Języki Programowania

dr inż. Tomasz Kubik
tomasz.kubik.staff.iiar.pwr.edu.pl

Pętla `for`

```java
// Inicjacja tablicy wielowymiarowej
int[][] tab = { { 2, 3, 4, 5 },
                { 2, 3, 4, 5 },
                { 2, 3, 4, 5 } };
int end=0;
int i=0,j=0;
// Tradycyjne przerwanie pętli zewnętrznej (1)
for (i = 0; i < tab.length && end==0; i++) {
  for (j = 0; j < tab[0].length; j++) {
     if(tab[i][j]==3) { end=1; break;}
  }
}
System.out.println(i-1+" "+j); // i-1 (!)

// Tradycyjne przerwanie pętli zewnętrznej (2)
end=0;
for (i = 0; i < tab.length && end==0; i++) {
  for (j = 0; j < tab[0].length; j++) {
     if(tab[i][j]==3) { end=1; break; }
  }
  if(end==1) break;
}
System.out.println(i+" "+j); // i-1 (!)

// Przerwanie etykietowanej pętli zewnętrznej
outer: for (i = 0; i < tab.length ; i++) {
   for (j = 0; j < tab[0].length; j++) {
     if(tab[i][j]==3) { break outer; }
   }
}
System.out.println(i+" "+j); // i (!)
```

```java
// foreach - specjalna instrukcja for
// - zmienna jest deklarowana wewnątrz pętli
// - poza pętlą zmienna jest niedostępna
//  for ([typ] [zmienna]: [tablica]) {
//    // TO DO: ...
//  }
//  for( [typ] [zmienna]: [kolekcja] ) {
//    // TO DO: ...
//  }

// Przerwanie etykietowanej pętli zewnętrznej
outer1: for(int[] row: tab) {
    for(int item: row) {
        if(item == 3) { break outer1; }
    }
} // brak jest informacji o indeksie elementu

// forEach - metoda kolekcji
//  void forEach(Consumer<? super T> action)
//
var names = new ArrayList<String>();
names.add("John");
// Dostęp do elementów kolekcji w pętli foreach
for (String name : names) {
    System.out.println(name);
}
// Dostęp do elementów kolekcji metodą forEach
// i wyrażeniem labda
names.forEach(name -> {
    System.out.println(name);
});
```

Proszę zajrzeć do `ex01.Main` w `Wyk04-2021.zip`

Wyrażenia lambda

```
// Wyrażenia lamba pojawiają się w miejscach,
// w których oczekiwana jest implementacja
// interfejsu funkcyjnego
@FunctionalInterface
interface I {
    public void m(String s);
}
class CI implements I {
  @Override // nie może być static
  public void m(String s) {
    System.out.println("m:" + s);
  }
}

public class C {
  I i = null;
  void k(String s) {
    System.out.println("k:" + s);
  }
  void n(I i) {
    i.m("Call from n(I)");
  }
  static void m(String s) {
    System.out.println("sm:" + s);
  }
```

```
  C() {
  // instancja klasy implementującej interfejs
     i = new CI();
  // instancja klasy anonimowej
     i = new I() {
            @Override
            public void m(String s) {
               System.out.println("i:" + s);
            }
        };
  // wyrażenia lambda
     i = (String s) -> {System.out.println("l:"+s);};
     i = (s) -> {System.out.println("l:"+s);};
     i = s -> System.out.println("l:"+s);
  // referencje do metod (o sygnaturze jak I.m() !!!)
     i = C::m; // referencja do metody statycznej
     i = this::k; //referencja do metody instancyjnej
  }

  // Istnieją cztery rodzaje referencji do metod
  // (mogących posłużyć zamiast wyrażenia lambda):
  // 1. referencja do statycznej metody
  // 2. referencja do metody instancyjnej jakiegoś
  obiektu
  // 3. referencja do metody instancyjnej jakiegoś
  obiektu szczególnego typu
  // 4. referencja do konstruktora
```

Proszę zajrzeć do `ex01.C` w `Wyk04-2021.zip`

Do poczytania:
https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html
Przy okazji proszę zwrócić uwagę na formalny opis przypadku użycia. Może się to kiedyś przydać (przy tworzeniu wymagań we własnym projekcie !!!).

