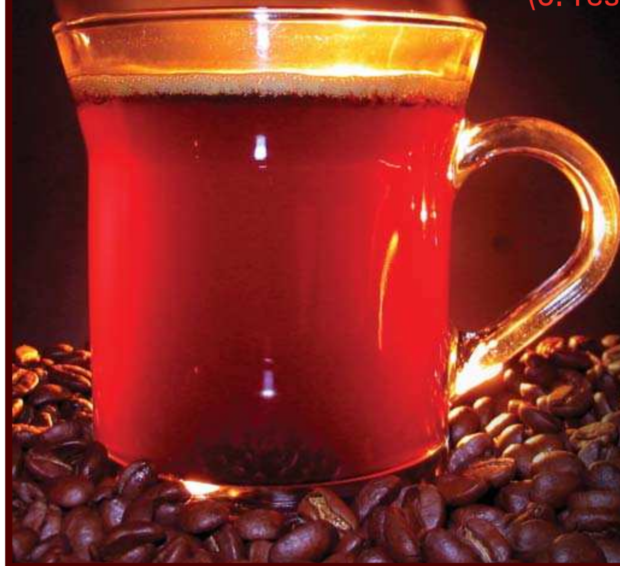
Kávéfőzés lépésről lépésre

(6. rész)



Négy labda, két egérgomb, és egyetlen cél – minél több pontot gyűjteni. Útmutató a világ legidegesítőbb játékának létrehozásához.

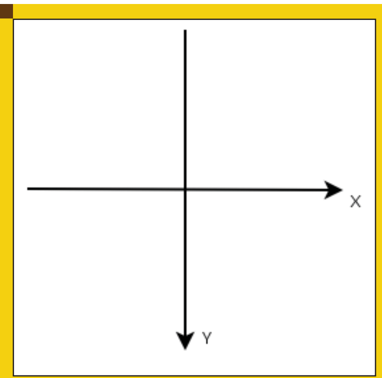
A múlt hónapban lerántottuk a leplet a grafikus felületek létrehozásának mikéntjéről. Szó esett arról, hogyan hozhatunk létre egy ablakot, és miként tölthetjük fel vezérlőkkel. A kedves Olvasónak lehetősége nyílt továbbá arra, hogy megismerje az eseménykezelés módszerét. Mindez egy bájosan trükkös alkalmazás példáján keresztül került bemutatásra, amely a grafikus kezelőfelület ellenére gonosz módon épp a felhasználóbarátság tökéletes ellenpéldájának bizonyult.
Be kell, hogy ismerjem, annak ellenére, hogy a ***GUI*** programozást a felhasználó kényeztetése címén harangoztam be, a most bemutatásra kerülő alkalmazás célja úgyszintén az idegek borzolása lesz. Egy valódi játékot fogunk írni, mely stílusát tekintve egészen a gyökerekig fog visszanyúlni. Azokba az időkbe, amikor egy őrületbe kergető egyszólamú muzsika mellett kellett elhasználni három botkormányt, különben a nagy piros kör megette a kis kék négyzetet.
Nem csalás, és nem is ámítás, valódi játékot készítünk. Ez első hallásra nem tűnhet akkora kihívásnak, viszont gondoljunk bele, mennyi feladatot foglal magába egy ilyen alkalmazás írása. Szükség van animációra, aminek objektumközpontú környezetben egyenes következménye a szálkezelés. Bizonyos fokú fizikát is meg kell valósítanunk annak érdekében, hogy a piros kör élethűen fogyassza el a kék négyzetet. Végül ne feledkezzünk meg a beviteli eszközök kezeléséről sem, ami itt eseménykezelést jelent. Ezek már önmagukban elgondolkodtató problémákat vetnek fel akkor is, ha a játék szabályait még nem is részleteztük. Mi több, nem szóltunk olyan további lehetőségekről, amelyek egyfajta játékélmény elengedhetetlen forrásait jelentik. Ide sorolhatjuk a zenét, a hálózaton keresztüli játékot, vagy a mesterséges intelligenciát. Ezekre első játékunk írása során nem térünk ki, csupán érzékeltetni szeretném azt, hogy játékprogramot írni már csak azért is komoly kihívás, mert szinte elképzelhetetlen olyan terület, amit ne érintene.
Sok munka áll előttünk, és már most szólok, hogy bármennyi szabadideje és türelme van a kedves Olvasónak, biztos, hogy mindre szüksége lesz. Természetesen megteheti, hogy a gondolkodás terhétől megszabadulva vadul begépeli az itt látható kódokat, és azonnal beleveti magát a játékba. Viszont még ha így is tesz, kérem, hogy ha valamilyen szerencsés véletlen folytán alábbhagyna a játékszenvedélye, tegyen egy kísérletet a program működésének elemzésére. Nem is szaporítom tovább a szót, vágjunk bele!

