

# Reflexivita

(Java Reflection Model)

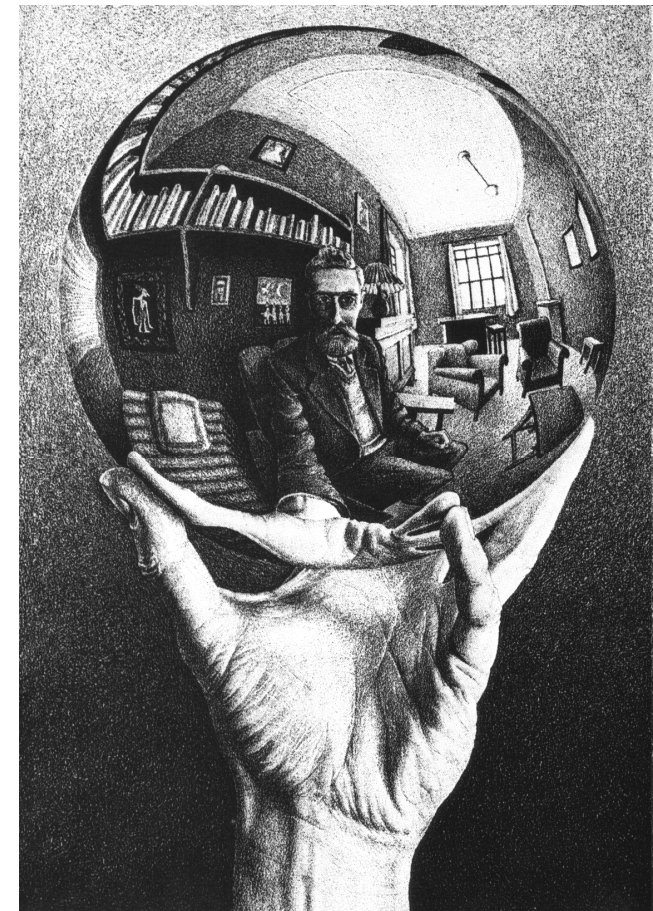
- možnosť čítať, vykonávať, resp. modifikovať program, ktorý sa práve vykonáva
- je to vlastnosť, ktorá sa vyskytuje v interpretovaných jazykoch, napr. `exec` a `eval` v Pythone, nie v kompilovaných (v skutočných) jazykoch ako C, C++)

Prečo ??

- JAVA je niekde medzi, lebo sa kompiluje do byte kódu, ktorý je ale interpretovaný

JAVA poskytuje

- Introspection: triedy `Class` a `Field` pre čítanie vlastného programu
- Reflexívne volanie: triedy `Method`, `Constructor`



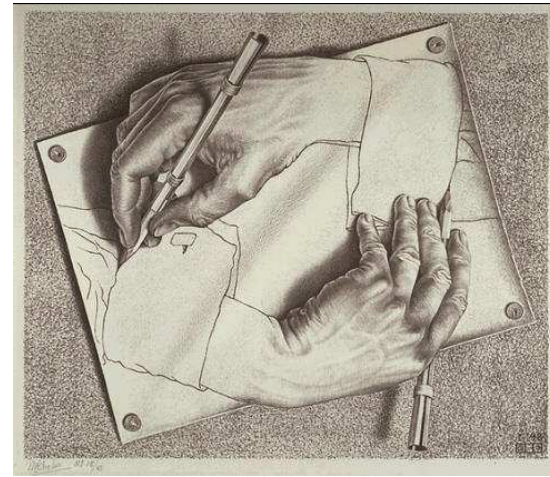
# Nadtrieda a Podtrieda

(ilustračný príklad)

```
public class Nadtrieda implements Runnable {  
  
    public int variabla;  
    public int[] pole = {1,2,3};  
    public String[] poleStr = {"janko", "marienka" };  
    public Nadtrieda() { }  
    public Nadtrieda(int a) { }  
  
    public void Too(double r) { }  
    public void run() { ... kvôli Runnable ... }  
}
```

```
public class Podtrieda extends Nadtrieda {  
    public Podtrieda(String s) { }  
  
    public class Vnorena { }  
    public interface Prazdny {}  
}
```

# Trieda Class<T>



Každý objekt pozná metódu getClass():

```
Class nt = new Nadtrieda().getClass();
```

**Class:**

- hodnotou sú reflexívne obrazy tried nášho programu,
- umožní nám čítať a spúšťať časti nášho programu,
- o.i. pozná metódu String getName()

```
System.out.println(nt.getName());
```

```
// Nadtrieda
```

```
Class nt1 = Nadtrieda.class;
```

```
System.out.println(nt1.getName());
```

```
// Trieda.class
```

```
// Nadtrieda
```

• meta-trieda:

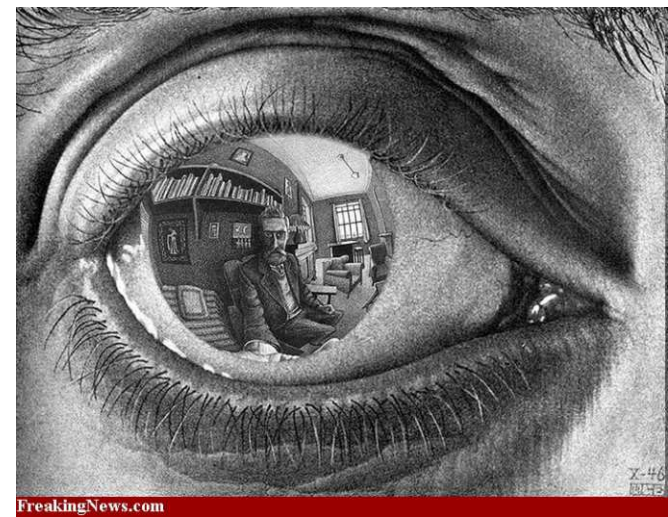
```
Class klas = Class.class;
```

```
Class klas1 = nt.getClass();
```

