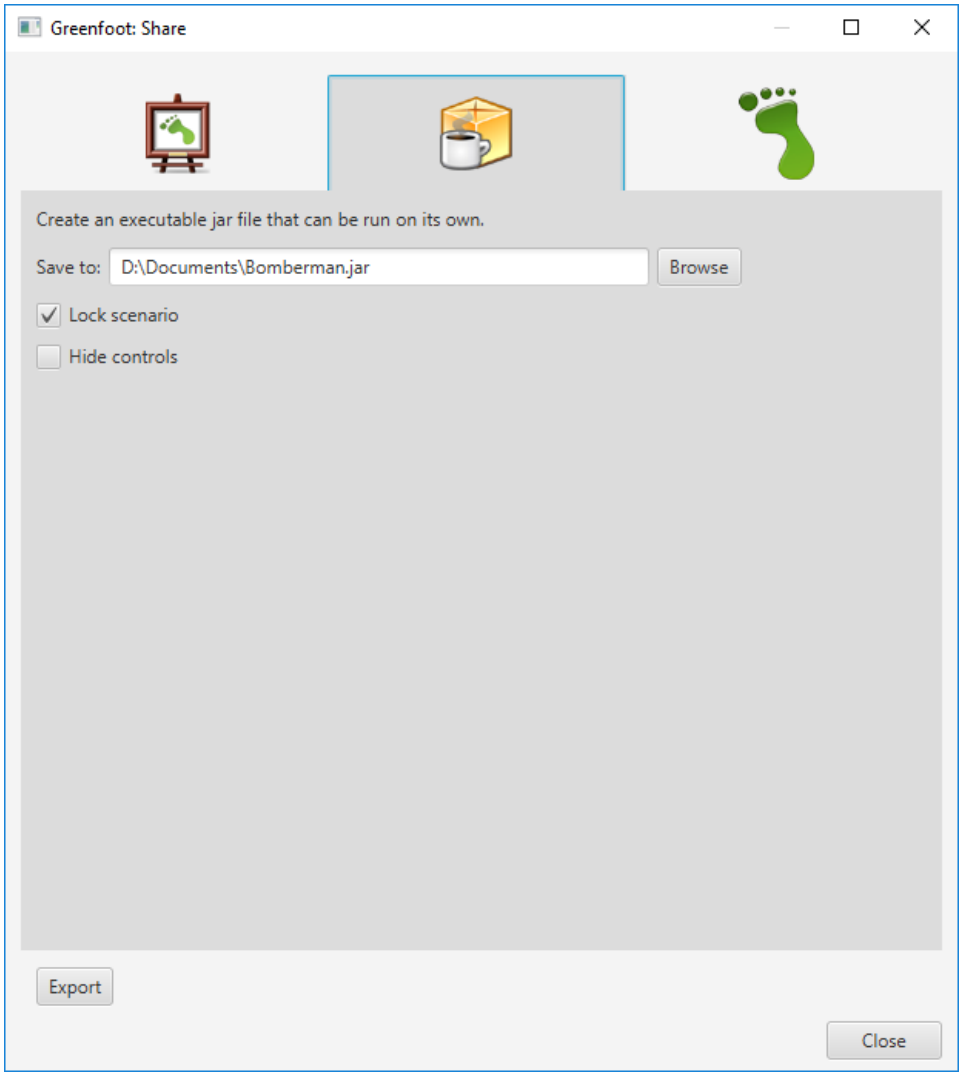
Pre spustenie aplikácií napísaných v jazyku Java je potrebné prostredie **`Java Runtime Environment (JRE)`**. (Ak máte nainštalovaný Greenfoot, máte aj JRE. Ak však chcete spustiť Vašu aplikáciu na počítači, ktorý nemá Javu nainštalovanú, môžete ju stiahnuť z tohto odkazu https://www.oracle.com/technetwork/java/javase/downloads/jre8-downloads-2133155.html a nainštalovať). JRE zabezpečí, že aplikáciu je možné spustiť nezávisle na operačnom systéme zariadenia, preto je aplikácia napísaná v jazyku Java spustiteľná na rôznych zariadeniach s JRE.

