

Podstawy języka JAVA

Wykład nr 6 – Deklaracje i nadpisywanie metod w języku Java. Komentarze dokumentacyjne w języku Java.
Debugowanie aplikacji Java.



Java – deklaracje metod

- **Metody** są zgrupowanymi zestawami instrukcji do wykonania.
- Metody wykorzystuje się w przypadku, kiedy określony zestaw instrukcji ma zostać wykorzystany w kilku miejscach w aplikacji – dzięki temu, jeżeli w zadanym zestawie instrukcji jest błąd to należy go poprawić w jednym miejscu, a nie wielu.
- Metody służą również zmniejszeniu złożoności programów tj. bez odpowiedniego podziału kodu, znalezienie odpowiedniego fragmentu zajmuje znacznie więcej czasu i trudniej go odnaleźć.



Java – deklaracje metod

- Definicja metody wygląda następująco:

```
<typ_zwracany> NazwaMetody (opcjonalne_argumenty)  
{  
    ciało metody  
}
```

- Typ_zwracany** określa typ zwracanej wartości z metody.
- NazwaMetody** służy do jej wywoływania oraz identyfikacji.
- Opcjonalne argumenty umożliwiają przekazywanie wartości do metody.
- Każdy argument musi posiadać **typ oraz nazwę**.



Java – deklaracje metod

- **Ciało metody** to zestaw instrukcji, które zostaną wykonane przy wywołaniu metody.
- W celu zwrócenia wartości z metody wykorzystuje się słowo kluczowe **return**, po którym podaje się wartość do zwrócenia.
- Metoda może również nie zwracać żadnych wartości (np. zapis informacji do pliku) i w takim przypadku jej zwracany typ określa się słowem kluczowym **void**.
- W Java przed deklaracją metody stosuje się również **modyfikator dostępu** tj. **public**, **private**



Java – deklaracje metod

- W Java istnieje możliwość zagnieżdżania wywołania metod tzn. można przekazać wynik wykonania jednej metody jako argument dla drugiej metody.

```
public class Class1 {  
  
    int i;  
    public Class1()  
    {  
        i=100;  
    }  
    public void setIterator(int i)  
    {  
        this.i=i+300;  
    }  
    public int getIterator()  
    {  
        return this.i;  
    }  
}
```

```
public class MainClass {  
  
    public static void main(String[] args) {  
        Class1 o = new Class1();  
        o.setIterator(o.getIterator());  
        System.out.println(o.getIterator());  
    }  
}
```



Java – nadpisywanie metod

- **Nadpisywanie metod (*ang. Method overriding*)** jest cechą, która pozwala klasie pochodnej (dziedziczącej) na specyficzną implemencję metody z klasy bazowej (po której dziedziczy).
- Nadpisanie metody w klasie pochodnej **powoduje zastąpienie metody z klasy bazowej za pomocą metody o tej samej nazwie, tych samych parametrach i tym samym zwracanym typie. Różnica natomiast może polegać na innym ciele metody.**
- Wersja metody, która zostanie wykonana określana jest przez obiekt, na rzecz którego zostanie wywołana metoda.



Java – nadpisywanie metod

- Jeżeli obiekt jest typu klasy bazowej, to zostanie wywołana metoda z klasy bazowej.
- Jeżeli obiekt jest typu klasy pochodnej, to zostanie wywołana metoda z klasy pochodnej (nadpisana).



Java – nadpisywanie metod

```
public class Class1 {  
  
    int i;  
    public Class1()  
    {  
        i=100;  
    }  
    public void setIterator(int i)  
    {  
        this.i=i+300;  
    }  
    public int getIterator()  
    {  
        return this.i;  
    }  
}
```

```
public class Class2 extends Class1 {  
  
    int i;  
    public Class2()  
    {  
        i=0;  
    }  
    public void setIterator(int i)  
    {  
        this.i=i+3000;  
    }  
    public int getIterator()  
    {  
        return this.i+500;  
    }  
}
```

```
public class MainClass {  
  
    public static void main(String[] args) {  
        Class1 o = new Class1();  
        Class2 o2 = new Class2();  
        o.setIterator(o.getIterator());  
        o2.setIterator(o2.getIterator());  
        System.out.println(o.getIterator());  
        System.out.println(o2.getIterator());  
    }  
}
```

400
4000



Java – nadpisywanie metod

- W klasie pochodnej można wykorzystać słowo kluczowe `super` i po kropce podać nazwę metody z klasy bazowej.
- Spowoduje to wykonanie metody o wskazanej nazwie z klasy bazowej.
- Po słowie kluczowym `super` można podać inną metodę z klasy bazowej niż ta, która została nadpisana.



Java – nadpisywanie metod

```
public class Class1 {  
  
    public Class1()  
    {  
    }  
    public void Method()  
    {  
        System.out.println("Metoda klasy Class1");  
    }  
}
```

```
public class MainClass {  
  
    public static void main(String[] args) {  
        Class1 o1 = new Class1();  
        Class2 o2 = new Class2();  
        o1.Method();  
        o2.Method();  
    }  
}
```

```
public class Class2 extends Class1 {  
  
    public Class2()  
    {  
    }  
    public void Method()  
    {  
        super.Method();  
        System.out.println("Metoda klasy Class2");  
    }  
}
```

Metoda klasy Class1
Metoda klasy Class1
Metoda klasy Class2



Java – nadpisywanie metod

- Zasady nadpisywania metod w Java:
 - Metoda może zostać nadpisana jedynie w klasie pochodnej, a nie w klasie bazowej.
 - Lista parametrów w metodzie nadpisującej powinna być dokładnie taka sama jak w metodzie nadpisywanej.
 - Typ zwracany przez nadpisaną metodę powinien być dokładnie taki sam jak typ zwracany przez metodę nadpisywaną (typ zwracany może też być podtypem typu z metody bazowej).
 - Modyfikator dostępu w metodzie nadpisywanej nie może być bardziej restrykcyjny niż modyfikator dostępu metody nadpisywanej.