**Russellov paradox**  
(antinómia)

$$S = \{X | X \notin X\}$$

# Trieda Class<T>



```
try {
```

```
    Class pt = Class.forName("Podtrieda");
```

```
    System.out.println(pt.getName());
```

```
    Class nt2 = pt.getSuperclass();
```

```
    System.out.println(nt2.getName());
```

```
    for(Class cl:pt.getClasses())
```

```
        System.out.print(cl.getName());
```

```
} catch (ClassNotFoundException e) {
```

```
    e.printStackTrace();
```

```
}
```

```
// forName("...")
```

```
// Podtrieda
```

```
// getSuperClass()
```

```
// Nadtrieda
```

```
// getClasses()
```

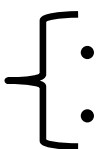
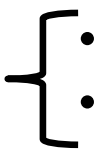
```
// public classes & interf
```

```
// Podtrieda$Prazdny
```

```
// Podtrieda$Vnorena
```

```
public class Podtrieda extends Nadtrieda {  
    public Podtrieda(String s) { }  
  
    public class Vnorena { }  
    public interface Prazdny {}  
}
```

# Metódy Class<T>

- T cast(Object obj)      pretypuje obj do triedy T
- static Class<?> **forName**(String name)      vráti Class objekt zodpovedajúci triede s menom name
- Class[] getClasses()      public triedy a interface implementované touto triedou
- 
  - Constructor[] getConstructors()      všetky konštruktory triedy
  - Constructor<T> getConstructor(Class... parameterTypes)      konštruktor triedy pre parameterTypes
- 
  - Field[] getFields()      všetky položky (premenné) triedy
  - Field getField(String name)      položka s menom name
- 
  - Method[] getMethods()      všetky metódy triedy
  - Method getMethod(String name, Class... parameterTypes)
- int getModifiers()      atribúty triedy (public, abstract, ...)
- String getName()      meno triedy
- boolean isInstance(Object obj)      je inštanciou triedy ?
- boolean isArray()      je pole ?
- boolean isPrimitive()      je primitívny typ ? (int, double, boolean...)

# Class<T>

Trieda Class umožňuje prístup k **atribútom** triedy

```
int m = nt.getModifiers();  
if (Modifier.isPublic(m))  
    System.out.println("public");
```

podobne:

isPrivate(), isProtected(), isStatic, isFinal(), isAbstract(), isFinal(),  
isSynchronized(),

Trieda Class umožňuje prístup k **interface** triedy

```
Class[] theInterfaces = nt.getInterfaces();  
for (int i = 0; i < theInterfaces.length; i++) {  
    String interfaceName = theInterfaces[i].getName();  
    System.out.println(interfaceName);  
}
```

**java.lang.Runnable**

```
public class Nadtrieda implements Runnable {
```

# Premenné, konštruktory

```
Field[] publicFields = nt.getFields();
for (int i = 0; i < publicFields.length; i++) {
    String fieldName = publicFields[i].getName();
    Class typeClass = publicFields[i].getType();
    String fieldType = typeClass.getName();
    System.out.println("Name: " + fieldName + ", Type: " + fieldType);
}
```

Name: variabla, Type: int  
Name: pole, Type: [I  
Name: poleStr,  
Type: [Ljava.lang.String;

```
Class intArray = Class.forName("[I");
Class stringArray =
Class.forName("[Ljava.lang.String;");

Constructor[] theConstructors = nt.getConstructors();
for (int i = 0; i < theConstructors.length; i++) {
    System.out.print("( ");
    Class[] parameterTypes = theConstructors[i].getParameterTypes();
    for (int k = 0; k < parameterTypes.length; k++) {
        String parameterString = parameterTypes[k].getName();
        System.out.print(parameterString + " ");
    }
    System.out.println(")");
}
```

()  
( int )

```
public class Nadtrieda implements Runnable {
    public int variabla;
    public int[] pole = {1,2,3};
    public String[] poleStr = {"janko", "marienka" };
    public Nadtrieda() { }
    public Nadtrieda(int a) { }
```

# Premenné, konštruktory

```
for (Field f : nt.getFields() ) {  
    String fieldName = f.getName();  
    Class typeClass = f.getType();  
    String fieldType = typeClass.getName();  
    System.out.println("Name: " + fieldName + ", Type: " + fieldType);  
}
```

```
Name: variabla, Type: int  
Name: pole,      Type: [I  
Name: poleStr,  
Type: [Ljava.lang.String;
```

```
Class intArray = Class.forName("[I");  
Class stringArray = Class.forName("[Ljava.lang.String;");
```

```
for (Constructor c : nt.getConstructors()) {  
    System.out.print("( ");  
  
    for (Class parameterType : c.getParameterTypes() ) {  
        String parameterString = parameterType.getName();  
        System.out.print(parameterString + " ");  
    }  
    System.out.println(")");  
}
```

```
( )  
( int )
```

```
public class Nadtrieda implements Runnable {  
    public int variabla;  
    public int[] pole = {1,2,3};  
    public String[] poleStr = {"janko", "marienka" };  
    public Nadtrieda() { }  
    public Nadtrieda(int a) { }
```