Mire lesz szükségünk?

A játék igen egyszerű. Egy ketrecben négy pattogó labda van, két piros és két fekete. A játékos úgy szerez pontot, ha sikeresen rákattint valamelyikre. Minden labda egyenlő sebességgel mozog, viszont minél több pontot szerez a játékos, annál gyorsabbak lesznek a labdák. A piros labdákra csak a bal, míg a fekete labdákra csak a jobb egérgombbal történő kattintással lehet pontot elérni. A labdák csak a fallal ütköznek, egymással nem, így ha két azonos színű fedésbe kerül, és sikeres a kattintás, dupla pont jár.
Az alkalmazás 6 osztályból építkezik, ezeket fogjuk egyesével elkészíteni. Minden azonosítónál, legyen az akár osztály-, vagy változónév, az eddig megszokottaknak ellent mondva angol neveket használtam. Azért döntöttem így, mert bizonyos elemek esetén sokkal kényelmesebb a szakirodalomban is elterjedt jelöléseket alkalmazni. Jó példa erre egy személyes tagváltozó beállító, illetve lekérdező függvénye, amelynél a szokásos `set`, illetve `get` nevektől csak körülményes szóhasználattal lehetne eltérni.
Elsődleges célom itt is a következetesség megtartása volt, ezért nem alkalmaztam az egyes körökben elterjedt kevert megoldást, például `getSzin`, `setSzin`. A megjegyzések továbbra is magyar nyelvűek, és a leírás remélhe-

© Kiskapu Kft. Minden jog fenntartva

© Kiskapu Kft. Minden jog fenntartva



1. ábra A koordinátarendszer

tőleg kellő magyarázattal szolgál azoknak is, akik még kevés jártassággal rendelkeznek az angol nyelvű irodalomban.
Lássuk, melyek azok az osztályok, amiket meg kell írnunk:

- `Ball` (Labda)
- `Cage` (Ketrec)
- `Game` (Játék)
- `Main` (Fő)
- `Score` (Eredmény)
- `Speed` (Sebesség)

A `Main` jelenti az alkalmazás belépési pontját. Ez a legegyszerűbb osztályunk, hiszen egyetlen feladata, hogy létrehozzon egy `Game` objektumot, és elindítsa a játékot. A `Game` építi fel az alkalmazás ablakát, példányosítja a többi osztályt, és lekezeli a felhasználói beavatkozásokat. A `Speed` egy sebességvektort nyújt a `Ball` számára, ami a játék egyik legfontosabb szereplője. A `Cage` a játéktér határait biztosítja. A `Score` pedig a játékos pontszámait tartja nyilván.