Java – nadpisywanie metod

- Modyfikator dostępu w metodzie nadpisywanej nie może być bardziej restrykcyjny niż modyfikator dostępu metody nadpisywanej – np. jeżeli metoda z klasy bazowej ma modyfikator public, to metoda nadpisująca z klasy pochodnej nie może być protected.
- Metody zadeklarowane jako statyczne nie mogą być nadpisywane, ale mogą zostać przededefiniowane.
- Metody zadeklarowane jako final nie mogą być nadpisywane.
- Konstruktor nie może być nadpisany.



Java – komentarze dokumentacyjne

- JDK zawiera w sobie bardzo przydatne narzędzie, który jest javadoc.
- Pozwala ona na generowanie dokumentacji w formacie plików HTML z plików źródłowych.
- Profesjonalną dokumentację do napisanych klas można wykonać za pomocą javadoc przy wykorzystaniu komentarzy zaczynających się od znaków `/**`.
- Javadoc umożliwia przechowywanie kodu oraz dokumentacji do wytworzonego kodu w jednym miejscu. Jest to bardzo wygodne i praktyczne rozwiązanie.



Java – komentarze dokumentacyjne

- Jeżeli napisany kod i jego dokumentacja znajdowałyby się w innych miejscach to prędzej czy później doszłoby do rozbieżności pomiędzy nimi.
- Dzięki temu, że komentarze dokumentacyjne znajdują się w tym samym pliku co kod źródłowy to aktualizacja obu tych elementów jest znacznie ułatwiona.



Java – komentarze dokumentacyjne

- Narzędzie javadoc pobiera informacje o następujących elementach:
 - Pakiety,
 - Klasy i interfejsy publiczne,
 - Publiczne i chronione (protected) metody i konstruktory,
 - Pola publiczne i chronione.
- Każda z powyższych konstrukcji może być i powinna być opatrzona komentarzem.
- Komentarz powinien znajdować się bezpośrednio nad tym czego dotyczy.
- Początek komentarza określa sekwencja symboli `/**`, a koniec komentarza określa sekwencja symboli `*/`



Java – komentarze dokumentacyjne

- W komentarzu można umieścić dowolny tekst oraz specjalne znaczniki dokumentacyjne.
- Znaczniki dokumentacyjne rozpoczynają się od symbolu **@**.
- Przykładowo są to **@author** (autor metody), **@param** (parametry metody).
- Dobrą praktyką jest, aby pierwsze zdanie komentarza było streszczeniem.
- Narzędzie **javadoc** automatycznie generuje strony streszczeń, bazując na pierwszych zdaniach w komentarzach.



Java – komentarze dokumentacyjne

- W tekście komentarza można wykorzystywać również znaczniki HTML np. `` (pogrubienie), `` (wstawienie obrazu).
- Należy unikać natomiast znaczników związanych z nagłówkami np. `<h1>` oraz poziomymi kreskami `<hr>`.
- Mogą one spowodować złe formatowanie dokumentacji napisanych elementów w Java.
- Jeżeli programista chce załączyć dodatkowe elementy np. pliki graficzne to powinno się je przechowywać w folderze doc-files.
- Dodatkowo nazwa tego folderu powinna pojawić się w ścieżce odnośnika np. pliku graficznego.



Java – komentarze do klas

- Komentarze klasy muszą znajdować się za instrukcjami import i bezpośrednio przed definicją klasy.

```
/**  
 * Klasa wykorzystywana do zaprezentowania działania nadpisywania metod.  
 */  
public class Class1 {  
    public Class1()  
    {  
    }  
    public void Method()  
    {  
        System.out.println("Metoda klasy Class1");  
    }  
}
```



Java – komentarze do metod

- Komentarz do metody musi znajdować się bezpośrednio przed metodą, której dotyczy.
- Oprócz znacznika ogólnego przeznaczenia w komentarzu do metody można wykorzystać dodatkowe znaczniki:
 - **@param opis zmiennej** – dodaje pozycję do sekcji Parameters metody. Opis może znajdować kilka wierszy i zawierać znaczniki HTML. Wszystkie znaczniki @param dotyczące wybranej metody powinny znajdować się w jednym miejscu.
 - **@return opis** – opis zwracanej wartości. Powoduje dodanie sekcji Returns. Opis może zajmować kilka wierszy i zawierać znaczniki HTML.
 - **@throws opis** – powoduje dodanie informacji, że dana metoda może spowodować wyjątek.



Java – komentarze do metod

```
/**
 *
 * @param i - iterator do pierwszej pętli
 * @param j - iterator do drugiej pętli
 * @param s - nazwa pliku do zapisu wyników
 * @return zwraca sumę parametrów i oraz j
 */
public int Method(int i, int j, String s)
{
    return i+j;
}
```



Java – komentarze do pól

- Komentarze do pól są potrzebne tylko do pól publicznych.
- Oznacza to na ogół zmienne statyczne.

```
/**  
 * Stała wykorzystywana do obliczeń  
 */  
public static final k=1000;
```



Java – komentarze ogólne

- W komentarzach dokumentacji klas można wykorzystywać następujące znaczniki:
- **@author imię i nazwisko** – znacznik powodujący dodanie pozycji Author. W przypadku występowania kilku autorów, można wykorzystać ten znacznik kilkakrotnie.
- **@version tekst** – znacznik powodujący dodanie pozycji Version. Tekst może być opisem aktualnej wersji.
- We wszystkich komentarzach można wykorzystać również następujące znaczniki:
- **@since – tekst** – znacznik powodujący dodanie pozycji Since. Określa ona zazwyczaj w jakiej wersji wprowadzono funkcję.