# Metódy

```
Method[] theMethods = nt.getMethods();
for (int i = 0; i < theMethods.length; i++) {
    String methodString = theMethods[i].getName();
    System.out.println("Name: " + methodString);

    String returnString = theMethods[i].getReturnType().getName();
    System.out.println("  Return Type: " + returnString);

    Class[] parameterTypes = theMethods[i].getParameterTypes();
    System.out.print("  Parameter Types:");
    for (int k = 0; k < parameterTypes.length; k++) {
        String parameterString = parameterTypes[k].getName();
        System.out.print(" " + parameterString);
    }
    System.out.println();
}
```

```
Name: Too
  Return Type: void
  Parameter Types: double
Name: run
  Return Type: void
  Parameter Types:
... Metódy Object-u
```

# Je inštanciou

`cl.isInstance(obj)` je true, ak `obj` je inštanciou triedy reprezentovanej v `cl`.

```
Class nt = new Nadtrieda().getClass();
```

```
nt.isInstance(new Nadtrieda()) == true
```

`class1.isAssignableFrom(class2)` je true ak trieda reprezentovaná `class1` je nadtriedou/nadinterface triedy reprezentovanej `class2`, teda do premennej typu reprezentovaného `class1` môžeme priradiť objekt typu reprezentovaného `class2`.

Ergo:

```
cl.isAssignableFrom(obj.getClass()) == cl.isInstance(obj)
```

# Prístup k premennej

```
if (Integer.class.isAssignableFrom(Integer.class)) { // true
    Nadtrieda o = new Nadtrieda();
    Field f = o.getClass().getField("boxedInt");
    f.setAccessible(true);
    f.set(o, new Integer(88)); // o.boxedInt = 88;
    System.out.println(f.get(o)); // o.boxedInt;
}

if (int.class.isAssignableFrom(int.class)) { // true
    Nadtrieda o = new Nadtrieda();
    Field f = o.getClass().getField("variabla");
    f.setAccessible(true);
    f.set(o, new Integer(66)); // o.variabla = 66;
    alebo
    f.setInt(o, 77); // o.variabla = 77;
    System.out.println(f.get(o)); // o.variabla;
    System.out.println(f.getInt(o)); // o.variabla;
}
```

```
public class Nadtrieda implements Runnable {
    public int variabla;
    public Integer boxedInt;
    public Nadtrieda() { }
    public Nadtrieda(int a) { }
    public void Too(double r) { }
    public void run() { ... kvôli Runnable ... }
```

# Volanie konštruktora

```
try {  
    Nadtrieda nt2 = (Nadtrieda)(nt.getConstructor(int.class).newInstance(3));  
                                                // new Nadtrieda(3)  
} catch (InstantiationException e) {  
    e.printStackTrace();  
} catch (IllegalAccessException e) {  
    e.printStackTrace();  
} catch (IllegalArgumentException e) {  
    e.printStackTrace();  
} catch (InvocationTargetException e) {  
    e.printStackTrace();  
} catch (NoSuchMethodException e) {  
    e.printStackTrace();  
} catch (SecurityException e) {  
    e.printStackTrace();  
}
```

```
public class Nadtrieda implements Runnable {  
    public int variabla;  
    public Integer boxedInt;  
    public Nadtrieda() { }  
    public Nadtrieda(int a) { }  
    public void Too(double r) { }  
    public void run() { ... kvôli Runnable ... }
```

# Volanie konštruktora

V prípade konštruktora bez argumentov:

```
Class classDefinition = Class.forName(className);  
Object object = classDefinition.newInstance();
```

```
Class rectangleDefinition = Class.forName("java.awt.Rectangle");
```

```
// pole typov argumentov konštruktora, t.j. Class[]  
Class[] intArgsClass = new Class[] {int.class, int.class};
```

```
// daj mi konštruktor s daným typom argumentov  
Constructor intArgsConstructor =  
    rectangleDefinition.getConstructor(intArgsClass);
```

```
// pole hodnôt argumentov konštruktora, t.j. Object[]  
Object[] intArgs = new Object[] {new Integer(12), new Integer(34)};  
Rectangle rectangle =  
    (Rectangle) createObject(intArgsConstructor, intArgs);
```

# Volanie metódy

```
try {
```

```
(o.getClass()).getMethod("run").invoke(o);           // o.run();
```

```
Method met = (o.getClass()).getMethod("Too",new Class[]{double.class});  
met.invoke(o,new Object[]{new Double(Math.PI)});// o.Too(Math.PI);
```

```
(o.getClass()).getMethod("Too",double.class).invoke(o,Math.PI);  
                                           // o.Too(Math.PI);
```

```
} catch (SecurityException | NoSuchFieldException | IllegalAccessException |  
        IllegalArgumentException | InvocationTargetException |  
        NoSuchMethodException e) {  
    e.printStackTrace();  
}
```

```
public class Nadtrieda implements Runnable {  
    public int variabla;  
    public Integer boxedInt;  
    public Nadtrieda() { }  
    public Nadtrieda(int a) { }  
    public void Too(double r) { }  
    public void run() { ... kvôli Runnable ... }
```