Speed

Kezdjük egy kis fizikával. Mint az a játék leírásából kitűnik, a labdák egyenlő sebességgel mozognak.
Ez nagy könnyebbséget jelent. Ennek következtében ugyanis élhetünk egy apróbb egyszerűsítéssel, és ezzel komoly munkát takaríthatunk meg.
A `Speed` szigorú fizikai megközelítésben nem sebességvektort ad, hanem csak irányvektort. A labdák valódi sebességét nem a modell, hanem a megjelenítés szintjén kezeljük.
Ez egy takaros játékprogramban szentségtörésnek minősülne. Már a cikk írásának pillanatában látom az ablakom alatt a dühödt tömeget, és mintha már az akasztófát is elkezdték volna építeni. Viszont ne felejtsük el, hogy legelső játékunkban nem a tökéletesség a cél. Jogos kérdés, hogy az osztály az említettek ellenére miért viseli a `Speed` nevet. Azt szeretném, ha ebből a példaprogramból az Olvasóban egy általános kép maradna meg a kisebb játékok működéséről, és nem konkrét megvalósítási kérdések bosszantanák. Írjunk be magunknak egy rossz pontot ezért a csalásért most, és a következő programban küszöböljük ki.
Térjünk vissza a labdák mozgásához. Az osztályban külön tároljuk az X, illetve Y irányú összetevőt. Mi több, a későbbi számolások megkönnyítése érdekében az összetevőknek a nagyságát, azaz abszolút értékét, illetve az irányát, vagyis az előjelét is külön változókban tároljuk. Ez négy tagváltozót jelent, nevezetesen: `abs_dx`, `dir_dx`, `abs_dy`, `dir_dy`.
A következő példához tekintsük az *1. ábrán* látható koordinátarendszert. Elsőre furcsának tűnhet az Y tengely fordított helyzete, azaz, hogy a Y tengely negatív tartománya az X tengely felett, pozitív tartománya pedig az X tengely alatt helyezkedik el. Számítógépes grafikáknál legtöbbször kényelmes ilyen rendszerben számolni, mivel az esetek nagy részében a megjelenítés ezt várja el tőlünk. Ez a Java esetében sincs másként.
Ebben a rendszerben adott egy sebességvektor: (-3,2). Ez azt jelenti, hogy időegység alatt az objektum 3 egységnyi utat tesz meg balra, és 2 egységnyit lefelé. A `Speed` osztályban bevezetett változókkal ez így írható fel:

```
abs_dx = 3
dir_dx = -1
abs_dy = 2
dir_dy = 1
```

Nevezzük mozdulatnak az időegység alatt történő teljes elmozdulást.
A mozdulatot több körben tesszük meg. Egy körben 1-1 egységnyi lépést teszünk azon összetevők által meghatározott irányokban, amelyek abszolút értéke által előírt mennyiséget még nem értük el korábbi körökben. Visszatérve a példára, a vektor alatt értendő mozdulatot 3 körben tesszük meg:

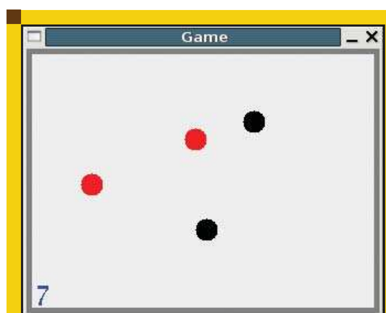
- 1-et lépünk balra, 1-et le
- 1-et lépünk balra, 1-et le
- 1-et lépünk balra