Java – komentarze ogólne

- **@deprecated tekst** – dodaje komentarz informujący, że dana klasa, metoda lub zmienna nie powinny być używane. Dodatkowo tekst powinien zawierać informacje o zamienniku, czyli jaka nowa klasa, metoda lub zmienna powinna być używana.
- **@see** – dodaje sekcję SeeAlso, która pozwala na odwołania do innych dokumentacji.



Java – komentarze do pakietów

- Komentarze do klas, metod i zmiennych znajdują się bezpośrednio w plikach źródłowych Java pomiędzy ciągami znaków `/**` i `*/`.
- Generowanie komentarzy do pakietów wymaga utworzenia osobnego pliku do każdego katalogu pakietu.
- Są dwie możliwości na realizację powyższego zapisu tj:
 - Utworzenie pliku HTML o nazwie `package.html`. Z tego pliku zostanie pobrane wszystko co znajduje się pomiędzy znacznikami **<body>** i **</body>**.
 - Utworzenie pliku Java o nazwie **package-info.java**. Na początku tego pliku musi się znajdować komentarz i instrukcja `package`.



Java – komentarze do pakietów

- Istnieje również możliwość utworzenia ogólnego komentarza do wszystkich plików źródłowych.
- Powinien się on znajdować w pliku o nazwie **overview.html**, który musi być zlokalizowany w katalogu macierzystym wszystkich plików źródłowych.
- Z tego pliku zostanie pobrane wszystko co znajduje się pomiędzy znacznikami **<body>** i **</body>**.
- Komentarz do wszystkich plików źródłowych zostanie wyświetlony, kiedy użytkownik kliknie opcję Overwie na pasku nawigacyjnym.



Java – generowanie dokumentacji

- W celu wygenerowania dokumentacji z komentarzy należy:
 - Przejść do katalogu z plikami źródłowymi, dla których chce się utworzyć dokumentację.
 - W celu wygenerowania dokumentacji dla jednego pakietu należy wywołać polecenie:

javadoc -d docDirectory nazwaPakietu

- W przypadku generowania dokumentacji dla kilku pakietów, należy podawać ich nazwy po spacji tj.

javadoc -d docDirectory nazwaPakietu1 nazwaPakietu2

- Jeżeli pliki znajdują się w domyślnym pakiecie to należy wywołać:

javadoc -d docDirectory *.java

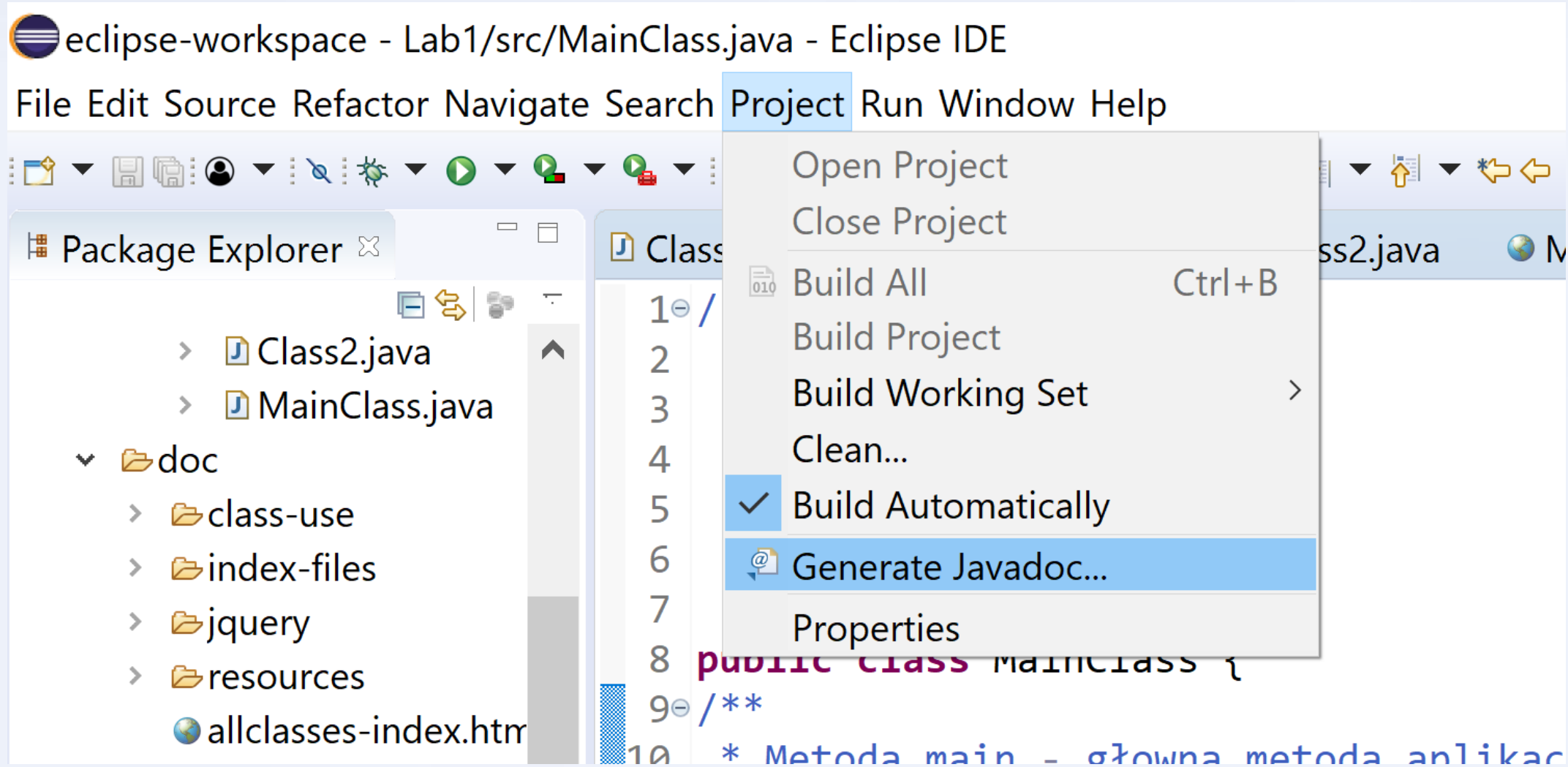


Java – generowanie dokumentacji


- W przypadku braku opcji **-d docDirectory** pliki HTML zostaną umieszczone w bieżącym katalogu.
- Takie rozwiązanie nie jest zalecane, ponieważ powoduje bałagan w projekcie i na dysku.
- Działaniem javadoc można sterować za pomocą różnych opcji, które przekazywane są do programu.
- Przykładowo opcje **-author** i **-version** powodują dodanie do dokumentacji znaczników **@author** i **@version**. Domyślnie te znaczniki są pomijane.




Java – generowanie dokumentacji - Eclipse




Java – generowanie dokumentacji - Eclipse

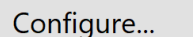
 Generate Javadoc

Javadoc Generation


 Javadoc generation may overwrite existing files

Javadoc command:

C:\Program Files\Java\jdk-11\bin\javadoc.exe 

 Configure...

Select types for which Javadoc will be generated:

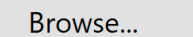
>  Lab1

Create Javadoc for members with visibility:

☐ Private ☐ Package ☐ Protected ☒ Public

Public: Generate Javadoc for public classes and members.


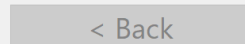
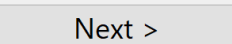
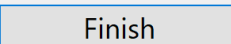
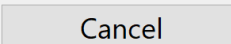
☒ Use standard doclet

Destination: C:\Users\Lukasz\eclipse-workspace\Lab1\doc 

☐ Use custom doclet


Doclet name:

Doclet class path:

  < Back  Next >  Finish  Cancel



Java – generowanie dokumentacji - Eclipse

 Generate Javadoc

Javadoc Generation
Configure Javadoc arguments for standard doclet.

☐ Document title:


Basic Options


- ☒ Generate use page
- ☒ Generate hierarchy tree
- ☒ Generate navigator bar
- ☒ Generate index
 - ☒ Separate index per letter

Document these tags

- ☒ @author
- ☒ @version
- ☒ @deprecated
 - ☒ deprecated list

Select referenced archives and projects to which links should be generated:

☐  jrt-fs.jar - not configured

☐  Lab1 - file:/C:/Users/Lukasz/eclipse-workspace/Lab1/doc/


Select All

Clear All

Browse...

☐ Style sheet:

Browse...



< Back


Next >

Finish

Cancel



Java – generowanie dokumentacji - Eclipse

 Generate Javadoc

Javadoc Generation
Configure Javadoc arguments.

☐ Overview:

VM options (prefixed with '-J', e.g. -J-Xmx180m for larger heap space):


Extra Javadoc options (path names with white spaces must be enclosed in quotes):

JRE source compatibility:

☐ Save the settings of this Javadoc export as an Ant script:

Ant Script:

☐ Open generated index file in browser





Java – generowanie dokumentacji - Eclipse

eclipse-workspace - file:///C:/Users/Lukasz/eclipse-workspace/Lab1/doc/index.html - Eclipse IDE

File Edit Navigate Search Project Run Window Help

Package Explorer

- > Class2.java
- > MainClass.java
- > doc
 - > class-use
 - > index-files
 - > jquery
 - > resources
 - allclasses-index.htm
 - allclasses.html
 - allpackages-index.h
 - Class1.html
 - Class2.html
 - constant-values.htm
 - deprecated-list.htm
 - element-list
 - help-doc.html
 - index.html
 - MainClass.html
 - member-search-inc
 - member-search-inc
 - overview-tree.html
 - package-search-inc
 - package-search-inc
 - package-summary.l
 - package-tree.html
 - package-use.html
 - script.js
 - search.js
 - stylesheet.css

Class1.java MainClass.java Class2.java <Unnamed>

file:///C:/Users/Lukasz/eclipse-workspace/Lab1/doc/package-summary.html

PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

ALL CLASSES SEARCH:

Package <Unnamed>

Class Summary

Class	Description
Class1	Klasa Class1 do testowania aplikacji
Class2	Klasa przedstawiająca nadpisywanie metod.
MainClass	Główna klasa aplikacji

PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

ALL CLASSES



Java – debugowanie aplikacji

- Każdy programista w trakcie pisania aplikacji natrafia na moment kiedy aplikacja nie działa.
- Jednym ze sposobów na poradzenie sobie z problemem jest wykorzystanie wbudowanego w środowisko **Eclipse Debuggera**.
- Debugger pozwala na wstrzymanie wykonywania kodu w wybranym przez programistę miejscu.
- Następnie można podejrzeć wartości zmiennych w tym konkretnym momencie, a następnie wykonywać aplikację np. „**krok po kroku**”.
- Dzięki temu możliwe jest odnalezienie przyczyny błędu lub nieporządkowanego zachowania aplikacji.



