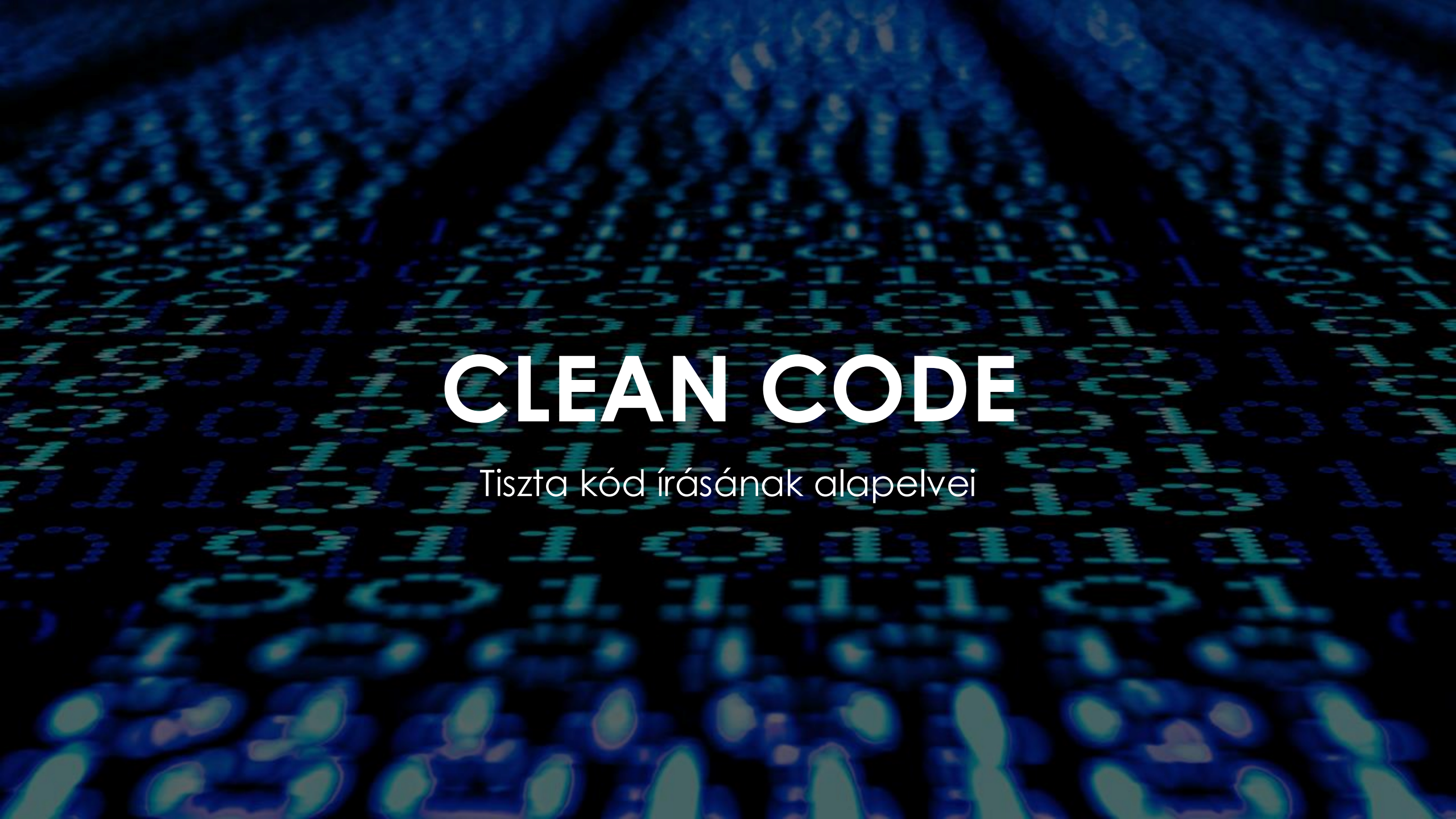

# CLEAN CODE

Tiszta kód írásának alapelvei

# TISZTA KÓD JELLEMZŐI

- Egyszerű és egyértelmű: a kódot olvasó azonnal megérti, mit csinál a program, olyan fejlesztő számára is olvasható, továbbfejleszthető, aki nem az eredeti szerző.
- Fókuszált: minden komponens egyetlen funkciót lát el, és azt jól csinálja
- Következetes: az elnevezések, formázások és struktúrák szabványosak
- Minimális: minden komponense csak azt tartalmazza, amire szükség van, és csak annyi függősége van, amennyi szükséges
- Tesztelhető: könnyen írhatóak tesztek hozzá