Ezek után újabb mozdulat következhet. Az adott irányba már megtett lépések nyilvántartására bevezetünk két újabb változót: `stepX`, `stepY`.
Ezek kezelésére később visszatérünk.
Az idáig vázolt modell nem sérti a sebességvektor definícióját. A csalást ott fogjuk elkövetni, hogy a különböző objektumoknak egyenlő hosszú köröket fogunk osztani. Így a teljes mozdulat nem egységes idő alatt történik. Emiatt függetlenül a sebességvektor abszolútértékétől, ugyanolyan gyors objektumokhoz jutunk. Például egy (1,1) és egy (2,2) vektor ebben a programban (sajnos) egyenlő.
Lássuk az osztály forráskódját (*1. kód*)!
A tárgyalt tagváltozók felsorolása után a konstruktort láthatjuk. Ez a paraméterként kapott vektor alapján tölti fel értékekkel a létrejövő objektum tulajdonságait. Mindkét összetevő esetén először az előjelet dönti el, majd az ezzel szorzott összetevőt tekinti abszolút értéknek. Így, ha negatív volt, -1-el szoroz, ha pozitív, 1-el, tehát helyes a leképezés.
A `stepX` és `stepY` változók nulla kezdőértéket kapnak.
Azok az objektumok, amelyeket mozgatni szeretnénk a programban, rendelkeznek majd egy `Speed` típusú tulajdonsággal. Mozgatásuk úgy fog történni, hogy előbb lekérdezik, hogy az adott körben az egyes tengelyek mentén lépniük kell-e, majd a lépést követően ezt jelentik.
A `step` metódusból látszik a `stepX` és `stepY` értelme. Ennek a függvénynek a meghívásával jelentheti egy objektum, hogy megtette a szükséges lépéseket. Ha mindkét változó túlszaladt, akkor lenullázza őket, egyébként mindkettőt növeli 1-el. Itt egy újabb csalást érhetünk tetten. A metódus mindkét változót növeli, függetlenül attól, hogy az elérte-e már a hozzá tartozó összetevő abszolút értékét, vagy sem. Ennek a látszólagos hanyagságnak az okára és árára hamarosan rátérünk.
Két lekérdező metódus következik, melyekkel a mozgó objektumok megtudhatják, hogy kell-e lépniük egy adott tengely mentén. Ezek csak akkor

adják vissza a megfelelő összetevőhöz tartozó irányt, ha a vonatkozó `step*` változó nem szaladt túl. Egyébként nullát adnak vissza. Ez a feltétel vizsgálat javítja az előző metódus hanyagságát. Ez időben nem jelent veszteséget, viszont rövidebb és áttekinthetőbb kódot biztosít.
A további négy metódussal az irányváltás oldható meg. Ezek egészen egyszerűen az irányokat jelentő tagváltozók átírását teszik lehetővé.

Ball

Következzen főhősünk, a labda. Egy labdát középpontjának X és Y koordinátája, sugara, színe és sebessége határoz meg. Ennek alapján tekintsük a forráskódot. (2. *kód*)
A tagváltozók nevei kellően beszédesek ahhoz, hogy eltekintsünk a tárgyalásuktól. A konstruktor ezeknek ad kezdőértéket. Látható, hogy létrehozunk egy `Speed` objektumot a sebességvektor jellemzéséhez.
A többi tulajdonság lekérdezhető a megfelelő `get` metódus meghívásával. Ne felejtsük el, hogy egy objektumközpontú környezetben érdemes minden tagváltozót személyesre állítani, és csak azokhoz biztosítani lekérdező, illetve beállító metódust, amelyekhez ez valóban szükséges.
A `move` metódus végzi a labda mozgatását. Röviden összefoglalva az eddigieket, ebben a modellben a labdák mozgatják saját magukat. Viszont a jelenlegi pozíciójukon kívül mást nem tudnak, ezért segítséget vesznek igénybe egy `speed` objektumtól.
Amikor valaki mozgásra kényszerít egy labdát, azaz meghívja a `move` metódusát, az módosítja saját pozícióját úgy, ahogy a `speed.getX()` és a `speed.getY()` előírja. Ezután becsületesen jelenti a `speed` objektumnak, hogy a mozgás megtörtént. Így a `speed` nyilván tudja tartani, hogy a labda hányadik körnél tart, és az egyes tengelyek mentén kell-e még lépni.
A `move` ezért először hozzáadja a `speed` által szolgáltatott lépéseket saját koordinátáihoz. Ezután jelent, majd ellenőrzi, hogy történt-e ütközés a ketrec valamelyik falával. Az ellenőrzésnél figyelembe veszi saját középpontját, és sugarát. Ütközés esetén elfordul.



