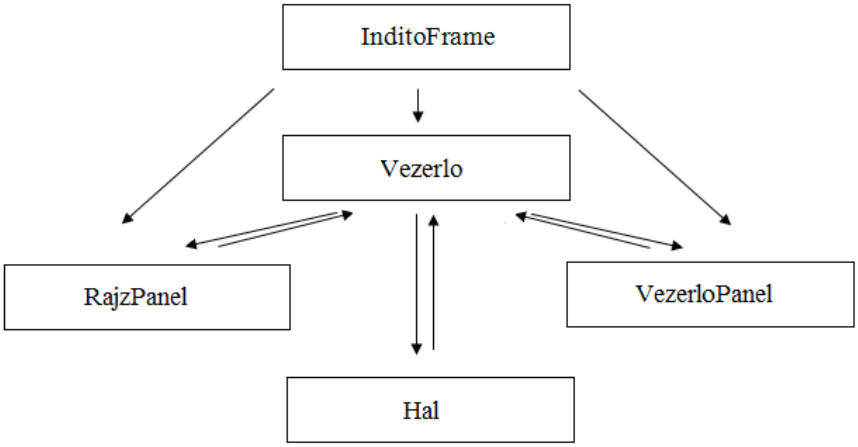
# Halas variációk


A baloldali vezérlőpanelen kiválasztott halak úszkáljanak az akváriumban.

A következőkben **három variációról** lesz szó.

## 1. variáció

A feladatmegoldás során az újrahasznosíthatóság is célunk, ezért úgy akarjuk megoldani, hogy az egyes osztályok minél függetlenebbek legyenek egymástól, vagyis úgy, hogy mindegyik csak a vezérlővel tartson kapcsolatot.




Elvileg így oldottuk meg a bemutatott pdf fájlban is, ténylegesen azonban nem, ugyanis a vezérlőpanel „titokban” mégiscsak kapcsolatot tartott a `Hal` osztállyal, hiszen `Hal` típusú objektumokat kezelt. A most bemutatandó variációk ezt a hibát küszöbölik ki.

Hogyan lehet függetlenné tenni a vezérlőpanelt a halaktól? Úgy, hogy nem használjuk a `Hal` típust. Helyette két lehetőségünk van: vagy az alapértelmezett ős `Object` osztályt használjuk, vagy generikus típust. Kezdjük az első változattal.

A modellt ekkor így deklaráljuk:

```
private DefaultListModel<Object> objektumModell = new DefaultListModel<>();
```

A panel a vezérlőtől kapja majd meg a beolvasott objektumokat, ezeket kell modellbe raknia:

```
public void ittVannakaBeolvasottAdatok(Object object) {
    objektumModell.addElement(object);
}
```

A vízbedobás gomb hatása:

```
private void btnObjektumValasztasActionPerformed(java.awt.event.ActionEvent evt) {
    List<Object> kivalasztottak = lstObjektumok.getSelectedValuesList();
    for (Object object : kivalasztottak) {
        vezerlo.objektumFeldolgozas(object);
        objektumModell.removeElement(object);
    }
}
```

Most persze mondhatja, hogy mégiscsak benne maradt valami, ami a halakra utal, hiszen a gomb felirata a vízbedobást emlegeti, és a felület tetejére írt címben is a halak szerepelnek. Ez igaz. Ha ugyanezt az osztályt más, hasonló jellegű feladatra szeretnénk használni (pl. a listában lévő diákok számára el akarjuk küldeni a zh-kat), akkor kicsit módosítanunk kell a felületen látható feliratokat. De csak ezeket, és semmi mást. Vagyis a kódhoz egyáltalán nem kell hozzányúlnunk. Ráadásul az is megoldható, hogy ezeket a feliratokat egy külső fájlból vegyük, így aztán valóban teljesen rugalmassá válik.

