

# Podstawy Java – dzień

4

v3.1

# Plan

- Wprowadzanie danych
- Operacje na plikach
- XML
- Debugowanie w IDE
- External jar

# Wprowadzanie danych

# Wprowadzanie danych

Jednym z podstawowych zadań programu jest pobieranie danych od użytkownika. Możemy w tym celu użyć:

- klasy **Scanner** z pakietu **java.util**,
- elementów dialogowych z pakietów graficznych,
- formularzy **html**.

Podczas kursu omówimy sobie pobieranie danych ze standardowego wejścia (konsoli) oraz przy pomocy formularzy **html**.

# Scanner

Aby pobrać dane musimy najpierw stworzyć obiekt klasy **Scanner**:

```
Scanner scan = new Scanner(System.in);
```

Pamiętać należy o dodaniu odpowiedniego importu:

```
import java.util.Scanner;
```

Jak już wiemy bez tego nasza klasa nie zostanie prawidłowo rozpoznana.

# Scanner

Aby pobrać dane musimy najpierw stworzyć obiekt klasy **Scanner**:

```
Scanner scan = new Scanner(System.in);
```

Źródło danych (oznacza standardowe wejście) – umożliwia pobranie danych wprowadzonych w konsoli podczas działania naszego programu.

Pamiętać należy o dodaniu odpowiedniego importu:

```
import java.util.Scanner;
```

Jak już wiemy bez tego nasza klasa nie zostanie prawidłowo rozpoznana.

# Scanner

Klasa **Scanner** posiada metodę **next()**, która wczyta pierwszy napotkany napis:

```
String next = scan.next();
```

Kolejne wywołanie metody **next()** pobierze następny wprowadzony napis.

Metoda **nextLine()** pobierze całą wprowadzoną linię:

```
String nextLine = scan.nextLine();
```

Klasa **Scanner** posiada również kilka przydatnych metod, służących do pobierania wartości określonego typu np.:

```
int nextInt = scan.nextInt();  
double nextDouble = scan.nextDouble();  
byte nextByte = scan.nextByte();
```

W przypadku podania niepoprawnej wartości otrzymamy wyjątek **InputMismatchException**. Należy pamiętać, by zapewnić obsługę wyjątku.

# Scanner

Klasa **Scanner** posiada metodę **next()**, która wczyta pierwszy napotkany napis:

```
String next = scan.next();
```

Kolejne wywołanie metody **next()** pobierze następny wprowadzony napis.

Metoda **nextLine()** pobierze całą wprowadzoną linię:

```
String nextLine = scan.nextLine();
```

Klasa **Scanner** posiada również kilka przydatnych metod, służących do pobierania wartości określonego typu np.:

```
int nextInt = scan.nextInt();  
double nextDouble = scan.nextDouble();  
byte nextByte = scan.nextByte();
```

W przypadku podania niepoprawnej wartości otrzymamy wyjątek **InputMismatchException**. Należy pamiętać, by zapewnić obsługę wyjątku.

Pobierze wartość typu **int**.



# Scanner

Klasa **Scanner** posiada metodę **next()**, która wczyta pierwszy napotkany napis:

```
String next = scan.next();
```

Kolejne wywołanie metody **next()** pobierze następny wprowadzony napis.

Metoda **nextLine()** pobierze całą wprowadzoną linię:

```
String nextLine = scan.nextLine();
```

Klasa **Scanner** posiada również kilka przydatnych metod, służących do pobierania wartości określonego typu np.:

```
int nextInt = scan.nextInt();  
double nextDouble = scan.nextDouble();  
byte nextByte = scan.nextByte();
```

W przypadku podania niepoprawnej wartości otrzymamy wyjątek **InputMismatchException**. Należy pamiętać, by zapewnić obsługę wyjątku.

Pobierze wartość typu **int**.

Pobierze wartość typu **double**.

# Scanner

Klasa **Scanner** posiada metodę **next()**, która wczyta pierwszy napotkany napis:

```
String next = scan.next();
```

Kolejne wywołanie metody **next()** pobierze następny wprowadzony napis.

Metoda **nextLine()** pobierze całą wprowadzoną linię:

```
String nextLine = scan.nextLine();
```

Klasa **Scanner** posiada również kilka przydatnych metod, służących do pobierania wartości określonego typu np.:

```
int nextInt = scan.nextInt();  
double nextDouble = scan.nextDouble();  
byte nextByte = scan.nextByte();
```

W przypadku podania niepoprawnej wartości otrzymamy wyjątek **InputMismatchException**. Należy pamiętać, by zapewnić obsługę wyjątku.

Pobierze wartość typu **int**.

Pobierze wartość typu **double**.

Pobierze wartość typu **byte**.

# Scanner – przykład

```
System.out.println("Podaj liczbę całkowitą:");
Scanner scan = new Scanner(System.in);
try {
    int number = scan.nextInt();
    System.out.println(number);
} catch (InputMismatchException e) {
    System.out.println("Niepoprawne dane");
}
```

# Scanner – przykład

```
System.out.println("Podaj liczbę całkowitą:");
Scanner scan = new Scanner(System.in);
try {
    int number = scan.nextInt();
    System.out.println(number);
} catch (InputMismatchException e) {
    System.out.println("Niepoprawne dane");
}
```

Pobieramy wartość typu **int**, a następnie wyświetlamy ją.

# Scanner – przykład

```
System.out.println("Podaj liczbę całkowitą:");
Scanner scan = new Scanner(System.in);
try {
    int number = scan.nextInt();
    System.out.println(number);
} catch (InputMismatchException e) {
    System.out.println("Niepoprawne dane");
}
```

Pobieramy wartość typu **int**, a następnie wyświetlamy ją.

Obsługujemy ewentualne wystąpienie wyjątku.

# Scanner – przykład

```
System.out.print("Podaj liczbę całkowitą: ");
Scanner scan = new Scanner(System.in);
while (!scan.hasNextInt()) {
    scan.next();
    System.out.print("Nieprawidłowe dane. Podaj jeszcze raz:");
}
int number = scan.nextInt();
System.out.println("Podałeś liczbę: " + number);
```

Klasa **Scanner** posiada również analogiczne metody dla innych typów:

- `scan.hasNextBoolean();`
- `scan.hasNextFloat();`
- `scan.hasNextLong();`

# Scanner – przykład

```
System.out.print("Podaj liczbę całkowitą: ");
Scanner scan = new Scanner(System.in);
while (!scan.hasNextInt()) {
    scan.next();
    System.out.print("Nieprawidłowe dane. Podaj jeszcze raz:");
}
int number = scan.nextInt();
System.out.println("Podałeś liczbę: " + number);
```

Pobieramy wartości do momentu aż wystąpi wartość typu **int**. Metoda **hasNextInt()** zwraca **true**, jeżeli wartość wczytana jest poprawną wartością typu **int**.

Klasa **Scanner** posiada również analogiczne metody dla innych typów:

- **scan.hasNextBoolean();**
- **scan.hasNextFloat();**
- **scan.hasNextLong();**

# Zadania

Wykonaj zadania z działu

Wprowadzanie danych



# Operacje na plikach

# Odczyt z pliku

Aby odczytać dane z pliku możemy użyć poznanej wcześniej klasy **Scanner**, oraz klasy **File** z pakietu **java.io**:

```
File file = new File("readFile.txt");
Scanner scan = new Scanner(file);
while (scan.hasNextLine()) {
    scan.nextLine();
}
```

# Odczyt z pliku

Aby odczytać dane z pliku możemy użyć poznanej wcześniej klasy **Scanner**, oraz klasy **File** z pakietu **java.io**:

```
File file = new File("readFile.txt");
Scanner scan = new Scanner(file);
while (scan.hasNextLine()) {
    scan.nextLine();
}
```

Tworzymy obiekt klasy **File**. Podajemy ścieżkę do pliku, z którego chcemy skorzystać (jeśli znajduje się w głównym katalogu projektu – jak powyższy – to tylko nazwę).

# Odczyt z pliku

Aby odczytać dane z pliku możemy użyć poznanej wcześniej klasy **Scanner**, oraz klasy **File** z pakietu **java.io**:

```
File file = new File("readFile.txt");
Scanner scan = new Scanner(file);
while (scan.hasNextLine()) {
    scan.nextLine();
}
```

Tworzymy obiekt klasy **File**. Podajemy ścieżkę do pliku, z którego chcemy skorzystać (jeśli znajduje się w głównym katalogu projektu – jak powyższy – to tylko nazwę).

Tworzymy obiekt klasy **Scanner**, wykorzystując wcześniej utworzony obiekt klasy **File**.

# Odczyt z pliku

Aby odczytać dane z pliku możemy użyć poznanej wcześniej klasy **Scanner**, oraz klasy **File** z pakietu **java.io**:

```
File file = new File("readFile.txt");
Scanner scan = new Scanner(file);
while (scan.hasNextLine()) {
    scan.nextLine();
}
```

Tworzymy obiekt klasy **File**. Podajemy ścieżkę do pliku, z którego chcemy skorzystać (jeśli znajduje się w głównym katalogu projektu – jak powyższy – to tylko nazwę).

Tworzymy obiekt klasy **Scanner**, wykorzystując wcześniej utworzony obiekt klasy **File**.

