

# PROGRAMMING BEST PRACTICES

Ciesielczyk Michał

# Agenda

---

- Cel prezentacji.
- Konwencje nazewnictwa.
- Konwencje stylu.
- Poprawianie wydajności:
  - ▣ pętle,
  - ▣ obiekty,
  - ▣ łańcuchy znaków (String),
  - ▣ kolekcje,
  - ▣ operacje I/O.
- Podsumowanie.

# Wymagania

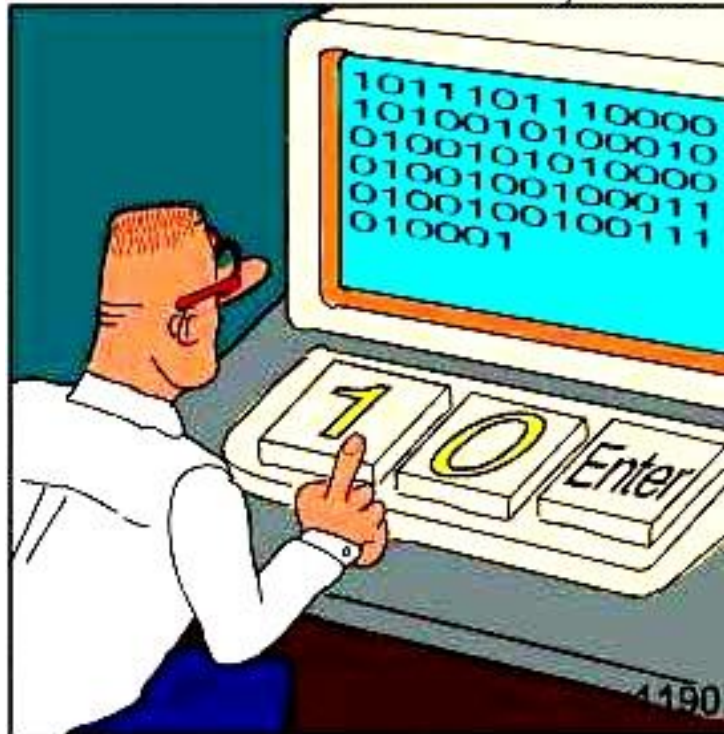
---

- Podstawowa znajomość jakiegoś języka programowania.
- Podstawowe pojęcia dotyczące programowania obiektowego.

# Cel prezentacji

„kod działający” vs „kod efektywny”.

# Pisać każdy może..



**REAL Programmers code in BINARY.**

# Po co?

- Maintenance - czytelny kod.
- Performance - szybkość działania.
- Reliability - odporność na niestandardowe zdarzenia.
  - *„If it compiles, sell it.”*
- Reusability - ponowne użycie.
  - *„Before software can be reusable it first has to be usable.”*
- *„The best is the enemy of the good.”* – Voltaire  
Lepsze wrogiem dobrego.

# Przykład sleep()

```
Thread.sleep(1000);
```

```
void sleep() {  
    for (int i = 0; i < Integer.MAX_VALUE; i++) {  
        // do nothing  
    }  
}
```

# Przykład isOdd()

```
public boolean isOdd(int number)
{
    return (number % 2) != 0;
}
```

```
public boolean isOdd(int number)
{
    boolean result;
    int i;
    result = true;
    for( i = 0; i < number; i++ )
    {
        if( result == true )
            result = false;
        else
            result = true;
    }
    return result;
}
```

# Jak?

- Przed (projektowanie):
  - wiedzieć co chcemy osiągnąć,
  - dzielić problemy na mniejsze.
- W trakcie (implementacja):
  - stosować się do konwencji,
  - korzystać z pracy innych,
    - „*Don't Reinvent The Wheel, Unless You Plan on Learning More About Wheels*”
  - jak najprościej,
    - „*Simplicity is the ultimate sophistication.*” - Leonardo da Vinci
  - nie zawsze najprościej znaczy najlepiej..
    - „*Premature optimization is the root of all evil.*” - Donald Knuth
  - przewidywać różne wyjątkowe sytuacje,
  - dokumentacja..
- Po (testowanie):
  - efektywna optymalizacja.