2. ábra A játék

Az utolsó metódus kissé szokatlan lehet, mert egy `Graphics` objektumot kap paraméterül. Ez nem lesz más, mint annak a rajzfelületnek a referenciája, amelyre ki kell rajzolni a labdát. A `paint` feladata tehát megjeleníteni a labdát a kapott felületen. Ezúton újabb megrovásban részesítem magam, mert egy komoly alkalmazásban külön kell választani a vezérlést a megjelenítéstől. Ezt legtöbbször úgy szokták elérni, hogy a megjelenítést végző kód külön osztályban kap helyet.
A működéséhez szükséges információ például egyszerű öröklődéssel nyerhető. Ebben az esetben viszont egyetlen kétsoros metódus miatt nem tettem meg külön osztálynak a labda grafikus változatát.
A ***Java API***-ban a `Graphics` osztály tanulmányozásával láthatjuk, hogy milyen rajzolási műveletek állnak rendelkezésünkre egy ilyen objektum esetében. A mi `paint` metódusunkban előbb a `setColor` segítségével beállítjuk a rajzolás színét. Ezután egy kitöltött ellipszist rajzolunk. A `fillOval` metódusnak kissé furcsa a paraméterezése.
Azt a téglalapot kell leírnunk, amely magába foglalja a kívánt ellipszist. Meg kell adnunk a téglalap bal felső sarkának X és Y koordinátáját, a szélességét és a magasságát.
Ez egy kör esetében nyilvánvalóan egy négyzet lesz, a paraméterek pedig a fent látható módon adódnak.

Cage

Lássuk most a játékteret határoló ketrec felépítését. Ezt falának vastagsága, színe, valamint szélessége és magassága jellemzi. (3. *kód*)
Az említett tulajdonságok leírását követően a konstruktor látható.
Ez ismét tartogat egy meglepetést, jelen esetben a `Dimension` osztály személyében. Ezzel az egyszerű könyvtári osztállyal egy 2 dimenziós méret fejezhető ki. Ne törődjünk most azzal, honnan fogjuk ezt megkapni a `Cage` példányosításához, pusztán tegyük fel, hogy egy ilyen objektum referenciájával hívjuk meg a konstruktort.
Először a `thick` és `color` tagváltozóknak adunk kezdőértéket. Ezután a `Dimension` típusú objektumnak a megfelelő metódusaival lekérdezzük a szélességet és magasságot, és ezeket átadjuk a `width` és `height` változóknak. A típuskényszerítés azért kötelező, mert ezek a metódusok dupla lebegőpontos értéket adnak vissza, nekünk pedig egész számokra van szükségünk.
A konstruktort az osztály `paint` metódusa követi. Ketrecet legegyszerűbben úgy tudunk rajzolni, hogy előbb a teljes rajzfelületet kitöltjük egy téglalappal, majd ebből kivágunk egy ablakot. A `fillRect` a `setColor` által meghatározott előtérszínnel kitöltött téglalapot rajzol, a `clearRect` pedig a háttérszínt használja. Mindkét metódusnak a bal felső sarok X és Y koordinátáját, valamint a téglalap szélességét és magasságát kell átadni.
Végül négy lekérdező függvény látható, melyekkel a falak belső oldalának megfelelő koordinátái kérdezhetők le. Nyilvánvalóan minden oldal esetében csak egy koordináta értelmes, hiszen például a plafonnak nincs X pozíciója.

Score

Következő osztályunk a `Score`, amely a játékos pontszámának tárolásáért és megjelenítéséért felel. Tulajdonságai a pontszám, a megjelenítéshez használt szín, betűtípus, és a helyének X és Y koordinátája. (4. *kód*)
A tagváltozók felsorolását megszokott módon a konstruktor követi. Ez paraméterként egy X és Y pozíciót, valamint egy színt kap. Ezeket kezdőértékként átadja a megfelelő változóknak, a pontszámot pedig 0-ra állítja. A betűtípus meghatározásához készítünk egy `Font` objektumot. Ennek létrehozásakor megadjuk a betűtípus nevét, stílusát és méretét. A név jelen esetben nem konkrét betűtípusra vonatkozik, hanem egy családot határoz meg, melynek minden virtuális gépet

© Kiskapu Kft. Minden jog fenntartva

© Kiskapu Kft. Minden jog fenntartva

futtató rendszeren biztosan található példánya. A stílus félkövér, a méret pedig 24 pont.
Az `increment` metódus a pontszámot növeli eggyel, a `getPoints` pedig visszaadja ezt. A `paint` előbb beállítja a színt, majd a betűtípust, végül meghívja a `Graphics drawString` tagfüggvényét, mellyel egy szövegfüzért lehet kirajzolni. Miután ez `String` típusú objektumot vár, az `Integer` osztály segítségével előbb át kell alakítanunk az `int` típusú pontszámot szövegfüzérré.