Wczytujemy plik, dopóki istnieją linie w pliku. Metoda **hasNextLine()** zwraca **true**, jeżeli są jeszcze linie do wczytania.

# Odczyt z pliku

Aby odczytać dane z pliku możemy użyć poznanej wcześniej klasy **Scanner**, oraz klasy **File** z pakietu **java.io**:

```
File file = new File("readFile.txt");
Scanner scan = new Scanner(file);
while (scan.hasNextLine()) {
    scan.nextLine();
}
```

Tworzymy obiekt klasy **File**. Podajemy ścieżkę do pliku, z którego chcemy skorzystać (jeśli znajduje się w głównym katalogu projektu – jak powyższy – to tylko nazwę).

Tworzymy obiekt klasy **Scanner**, wykorzystując wcześniej utworzony obiekt klasy **File**.

Wczytujemy plik, dopóki istnieją linie w pliku. Metoda **hasNextLine()** zwraca **true**, jeżeli są jeszcze linie do wczytania.

Pobieramy linię.

# Odczyt z pliku



Wymagana jest obsługa wystąpienia wyjątku: `FileNotFoundException`.

```
File file = new File("readFile.txt");
StringBuilder reading = new StringBuilder();
try {
    Scanner scan = new Scanner(file);
    while (scan.hasNextLine()) {
        reading.append(scan.nextLine() + "\n");
    }
} catch (FileNotFoundException e) {
    System.out.println("Brak pliku.");
}
System.out.println(reading.toString());
```

# Odczyt z pliku



Wymagana jest obsługa wystąpienia wyjątku: `FileNotFoundException`.

```
File file = new File("readFile.txt");
StringBuilder reading = new StringBuilder();
try {
    Scanner scan = new Scanner(file);
    while (scan.hasNextLine()) {
        reading.append(scan.nextLine() + "\n");
    }
} catch (FileNotFoundException e) {
    System.out.println("Brak pliku.");
}
System.out.println(reading.toString());
```

Dodajemy obsługę wyjątku poprzez dodanie klauzuli **try – catch**.



# Odczyt z pliku



Wymagana jest obsługa wystąpienia wyjątku: `FileNotFoundException`.

```
File file = new File("readFile.txt");
StringBuilder reading = new StringBuilder();
try {
    Scanner scan = new Scanner(file);
    while (scan.hasNextLine()) {
        reading.append(scan.nextLine() + "\n");
    }
} catch (FileNotFoundException e) {
    System.out.println("Brak pliku.");
}
System.out.println(reading.toString());
```

Dodajemy obsługę wyjątku poprzez dodanie klauzuli **try – catch**.

Do łańcucha dodawana jest kolejna linia.

# Zapis do pliku

Aby zapisać dane do pliku możemy użyć klasy **PrintWriter** z pakietu **java.io**:

```
PrintWriter out = new PrintWriter("writeFile.txt");  
out.println("first line");  
out.println("second line");  
out.close();
```

# Zapis do pliku

Aby zapisać dane do pliku możemy użyć klasy **PrintWriter** z pakietu **java.io**:

```
PrintWriter out = new PrintWriter("writeFile.txt");  
out.println("first line");  
out.println("second line");  
out.close();
```

Tworzymy obiekt klasy **PrintWriter**, podając mu nazwę pliku, z którego będziemy chcieli korzystać. Jeżeli plik nie istnieje – zostanie automatycznie utworzony.

# Zapis do pliku

Aby zapisać dane do pliku możemy użyć klasy **PrintWriter** z pakietu **java.io**:

```
PrintWriter out = new PrintWriter("writeFile.txt");  
out.println("first line");  
out.println("second line");  
out.close();
```

Tworzymy obiekt klasy **PrintWriter**, podając mu nazwę pliku, z którego będziemy chcieli korzystać. Jeżeli plik nie istnieje – zostanie automatycznie utworzony.

Wypełniamy kolejne linie pliku.

# Zapis do pliku

Aby zapisać dane do pliku możemy użyć klasy **PrintWriter** z pakietu **java.io**:

```
PrintWriter out = new PrintWriter("writeFile.txt");  
out.println("first line");  
out.println("second line");  
out.close();
```

Tworzymy obiekt klasy **PrintWriter**, podając mu nazwę pliku, z którego będziemy chcieli korzystać. Jeżeli plik nie istnieje – zostanie automatycznie utworzony.

Wypełniamy kolejne linie pliku.

Zamykamy strumień, zwalniając wszelkie zasoby systemowe z nim powiązane.

# Zapis do pliku

Podobnie jak w przypadku odczytu musimy napisać obsługę wystąpienia wyjątku: **FileNotFoundException**.

```
try {
    PrintWriter out = new PrintWriter("writeFile.txt");
    out.println("first line");
    out.println("second line");
    out.close();
} catch (FileNotFoundException ex) {
    System.out.println("Błąd zapisu do pliku.");
}
```

# Zapis do pliku

Podobnie jak w przypadku odczytu musimy napisać obsługę wystąpienia wyjątku: **FileNotFoundException**.

```
try {  
    PrintWriter out = new PrintWriter("writeFile.txt");  
    out.println("first line");  
    out.println("second line");  
    out.close();  
} catch (FileNotFoundException ex) {  
    System.out.println("Błąd zapisu do pliku.");  
}
```

Dodajemy obsługę wyjątku poprzez dodanie klauzuli **try – catch**.

# Zapis do pliku

Podobnie jak w przypadku odczytu musimy napisać obsługę wystąpienia wyjątku: **FileNotFoundException**.

```
try {  
    PrintWriter out = new PrintWriter("writeFile.txt");  
    out.println("first line");  
    out.println("second line");  
    out.close();  
} catch (FileNotFoundException ex) {  
    System.out.println("Błąd zapisu do pliku.");  
}
```

Dodajemy obsługę wyjątku poprzez dodanie klauzuli **try – catch**.

Przekazanie sterowania w przypadku problemu z odczytem lub utworzeniem pliku.



# Zapis do pliku

Do zapisu plików możemy również skorzystać z klasy **FileWriter**:

```
try {  
    FileWriter out = new FileWriter("writeFile.txt", true);  
    out.append("first line\n");  
    out.append("second line\n");  
    out.close();  
} catch (IOException ex) {  
    System.out.println("Błąd zapisu do pliku.");  
}
```

# Zapis do pliku

Do zapisu plików możemy również skorzystać z klasy **FileWriter**:

```
try {  
    FileWriter out = new FileWriter("writeFile.txt", true);  
    out.append("first line\n");  
    out.append("second line\n");  
    out.close();  
} catch (IOException ex) {  
    System.out.println("Błąd zapisu do pliku.");  
}
```

Drugi parametr informuje czy mamy zastąpić istniejące już dane, czy dopisać na końcu. Wartość **true** – oznacza, że dopisujemy do pliku.

# Klasa File

Klasy **File** używamy, by uzyskać informacje o plikach i katalogach, oraz wykonywać na nich operacje. Wybrane metody klasy **File**:

- **exists()** – sprawdza czy dany plik/katalog istnieje,
- **delete()** – usuwa plik lub katalog,
- **isFile()** – sprawdza czy jest plikiem,
- **isDirectory()** – sprawdza czy jest katalogiem,
- **lastModified()** – podaje czas ostatniej modyfikacji.

```
File dir = new File("newDir");  
dir.mkdir();  
// Tworzy katalog
```

```
File dirs = new File("newDirs/abc/des");  
dirs.mkdirs();  
// Tworzy katalog wraz z podkatalogami
```

```
File dirs = new File("newDirs");  
String[] dirNames = dirs.list();  
// Pobiera listę nazw katalogów  
System.out.println(  
    Arrays.toString(dirNames));
```

# Zadania

Wykonaj zadania z działu

Pliki

# Pakiet java.nio

# Pakiet java.nio

W języku **Java** – od wersji **1.7** – mamy do dyspozycji rozbudowaną klasę **Files** pakietu **java.nio.file**, która oferuje wiele użytecznych metod służących do pracy z plikami i katalogami.

Metody klasy **Files** przyjmują zwykle jako argument obiekt klasy **Path**, która reprezentuje, w sposób niezależny od systemu plikowego, ścieżki do plików lub katalogów.

```
Paths.get("/home/dell/c_opencv.sh");  
// Absolutna ścieżka do pliku  
Paths.get("file.txt");  
// Plik z katalogu bieżącego  
Paths.get(".");  
// Katalog bieżący  
Paths.get("/");  
// Katalog główny (root)  
Paths.get("../file.txt");  
// Plik z nadkatalogu katalogu  
// bieżącego  
Paths.get("/home", "dell", "Desktop");  
// Ścieżka /home/dell/Desktop
```

Jeśli chcesz dowiedzieć się więcej o pakiecie **java.nio**, informacje uzyskasz pod adresem:

➤ <http://docs.oracle.com/javase/tutorial/essential/io/fileio.html>

# Path

**Path** – posiada przydatne metody pozwalające na pobranie informacji o ścieżce:

```
Path path = Paths.get("/home/dell/compile_opencv.sh");  
path.getFileName();  
path.getName(1);  
path.getNameCount();  
path.getParent();  
path.getRoot();
```

# Path

**Path** – posiada przydatne metody pozwalające na pobranie informacji o ścieżce:

```
Path path = Paths.get("/home/dell/compile_opencv.sh");  
path.getFileName();  
path.getName(1);  
path.getNameCount();  
path.getParent();  
path.getRoot();
```

Tworzymy obiekt klasy **Path**.