Spustiteľné súbory v jazyku Java majú príponu **`.jar`** (**J**ava **AR**chive). Tento súbor obsahuje zabalené potrebné zdrojové súbory, metadáta, zdroje, atď. Taktiež obsahuje popis, ktorá trieda obsahuje metódu **`main`**, z ktorej sa celá aplikácia spúšťa. Pripomeňme, že nástroj Greenfoot vytvára jedinú metódu **`main`** automaticky a preto je tvorba výsledného **`.jar`** súboru jednoduchšia, ako v iných vývojových prostrediach.

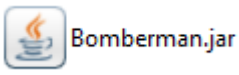
Pre vytvorenie spustiteľného **`.jar`** súboru je potrebné v dialógovom okne **`Share`** zvoliť druhú možnosť – **`Create an executable jar file that can be run on its own`** (pozri obrázok 14). V dialógu potom:

- zadáme plný názov výsledného **`.jar`** archívu (**`Save to`**). Zmenu ponúkanej cesty vykonáme cez tlačidlo **`Browse`**,
- zvolíme, či chceme uzamknúť projekt (**`Lock scenario`**) a
- zvolíme, či majú byť skryté ovládacie prvky prostredia Greenfoot (tlačidlá **`Run`** a **`Reset`** v dolnej časti okna).

Výsledný **`.jar`** súbor (v systéme Windows zobrazený na obrázku 15) vytvoríte tlačidlom **`Export`**.