A vezérlő eredeti `vizbedob()` metódusa most annyiban módosul, hogy nem `Hal`, hanem `Object` típust kap paraméterként, illetve mivel a vezérlőpanelnek fogalma sincs, hogy mi történik majd az átadott objektummal, ezért a metódus neve nem `vizbedob()`, hanem `objektumFeldolgozas()`. A metódusban összesen annyit kell tennünk, hogy a paramétert `Hal` típusúra kényszerítjük:

```
public void objektumFeldolgozas(Object object) {
    if (object instanceof Hal) {
        Hal hal = (Hal) object;

        ...

    }
}
```

Ha mégsem `hal` típusú példányt kap, akkor most nem csinál vele semmit, de úgy is meg lehetne írni, hogy az egészet berakjuk egy `try` blokkba, a `catch` ágon pedig hibajelzést adunk.

**2. variáció**

Szebb az a megoldás, ha az általános `Object` helyett generikust használunk – csak az eltéréseket emeljük ki (`T` tetszőleges típust jelent):

```
public class VezerloPanel<T> extends javax.swing.JPanel {

    private DefaultListModel<T> objektumModell = new DefaultListModel<>();

    private SzalVezerlo<T> szalVezerlo;


    public void ittVannakaBeolvasottAdatok(T object) {
        objektumModell.addElement(object);
    }
```

```
private void btnObjektumValasztasActionPerformed(java.awt.event.ActionEvent
    List<T> kivalasztottak = lstObjektumok.getSelectedValuesList();
    for (T object : kivalasztottak) {
        vezerlo.objektumFeldolgozas(object);
        objektumModell.removeElement(object);
    }
}
```

A vezérlőben ezek a sorok módosulnak:

```
public class Vezerlo<T> {

    private VezerloPanel<T> vezerloPanel;

    public void objektumFeldolgozas(T object) {
```

Ennyi változtatás mellett is működik, de ez még nem igazán különbözik az első variációtól, hiszen ha változatlanul hagyjuk a frame-t, akkor gyakorlatilag továbbra is `Object`-ként kezeljük a `T` típust, de persze, már így is léptünk előre, hiszen elvileg beállítottuk a típussal való paraméterezhetőséget. (Jó humora van a Word helyesírás-ellenőrzőjének a paraméterezhetőség szó helyett a paraméterehetőség szót javasolja. ☺)

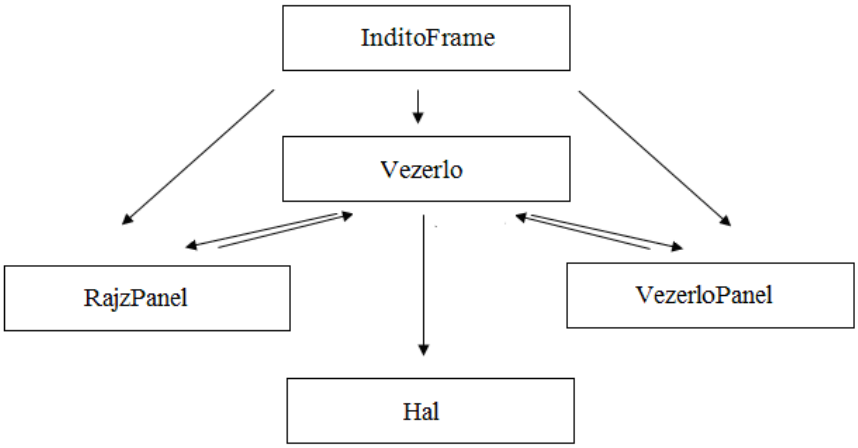
A generikus tényleges használata az lenne, ha mindkét osztályt valóban paramétereznénk is, és a paraméterezett példányokat raknánk fel a frame-re. Ezt azonban csak kódból tudjuk

megoldani, mégpedig pl. úgy, hogy létrehozunk egy `Main` osztályt, ennek `main()` metódusából hívjuk meg a felületet létrehozó metódust:

```
private void start() {

    RajzPanel rajzPanel = new RajzPanel();
    VezerloPanel<Hal> vezerloPanel = new VezerloPanel<>();
    Vezerlo<Hal> vezerlo = new Vezerlo(rajzPanel, vezerloPanel);

    rajzPanel.setVezerlo(vezerlo);
    vezerloPanel.setSzalVezerlo(vezerlo);

    JFrame frame = new JFrame();
    frame.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
    frame.add(vezerloPanel, BorderLayout.WEST);
    frame.add(rajzPanel, BorderLayout.CENTER);
    frame.setSize(szelesseg, magassag);
    frame.setTitle(cim);
    frame.setLocationRelativeTo(null);

    frame.setVisible(true);

    vezerlo.indit();
}
```