Game

Eddig a nagy képnek csak különálló részletein dolgoztunk. Hogy hogyan lesz ebből egy nagy műalkotás, már biztosan foglalkoztatja az Olvasót. Elérkezett az idő, hogy elkezdjük használni eddigi munkánk eredményeit, és lássuk, ahogy a dolgok összeállnak. Következzen a `Game.java` (*5. kód*).
A sorozat előző részében létrehozott grafikus felülethez használt osztályok közül több visszaköszön itt is. Viszont első ránézésre sok az újdonság. Az elemzést kezdjük a konstruktor vizsgálatával. Az első sorban az ismert módon létrehozunk egy ablakot, ***„Game”*** címmel. A második sorban állítjuk be az ablak elrendezéskezelőjét. A `BorderLayout` egy olyan elrendezéskezelő, ami öt részre vágja az ablakot. Van egy nagy középső rész, és négy kisebb a négy égtájnak megfelelően. Mi csak a középső részt használjuk, így az elhelyezésre kerülő egyetlen vezérlő kitölti az ablakot.
Ez a vezérlő a `Canvas` lesz, ami nem több, mint egy rajzvászon. Ezzel nem becsmérelni akartam a képességeit, csupán azt akartam jelezni, hogy egy nagyon alacsony szintű ***AWT*** elemről van szó. Gyakori, hogy nem is használják közvetlenül, hanem egy saját osztály terjeszti ki. Viszont arra a feladatra, amire nekünk kell, elegendő lesz önmagában is. A `Canvas` példányosítása után beállítjuk a méretét. Itt látható, hogy egy új `Dimension` objektumot adunk át a `setSize` metódusnak, így határozzuk meg a vászon szélességét és magasságát.
A következő sorban az ablak tartalmához hozzáadjuk a vásznat, és jelezzük, hogy középen szeretnénk elhelyezni. Ezután életre keltjük az elrendezéskezelőt, beállítjuk, hogy az ablakot ne lehessen átméretezni, továbbá, hogy az ablakkezelőn keresztül kezdeményezett bezárás hatására lépjen ki a program.
A következő sorban meghatározzuk, hogy dupla-pufferelt üzemmódban szeretnénk használni a rajzvásznat. A dupla-pufferelés azt jelenti, hogy két lapunk van. Egy amit a felhasználó lát, és egy, amin mi dolgozhatunk. A kettőt egy művelettel kicserélhetjük. Így a sok időt emésztő rajzolást a háttérben végezhetjük, és csak akkor mutatjuk meg a felhasználónak a képet, ha az elkészült. Ha nem használnánk ezt a módszert, az animációnk biztosan villogna, ami nem túl szép.
A pufferkezelő stratégia objektumával tudjuk a háttérlaphoz tartozó `Graphics` objektumot lekérdezni, illetve a cserét végrehajtani. Ezért ennek a referenciáját a következő sorban lekérdezzük, és eltároljuk egy személyes tagváltozóban. Fontos, hogy a dupla-puffer létrehozása, és a stratégia lekérdezése csak azután történhet, hogy a vásznat hozzárendeltük egy ablakhoz, és az elrendezéskezelő is befejezte tevékenységét, és így a vászonnak biztosan van valódi mérete. Ezen megfontolások miatt az utóbbi két sor nem előzheti meg a `frame.pack()` hívást!
Az ezt követő részben hozzuk létre azoknak a hozzávalóknak az objektumait, amiken idáig dolgoztunk. Már itt érezhetjük, milyen boldogító érzés saját osztályainkat felhasználni. Egyetlen jól irányzott sorral 4 helyett 5 labdánk lehetne a játékban, mi több, semmi sem állíthat meg bennünket még több és több labda létrehozásában.
A `gameSpeed` tagváltozó a labdák sebességében fog szerepet játszani.
A konstruktor utolsó két sorában az egérrel kapcsolatos események lekezelését bízzuk a létrejövő `Game` objektumra, és láthatóvá tesszük az ablakot. Ezek alapján látható az osztály egyes tagváltozóinak szerepe. Egyedül a `gameSpeed` tulajdonság nem tisztázott, ám erre is hamarosan fény derül.
„Jó, de hogyan mozog a labda?” – kérdezheti teljes joggal az Olvasó. Végtére is minden adott, már csak az isteni szikra hiányzik a gépezet beindításához. A labdák mozgása lényegében egy végtelen