# Path

**Path** – posiada przydatne metody pozwalające na pobranie informacji o ścieżce:

```
Path path = Paths.get("/home/dell/compile_opencv.sh");  
path.getFileName();  
path.getName(1);  
path.getNameCount();  
path.getParent();  
path.getRoot();
```

Tworzymy obiekt klasy **Path**.

Zwraca ostatni element ścieżki lub nazwę pliku.

# Path

**Path** – posiada przydatne metody pozwalające na pobranie informacji o ścieżce:

```
Path path = Paths.get("/home/dell/compile_opencv.sh");  
path.getFileName();  
path.getName(1);  
path.getNameCount();  
path.getParent();  
path.getRoot();
```

Tworzymy obiekt klasy **Path**.

Zwraca ostatni element ścieżki lub nazwę pliku.

Zwraca określony element ścieżki.

# Path

**Path** – posiada przydatne metody pozwalające na pobranie informacji o ścieżce:

```
Path path = Paths.get("/home/dell/compile_opencv.sh");  
path.getFileName();  
path.getName(1);  
path.getNameCount();  
path.getParent();  
path.getRoot();
```

Tworzymy obiekt klasy **Path**.

Zwraca ostatni element ścieżki lub nazwę pliku.

Zwraca określony element ścieżki.

Liczba elementów w ścieżce.

# Path

**Path** – posiada przydatne metody pozwalające na pobranie informacji o ścieżce:

```
Path path = Paths.get("/home/dell/compile_opencv.sh");  
path.getFileName();  
path.getName(1);  
path.getNameCount();  
path.getParent();  
path.getRoot();
```

Tworzymy obiekt klasy **Path**.

Zwraca ostatni element ścieżki lub nazwę pliku.

Zwraca określony element ścieżki.

Liczba elementów w ścieżce.

Zwraca ścieżkę do rodzica bieżącego pliku/katalogu.

# Path

**Path** – posiada przydatne metody pozwalające na pobranie informacji o ścieżce:

```
Path path = Paths.get("/home/dell/compile_opencv.sh");  
path.getFileName();  
path.getName(1);  
path.getNameCount();  
path.getParent();  
path.getRoot();
```

Tworzymy obiekt klasy **Path**.

Zwraca ostatni element ścieżki lub nazwę pliku.

Zwraca określony element ścieżki.

Liczba elementów w ścieżce.

Zwraca ścieżkę do rodzica bieżącego pliku/katalogu.

Katalog główny.

# Files

Do operacji na plikach i katalogach służy klasa **Files**:

```
Path path = Paths.get("/home/dell/compile_opencv.sh");  
Files.exists(path);  
Files.copy(path1, path2);  
Files.move(path2, path3);  
Files.delete(path4);  
Files.isSymbolicLink(path1);
```

# Files

Do operacji na plikach i katalogach służy klasa **Files**:

```
Path path = Paths.get("/home/dell/compile_opencv.sh");  
Files.exists(path);  
Files.copy(path1, path2);  
Files.move(path2, path3);  
Files.delete(path4);  
Files.isSymbolicLink(path1);
```

Tworzymy obiekt klasy **Path**.

# Files

Do operacji na plikach i katalogach służy klasa **Files**:

```
Path path = Paths.get("/home/dell/compile_opencv.sh");  
Files.exists(path);  
Files.copy(path1, path2);  
Files.move(path2, path3);  
Files.delete(path4);  
Files.isSymbolicLink(path1);
```

Tworzymy obiekt klasy **Path**.

Sprawdza czy plik istnieje.



# Files

Do operacji na plikach i katalogach służy klasa **Files**:

```
Path path = Paths.get("/home/dell/compile_opencv.sh");  
Files.exists(path);  
Files.copy(path1, path2);  
Files.move(path2, path3);  
Files.delete(path4);  
Files.isSymbolicLink(path1);
```

Tworzymy obiekt klasy **Path**.

Sprawdza czy plik istnieje.

Kopiuje plik ze ścieżki **path1** do **path2**.

# Files

Do operacji na plikach i katalogach służy klasa **Files**:

```
Path path = Paths.get("/home/dell/compile_opencv.sh");  
Files.exists(path);  
Files.copy(path1, path2);  
Files.move(path2, path3);  
Files.delete(path4);  
Files.isSymbolicLink(path1);
```

Tworzymy obiekt klasy **Path**.

Sprawdza czy plik istnieje.

Kopiuje plik ze ścieżki **path1** do **path2**.

Przenosi plik ze ścieżki **path2** do **path3**.

# Files

Do operacji na plikach i katalogach służy klasa **Files**:

```
Path path = Paths.get("/home/dell/compile_opencv.sh");  
Files.exists(path);  
Files.copy(path1, path2);  
Files.move(path2, path3);  
Files.delete(path4);  
Files.isSymbolicLink(path1);
```

Tworzymy obiekt klasy **Path**.

Sprawdza czy plik istnieje.

Kopiuje plik ze ścieżki **path1** do **path2**.

Przenosi plik ze ścieżki **path2** do **path3**.

Usuwa plik lub niepusty katalog.

# Files

Do operacji na plikach i katalogach służy klasa **Files**:

```
Path path = Paths.get("/home/dell/compile_opencv.sh");  
Files.exists(path);  
Files.copy(path1, path2);  
Files.move(path2, path3);  
Files.delete(path4);  
Files.isSymbolicLink(path1);
```

Tworzymy obiekt klasy **Path**.

Sprawdza czy plik istnieje.

Kopiuje plik ze ścieżki **path1** do **path2**.

Przenosi plik ze ścieżki **path2** do **path3**.

Usuwa plik lub niepusty katalog.

Sprawdza czy ścieżka jest linkiem symbolicznym.

# Kopiowanie i przenoszenie plików

Klasa **Files** udostępnia metody pozwalające m.in. kopiować i przenosić pliki:

```
Path path1 = Paths.get("path1.txt");
Path path2 = Paths.get("path2.txt");
try {
    Files.copy(path1, path2);
} catch (IOException e) {
    e.printStackTrace();
}
```

Kopiowanie pliku.

```
Path path1 = Paths.get("path1.txt");
Path path2 = Paths.get("path2.txt");
try {
    Files.move(path1, path2);
} catch (IOException e) {
    e.printStackTrace();
}
```

Przenoszenie pliku.

# Kopiowanie i przenoszenie plików

Możemy również określić opcje, które decydują np. co ma się dzieć, gdy docelowy plik istnieje:

```
Files.copy(path1, path2, StandardCopyOption.REPLACE_EXISTING);
```

Spowoduje to zastąpienie istniejącego pliku.

Jeżeli jako trzeci argument metody nie podamy wartości:

```
StandardCopyOption.REPLACE_EXISTING
```

to w przypadku próby nadpisania pliku, zostanie zwrócony wyjątek:

```
FileAlreadyExistsException
```

Pełen zestaw operacji które udostępnia klasa **Files**:

<https://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html>

# Wczytanie pliku

```
Path path1 = Paths.get("path1.txt");
try {
    for (String line : Files.readAllLines(path1)) {
        System.out.println(line);
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

Zamiast wczytywać plik wiersz po wierszu, tak jak to miało miejsce w poprzednich przykładach, możemy wykorzystać metodę **readAllLines**, dzięki której otrzymamy listę wszystkich wierszy.

Pamiętajmy, że taki sposób powinniśmy stosować jedynie dla niewielkich plików.

# Pisanie do pliku

```
Path path1 = Paths.get("path1.txt");
ArrayList<String> outList = new ArrayList<>();
outList.add("first line");
outList.add("second line");
try {
    Files.write(path1, outList);
} catch (IOException ex) {
    System.out.println("Nie można zapisać pliku.");
}
```



# Pisanie do pliku

```
Path path1 = Paths.get("path1.txt");
ArrayList<String> outList = new ArrayList<>();
outList.add("first line");
outList.add("second line");
try {
    Files.write(path1, outList);
} catch (IOException ex) {
    System.out.println("Nie można zapisać pliku.");
}
```

**ArrayList** – ten element to kolekcja danych – będziemy omawiać kolekcje w kolejnych modułach, na razie możemy go postrzegać jako tablicę z metodą **add**.

# Pisanie do pliku

```
Path path1 = Paths.get("path1.txt");
ArrayList<String> outList = new ArrayList<>();
outList.add("first line");
outList.add("second line");
try {
    Files.write(path1, outList);
} catch (IOException ex) {
    System.out.println("Nie można zapisać pliku.");
}
```

**ArrayList** – ten element to kolekcja danych – będziemy omawiać kolekcje w kolejnych modułach, na razie możemy go postrzegać jako tablicę z metodą **add**.

Uzupełniamy listę danymi.

# Pisanie do pliku