### 3. variáció

Az osztályok közötti függőség tovább csökkenthető, ha másképp építjük fel a `Hal` osztályt.




Ráadásul a most bemutatandó megoldás gyógyírt jelent arra a kis szépséghibára is, hogy az eddig tárgyalt megoldás önálló (azaz nem szálként viselkedő, és nem állandóan frissülő) képként nem tud animált gifet kezelni, ill. nem animálja a megjelenített gif képet.

Az alapötlet az, hogy a halakat komponensként kezeljük. A megoldás előnye, hogy saját maga ki tudja rajzolni magát, illetve képes frissülni, hátránya, hogy jobban oda kell figyelni a layout beállításra, illetve esetenként kezelni kell, hogy ki kit takarjon (melyik halnak a képe látszódjon, ha „z” irányban egymás fölött haladnak el – esetünkben most ez teljesen lényegtelen).

**FONTOS**:
Csak akkor látszódik az így megadott komponens, ha a rajzpanelt `BorderLayout`-ra állítjuk!

Az ilyen elvű megoldás részletei (most csak a mozgás és az elkapás van benne, de nyilván bővíthető az egyéb funkciókkal is):

```
public class Hal extends JComponent implements Runnable{

    private String nev;
    private Image balKep, jobbKep;
    private int szelesseg, magassag;
    private Image kep;
    private boolean aktiv;
    private int lepeskoz;
    private long ido;

    private static int akvariumSzelesseg;
```

```
public Hal(String nev, KepPar kepPar, int szelesseg, int magassag) {
    this.nev = nev;
    this.balKep = kepPar.getBalKep();
    this.jobbKep = kepPar.getJobbKep();
    this.szelesseg = szelesseg;
    this.magassag = magassag;
    // A megadott méret most a komponens mérete lesz.
    this.setSize(szelesseg, magassag);
}

public void beallitas(boolean fut, int lepeskoz, long ido) {
    this.aktiv = fut;
    this.lepeskoz = lepeskoz;
    this.ido = ido;

    if(lepeskoz < 0) kep = balKep;
    else kep = jobbKep;
}

// Mivel komponens, ezért most ki tudja rajzolni magát. A hal képe
// a komponens háttérképe.
@Override
protected void paintComponent(Graphics g) {
    super.paintComponent(g);
    if(kep != null){
        g.drawImage(kep, 0, 0, this.getWidth(), this.getHeight(), this);
    }
}

// Runnable-ként deklaráltuk, ezért így lehet elindítani.
public void start() {
    Thread szal = new Thread(this);
    szal.start();
}

@Override
public void run(){
    while(aktiv){
        mozdul();
        frissit();
        kesleltet();
    }
}
```

Mivel most önálló komponens, ezért a kép bal sarkának koordinátái a komponens origója – pontosan ezért nem volt rá szükség, hogy a beállítás során megadjuk a bal sarok helyzetét. Viszont a mozgatáshoz erre mégiscsak szükségünk lenne. Ez most úgy oldható meg, hogy lekérjük a komponens `getX()`, `getY()` koordinátáit (ezek határozzák meg a „hordozó” panelen – parent component – lévő helyét), majd a megváltozott értékkel ismét beállítjuk a helyzetét (`setLocation()`).

```
private void mozdul() {
    int kx = this.getX();
    int ky = this.getY();
    kx += lepeskoz;
    this.setLocation(kx, ky);
    if(kx > akvariumSzelesseg - this.getWidth() || kx < 0){
        lepeskoz = -lepeskoz;
        if(lepeskoz < 0) kep = balKep;
        else kep = jobbKep;
    }
}

// Tudja frissíteni saját magát.
private void frissit() {
    this.repaint();
}
```

A többi metódus változatlan. A vezérlőpanelen sincs változás.