# FONTOS FOGALMAK

- Refaktorálás = kódújraszervezés: a kód belső szerkezetének javítása anélkül, hogy a külső viselkedés változna, ez javítja az olvashatóságot és csökkenti a komplexitást

- Kódszagok (Code smells): rosszul tervezett kód tünetei, amelyek mélyebb problémákat jelezhetnek, pl.: túl hosszú metódusok, ismétlődő kódok, vagy nem beszédes változónevek

# 1. ÉRTELMES NEVEK

- változók, metódusok és osztályok neve tükrözze a céljukat/szándékukat
- egybetűs változók csak közismert esetekben
  pl.: koordináták (x,y,z), ciklusváltozók (i,j,k)
- kerüljük a félrevezető neveket
  pl.: accountList, amikor nem is listáról van szó, helyette: accounts vagy accountGroup
- használjunk egyértelműen megkülönböztető neveket
  pl.: copyChars(char a1[], char a2[]) helyett copyChars(char source[], char target[])

# 1. ÉRTELMES NEVEK

- kerüljük az általános zajszavakat
  pl.: Data, Info, Object, Variable
- használjunk kiejthető és kereshető neveket
  pl. "a" vagy "acnt" helyett "account"
- név hossza feleljen meg a hatásköre méretének
  pl.: BUTTONS_PANEL_ROW_WIDTH
- név ne árulkodjon a hatásköréről
  pl.: IUser, prvName

# 1. ÉRTELMES NEVEK

- osztályok legyenek főnevek egyes számban, függvények legyenek igék
  pl.: FileReader, readByte
- következetesen adjunk neveket
  pl.: handler/manager/factory közül csak egyet
- önmagukban nem értelmes nevek kontextusba helyezése
  pl.: "street", "city", "zipCode", "state" egyértelműen "address"-re utalnak, viszont csak a "state" önmagában sok mindent jelenthet

# 1. ÉRTELMES NEVEK

```
// Rossz
int x = 60;
int y = 40;
int z = (x + y) / 2;

// Jó
int mathScore = 60;
int physicsScore = 40;
int averageScore = (mathScore + physicsScore) / 2;
```

# 1. ÉRTELMES NEVEK

```
// Rossz
public List<int[]> getThem() {
  List<int[]> list1 = new ArrayList<int[]>();
  for (int[] x : theList)
    if (x[0] == 4)
      list1.add(x);
  return list1;
}
```

# 1. ÉRTELMES NEVEK

```java
// Jó
public List<int[]> getElementsWhereFirstValueIsFour(List<int[]> elements) {
    List<int[]> filteredElements = new ArrayList<>();

    for (int[] element : elements) {
        if (isFirstValueFour(element)) {
            filteredElements.add(element);
        }
    }

    return filteredElements;
}

private boolean isFirstValueFour(int[] element) {
    return element != null && element.length > 0 && element[0] == 4;
}
```

# 2. RÖVID, JÓL DEFINIÁLT FÜGGVÉNYEK

- egy függvény maximum 20, de inkább 5-15 soros legyen
- egy dolgot csináljon, de azt jól
- legyenek mellékhatásmentesek -> ne csináljanak mást, mint amit a nevükben ígérnek
- utastásblokkok (for, if) csak egy sor függvényhívást tartalmazzanak alapvető eseteknél

# 2. RÖVID, JÓL DEFINIÁLT FÜGGVÉNYEK

- 0-2 paraméterrel rendelkezzen, 3 a maximum, ami még elfogadható
- több paraméter szüksége esetén szervezzük a paramétereket egy objektumba
- ne használjunk jelző logikai paramétert, mert akkor két különböző dolgot fog csinálni a függvény
- kerüljük az output paramétereket

```
// Rossz:
Circle makeCircle(double x, double y, double radius);

// Jó:
Circle makeCircle(Point center, double radius);
```

# 2. RÖVID, JÓL DEFINIÁLT FÜGGVÉNYEK

- ne legyen 2-nél több indentálás függvényen belül, hanem ilyenkor szervezzük ki a kódot több függvénybe
- több return/break használatával is elkerülhetőek a további elágazásokkal járó indentálások
- switchet csak akkor használjunk, ha muszáj, és szervezzük ki alacsony szintű osztályba
- függvény vagy csináljon valamit vagy válaszoljon valamit, de lehetőleg ne mind a kettőt -> vagy változtassa meg egy objektum állapotát, vagy adjon vissza információt az objektumról