Java – debugowanie aplikacji

- Z pojęciem **debugger** jest również połączone pojęcie **breakpoint**.
- Jest to miejsce wskazane przez programistę, w którym działanie aplikacji zostanie „wstrzymane” w celu dalszej analizy.
- Należy pamiętać że włączenie debugowania znacznie spowolni wykonywanie aplikacji – maszyna wirtualna Java komunikuje się wtedy dużo częściej ze środowiskiem oraz nie wykonuje optymalizacji kodu.



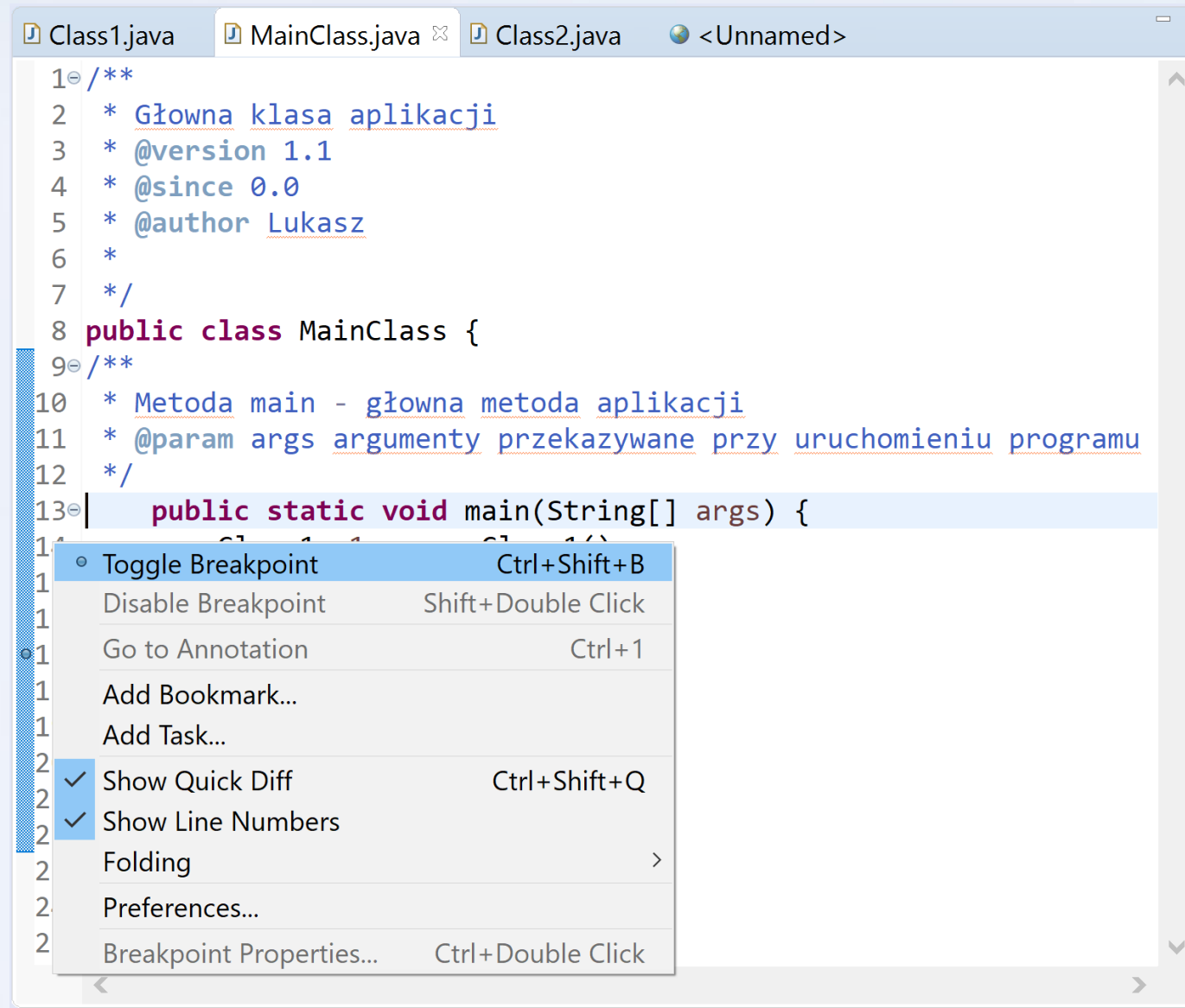
Java – debugowanie aplikacji

- Pierwszym krokiem jest ustawienie breakpoint w naszej aplikacji.
- Można to zrealizować poprzez wybranie Run->Toggle Breakpoint lub dwukrotne kliknięcie na numerze linii, w której chcemy ustawić breakpoint.

```
13  public static void main(String[] args) {  
14      Class1 o1 = new Class1();  
15      Class2 o2 = new Class2();  
16      o1.Method(10, 20, "");  
17      o2.Method(10, 10, "");
```



Java – debugowanie aplikacji



The screenshot shows an IDE window with three tabs: Class1.java, MainClass.java (active), and Class2.java. The active tab contains the following Java code:

```
1 /**
2  * Główna klasa aplikacji
3  * @version 1.1
4  * @since 0.0
5  * @author Lukasz
6  *
7  */
8 public class MainClass {
9 /**
10  * Metoda main - główna metoda aplikacji
11  * @param args argumenty przekazywane przy uruchomieniu programu
12  */
13 public static void main(String[] args) {
```

A context menu is open over line 13, listing the following options:

- Toggle Breakpoint (Ctrl+Shift+B)
- Disable Breakpoint (Shift+Double Click)
- Go to Annotation (Ctrl+1)
- Add Bookmark...
- Add Task...
- ✓ Show Quick Diff (Ctrl+Shift+Q)
- ✓ Show Line Numbers
- Folding >
- Preferences...
- Breakpoint Properties... (Ctrl+Double Click)



Java – debugowanie aplikacji

- Breakpoint należy zawsze postawić w linii, w której są operacje do wykonania.
- Nie należy stawiać **breakpoint**ów w liniach które zawierają deklaracje metod, klas lub nawias.
- Warunkiem zatrzymania aplikacji przez debugger jest wykonanie linii, w której został pozostawiony breakpoint.
- W związku z tym należy uważać na instrukcje if, w które nasza aplikacja może nie wejść (brak spełnienia warunku).
- Breakpoint powinien być ustawiany przed miejscem, w którym podejrzewamy, że aplikacja nie działa poprawnie.




Java – debugowanie aplikacji

- W jednej aplikacji można wstawić wiele breakpoint'ów.
- Aplikacja zostanie w takim przypadku zatrzymana w każdej wykonywanej linii, dla której wstawiono breakpoint.
- Należy jednak pamiętać, że aplikacja jest wstrzymywana przed wykonaniem linii, dla której wstawiono breakpoint.



Java – debugowanie aplikacji

- Po wstawieniu breakpoint'ów w określonych linii należy uruchomić aplikację w trybie debugowania.
- Służy do tego nie standardowy przycisk uruchom, a przycisk z „robakiem” (ang. Bug)  ▼
- Po napotkaniu pierwszego breakpoint'a środowisko Eclipse zapyta się czy chcemy zmienić perspektywę na debugowania (na co oczywiście należy się zgodzić).



Java – debugowanie aplikacji

Class1.javaMainClass.javaClass2.java<Unnamed>

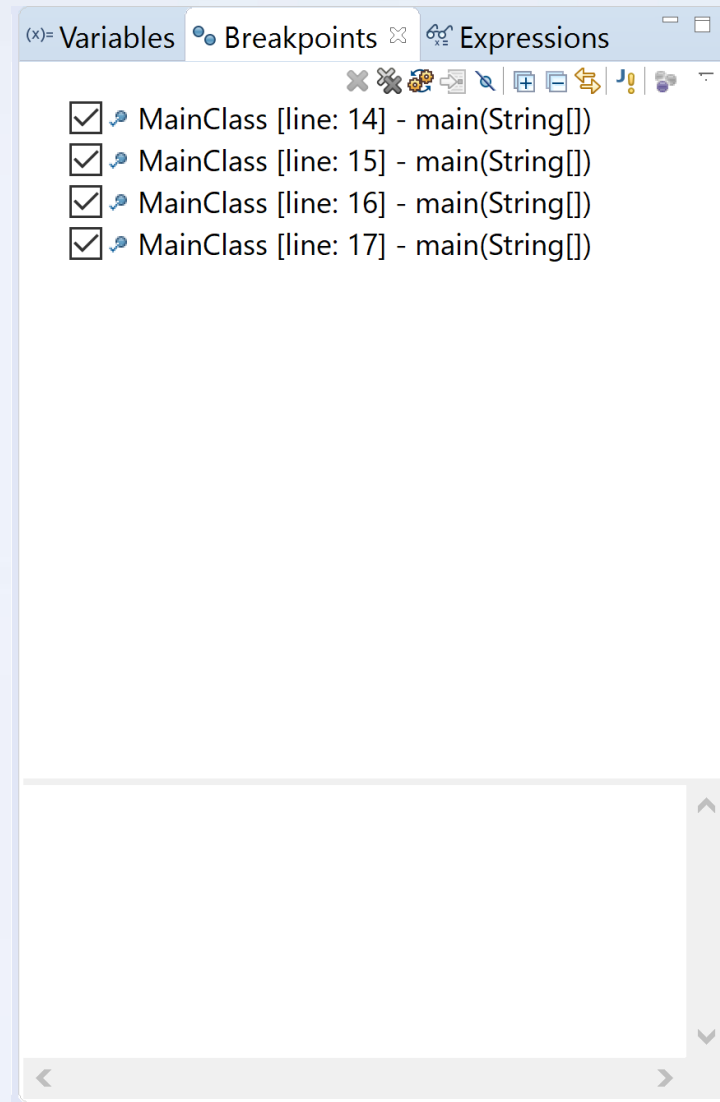
```
1 /**
2  * Główna klasa aplikacji
3  * @version 1.1
4  * @since 0.0
5  * @author Lukasz
6  *
7  */
8 public class MainClass {
9 /**
10  * Metoda main - główna metoda aplikacji
11  * @param args argumenty przekazywane przy uruchomieniu programu
12  */
13 public static void main(String[] args) {
14     Class1 o1 = new Class1();
15     Class2 o2 = new Class2();
16     o1.Method(10, 20, "");
17     o2.Method(10, 10, "");
18     while(true)
19     {
20
21     }
22 }
23
24 }
25
```

(x)= VariablesBreakpointsExpressions


Name	Value
no method return val	
args	String[0] (id=19)
o1	Class1 (id=20)
o2	Class2 (id=23)



Java – debugowanie aplikacji



Java – debugowanie aplikacji

- Za pomocą przycisków  możliwe jest dalsze sterowanie aplikacją.
- Programista ma następujące opcje:
- Uruchomić dalej aplikację bez zatrzymywania (do napotkania kolejnego breakpointa).
- Zapauzować lub wyłączyć aplikację
- **Step into** – wykonuje jedną operację (przechodzi linię dalej). Jeżeli obecna linia jest wywołaniem funkcji, następuje przejście do wnętrza funkcji.