Obrázok 14: Dialógové okno pre vytvorenie spustiteľného súboru `.jar`

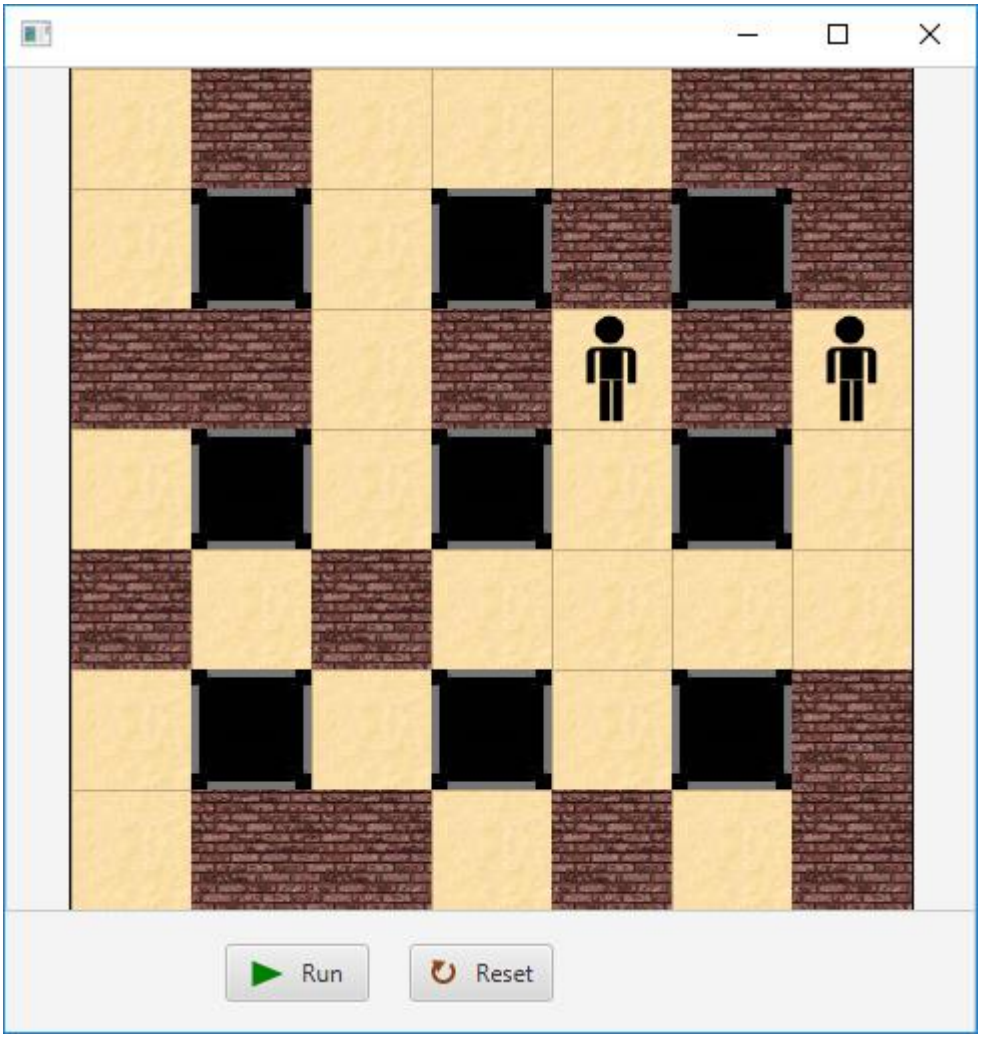


Obrázok 15: Vytvorený `.jar` súbor pre hru Bomberman v prostredí Windows

Výsledný súbor môžeme spustiť dvoma spôsobmi:

1) po dvojkliku myšou naň alebo
2) po zadaní príkazu **`java -jar súbor`** do príkazového riadku, v našom prípade **`java -jar D:\Documents\Bomberman.jar`**.

Spustená aplikácia je zobrazená na nasledujúcom obrázku.



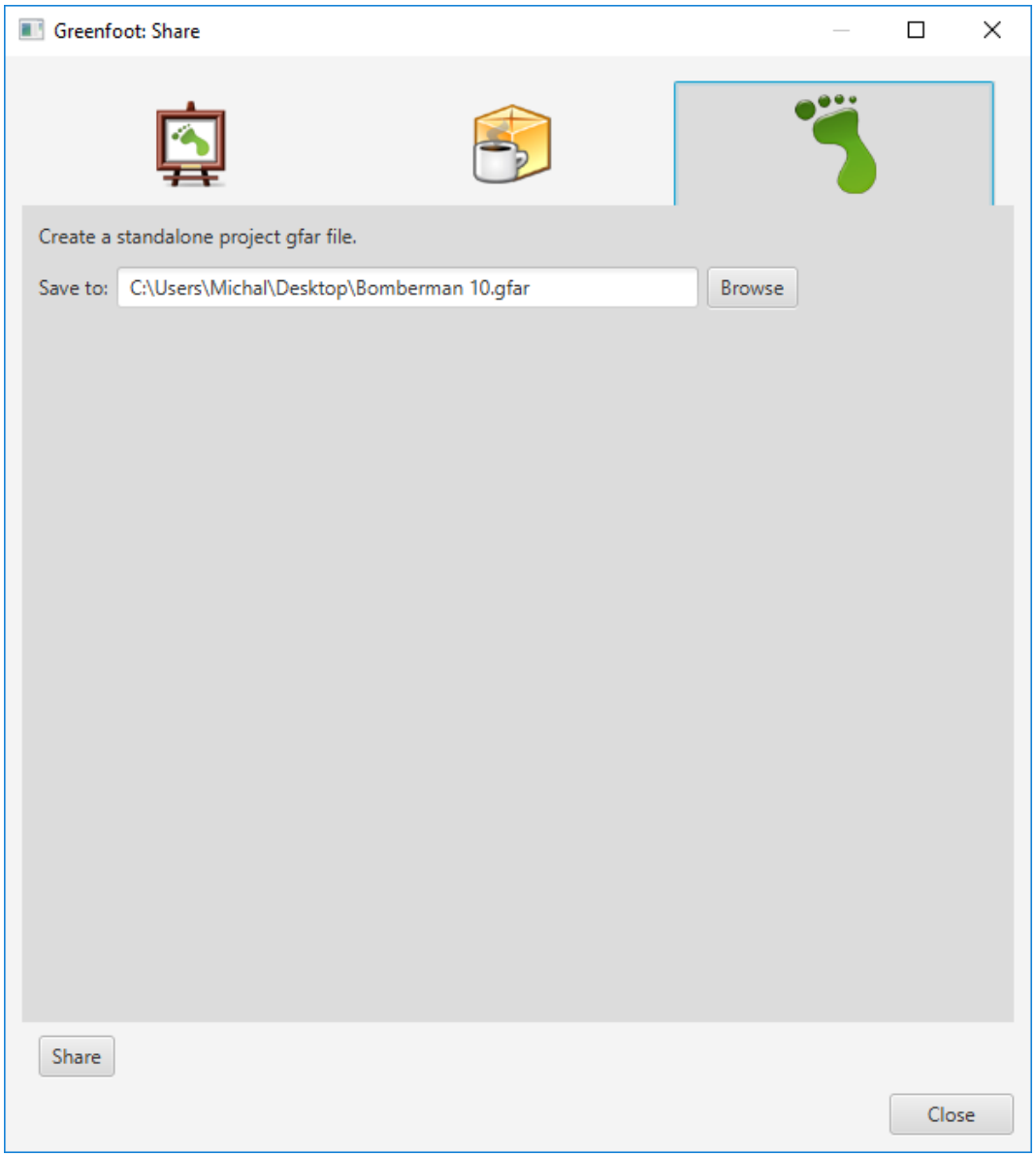
Obrázok 16: Spustená aplikácia Bomberman