A rajzpanelnek többé nem kell halakat rajzolnia, elég, ha csak a saját háttérképét rajzolja, és majd a szálvezérlő kérésére felrakja az újabb és újabb hal-komponenseket.

A `paintComponent()` tehát csak ennyi:

```
@Override
protected void paintComponent(Graphics grphcs) {
    super.paintComponent(grphcs);
    grphcs.drawImage(hatterKep, 0, 0, this.getWidth(), this.getHeight(), null);
}
```

viszont szükség van a halak felrakását végző metódusra:

```
// Ha így oldjuk meg, akkor a rajzpanelnek nem is kell tudnia, hogy
// milyen komponens kerül rá, azaz nem kell ismernie a Hal osztályt.
public void felrak(JComponent hal, int kx, int ky) {
    hal.setLocation(kx, ky);
    this.add(hal);
    //Ez most nem kell, akkor érdekes, ha fontos, hogy a frissen
    //felrakott elem z irányban a legtetején legyen.
    this.setComponentZOrder(hal, 0);
}
```

A vezérlő is egyszerűsödik, mert nincs szükség sem a `frissit()` sem a `rajzol()` metódusra, illetve a halak listájára sincs, hiszen az a `rajzol()` metódushoz kellett.

Felrakni és indítani így tudjuk (a vezérlőnek ugyanabban a „vízbedobó” metódusában, ahol eddig):

```
boolean aktiv = true;
hal.beallitas(aktiv, lepesKoz, ido);
rajzPanel.felrak(hal, kepX, kepY);
hal.start();
```

Mivel ez a szemléletmód újdonság, ezért még azt is beszéljük meg, hogyan lehet kilőni egy ilyen halat. A feladat tehát most az, hogy a rajzpanelre kattintva, ha eltaláltunk egy halat, akkor az tűnjön el.

Különösebb magyarázat nélkül a szükséges metódusok:

`RajzPanel`:

```
private void formMouseClicked(java.awt.event.MouseEvent evt) {
    vezerlo.vadaszat(evt.getX(),evt.getY());
}
```

(Vagyis látjuk, hogy a rajzpanelnek továbbra is szüksége van a vezérlőre.)

A vezérlőnek most mégis szüksége lesz a halak listájára (amit a rajzolás kedvéért még nem kellett volna definiálnunk), hiszen egyenként meg kell kérdeznie, hogy kit találtak el. Ehhez viszont az elindított halakat be kell rakni ebbe a listába:

```
boolean aktiv = true;
hal.beallitas(aktiv, lepesKoz, ido);
rajzPanel.felrak(hal, kepX, kepY);
halak.add(hal);
hal.start();
```

Mivel nem akarjuk, hogy a hal is kénytelen legyen „visszaszólni a vezérlőnek”, ezért nem kérjük, hogy szóljon neki, ha eltalálták, hanem csak annyit, hogy igaz/hamis értékkel jelezze ezt. Ha eltalálták, akkor a vezérlő már tudja, hogy törölnie kell (a `Vezerlo` -ben):

```
public void vadaszat(int x, int y) {
    for (Hal hal : halak ) {
        if(hal.elkaptak(x,y)){
            halak.remove(hal);
            rajzPanel.torol(hal);
            break;
        }
    }
}
```

A `Hal` `elkaptak()` metódusa:

```
public boolean elkaptak(int x, int y) {
    if(this.contains(new Point(x-this.getX(),y-this.getY()))) return  true;
    return false;
}
```

A `RajzPanel`-en ennyi a törlés:

```
public void torol(JComponent hal) {
    this.remove(hal);
    this.repaint();
}
```