Wyrażenia lambda

- Z kompilacją wyrażeń lambda do kodu bajtowego i ich użyciem jest inaczej niż z kompilacją i użyciem interfejsów, klas i typów wyliczeniowych.
- Za szczegóły odpowiada witrualna maszyna, a nie kompilator
  - w wirtualnej maszynie musi działać mechanizm dynamicznego wywoływania metod, którego implementacja zależeć może od danej wirtualnej maszyny,
  - można się o tym przekonać dekompilując kod bajtowego danej klasy.

```
> javap -private Main
Warning: File .\Main.class does not contain class Main
Compiled from "Event.java"
class ex02.Main {
  ex02.Main();
  public static void main(java.lang.String...);
  private static void lambda$0(ex02.Event);
}
```

Proszę zajrzeć do `ex02.Main` w `Wyk03-2021.zip`

Do poczytania:
https://www.edureka.co/community/40239/how-are-java-lambda-functions-compiled
https://stackoverflow.com/questions/16827262/how-will-java-lambda-functions-be-compiled
https://stackoverflow.com/questions/34589435/get-the-enclosing-class-of-a-java-lambda-expression

Wyrażenia lambda

- Wyrażenie lambda może być:
  - *void-compatible*
    - jeśli każdy `return` w bloku ma postać `return;` (żadna ścieżka wykonania nie doprowadzi do zwrócenia wartości),
  - *value-compatible*
    - jeśli nie może zakończyć się normalnie albo każde wyrażenie `return` w bloku ma postać: `return Expression`; (są wyrzucane wyjątki lub każda ścieżka wykonania prowadzi do zwrócenia wartości).
- Kompilator musi ocenić z kontekstu, z jakim rodzajem wyrażenia lambda ma do czynienia i od tego uzależnić, którą metodę faktycznie należy wykonać (jeśli metod kandydujących do wywołania jest więcej niż jedna).

```
interface I {  void m(); }
interface J { int m() throws Exception; }

public class A {
  void ma(I i) { i.m(); }
  void ma(J i) {
    try { i.m();
    } catch (Exception e) {
    }
  }
  public static void main(String[] args) {
  A a = new A();
  a.ma(() -> { // o tym, która metoda ma() zostanie wywołana zadecyduje kompilator
    boolean b = true;
    while (true)
    // jeśli w warunku wstawione jest b,
    // to wtedy jest problem
    // wyjaśnienie na stronie:
https://stackoverflow.com/questions/56149752/thread-sleep-inside-infinite-while-loop-in-lambda-doesnt-require-catch-interr
    // dokładniejszy opis można znajduje się
    // w specyfikacji języka:
    //
https://docs.oracle.com/javase/specs/jls/se8/html/jls-15.html#jls-15.27.2
      Thread.sleep(100); } );
  }
}
```

Proszę zajrzeć do `ex01` w `Wyk03-2021.zip`

Przykłady wyrażeń lambda

```
class Pair {
  int w=0, h=0;
  Pair enlarge() {
    w++; h++;
    return this; // metoda zwraca this,
  }              // dzięki czemu można tworzyć
}                // łańcuchy wywołań: p.enlarge().enlarge();
// interfejs standardowy
@FunctionalInterface
interface Processing01 {
  void process01(int i); }

// interfejs generyczny (z dwoma parametrami)
@FunctionalInterface
interface Processing02<U,V> {
   V process02(U u); }

//interfejs generyczny (z jednym parametrem)
@FunctionalInterface
interface Processing03<U> {
   U process03(); }

public class Tester {
  // metoda z atrybutem typu interface
  public void proc01(Processing01 p, int input) {
    p.process01(input);
  }
  // metoda z atrybutem typu sparametryzowany interfejs generyczny
  public Pair proc02(Processing02<Pair, Pair> p, Pair input) {
    return p.process02(input);
  }

  // metoda z atrybutem typu sparametryzowany interfejs generyczny
  public Pair proc03(Processing03<Pair> p) {
    return p.process03();
  }

  public void m2(int i) {
    System.out.println("m2="+i);
  }
  // metoda statyczna
  public static void ms(int i) {
    System.out.println("ms(i)="+i);
  }
  // metoda instancyjna
  public void mi(int i) {
    System.out.println("mi(i)="+i);
  }

  public static void main(String ... args) {
    // instancja klasy
    Tester t1 = new Tester();

    // Gdy chcemy przetworzyć przekazany obiekt

    // 1a. Lambda z refencją do metody statycznej
    t1.proc01(a->Tester.ms(a),1);
    // 1b. Referencja do statycznej metody
    t1.proc01(Tester::ms,1);

    // Gdy chcemy przetworzyć przekazany obiekt

    // 2a. Lambda z refencją do metody instancyjnej
    t1.proc01(a->t1.mi(a),1);
    // 2b. Referencja do metody instancyjnej
    t1.proc01(t1::mi,1);
    // t1.method(t1::ms, 0, 0); // Źle, bo ms statyczna

    // Gdy chcemy zmienić przekazany obiekt

    // 3a. Lambda z referencją do metody przekazanej instancji
    Pair input = new Pair();
    Pair output = t1.proc02(sq -> sq.enlarge(), input);
    System.out.println("x="+output.w+" y"+output.h);

    // 3b. Referencja do metod instancji za pomocą nazwy klasy
    output = t1.proc02(Pair::enlarge, input);

    // Gdy chcemy przekazać metodę, która utworzy obiekt

    // 4a. Lambda z referencją do konstruktora
    output = t1.proc03(()-> new Pair());
    // 4b. Referencja do konstruktora
    output = t1.proc03(Pair::new);
  }
}
```