VYTVORENIE PROJEKTOVÉHO SÚBORU .GFAR

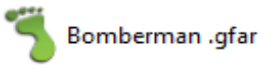
Súbory s príponou **`.gfar`** (**G**reen**F**oot **AR**chive) nie sú samostatne spustiteľné súbory (tak, ako napr. súbory s príponou **`.jar`** alebo **`.exe`**), obsahujú však všetky potrebné zdrojové súbory, obrázky a zvuky potrebné pre otvorenie projektu v prostredí Greenfoot. Tieto súbory sú teda vhodné na to, ak chceme zdieľať projekty, pretože nie je nutné ručne otvárať projektovú zložku a postupne vyberať potrebné súbory.

Je dobré podotknúť, že sa stále jedná o archív a je teda možné prezerať a modifikovať jeho obsah s využitím vhodného softvéru (napríklad 7zip).

Pre vytvorenie archívu **`.gfar`** postupujeme takmer rovnako, ako v prípade vytvorenia **`.jar`** súboru. Z dialógu **`Share`** vyberieme tretiu možnosť – **`Create a standalone project gfar file`** (pozri obrázok 14). V dialógu potom vyplníme celú cestu (**`Save to`**), ktorú v prípade potreby môžeme modifikovať pomocou tlačidla **`Browse`**. Výsledný **`.gfar`** súbor (v systéme Windows zobrazený na obrázku 18) vytvoríte tlačidlom **`Share`**.



Obrázok 51: Dialógové okno pre vytvorenie projektového súboru `.gfar`



Obrázok 18: Vytvorený `.gfar` súbor pre hru Bomberman v prostredí Windows

Výsledný projektový súbor môžeme otvoriť dvomi spôsobmi:

1) Po dvojkliku myšou naň sa otvorí prostredie Greenfoot s daným projektom.
2) Otvoríme prostredie Greenfoot a z hlavnej ponuky vyberieme možnosť **`Scenario`** a následne **`Open GFAR...`** . V dialógu vyberieme a potvrdíme požadovaný súbor s projektom, ktorý sa následne otvorí v prostredí Greenfoot.