# Volanie metódy

```
public static String append(String firstWord, String secondWord) {  
    String result = null;  
  
    try {  
  
        // pole typov argumentov metódy, t.j. Class[]  
        Class[] parameterTypes = new Class[] {String.class};  
        Class c = String.class;  
  
        // daj mi metódu s daným typom argumentov  
        Method concatMethod = c.getMethod("concat", parameterTypes);  
  
        // pole hodnôt argumentov metódy, t.j. Object[]  
        Object[] arguments = new Object[] {secondWord};  
        result = (String) concatMethod.invoke(firstWord, arguments);  
  
    } catch (Exception e) {  
        . . . .  
    }  
    return result;  
}
```

# Polia

(**java.lang.reflect.Array**)

```
int[] pole = (int[]) Array.newInstance(int.class, 5);    // int[] pole = new int[5];
```

```
for(int i = 0; i < Array.getLength(pole); i++) {  
    Array.set(pole, i, i);                                // pole[i] = i;  
    Array.setInt(pole, i, i);                             // pole[i] = i;  
}
```

```
for(int i = 0; i < Array.getLength(pole); i++ ) {  
    System.out.println("pole["+i+"] = " + Array.get(pole, i));    // pole[i] = i;  
    System.out.println("pole["+i+"] = " + Array.getInt(pole, i)); // pole[i] = i;  
}
```

```
pole[0] = 0  
pole[1] = 1  
pole[2] = 2  
pole[3] = 3  
pole[4] = 4
```



# Polia

(**java.lang.reflect.Array**)

```
Nadtrieda o = new Nadtrieda();  
Field f = o.getClass().getField("pole");  
Object oo = f.get(o);  
if (oo.getClass().isArray()) {  
    System.out.println(Array.getLength(oo));  
    for(int i=0; i<Array.getLength(oo); i++)  
        System.out.println(Array.getInt(oo,i));  
}
```

3  
1  
2  
3

```
Object ooo = o.getClass().getField("poleStr").get(o);  
if (ooo.getClass().isArray()) {  
    System.out.println(Array.getLength(ooo));  
    for(int i=0; i<Array.getLength(ooo); i++)  
        System.out.println(Array.get(ooo,i));  
}
```

2  
janko  
marienka

# Efektivita

```
Nadtrieda nt=new Nadtrieda();
```

```
start=System.nanoTime();  
for(int i=0;i<MAX;i++)  
    nt.Too(Math.PI);  
end=System.nanoTime();
```

```
Method m=nt.getClass().getMethod("Too",double.class);  
startReflex=System.nanoTime();  
for(int i=0;i<MAX;i++)  
    m.invoke(nt, Math.PI);  
endReflex=System.nanoTime();
```

Regular **method call**: 0.05669715

reflexive **method call**:1.47600883

Slowdown factor:26x

regular **new (constructor)**: 0.56120261

reflexive **new (constructor)**:2.3079218200000002

Slowdown factor:4x

# Case 3

- zadanie Plumber z predtermínu 2008:
- tri vzorové skúšky (zadania) visia na stránke predmetu
- príklad ilustruje štruktúru skúšky:
  - **čítanie konfigurácie** hry súboru a vykreslenie plochy, konštrukcia scény,  
8  
4  
12345623  
34613532  
35216311  
23654545
  - **ošetrenie udalostí** a rozpohybovanie scény v intenciách pravidiel danej hry,
  - **počítanie a zobrazovanie** krokov, životov, časomiera, zistenie, či v danej konfigurácii už sme boli a pod,
  - **škálovateľnosť** hracej plochy,
  - **load a save** konfigurácie (serializácia),
  - **algoritmus** (napr. kam dotečie voda - hľadanie cesty v grafe (labyrinte), analýza víťaznej konfigurácie, ...)