```
Path path1 = Paths.get("path1.txt");
ArrayList<String> outList = new ArrayList<>();
outList.add("first line");
outList.add("second line");
try {
    Files.write(path1, outList);
} catch (IOException ex) {
    System.out.println("Nie można zapisać pliku.");
}
```

**ArrayList** – ten element to kolekcja danych – będziemy omawiać kolekcje w kolejnych modułach, na razie możemy go postrzegać jako tablicę z metodą **add**.

Uzupełniamy listę danymi.

Zapisujemy dane do pliku.

# Zadania

Wykonaj zadania z działu

Pakiet java.nio

# XML

# XML

**XML** (ang. eXtensible Markup Language) – rozszerzalny język znaczników (można go dostosować do własnych potrzeb).

- Zaprojektowany by przenosić dane, a nie format.
- Składniowo podobny do **HTML**.
- Znaczniki nie są predefiniowane, możemy sami je określać.

- **XML** – podobnie jak **HTML** – korzysta:
  - z tagów (elementy ujęte w nawiasy '<' oraz '>'),
  - z atrybutów (np. **name="value"**).
- Jest plikiem tekstowym.
- W **Javie** często używany do przechowywania danych konfiguracyjnych.
- Niezależny od języka programowania.

# Znaczniki XML

## Przykładowy tag XML:

```
<title lang="pl">Coderslab</title>
```

**<title>** – tag otwierający,

**Coderslab** – zawartość elementu,

**lang** – atrybut,

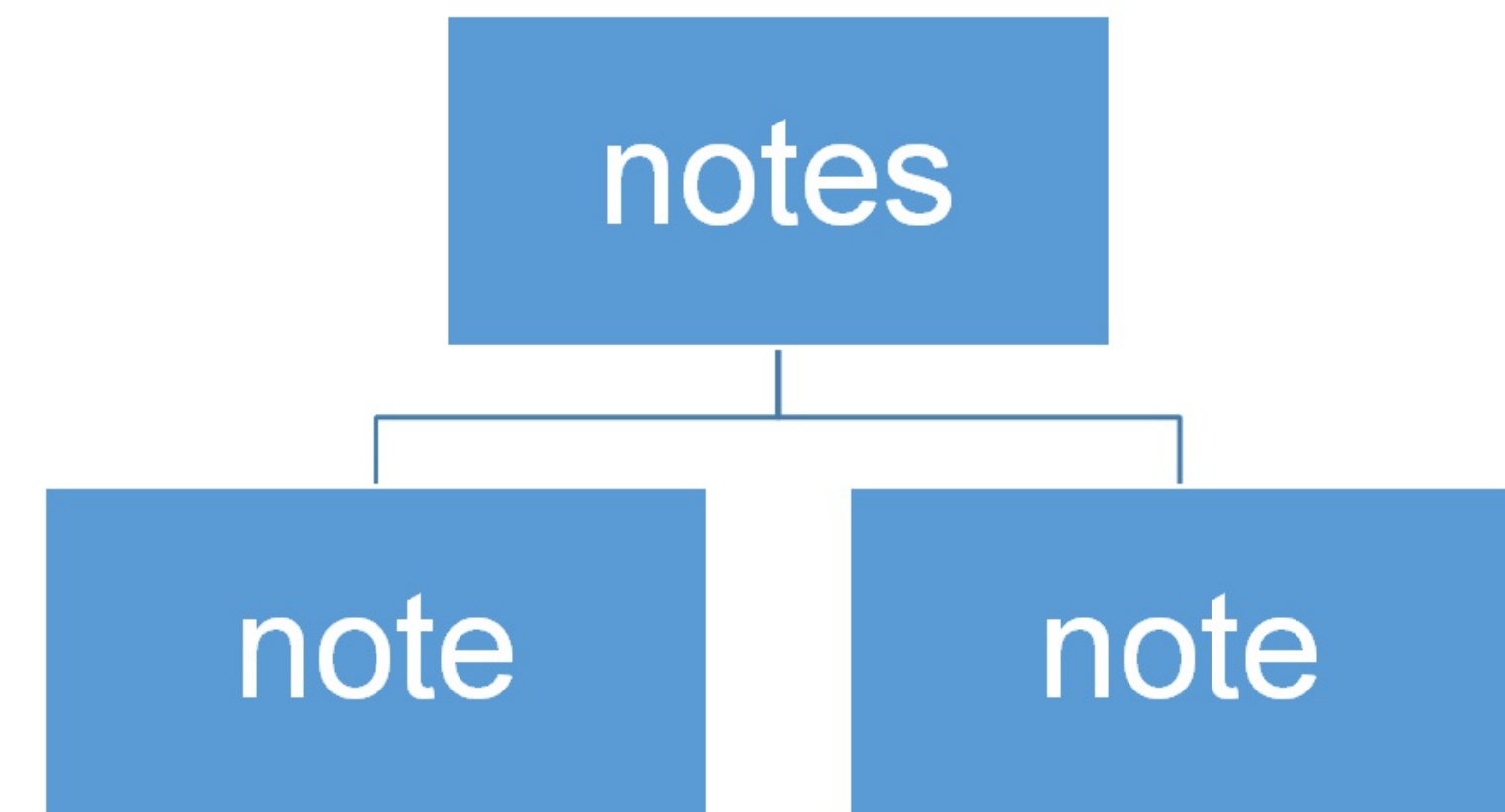
**"pl"** – wartość atrybutu,

**</title>** – tag zamykający.

# Budowa dokumentu

```
<?xml version="1.0" encoding="UTF-8"?>
<notes>
  <note>
    <from>Coderslab</from>
    <to>Mentor</to>
    <body>Do not hit students!!</body>
  </note>
  <note>
    <from>Mentor</from>
    <to>Students</to>
    <body>Hello Students!</body>
  </note>
</notes>
```

Hierarchia elementów:





# Struktura XML

## Nagłówek dokumentu:

```
<?xml version="1.0" encoding="UTF-8"?>
```

Jest to instrukcja przetwarzania, mówiąca z której wersji **XML** i z jakiej strony kodowej korzystamy.

Cały dokument poza instrukcjami przetwarzania musi być zamknięty w jednym głównym elemencie. W naszym przykładzie jest to element: **<notes>**

## Elementy dokumentu:

- wyznaczone za pomocą znaczników,
- mogą zawierać elementy podrzędne,
- wielkość liter ma znaczenie,
- znacznik może zawierać tylko jeden atrybut o danej nazwie.

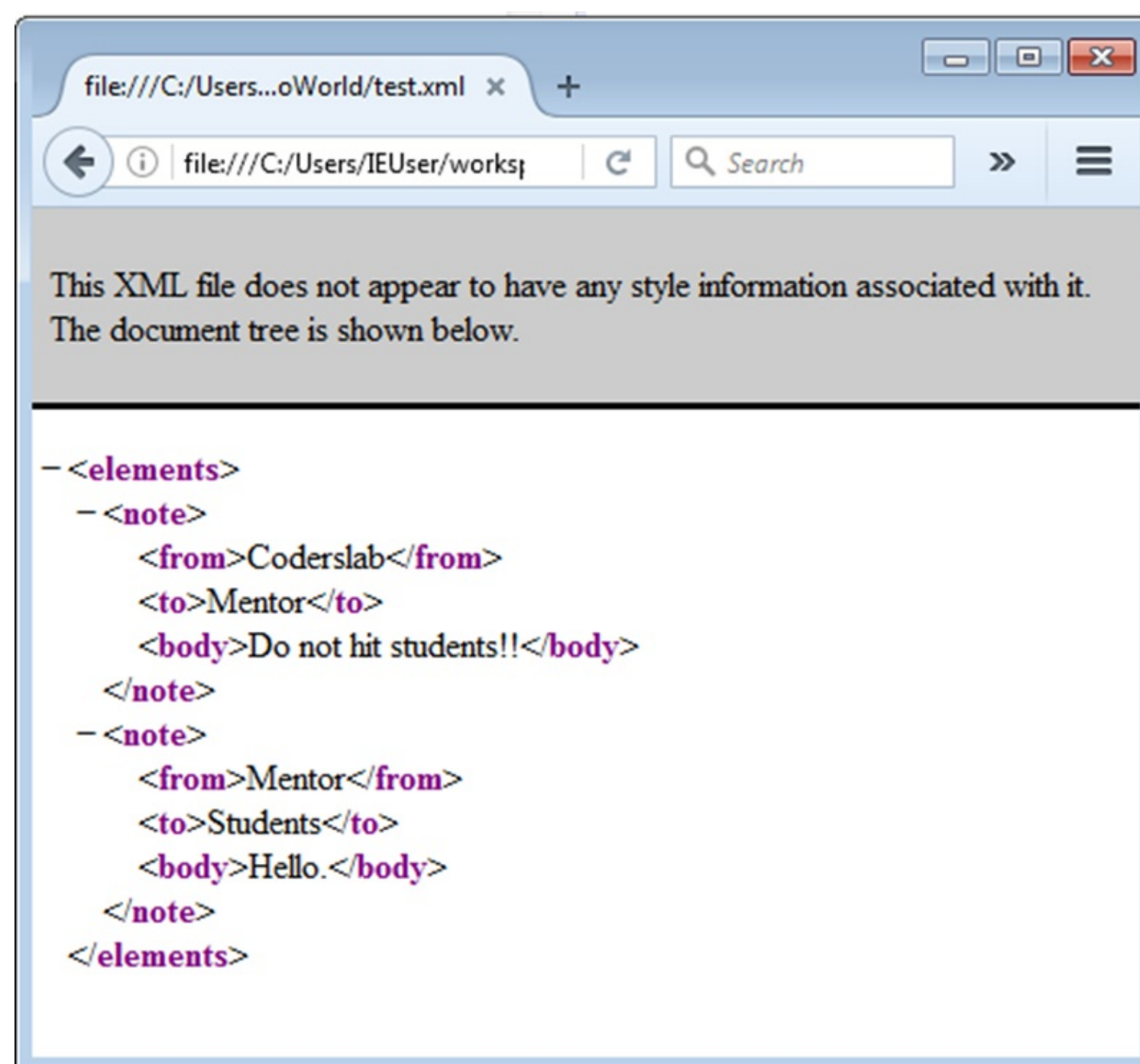
# Poprawność składniowa

## Główne zasady:

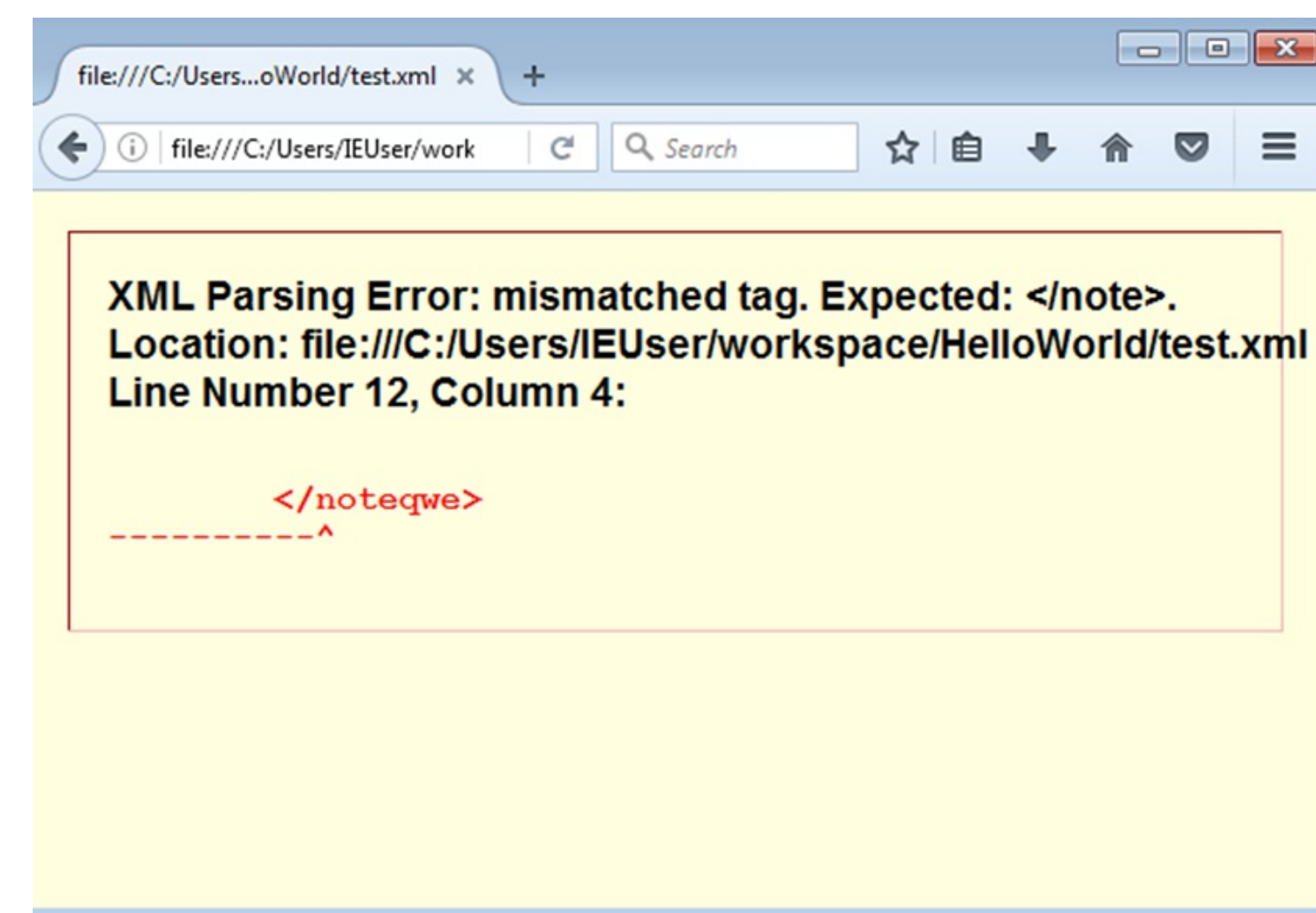
- każdy element musi zaczynać się znacznikiem początku elementu oraz kończyć identycznym znacznikiem zamykającym (wyjątek stanowią elementy puste),
- informacje, które zawiera element, muszą być zapisane pomiędzy znacznikiem początku i końca elementu,
- może zawierać tylko jeden element główny,
- nazwy elementów mogą zawierać:
  - litery **a-z**, **A-Z**,
  - cyfry **0-9**,
  - znaki interpunkcyjne: **podkreślenie**, **myślnik** i **kropkę**,
- nazwy nie mogą zaczynać się od myślnika, kropki ani cyfry.

# Poprawność składniowa

Najprostszym sposobem sprawdzenia poprawności składniowej pliku **XML** jest otworzenie go w dowolnej przeglądarce internetowej.



W przypadku wystąpienia błędu zostaniemy o nim poinformowani odpowiednim komunikatem:



# Debugowanie IDE

# Debugowanie

**Debugger** – program służący do dynamicznej analizy innego programu w celu odnalezienia błędów.

Debugowanie nazywane jest często krokowym wykonywaniem programu.

Debugowanie wykonujemy w celu wykrycia i wyeliminowania błędów naszego programu.

Większość współczesnych środowisk programistycznych zawiera wbudowaną obsługę procesu debugowania.

**Wikipedia** podaje, że nazwa **bug** określająca błąd, pojawiła się po raz pierwszy podczas poszukiwań przepalonej lampy w pierwszych komputerach lampowych.

Zepsutej lampy nie odnaleziono – okazało się jednak, że na jednej z wtyczek znaleziono martwego owada, a proces jego usunięcia nazwano **debugowaniem** (ang. bug – robak).



# Breakpoint

**Breakpoint** – punkt wstrzymania wykonywania programu.

Program uruchomiony w trybie debugowania, wstrzyma swoje wykonanie we wskazanym przez breakpoint miejscu.

Po wstrzymaniu programu mamy możliwość zbadania wartości zmiennych w danym momencie oraz wznowienia dalszego wykonywania programu „krok po kroku”.

Wielokrotnie już wyświetlaliśmy wartości zmiennych używając do tego wbudowanej klasy

**System:**

```
System.out.println(variable1);
```

Jednakże debugowanie bardziej skomplikowanego programu w ten sposób byłoby bardzo uciążliwe, a czasem wręcz niemożliwe.

# Breakpoint

Program można wstrzymać przy pomocy **breakpointa**:

- w określonej linii,
- przy wejściu lub wyjściu z metody,
- podczas nadania lub edycji wartości pola,
- w momencie wystąpienia wyjątku,
- podczas ładowania klasy.

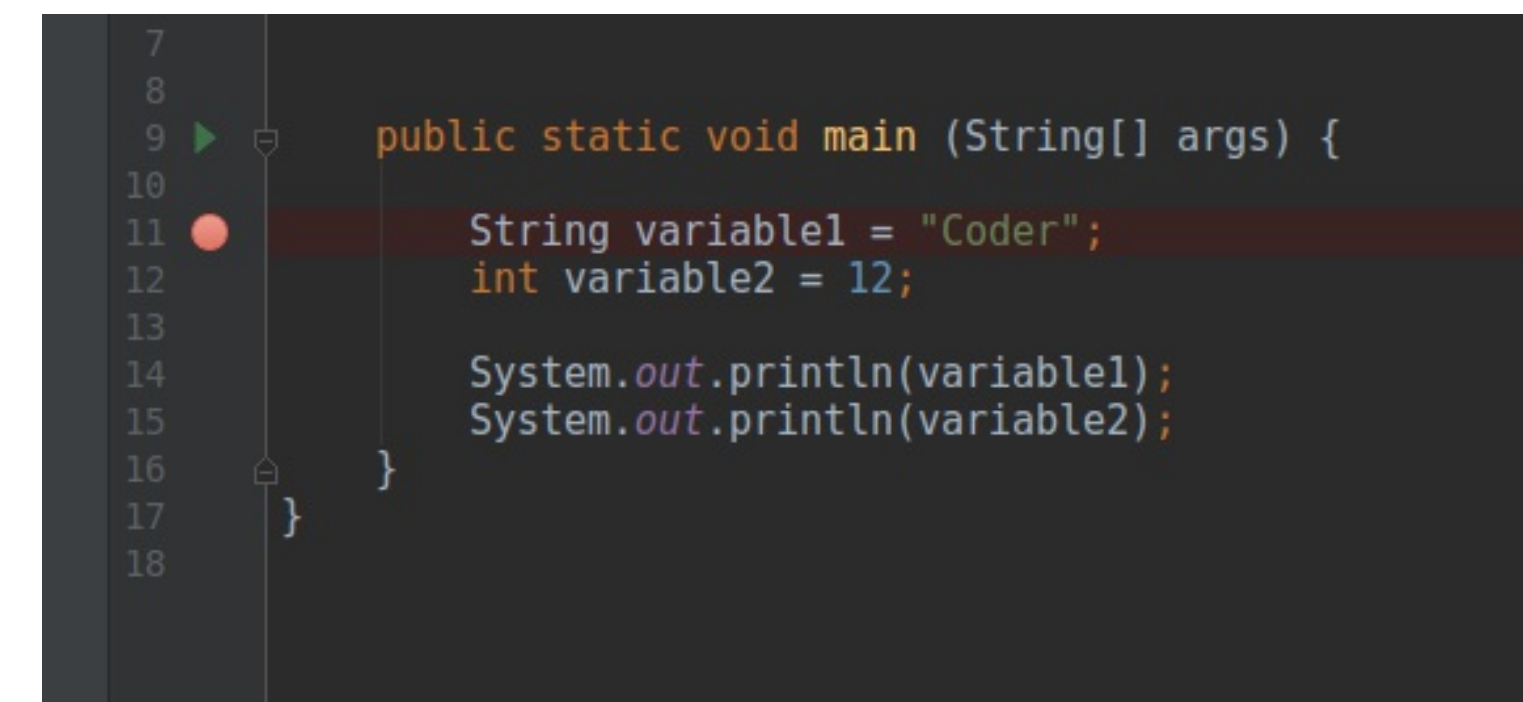
# Debugowanie IntelliJ

Pierwszym krokiem, jaki musimy wykonać, jest ustawienie **breakpointa**.

W **IntelliJ** możemy wykonać to na dwa sposoby:

- klikając z lewej strony linii, w której chcemy ustawić breakpoint,
- umieszczając kursor w linii, w której chcemy ustawić breakpoint i wciskając skrót **Ctrl + F8**.

Po poprawnym ustawieniu punktu wstrzymania w linii pojawi się czerwona kropka jak poniżej:



```
7  
8  
9 ▶ public static void main (String[] args) {  
10  
11 ● String variable1 = "Coder";  
12   int variable2 = 12;  
13  
14   System.out.println(variable1);  
15   System.out.println(variable2);  
16 }  
17  
18
```

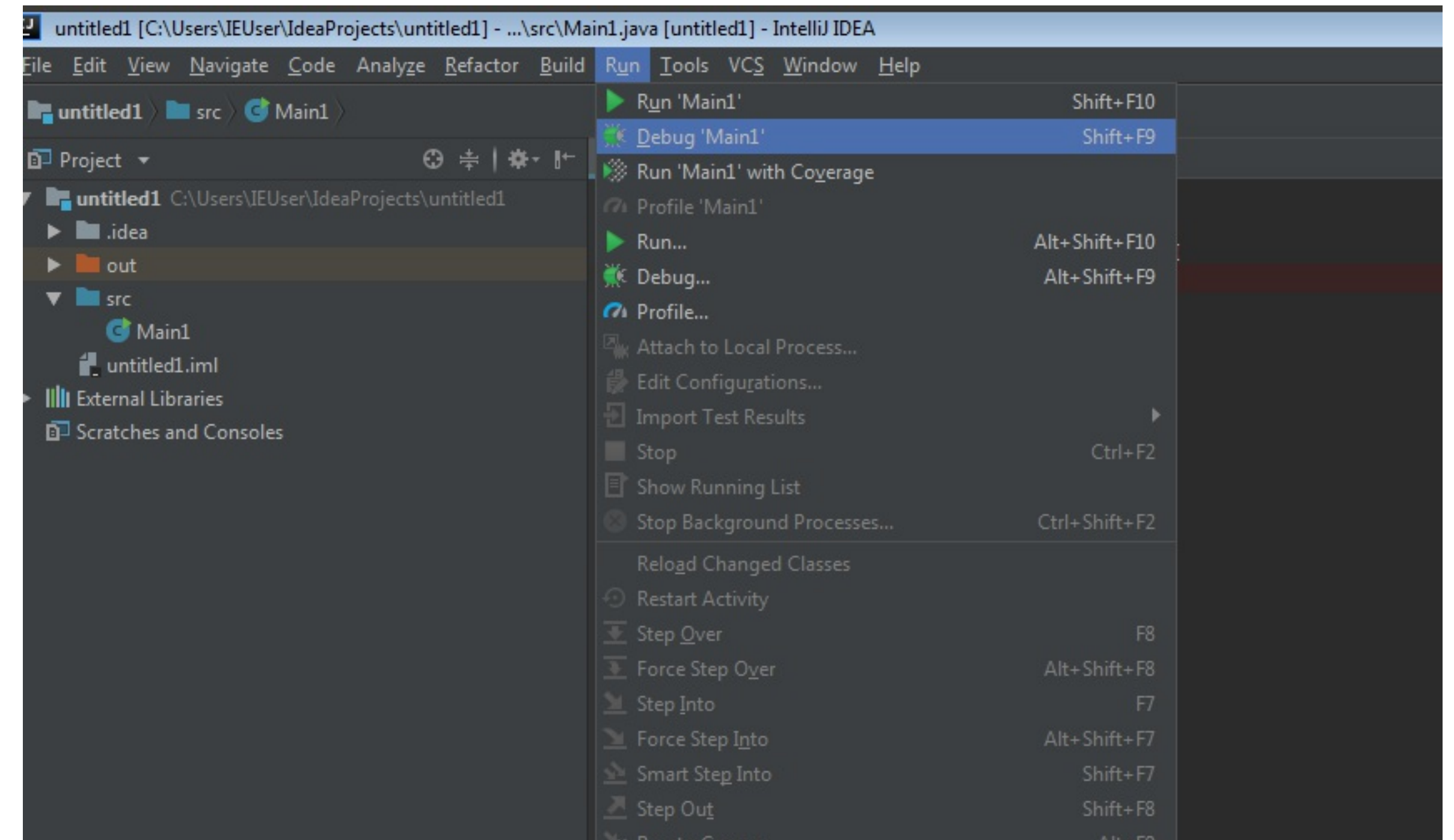
The screenshot shows a Java code editor with a breakpoint (red dot) set on line 11, which contains the line `String variable1 = "Coder";`. The code is part of a `main` method. The line numbers 7 through 18 are visible on the left margin.



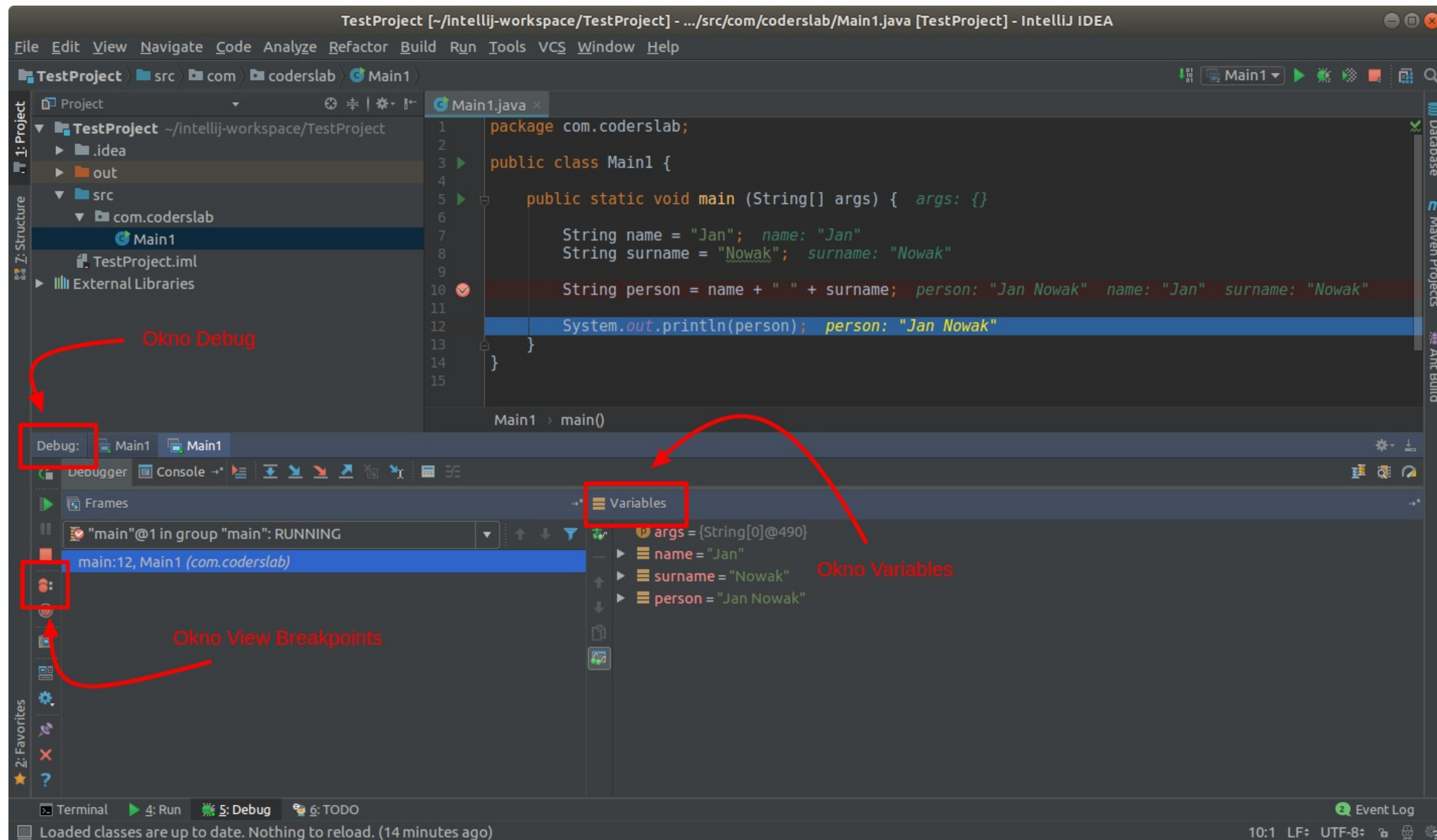
# Debugowanie IntelliJ

Po poprawnym ustawieniu **breakpointów**, musimy uruchomić nasz program w trybie **debugowania**, wybierając z górnego menu zakładkę **Run**, a z listy rozwijanej **Debug**.

Aplikacja uruchomi się, otwierając automatycznie widok **Debuggera** w dolnej części okna programu.



# Widok: Debugger





# Widok: Debugger

Okno **View Breakpoints** – po kliknięciu tej ikonki, otworzy się okno z listą wszystkich breakpointów wraz z ich właściwościami np. ustawienia lub modyfikacji

Okno **Variables** – dostarcza informacje na temat wszystkich zmiennych dostępnych w danym momencie.

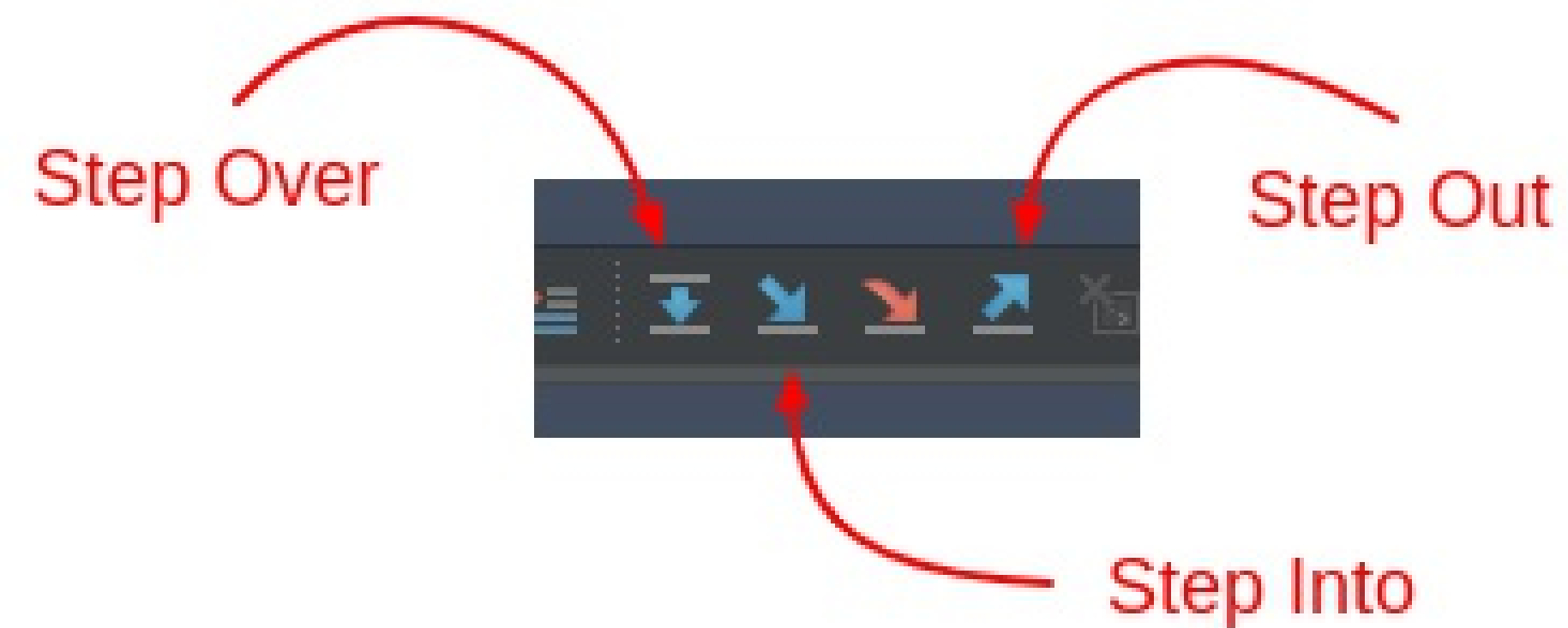
Okno **Debug** – zawiera informacje o miejscu wystąpienia błędu: nazwę klasy, metody, linię wystąpienia.

Perspektywa **Debug** zawiera specjalne przyciski służące do sterowania działaniem programu, najważniejsze z nich to:



# Perspektywa Debug

Do przechodzenia „krok po kroku” przez aplikację służą przyciski:



**Step Over** – przechodzi do następnej linii naszego programu.

**Step Into** – wykonuje aktualną linię i idzie do następnej, jeżeli linia jest metodą, wchodzi do jej kodu.

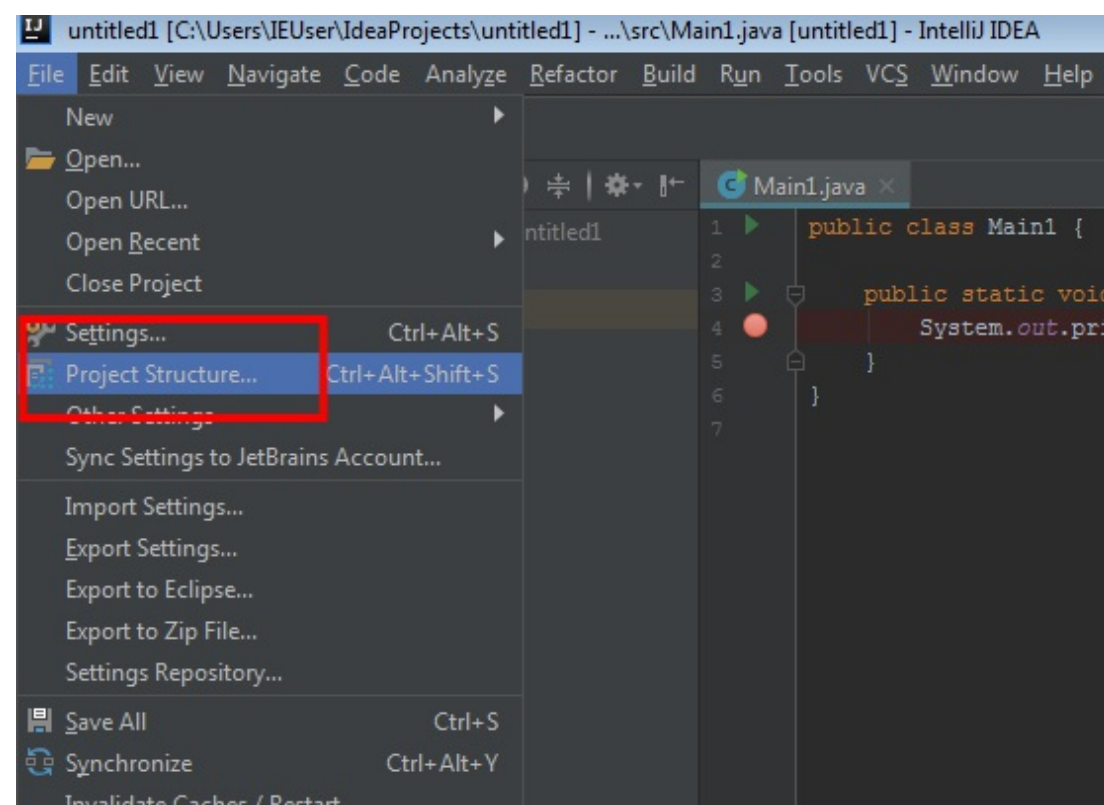