ciklus eredménye. Ez a ciklus nem csinál mást, csak frissíti a labdák helyzetét, azaz megmozdítja őket, majd újrarajzolja a teljes vásznat. Majd megint mozdít, újrarajzol. Egy ilyen végtelen ciklust viszont nem tehetünk közvetlenül a programunkba.
Ez a lépés azt eredményezné, hogy teljesen elveszítjük a vezérlést a programunk felett. Egy alkalmazás ugyanis számtalan eseményre kell, hogy reagáljon. Ezek egy részét tudjuk csak programozni ***Java***-ban, például az egérkezelést. A többség jelzések kezeléséből áll, amelyeket ilyen magas szinten nem is tudunk befolyásolni. Egy végtelen ciklus megölné a programot, nem tudna ellátni számos adminisztratív teendőt.
A megoldás egy új szál bevezetése a programba. A szálat szokás könnyűsúlyú folyamatnak is hívni, mert igen hasonlóan kezeli egy program a szálait ahhoz, ahogy az operációs rendszer kezeli a folyamatokat. Ezeket mellesleg az előbbivel szemben szokás nehézsúlyú folyamatoknak is nevezni. Ha tehát egy önálló szálra bíznánk ezt

a végtelen ciklust, azzal megoldanánk a problémát. Az egyetlen kérdés ezek után, hogy mennyire nehéz feladat a szálkezelés *Java*-ban.
A válasz természetesen az, hogy mint minden, ez is roppant egyszerű. Minden olyan objektum lehet új szál egy programon belül, ami egy `Runnable` interfészt megvalósító osztály példánya. Az interfész megvalósításán túl lehetőség van a `Thread` osztály kiterjesztésére is. Ennek részleteiért lásd a ***Java API***-ban a `Thread` leírását.
A `Runnable` interfész a `run` metódust megvalósítását írja elő. Ennek a metódusnak a tartalma jelenti a szál kódját.
A `Game` osztályban ez egy azonnal szembetűnő végtelen ciklus, pont az, amire szükségünk van. A ciklus hasa három lépést tartalmaz. Először annyi `update` hívást végez, amennyi a `gameSpeed`. Másodszor meghívja a `repaint` függvényt. Végül elalszik 10 millimásodpercre.
Az első lépésben érvényesítjük azt a csalást, amiről a sebességvektor tárgyalásánál ejtettem szót. Úgy gyorsítjuk a labdákat, hogy kevesebbszer rajzolunk, más szóval, több számolásra jut csak egy rajzolás. A `repaint` utáni alvás azért kell, mert még szálkezeléssel együtt is 100%-os processzorkihasználtságot eredményezne a végtelen ciklusunk, ha nm függesztené fel bizonyos időközönként a futását. A `try - catch` pedig azért kell, mert a `Thread.sleep`, ami a jelenlegi szálat altatja el, `InterruptedException` kivételt dob. Ezt most üres `catch` ággal kezeljük, de ez normális esetben nem gond.
A kellően kielemzett `run` után következik annak két segítő függvénye, az `update` és a `repaint`. Az `update` feladata a labdák léptetése. Egy ciklusban végigmegy a labdák tömbjének elemein, és egyesével meghívja a `move` metódust. Ennek átadja a ketrec referenciáját, így a labda le tudja kérdezni a ketrectől, hogy hol vannak a szélei, és így képes ellenőrizni az ütközéseket.
A `repaint` újrarajzolja a teljes pályát. Ehhez előbb lekérdezi a vászon pufferkezelő stratégiájától a háttérlap `Graphics` objektumát. Ezután ezt átadja a ketrec rajzoló metódusának, majd az összes labda rajzoló metódusának, végül a pontszám rajzoló metódusának. Végezetül kicseréli a két puffert, és így előtérbe hozza azt a lapot, amire idáig a háttérben rajzoltunk.