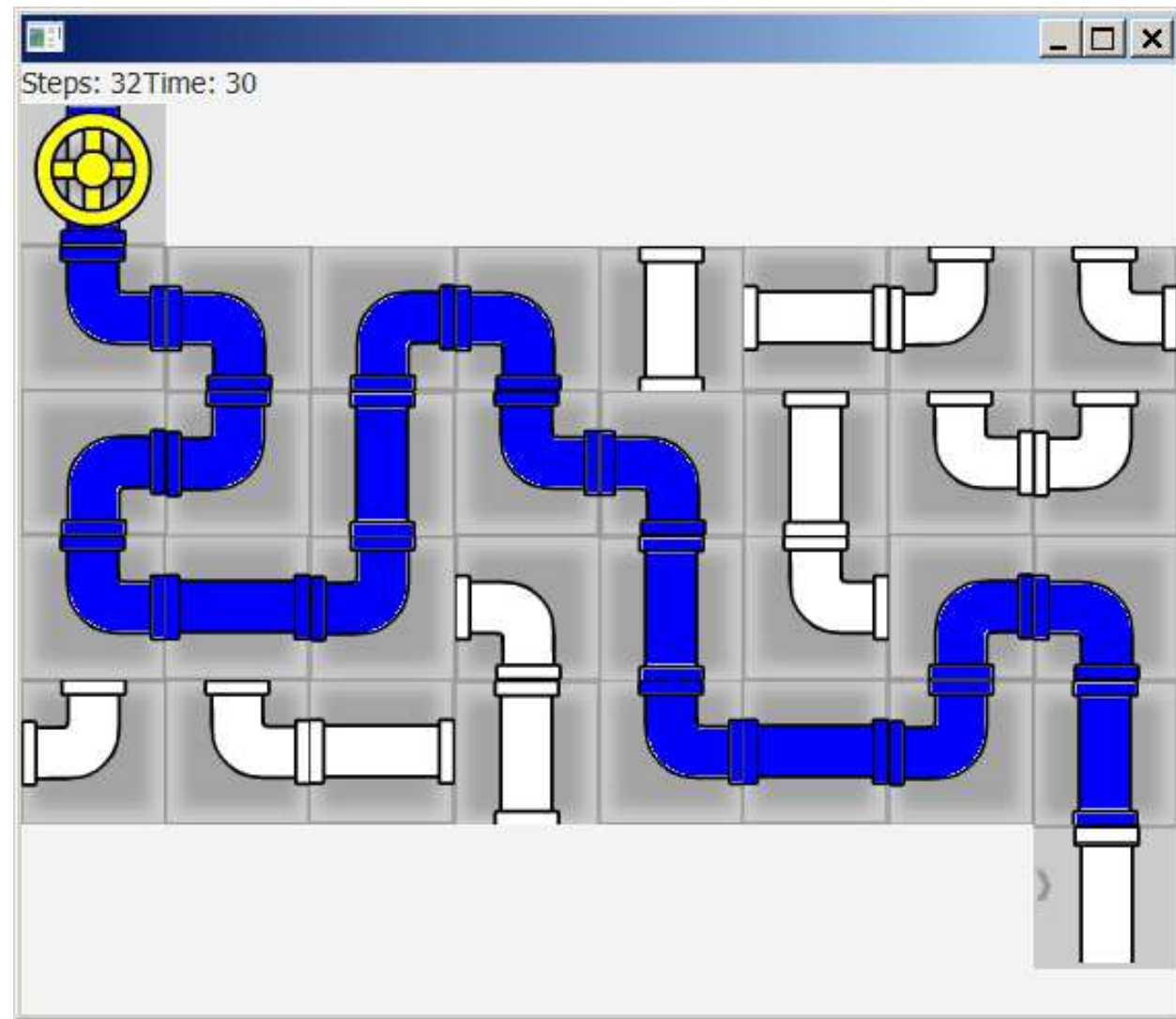
# Plumber (inštalatér)

## Oddel'te GUI

- kreslenie objektov,
- komponentov,
- appletu

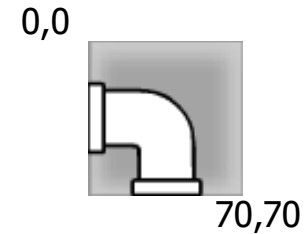
## od logiky hry

- analýza ťahov,
- víťazná konfigurácia,
- zacyklenie, ...



- Plumber – BorderPane/GridPane/Canvas,
- PlumberCanvas – Mouse Event Handler, kreslenie rúr .png,
- PlumberThread - časomiera,

# Plumber



- čítanie obrázkov:

```
for (int i = 1; i <= 8; i++) {  
    img[i] = new Image("plumber" + i + ".png");  
    img_blue[i] = new Image("plumber" + i + "_blue.png");  
    – ak vám nekreslí obrázok, pravdepodobne ste ho nenačítali správne,  
    – najčastejšie nie je v správnom adresári
```

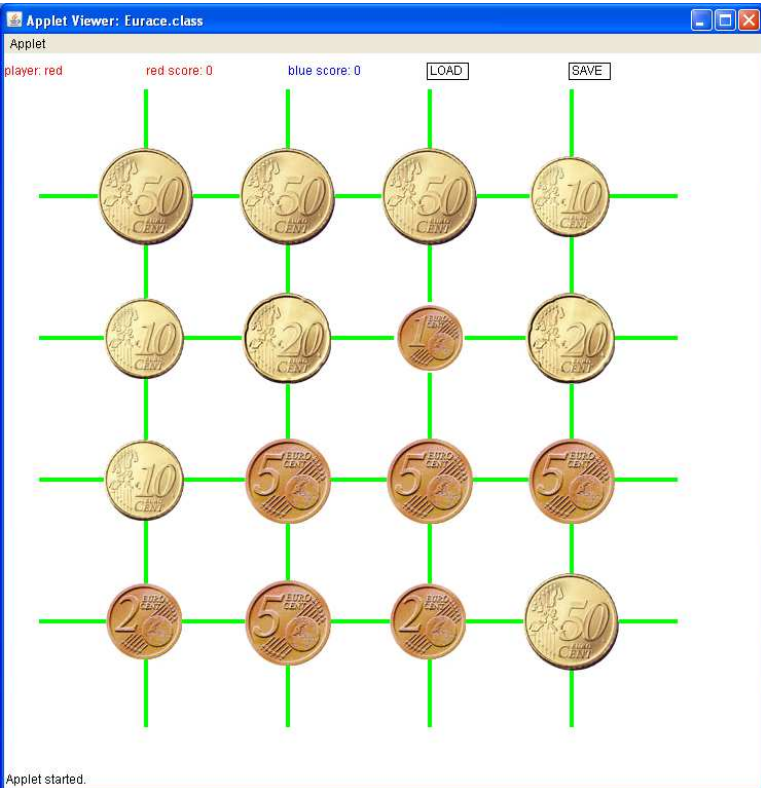
- čítanie vstupnej konfigurácie

```
try {  
    BufferedReader br =  
        new BufferedReader(new FileReader(new File("Plumber.txt")));  
    ... // čítanie textového súboru  
} catch (Exception E) {  
    System.out.println("file does not exist");  
}  
  
– nezanedbajte výnimky,  
– píšete na konzolu, čo čítate, kontrolné pomocné výpisy vás nijako nehandicapujú,  
– ak čítate vstup po znakoch (celé zle), nezabudnite, že riadok končí znakmi 13, 10,  
– rozdiel medzi cifrou a jej ascii kódom je 48, úplne zle, ...
```

- uloženie konfigurácie počas hry

- najjednoduchšie pomocou serializácie (pozri prednášku java.io)
- neserializujte celú aplikáciu, ale len triedu popisujúcu konfiguráciu hry – PiškyState..