Java – debugowanie aplikacji

- **Step over** – wykonuje jedną operację (przechodzi linię dalej). Jeżeli obecna linia jest wywołaniem funkcji, wywołuje ją i wznowia debugowanie po jej wykonaniu.
- **Step return** – kontynuuje działanie aż do końca bieżącej metody i wraca do trybu debugowania po zwróceniu wartości z tej metody.



Java – debugowanie aplikacji

- Inne sposoby radzenia sobie w przypadku niepoprawnego działania aplikacji:
 1. Wartość każdej zmiennej można wydrukować lub zarejestrować. Wykorzystując `System.out.println` (do wypisania) lub `Logger.global.info` (do zarejestrowania w dzienniku). Jeżeli wartość, którą zapisujemy lub rejestrujemy jest liczbą to zostanie przekonwertowana na łańcuch. Jeżeli jest obiektem to zostanie na jej rzecz wywołana metoda `toString`.



Java – debugowanie aplikacji

2. Kolejnym sposobem jest umieszczenie w każdej klasie metody main. W niej można umieścić namiastkę testu jednostkowego umożliwiając przetestowanie klasy w odosobnieniu.
3. W celu przeprowadzenia prawdziwych testów jednostkowych dobrze jest wykorzystać np. Junit. Jest to bardzo popularny framework testowy, która ułatwia organizację zestawów przypadków testowych. Testy powinno się przeprowadzać po wprowadzeniu zmian do klasy, a po znalezieniu błędu powinno się utworzyć dodatkowe testu.



Java – debugowanie aplikacji

4. Stos można uzyskać z każdego obiektu wyjątku za pomocą metody `printStackTrace` z klasy `Throwable`.

```
try
{
}
catch (Throwable e)
{
    e.printStackTrace();
    throw e;
}
```

5. Normalnie dane ze śledzenia stosu są wysyłane do strumienia `System.err`. Można użyć metody `printStackTrace (PrintWriter s)`, aby zapisać je do pliku.



Java – debugowanie aplikacji

6. Przydatne potrafi być zapisanie błędów programu do pliku. Są one jednak wysyłane do strumienia `System.err`, a nie `System.out`. Jeżeli chcemy błędy programu przekierować do pliku to należy program w Java uruchomić następująco:

`java Program 2> error.txt`

Spowoduje to przekierowanie strumienia `System.err` do wskazanego pliku. Jeżeli chcemy umieścić w pliku zarówno strumień `System.err` jak i `System.out` to należy program uruchomić następująco:

`java Program >& plik.txt`

Powyższe sposoby działają zarówno w systemach Linux jak i Windows



Java – debugowanie aplikacji

7. Zapisywanie danych ze śledzenia stosu nieprzechwyconych wyjątków w strumieniu System.err nie jest dobrym rozwiązaniem. Komunikaty te wprowadzają zamęt u zwykłego użytkownika, kiedy je zobaczą oraz są niedostępne kiedy są potrzebne do diagnostyki. Można zmienić procedurę nieprzechwyconych wyjątków za pomocą metody `Thread.setDefaultUncaughtExceptionHandler`.

```
Thread.setDefaultUncaughtExceptionHandler(  
    new Thread.UncaughtExceptionHandler()  
    {  
        public void uncaughtException(Thread t, Throwable e)  
        {  
            zapis informacji w pliku dziennika  
        };  
    });
```



Java – debugowanie aplikacji

8. W celu zobaczenia, jakie klasy są ładowane po kolei, należy uruchomić maszynę wirtualną Java przy użyciu opcji `-verbose`. Zostaną wtedy wydrukowane np. takie informacje:

```
[Opened /usr/local/jdk5.0/jre/lib/rt.jar]
[Opened /usr/local/jdk5.0/jre/lib/jsse.jar]
[Opened /usr/local/jdk5.0/jre/lib/jce.jar]
[Opened /usr/local/jdk5.0/jre/lib/charsets.jar]
[Loaded java.lang.Object from shared objects file]
[Loaded java.io.Serializable from shared objects file]
[Loaded java.lang.Comparable from shared objects file]
[Loaded java.lang.CharSequence from shared objects file]
[Loaded java.lang.String from shared objects file]
[Loaded java.lang.reflect.GenericDeclaration from shared objects file]
[Loaded java.lang.reflect.Type from shared objects file]
[Loaded java.lang.reflect.AnnotatedElement from shared objects file]
[Loaded java.lang.Class from shared objects file]
[Loaded java.lang.Cloneable from shared objects file]
...
```



Java – debugowanie aplikacji

9. Opcja `-Xlint` pozwala znaleźć najczęstsze problemy z kodem. Przykładowo jeżeli skompilujemy program `javac -Xlint:fallthroug` to kompilator zgłosi brakujące instrukcje `break` w instrukcjach `switch`.