**Step Out** – jeżeli używaliśmy opcji **Step Into**, wybranie tej opcji spowoduje wyjście z metody o poziom wyżej.

# External Jar

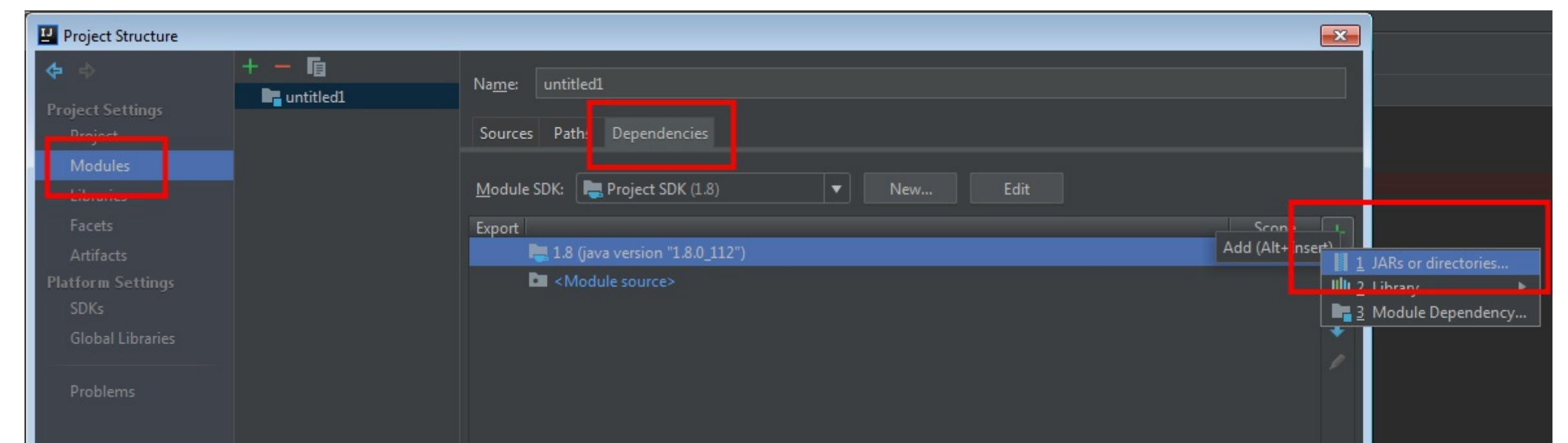
# External Jar

Często zdarza się, że potrzebujemy napisać program z użyciem zewnętrznych bibliotek – mamy możliwość dołączenia ich za pomocą naszego **IDE**.

Aby to zrobić, trzeba kliknąć zakładkę **File**, a z listy **Project Structure...**

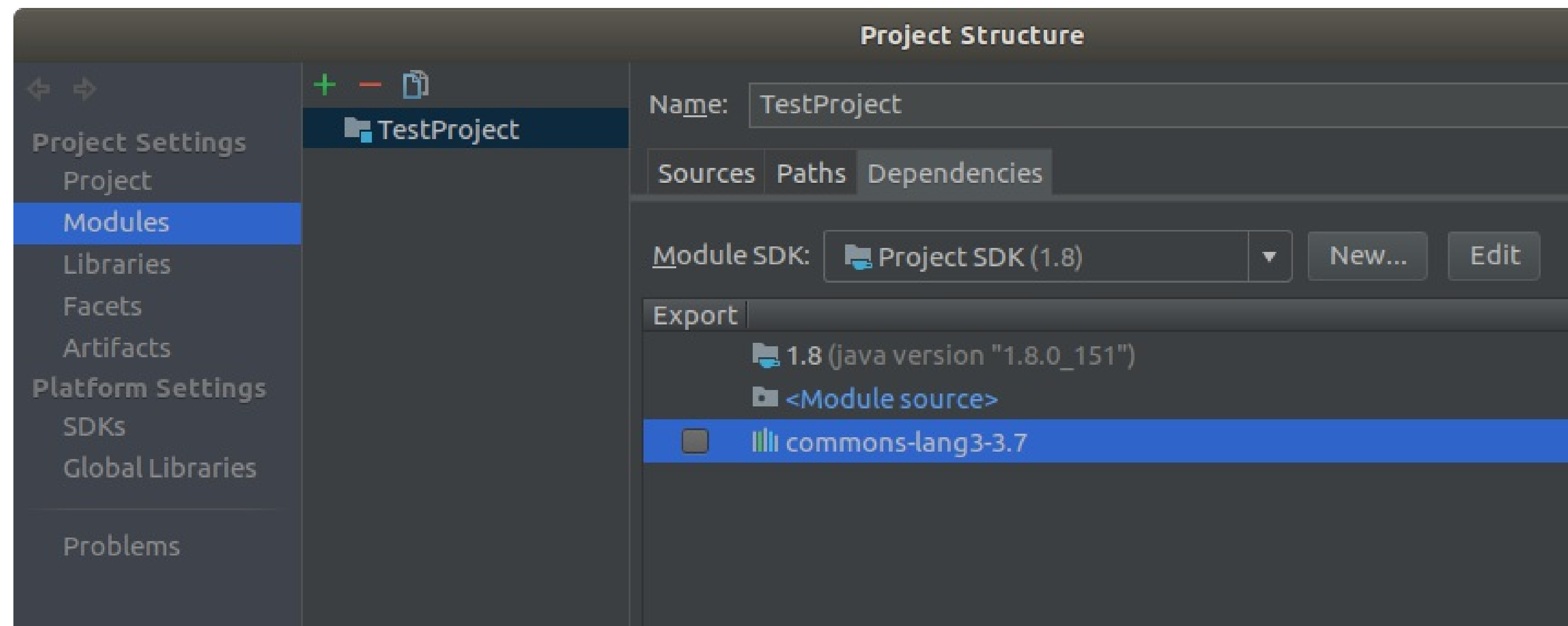


Otworzy się nowe okienko, w którym wybieramy z menu po lewej stronie opcję **Modules**, a w niej zakładkę **Dependencies**.



# External Jar

Dodajemy bibliotekę klikając w znak "+" po prawej stronie, wyszukujemy pobraną wcześniej bibliotekę w systemie plików i potwierdzamy klikając **OK**. Plik zostanie zaimportowany i od tej pory możemy używać klas udostępnianych przez daną bibliotekę.



# Apache Commons

Przykładowo zaimportujemy bibliotekę **lang** z pakietu **Apache Commons** – udostępniającą wiele przydatnych metod do operacji m.in. na Stringach. Dokumentację na ten temat znajdziemy pod adresem:

- <https://commons.apache.org/proper/commons-lang/javadocs/api-3.6/>

Natomiast pełną listę bibliotek grupy **Apache Commons** możemy zobaczyć pod adresem:

- <https://mvnrepository.com/artifact/org.apache.commons>

Aby skorzystać z biblioteki, najpierw pobieramy odpowiedni plik **jar**:

- <https://mvnrepository.com/artifact/org.apache.commons/commons-lang3>

następnie możemy go już zaimportować do naszego projektu wg instrukcji opisanych na poprzednim slajdzie.



# Apache Commons Lang

Biblioteka ta wprowadza wiele funkcjonalności niedostępnych w standardowych bibliotekach Javy. Poznamy bliżej klasę **StringUtils**, która pozwala w łatwy sposób modyfikować łańcuchy znaków.

Pamiętajmy o odpowiedniej instrukcji importu!