Súbor: [Plumber.java](#)



# Škálovanie

Naprogramujte mriežku škálovateľnú od rozmeru okna (štvorcová mriežka sa rozťahuje podľa veľkosti okna, v ktorom sa nachádza, NIE KONŠTANTA V PROGRAME)



```
private static int dist = 120;
private static int offset = dist;
```

// počiatočné nastavenia

```
public void init() {
    setSize(offset+N*dist, offset+N*dist); // originálna veľkosť hracej pl.
```

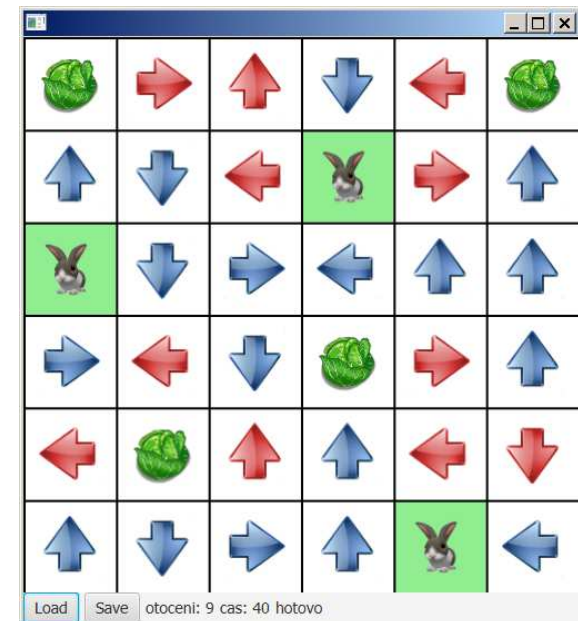
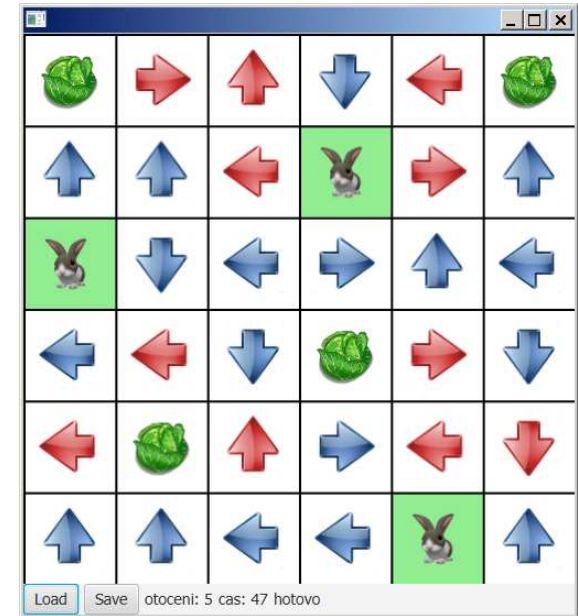
```
public void paint() {
    // nastavenia podľa aktuálnej
    // veľkosti okna
```

```
offset = dist = (Math.min(getHeight(),getWidth()))/(N+1);
```