<code>-Xlint: lub -Xlint:all</code>	Przeprowadza wszystkie testy.
<code>-Xlint:depreciation</code>	Działa tak samo jak opcja <code>-depreciation</code> — wyszukuje odradzane metody.
<code>-Xlint:fallthrough</code>	Szuka brakujących instrukcji <code>break</code> w instrukcjach <code>switch</code> .
<code>-Xlint:finally</code>	Ostrzega o klauzulach <code>finally</code> , które nie mogą się normalnie zakończyć.
<code>-Xlint:none</code>	Nie przeprowadza żadnego testu.
<code>-Xlint:path</code>	Sprawdza, czy wszystkie katalogi na ścieżce klas istnieją.
<code>-Xlint:serial</code>	Ostrzega o serializowalnych klasach bez <code>serialVersionUID</code>
<code>-Xlint:unchecked</code>	Ostrzega przed niebezpiecznymi konwersjami pomiędzy typami uogólnionymi a surowymi

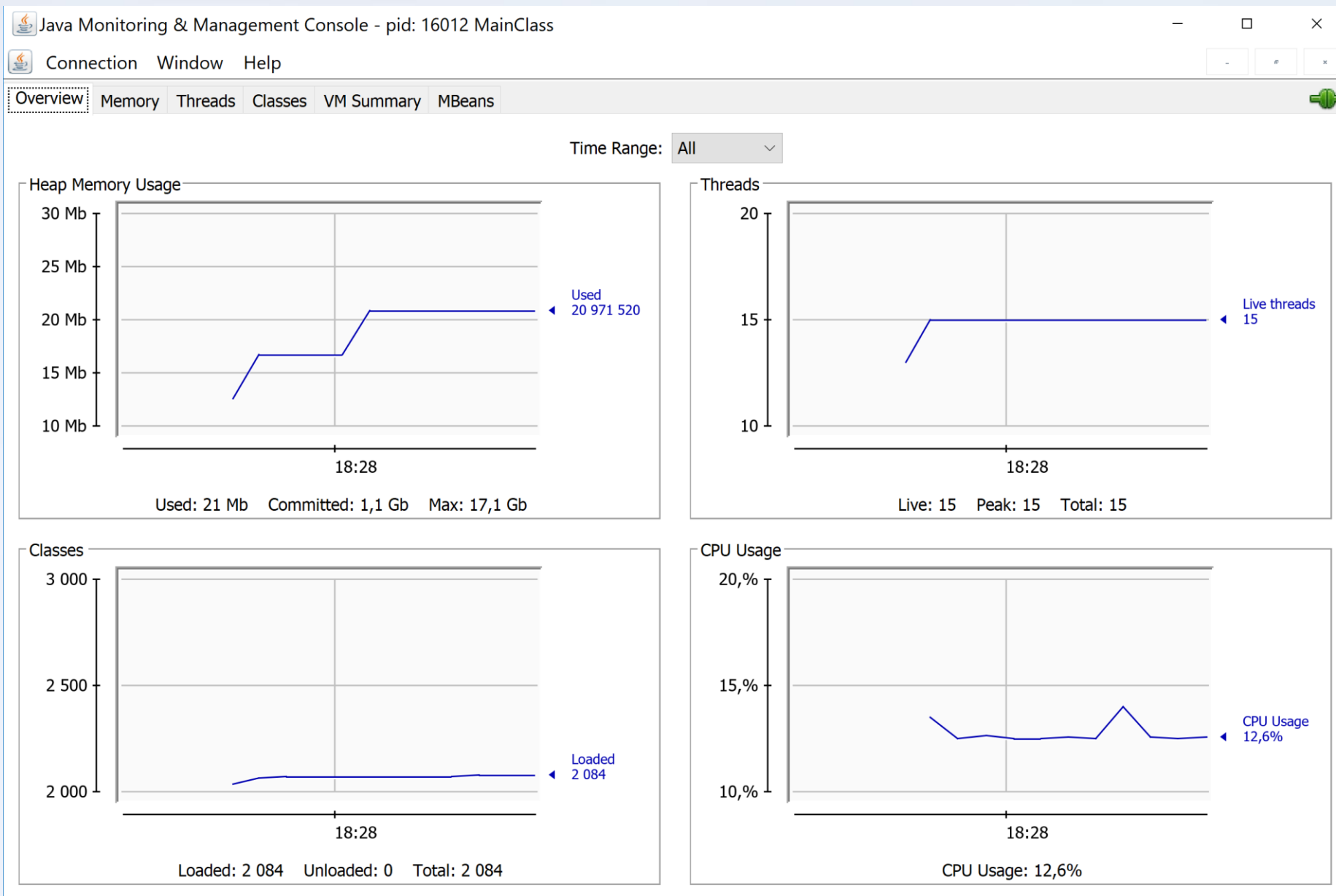


Java – debugowanie aplikacji

10. Maszyna wirtualna Javy może monitorować aplikacje i zarządzać nimi. Polega to na instalacji agentów w maszynie wirtualnej Javy, które śledzą zużycie pamięci, wykorzystanie wątków, ładowanie klas itp. Funkcje te są bardzo przydatne przy zastosowaniach serwerowych, kiedy aplikacje muszą pracować dłuższy okres czasu. Przykładowym narzędziem jest tutaj jconsole, które wyświetla statystyki dotyczące działania maszyny wirtualnej. Należy odszukać w systemie operacyjnym id procesu (w Linux za pomocą polecenia ps, w Windows za pomocą Menadżera Zadań). Następnie należy wywołać polecenie jconsole IDProcesu.



Java – debugowanie aplikacji



Java – debugowanie aplikacji

11. Uruchomienie maszyny wirtualnej ze znacznikiem `-Xprof` powoduje wywołanie prostego narzędzia profilującego, które śledzi najczęściej wywoływane metody w kodzie. Informacje te są wysyłane do strumienia `System.out` i informuje m.in. o tym, które metody zostały skompilowane przez kompilator JIT.



Kolejny wykład:
Konstruktory i ich przeciążanie w języku Java.
Bloki inicjalizujące i niszczenie obiektów.
Dziedziczenie w języku Java.



DZIĘKUJĘ ZA UWAGĘ