```
import org.apache.commons.lang3.StringUtils;
```

**Uwaga:** załączanie bibliotek za pomocą IntelliJ nie jest zalecaną praktyką. W dalszej części kursu poznamy narzędzia pozwalające na automatyzację procesu zarządzania bibliotekami w projekcie.

# Apache Commons Lang

```
package pl.coderslab;  
import org.apache.commons.lang3.StringUtils;  
public class TestStringUtils {  
    public static void main(String[] args) {  
        String str = "Programuj z CodersLab!";  
        System.out.println(StringUtils.deleteWhitespace(str));  
        System.out.println(StringUtils.reverse(str));  
        System.out.println(StringUtils.swapCase(str));  
    }  
}
```

# Apache Commons Lang

```
package pl.coderslab;  
import org.apache.commons.lang3.StringUtils;  
public class TestStringUtils {  
    public static void main(String[] args) {  
        String str = "Programuj z CodersLab!";  
        System.out.println(StringUtils.deleteWhitespace(str));  
        System.out.println(StringUtils.reverse(str));  
        System.out.println(StringUtils.swapCase(str));  
    }  
}
```

Klasa z załączonej biblioteki,

# Apache Commons Lang

```
package pl.coderslab;  
import org.apache.commons.lang3.StringUtils;  
public class TestStringUtils {  
    public static void main(String[] args) {  
        String str = "Programuj z CodersLab!";  
        System.out.println(StringUtils.deleteWhitespace(str));  
        System.out.println(StringUtils.reverse(str));  
        System.out.println(StringUtils.swapCase(str));  
    }  
}
```

Klasa z załączonej biblioteki,

**deleteWhitespace** – usuwa białe znaki (zwróci: **ProgramujzCodersLab!**),

# Apache Commons Lang

```
package pl.coderslab;  
import org.apache.commons.lang3.StringUtils;  
public class TestStringUtils {  
    public static void main(String[] args) {  
        String str = "Programuj z CodersLab!";  
        System.out.println(StringUtils.deleteWhitespace(str));  
        System.out.println(StringUtils.reverse(str));  
        System.out.println(StringUtils.swapCase(str));  
    }  
}
```

Klasa z załączonej biblioteki,

**deleteWhitespace** – usuwa białe znaki (zwróci: **ProgramujzCodersLab!**),

**reverse** – odwraca kolejność znaków w łańcuchu (zwróci: **!baLsredoC z jumargorP**),

# Apache Commons Lang

```
package pl.coderslab;  
import org.apache.commons.lang3.StringUtils;  
public class TestStringUtils {  
    public static void main(String[] args) {  
        String str = "Programuj z CodersLab!";  
        System.out.println(StringUtils.deleteWhitespace(str));  
        System.out.println(StringUtils.reverse(str));  
        System.out.println(StringUtils.swapCase(str));  
    }  
}
```

Klasa z załączonej biblioteki,

**deleteWhitespace** – usuwa białe znaki (zwróci: **ProgramujzCodersLab!**),

**reverse** – odwraca kolejność znaków w łańcuchu (zwróci: **!baLsredoC z jumargorP**),

**swapCase** – zamieni małe litery na duże i odwrotnie (zwróci: **pROGRAMUJ Z cODERSLAB!**).

# Zadania

Wykonaj zadania z działu

External Jar