# Konwencje nazewnictwa

# Nazwy

- CamelCase!
  - BumpyCaps, InterCaps, ..
  - lowerCamelCase, ..
- Klasy - rzeczowniki, rozpoczynające się od wielkiej litery
  - Line, AudioSystem
- Zmienne – rozpoczynające się od małej litery
  - line, audioSystem
- Metody – czasowniki, rozpoczynające się od małej (wielkiej dla C#) litery
  - getName(), computeTotalWidth()
- Stałe – wielkimi literami, słowa oddzielone ‘\_’
  - MAX\_ITERATIONS, COLOR\_RED
- Zmienne o większym zasięgu powinny mieć dłuższe nazwy, a o mniejszym zasięgu krótsze nazwy.
- Zmienne iterujące powinny być nazywane: *i, j, k..*
  - `for (int i = 0; i < nTables; i++) { ... }`

# Kolejność deklaracji w klasach/interfejsach

- Dokumentacja klasy/interfejsu.
- Deklaracja klasy/interfejsu.
- Zmienne statyczne (public, protected, package - bez modyfikatora, private)
- Zmienne instancji (public, protected, package - bez modyfikatora, private)
- Konstruktory.
- Metody (w dowolnej kolejności).

# Modyfikatory metod

- `<access>` `static` `abstract` `synchronized` `<unusual>`  
`final` `native`
- Modyfikator dostępu (access modifier) powinien być zawsze jako pierwszy.

```
public static double square(double a);
```

```
// NOT: static public double square(double a);
```

# Zmienne klas (class variables)

- Nie powinny być dostępne publicznie (declared public).  
private String name;
- W Javie korzystamy z „access fuctions”  
`public String getName() { return name; }`
- W C# mamy Properties:  
`public String Name { get { return name; } }`
- Wyjątek: klasy będące strukturami danych (bez ich przetwarzania)
- Minimalny zasięg; zmienne powinny być ‘żywe’ przez jak najkrótszy czas.

# Komentarze

- Nie opisujemy oczywistych rzeczy.  
`// Now we increase size by one.`  
`size = size + 1;`
- Nazwy metod oraz zmiennych powinny być zrozumiałe.
- Jak powinny wyglądać:
  - `// This is a comment`
  - `/**`
    - `* This is a javadoc`
    - `* comment`
    - `*/`
  - `/// This is a C# doc comment`

# Konwencje stylu

# Wcięcia

```
while (!done) {  
    doSomething();  
    done = moreToDo();  
}
```

```
while (!done)  
{  
    doSomething();  
    done = moreToDo();  
}
```

# Sprawdzanie poprawności danych

```
void changeScore(short numPoints)
{
    score += numPoints;
}
```

```
void changeScore(short numPoints)
{
    if (numPoints > 0)
    {
        score += numPoints;
    }
    // else maybe some error message
}
```

# Magic Numbers

---

```
private static final int TEAM_SIZE = 11;
```

```
...
```

```
Player[] players = new Player[TEAM_SIZE];
```

```
// NOT: Player[] players = new Player[11];
```

# Ułamki

```
double total = 0.0;
```

```
// NOT: double total = 0;
```

```
double speed = 3.0e8;
```

```
// NOT: double speed = 3e8;
```

```
double sum;
```

```
...
```

```
sum = (a + b) * 10.0;
```

# Mniejsze jest lepsze

---

- Pisać jak najkrótsze funkcje.

# Niepotrzebne zmienne

```
public int add(int a, int b) {  
    int sum = a + b;  
    return sum;  
}
```

```
public int add(int a, int b) {  
    return (a + b);  
}
```

# Niepotrzebne zmienne

```
public String translateAge(int age) {
    String result;
    String child = "child";
    String teen = "teen";
    String adult = "adult";
    String middleAged = "middle aged";
    String old = "old";
    if( age < 13 )
        result = child;
    else if( age < 20 )
        result = teen;
    else if( age < 40 )
        result = adult;
    else if( age < 65 )
        result = middleAged;
    else
        result = old;
    return result;
}
```

```
public String translateAge(int age) {
    if( age < 13 )
        return "child";
    else if( age < 20 )
        return "teen";
    else if( age < 40 )
        return "adult";
    else if( age < 65 )
        return "middle aged";
    else
        return "old";
}
```

# Nadmiarowe warunki

```
public void sayHello(String name)
{
    if(name.equals(""))
    {
        return;//Don't do anything
    }
    else
    {
        System.out.println(
            "Hello, "+ name);
    }
}
```

```
public void sayHello(String name)
{
    if(name.equals(""))
    {
        return;//Don't do anything
    }
    System.out.println("Hello, " +
        name)
}
```