# Niečo novšie

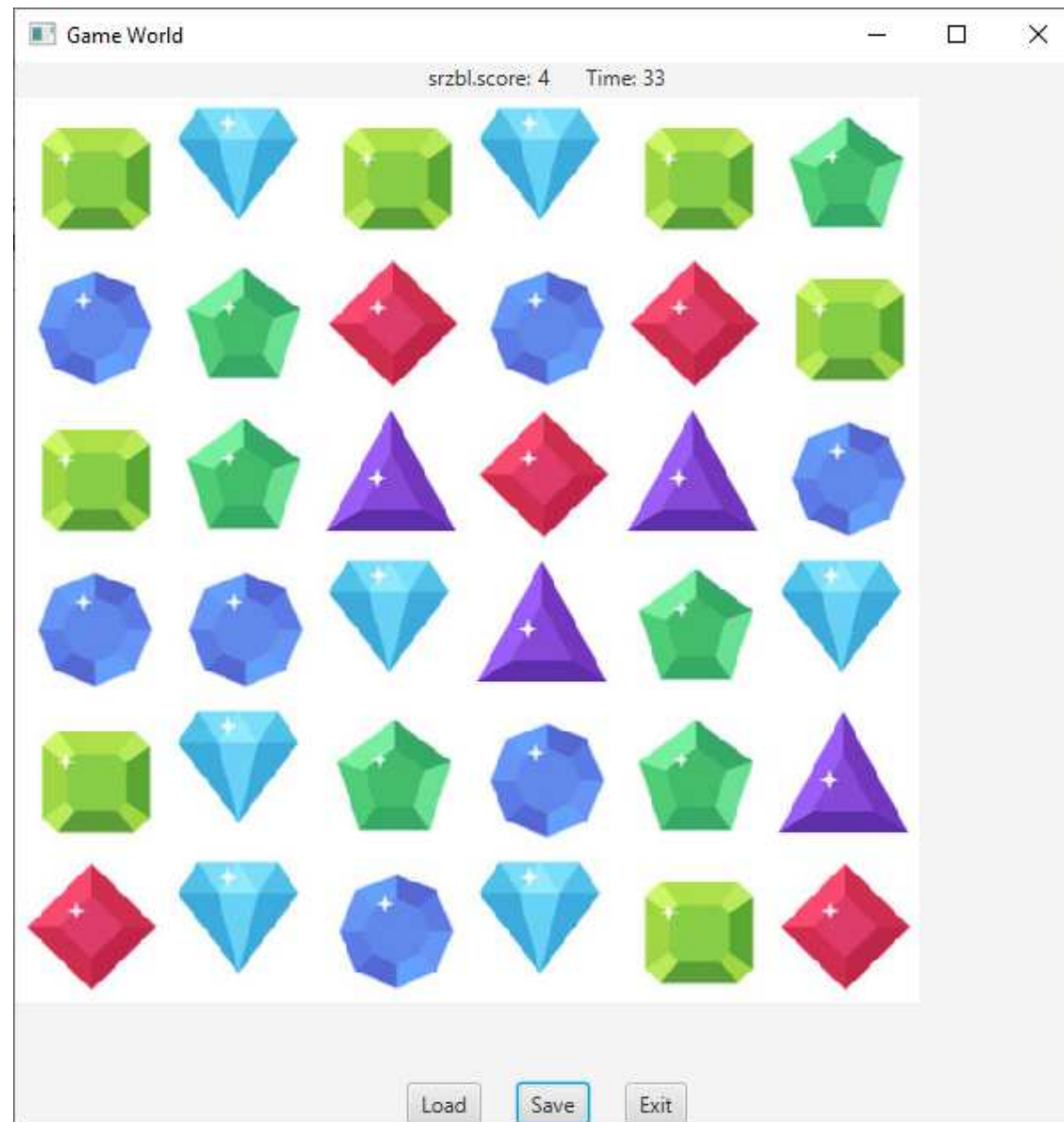
## (Zajace a kapusty)

- Naprogramujte hru pre jedného hráča Zajace a kapusty. Hrá sa na štvorcovej mriežke NxN štvorcov. V niektorých štvorcoch sa nachádzajú zajace, v niektorých kapusty a v ostatných šípky smerujúce jedným zo štyroch smerov. Počet zajacov, kapusty a šípok môže byť vzhľadom na N rôzny. Niektoré šípky sú červené - tie smerujú stále rovnakým smerom, niektoré sú modré a tie sa pri kliknutí myšou otáčajú o 90°. Príklad hernej situácie je na obrázku:
- Cieľom hráča je pootáčať modré šípky tak, aby sa všetky zajace mohli podľa šípok dostať ku kapuste. Keď zajac stúpi na políčko, kde je šípka, musí pokračovať smerom podľa šípky. Ak narazí na okraj poľa, alebo sa medzi nejakými šípkami zacyklí, ku kapuste sa nedostal. Začiatočná konfigurácia hry je uložená v súbore ...





# Bypass Excellencie



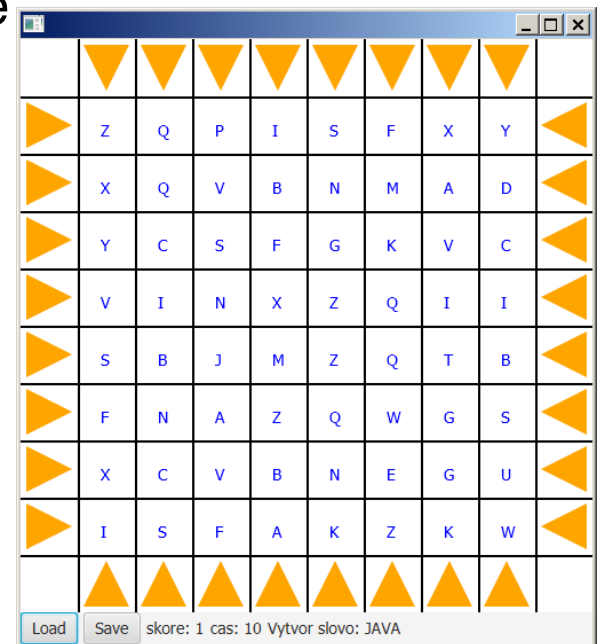
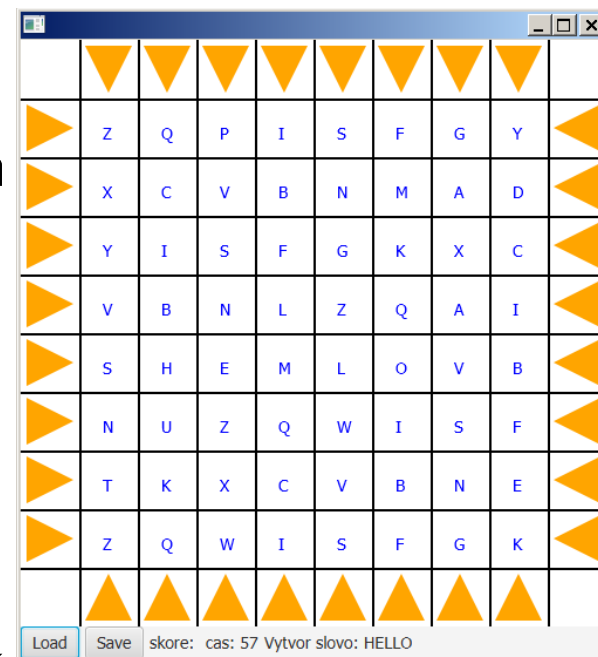