# 2. RÖVID, JÓL DEFINIÁLT FÜGGVÉNYEK

```
// Rossz:
public void processOrder(Order order, boolean isExpress, Result result) {
    if (isExpress) {
        // Gyorsított rendelés feldolgozása
        result.status = "Processed as express";
        result.deliveryTime = "1 day";
    } else {
        // Normál rendelés feldolgozása
        result.status = "Processed as standard";
        result.deliveryTime = "3-5 days";
    }
}
```

# 2. RÖVID, JÓL DEFINIÁLT FÜGGVÉNYEK

```
// Jó:
public Result processStandardOrder(Order order) {
    return new Result("Processed as standard", "3-5 days");
}

public Result processExpressOrder(Order order) {
    return new Result("Processed as express", "1 day");
}
```

# 3. MEGJEGYZÉSEK MINIMÁLIS HASZNÁLATA

- legjobb esetben is szükséges rosszak -> pontatlan, elavult információt közölhetnek, nehezen karbantarthatóak (kód változik, komment nem)
- a kód beszéljen önmagáért -> ha túl sok kommentet kell írni, valószínűleg a kódot kell átszervezni vagy részeit átnevezni
- csak a összetettebb logikát magyarázó kommenteket hagyjunk
- jó megjegyzések: informatív, szándékot magyarázó, tisztázó, következményekre figyelmeztető, TODO

# 3. MEGJEGYZÉSEK MINIMÁLIS HASZNÁLATA

```
// Ellenőrzi, hogy az adott jelszó megfelel-e a minimális biztonsági követelményeknek.
public bool IsValidPassword(string password)
{
    return password.Length >= 8 && password.Any(char.IsDigit);
}

// A fizetési műveletet tranzakcióban kell végezni az adatok konzisztenciájának biztosítása érdekében.
using (var transaction = database.BeginTransaction())
{
    ProcessPayment(order);
    transaction.Commit();
}

// A negyedéves jelentéseknél a hónap 1. napjától kezdjük a számítást.
DateTime reportStartDate = new DateTime(currentYear, currentQuarter * 3 - 2, 1);

// Figyelem: A fájl törlése végleges, nincs visszaállítási lehetőség!
File.Delete(filePath);

// TODO: Támogatni kell a többnyelvűség kezelését a hibaüzenetekben.
Console.WriteLine("Error: Invalid input");
```

# 3. MEGJEGYZÉSEK MINIMÁLIS HASZNÁLATA

- rossz megjegyzések: motyogás, zaj-megjegyzés, túl sok információt közlő, vagy nem (közvetlen) kódhoz kapcsolódó komment, elválasztóvonal/szalagcím
- motyogás: hanyagul odavetett megjegyzés, mely legfeljebb a szerzőnek jelent valamit, más számára nem érthető
- zaj-megjegyzés: új információval nem szolgáló, a magától értetődőt újra megfogalmazó megjegyzés
  pl.: redundáns, félrevezető, kötelező dokumentálást hanyagul letudó megjegyzések
- verziókezelők óta feleslegessé vált kommentek: szerző neve, módosítások naplózása, kikommentezett régi kódok

# 3. MEGJEGYZÉSEK MINIMÁLIS HASZNÁLATA

```
// Valami itt történik...
ProcessData();

// A felhasználó nevének beállítása
user.Name = "Kiss Pista";

// Szerző: Kovács J.
// Módosítva: 2024.11.15.

////////////////////////////////////
// Felhasználói műveletek
////////////////////////////////////
```

# 3. MEGJEGYZÉSEK MINIMÁLIS HASZNÁLATA

```
// Ez a függvény beállítja a felhasználó nevét az osztály példányán belül
// a megadott paraméter értékével. A paraméternek string típusúnak kell lennie,
// és a neve meg fog jelenni a felhasználói adatok között.
public void SetUserName(string name)
{
    user.Name = name;
}

//string connectionString = "Server=localhost;Database=test;";
//connectionString = "Server=production;Database=live;";

// Ez a cégünk legnépszerűbb funkciója, amit a tavalyi évben fejlesztettünk ki.
public void PopularFeature()
{
    // ...
}
```

# 4. JÓL OLVASHATÓ FORMÁZÁS

- egyszerű, közösen megbeszélt kódolási szabályok követése
  pl.: behúzások mértéke, új sorok, szóközök, zárójelek helye, stb.
- a) függőleges formázás:
  - az osztály felülről lefele legyen egyre részletesebb: elején a legmagasabb, végén a legalacsonyabb szintű függvények szerepeljenek
  - üres sor válasszon el minden fő egységet (csomagok, importok, változók, függvények, stb.)
  - szorosan kapcsolódó sorok között ne legyen elválasztás, pl. üres sor vagy komment