# Zwracanie wartości boolean

```
public boolean isPositive(int n) {  
    if( n > 0 )  
        return true;  
    else  
        return false;  
}
```

```
public boolean isPositive(int n) {  
    return (n > 0);  
}
```

# Optymalizacja

jak pisać by działało efektywnie

# „Poziomy” na których może się pojawić optymalizacja

- Projekt (design).
- Kod źródłowy (source code).
- Kompilacja (compile).
- Asemblacja (assembly).
- Run time

# Rules of Program Optimisation

---

1. Don't do it.
2. (For experts only!): Don't do it yet.

Michael A. Jackson

# Pętle

- Stosować typ **int** jako indeks pętli.
- Stosować odpowiednich metod do kopiowania tablic.
- Unikać wywołań metod w pętlach.
- Obsługiwać wyjątki poza pętlą.

# Pętle - kopiowanie tablic

```
for (int j = 0; j < a.length; j++) {  
    b[j] = a[j];  
}
```

```
int size = a.length  
for (int j = 0; j < size; j++) {  
    b[j] = a[j];  
}
```

```
System.arraycopy(a, 0, b, 0, a.length); // Java  
Array.Copy(a, b, a.Length); // C#
```

# Pętle - kopiowanie tablic

Liczba elementów: $2^{24}$	Java	C#
kopiowanie w pętli z wywołaniem metody length	204 ms	698 ms
kopiowanie w pętli bez wywołania metody length	161 ms	620 ms
<code>System.arraycopy</code> / <code>Array.Copy</code>	126 ms	203 ms

# Tworzenie obiektów

- Unikać tworzenia obiektów w pętlach.
- Korzystać z tzw. *String literals* zamiast tworzyć nowe obiekty (słowo kluczowe **new**).
  - `String str2= "Hello"; //String literal`
  - `String str3 = new ("Hello"); //String Object`
- Usuwać referencje do obiektów.
  - `obj = null;`
- Nie tworzyć zbyt długich łańcuchów dziedziczenia (inheritance chains).
- Dostęp do zmiennych lokalnych jest szybszy niż do zmiennych klasy.
- Korzystać z tzw. *lazy evaluation*, czyli opóźniać tworzenie obiektów jeśli to tylko możliwe.

# String.concat() vs StringBuilder.append()

```
// String Concatenation using + operator
String result = "";
for (int i = 0; i < testNo; i++) {
    result += "hello";
}
```

```
// String Concatenation using StringBuilder
StringBuilder result1 = new StringBuilder();
for (int i = 0; i < testNo; i++) {
    result1.append("hello");
}
```

# String.concat() vs StringBuilder.append()

Liczba powtórzeń: 5000	Java	C#
String.concat()	596 ms	178 ms
StringBuilder.append()	1 ms	1 ms

# StringBuilder()

```
// String concat using StringBuilder
StringBuilder sb1 = new StringBuilder();
for (int i = 0; i < testNo; i++) {
    sb1.append("hello");
}
```

```
// String concat using initialized StringBuilder
StringBuilder sb2 = new StringBuilder(testNo * 5);
for (int i = 0; i < testNo; i++) {
    sb2.append("hello");
}
```

# StringBuilder()

Liczba powtórzeń: $2^{20}$	Java	C#
StringBuilder ()	306	287
StringBuilder (length)	166	201

# String.concat() vs StringBuilder.append()

```
//Test the String Concatination
for (int i = 0; i < testNo; i++) {
    String result = "This is" + "testing the" + "difference" +
                    "between" + "String" + "and" + "StringBuffer";
}

//Test the StringBuffer Concatination
for (int i = 0; i < testNo; i++) {
    StringBuilder result = new StringBuilder();
    result.append("This is");
    result.append("testing the");
    result.append("difference");
    result.append("between");
    result.append("String");
    result.append("and");
    result.append("StringBuffer");
}
```

# String.concat() vs StringBuilder.append()

Liczba powtórzeń: 50000	Java	C#
String.concat()	2 ms	7 ms
StringBuilder.append()	82 ms	84 ms