# Niečo novšie

## (Písmenkovica)

- V hre Písmenkovica sú v štvorcovej mriežke rozmiestnené písmená anglickej abecedy. Na okrajoch všetkých strán štvorca sú šípky. Ich stlačením dojde k otočeniu riadka alebo stĺpca o jedno písmenko podľa smeru šípky. Niekde v okne je zobrazené slovo, ktoré treba zo susedných písmen v mriežke vytvoriť: buď vodorovne zľava-doprava, zvislo zhora-nadol, šikmo nadol vpravo, alebo šikmo nahor vpravo. Ak sa to hráčovi podarí, písmená vytvoreného slova zmiznú a sú nahradené za ďalšie. Hráč tým získava bod, cieľové slovo sa zmení a hra pokračuje ďalej. Na každé slovo má 60 sekúnd času, ktoré sa mu odpočítavajú a zostávajúci čas sa zobrazuje. Ak to nestihne, hra končí. Tlačidlami Save/Load uloží/načíta aktuálny stav hry, pričom z načítaného stavu môže pokračovať v hre ďalej. Začiatočná situácia hry, cieľové slová a písmená, ktoré postupne nahrádzajú písmená z vytvorených slov, sú uložené v súbore a na začiatku hry sa z neho načítajú. Formát súboru je nasledujúci

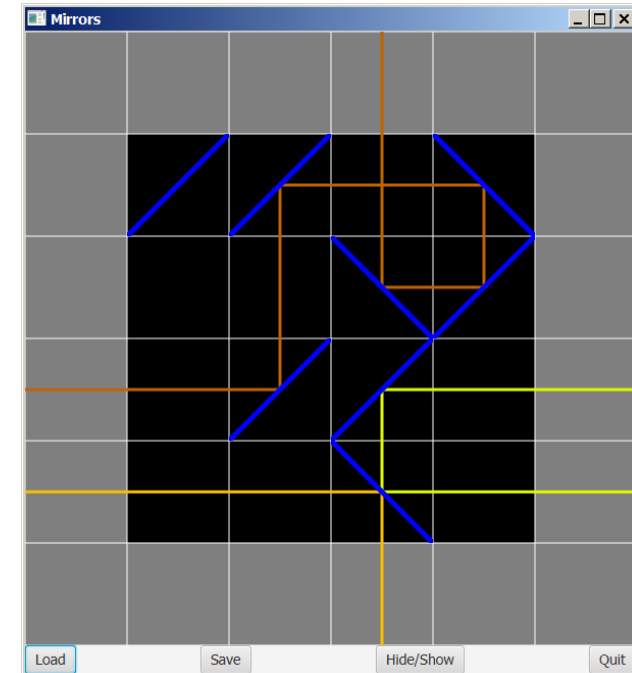
...



# Niečo novšie

## (Zrkadlová sieň)

- V štvorcovej sále s rozmermi  $N \times N$  sú v niektorých políčkach umiestnené diagonálne zrkadlá, v ilustráciach sú zobrazené modrou farbou. Môžu byť dvoch typov, / alebo \. Na kraji štvorcovej sály sú políčka, ktoré obsahujú zdroje svetla rôznych farieb. Krajné ľavé a pravé políčka ( $N+1$ ) obsahujú vodorovný zdroj svetla, krajné horné a dolné políčka ( $N+1$ ) obsahujú zvislý zdroj svetla. Rožné políčka (4) nemajú žiadnu funkciu. Pre jednoduchosť znázornenia scény plochu kreslíme do štvorcovej mriežky s rozmermi  $(N+2) \times (N+2)$ .



Ak zapneme svetelný zdroj, vodorovný či zvislý, svetlo sa začne šíriť daným smerom cez hráciu plochu. Ak je políčko prázdne, prejde ním. Ak je v ňom diagonálne zrkadlo, odrazí sa od neho presne v duchu príslovia: uhol odrazu je uhol dopadu. Samozrejme, keďže ide o diagonálne zrkadlá, tak tento uhol môže byť len 45 stupňov. Pri rôznych polohách zrkadiel môžu vzniknúť 4 situácie (premýslite si...). Niektorým políčkom lúč opustí hráciu plochu, vy znázorňujete jeho cestu. Svetelné zdroje sú rôznych farieb, a farieb máte dosť (stačí si ich nejako vygenerovať).

# Niečo novšie

## (Atomix)

- V hre Atomix sa z atómov konštruujú molekuly rozličných zlúčenín. Každý atóm má naznačený smer väzby a počas hry sa nedokáže otočiť - väzba smeruje stále tým istým smerom. V našej verzii hry sa zameriame iba na molekulu vody. Po kliknutí myšou na niektorý atóm sa tento atóm zvýrazní. Druhé kliknutie na voľné políčko v prázdnej uličke, ktorá vychádza od atómu jedným zo štyroch smerov, atóm uvedie do pohybu. Atóm sa však zastaví, až keď narazí do steny, alebo do iného atómu. Potom je možné kliknúť na nejaký atóm znova. V prípade, že sa podarí vytvoriť molekulu vody, t.j. vedľa seba sa bude vodorovne alebo zvislo nachádzať atóm vodíka, kyslíka a zasa vodíka a budú previazané vzájomnými väzbami, hráč level splnil a môže postúpiť do ďalšieho levelu.