# 4. JÓL OLVASHATÓ FORMÁZÁS

- a) függőleges formázás (folyt.):
  - szorosan egymáshoz kapcsolódó fogalmakat tartsuk függőlegesen egymáshoz közel, mivel egymás megértéséhez szükségesek
  - konstans és példány változókat az osztályok elején deklarálunk
  - többi változót deklaráljuk a használatuk helyéhez a lehető legközelebb, akár az első kapcsolódó függvényhívás felett közvetlenül
  - a hívó függvény mindig előzze meg a hívottat

# 4. JÓL OLVASHATÓ FORMÁZÁS

- b) vízszintes formázás:
  - sorok ne legyenek hosszabbak 80-120 karakternél
  - ne írjunk egynél több utasítást egy sorba
  - tegyünk szóközt gyengén összetartozó elemek közé, pl. értékadásnál, műveleteknél
  - ne tegyünk szóközt zárójelezett paraméterek megadása elé, pl. konstruktor vagy más függvény deklarációknál és hívásoknál
  - egységes behúzás -> ne szegjük meg rövid if utasítások, ciklusok vagy függvények kedvéért sem!

# 4. JÓL OLVASHATÓ FORMÁZÁS

```
// Rossz
void Add(){int a=5;int b=10;Console.WriteLine(a+b);}

// Jó
void Add()
{
    int a = 5;
    int b = 10;
    Console.WriteLine(a + b);
}
```

# 4. JÓL OLVASHATÓ FORMÁZÁS

```
// Rossz
package fitnesse;
import java.io.IOException; import java.net.Socket; import java.util.concurrent.ExecutorSer
public class FitNesseServer implements SocketServer {
  @Override
  public void serve(Socket s) throws IOException {
    serve(s,10000);
  }
  private final FitNesseContext context; private final ExecutorService executorService;
  public FitNesseServer(FitNesseContext context,ExecutorService executorService) {
    this.context = context; this.executorService = executorService;
  }
  ...
}
```

# 4. JÓL OLVASHATÓ FORMÁZÁS

```java
// Jó
package fitnesse;

import java.io.IOException;
import java.net.Socket;
import java.util.concurrent.ExecutorService;
import fitnesse.socketservice.SocketServer;

public class FitNesseServer implements SocketServer {
  private final FitNesseContext context;
  private final ExecutorService executorService;

  public FitNesseServer(FitNesseContext context, ExecutorService executorService) {
    this.context = context;
    this.executorService = executorService;
  }

  @Override
  public void serve(Socket s) throws IOException {
    serve(s, 10000);
  }
  ...
}
```

# TOVÁBBI SZABÁLYOK

- DRY = Don’t Repeat Yourself (Ne ismételjük magunkat):
  - a kód egy dolog elvégzéséhez egy módot biztosít több helyett
  - ismétlődő kódokat ki kell szervezni függvényekbe

```
// Rossz
int square1 = 5 * 5;
int square2 = 10 * 10;

// Jó
int CalculateSquare(int number) => number * number;
int square1 = CalculateSquare(5);
int square2 = CalculateSquare(10);
```

# TOVÁBBI SZABÁLYOK

- KISS = Keep It Simple, Stupid (Egyszerűsítés és minimalizálás):
  - törekedjünk a kódban előforduló bonyolult logikák lehető legnagyobb mértékű leegyszerűsítésére
  - nem hagyhatunk túlbonyolított megoldásokat hosszútávon a kódban

```
// Rossz
bool IsEven(int number) => number % 2 == 0 ? true : false;

// Jó
bool IsEven(int number) => number % 2 == 0;
```

# TOVÁBBI SZABÁLYOK

- Alapos hibakezelés:
  - beszédes kivételek dobása hibaüzenetek helyett
  - try/catch blokkokat emeljük ki önálló függvényekbe

```
public void delete(Page page) {
    try {
        deletePageAndAllReferences(page);
    } catch (Exception e) {
        logError(e);
    }
}

private void deletePageAndAllReferences(Page page) throws Exception {
    deletePage(page);
    registry.deleteReference(page.name);
    configKeys.deleteKey(page.name.makeKey());
}
```

# TOVÁBBI SZABÁLYOK

- Konvenciók és szabványok használata:
  - az adott nyelv best practiceit, azaz legjobb bevált szokásait és gyakorlatait, tartsuk be
  - pl. C#-ban "PascalCase" metódusoknál és konstansoknál, "camelCase" más változóknál, LINQ használata, stb.

```
private const int FilterThreshold = 10;

var filteredNumbers = numbers.Where(n => n > FilterThreshold);

private static void PrintNumbers(int[] numbers) {
    ...
}
```