# Kolekcje – wybór odpowiedniej

- Zależny od rodzaju operacji.
  - ▣ ArrayList(Java) / List(C#) vs LinkedList
- Zależny od wielkości kolekcji.
  - ▣ Zalecana inicjalizacja.
- Kolekcje „bezpieczne wątkowo” (thread safe) w Javie:
  - ▣ Vector vs **ArrayList**
  - ▣ HashTable vs **HashMap**

# Kolekcje – dodawanie elementów na początku

```
ArrayList<Integer> list1 = new ArrayList<Integer>();  
for (int i = 0; i < testNo; i++) {  
    list1.add(0, i);  
}
```

```
ArrayList<Integer> list2 = new ArrayList<Integer>(testNo);  
for (int i = 0; i < testNo; i++) {  
    list2.add(0, i);  
}
```

```
LinkedList<Integer> list3 = new LinkedList<Integer>();  
for (int i = 0; i < testNo; i++) {  
    list3.addFirst(i);  
}
```

# Kolekcje – dodawanie elementów na początku

Liczba elementów: 20000	Java	C#
ArrayList bez inicjalizacji	438 ms	387 ms
ArrayList z inicjalizacją	434 ms	341 ms
LinkedList	4 ms	9 ms

# Operacje I/O

```
InputStream in = new FileInputStream(fileFrom);
OutputStream out = new FileOutputStream(fileTo);
while (true) {
    int bytedata = in.read();
    if (bytedata == -1) {
        break;
    }
    out.write(bytedata);
}
```

```
InputStream inBuffer = new BufferedInputStream(new FileInputStream(fileFrom));
OutputStream outBuffer = new BufferedOutputStream(new FileOutputStream(fileTo));
while (true) {
    int bytedata = inBuffer.read();
    if (bytedata == -1) {
        break;
    }
    outBuffer.write(bytedata);
}
```

```
InputStream inCustom = new FileInputStream(fileFrom);
OutputStream outCustom = new FileOutputStream(fileTo);
int availableLength = inCustom.available();
byte[] totalBytes = new byte[availableLength];
int bytedata = inCustom.read(totalBytes);
outCustom.write(totalBytes);
```

# Operacje I/O

```
Stream in = new FileStream(fileFrom, FileMode.Open);
Stream out = new FileStream(fileTo, FileMode.Create);
while (true) {
    int bytedata = in.ReadByte();
    if (bytedata == -1) {
        break;
    }
    out.WriteByte((byte)bytedata);
}
```

```
Stream inBuffer = new BufferedStream(new FileStream(fileFrom, FileMode.Open));
Stream outBuffer = new BufferedStream(new FileStream(fileTo, FileMode.Create));
while (true) {
    int bytedata = inBuffer.ReadByte();
    if (bytedata == -1) {
        break;
    }
    outBuffer.WriteByte((byte)bytedata);
}
```

```
Stream inCustom = new FileStream(fileFrom, FileMode.Open);
Stream outCustom = new FileStream(fileTo, FileMode.Create);
long availableLength = inCustom.Length;
byte[] totalBytes = new byte[availableLength];
int bytedata = inCustom.Read(totalBytes, 0, (int)availableLength);
outCustom.Write(totalBytes, 0, (int)availableLength);
```

# Operacje I/O

wielkość pliku: 613kB	Java	C#
domyślny strumień	5135 ms	61 ms
zbuforowany strumień	2859 ms	37 ms
własny bufor	16 ms	25 ms

# Podsumowanie

- Oszczędzaj swój czas.
- 90% czasu zajmuje wykonanie 10% kodu programu.
- Złożone algorytmy i struktury są odpowiednie dla większej ilości danych, proste posłużą nam lepiej dla małej ilości elementów.
- Optymalizacja może powodować zmniejszenie czytelności kodu, przez to często jest wykonywana na końcu.
  - profiler / benchmark
- Coś za coś (trade-offs).
- Zdrowy rozsądek.

Dziękuję za uwagę

# Bibliografia

- <http://java.sun.com/docs>
- <http://www.ibm.com/developerworks>
- <http://msdn.microsoft.com/en-us/practices>
- <http://en.wikipedia.org/wiki>
- <http://www.javaworld.com>
- <http://www.codinghorror.com/blog>
- <http://www.dotnetspider.com/tutorials/BestPractices.aspx>
- <http://www.cs.indiana.edu/~sstamm/c212/c212-sp04/dos-and-donts.html>
- <http://geosoft.no/development>