Az osztály utolsó metódusa az egérkattintás eseményét kezelő `mouseClicked`. Ez két részből áll. Egyrészt ellenőrzi, hogy megfelelő kattintás történt-e labdán, majd a játékos pontszáma alapján állítja a játék sebességét.
Az első részben egy ciklussal végigmegy az összes labdán. Mindegyiknek lekérdezi a középpontját és a sugarát, majd a kör egyenletét felhasználva kiszámolja, hogy a kattintás a kör területén belül történt-e. Ez esetben még azt is megnézi, hogy a megfelelő gombbal kattintott-e a játékos, és ha minden rendben, odaítél egy pontot.
A második részben a pontszám növekedésével arányban gyorsítja a játékot.

Main

Végül, de nem utolsó sorban futtathatóvá tesszük fáradozásaink eredményét egy burkoló osztály segítségével. Íme:

```
/**
* Az osztaly az alkalmazast
* inditja.
*/
public class Main {

   /**
   * A jatek.
   */
   private Game game;

   /**
   * Letrehozza a jatek
   * objektumat
   * es elinditja a szalat.
   */
   public Main() {
      game = new Game();
      Thread t = new
      ➥Thread(game);
      t.start();
   }

   /**
   * A jatek inditasa.
   */
   public static void
   ➥main(String[] args) {
      new Main();
   }

}
```

A belépési pontot jelentő `main` tagfüggvény egy `Main` típusú objektumot hoz létre. Ennek a konstruktora így létrehoz egy `Game` típusú objektumot. Ezután a `game` referenciával létrehoz egy új szálat. A `Thread` ezen konstruktora egy `Runnable` interfészt megvalósító objektumot vár. Végül elindítja a szálat a `start` metódussal.

Hogyan terjesszük?

Ha a képen látható játékélményt mással is szeretnénk megosztani, kényelmetlen lehet a 6 `.class` fájlt kezelni. Szerencsére a `jar` parancs segítségével saját csomagot hozhatunk létre. Így elég egyetlen állománnyal bíbelődnie annak, aki a programot futtatni szeretné. Ám ehhez nem elég csupán a `.class` fájlokat összecsomagolni, azt is meg kell mondani, hogy melyik osztály tartalmazza a belépési pontot. Ezt egy úgynevezett `manifest` állománnyal írhatjuk le. Készítsünk egy `Manifest.txt` fájlt az alábbi tartalommal:

```
Main-Class: Main
```

Itt a kettőspont után álló `Main` vonatkozik arra, hogy a `Main` osztály tartalmazza a belépési pontot. Ezután adjuk ki a következő parancsot abból a könyvtárból, ahova idáig dolgoztunk:

```
$ jar cmf Manifest.txt
Balls.jar *.class
```

Ezzel el is készítettük a csomagot. Elég a `Balls.jar` fájlt elküldeni ismerőseinknek. A programot ezek után az alábbi paranccsal lehet elindítani:

```
$ java -jar Balls.jar
```

Jó játékot!

Fülöp Balázs
(bigwig42@gmail.com)
21 éves, imádja a Túró Rudit, a Debian Linuxot és a teheneket. Kedvenc írója Slawomir Mrožek. Leginkább a számítógépes hálózatok biztonsága érdekli. A BME VIK műszaki informatikus szak hallgatója.

KAPCSOLÓDÓ KÓDOK

A kódok a Linuxvilág honlapjáról tölthetők le.

© Kiskapu Kft. Minden jog fenntartva