Proszę zajrzeć do `ex01` w `Wyk04-2021.zip`

Typy parametryzowane/generyczne (ang. *generics*)

- Mechanizm typów generycznych wprowadzono w JDK 1.5 (2004)
- Pozwala on **deklarować** klasy, interfejsy i metody z wykorzystaniem **parametrów**, a potem **używać** tych klas, interfejsów i metod z **różnymi wartościami tych parametrów.**
- **Typ generyczny** (sparametryzowany typ) i jego **parametr** (którego wartością może być jakiś typ) to **dwa różne pojęcia.**
- **Deklaracja** typu generycznego (gdy pojawiają się parametry) oraz jego **użycie** (gdy w miejsce parametrów wstawiane są typu) to **dwa różne konteksty**. Podczas deklaracji parametr reprezentuje jakiś typ. Podczas użycia w miejsce parametru musi pojawić się jakiś konkretny typ (typ ten staje się wartością parametru).
  - deklaracja (tu typ `A` zostaje zadeklarowany z parametrem `T`): `public class A<T> {}`
  - użycie (tu typ `A` zostaje użyty z parametrem o wartości `Integer`): `A<Iteger> ai;`
- Typy generyczne nazywane są czasem „szablonami”, choć znaczenie tego pojęcia w Java **jest inne** niż C++.
- Zgodnie z konwencją parametry deklarowane są z wykorzystaniem końcowych liter alfabetu (`T,U,V,..`).
- Zalety
  - większa kontrola typów podczas kompilacji,
  - brak konieczności rzutowania typów,
  - możliwość pisania uniwersalnych algorytmów.

- **Kolekcje Java napisano z użyciem typów generycznych (można jednak wciąż używać kolekcji bez parametryzacji, operujących na typie `Object`).**

Parametryzowane typy

- Deklaracja typu generycznego z parametrem `T`
  ```
  public class A<T> { … }
  ```
- Deklaracja pola, którego typ odpowiada zadeklarowanemu wcześniej parametrowi (tj. parametrowi towarzyszącemu deklaracji zewnętrznego typu)
  ```
  public T fT;
  ```
- Deklaracja pola, którego typ jest typem sparametryzowanym zadeklarowanym wcześniej parametrem
  ```
  public A<T> fAT;
  ```
- Deklaracja metody sparametryzowanej parametrem `U`
  ```
  public <U> void m(…){…}
  ```
- Deklaracja atrybutu metody, którego typ jest jak
  - parametr metody
    ```
    public <U> void mUU(U pA){…}
    ```
  - parametr zadeklarowany wcześniej
    ```
    public <U> void mUT(T pA){…}
    ```

Proszę zajrzeć do `ex01.B.m()`
w `Wyk04-2021.zip`

```
/***** Deklaracja typu generycznego
class C<T> { T f = null; }
/***** Użycie typu generycznego
C<String> cS1 = new C<String>(); // kiedyś
C<String> cS2 = new C<>();       // teraz
var cI = new C<Integer>();       // teraz
cS1.f = "ALA";
cS2.f = cS1.f;
cI.f = 10;
// cI.f = cS.f; // ŹLE (różne typy pól)
System.out.println(cS1.getClass());
System.out.println(cI.getClass());
// powyższe wypisze: class C

/***** Użycie listy typu standardowego
List lo = new ArrayList();
lo.add(1);              // opakowanie do Integer
lo.add("2");
int i = (int)lo.get(0); // rozpakowanie do int
Integer oi = (Integer)lo.get(0);
String os = (String)lo.get(1);
/***** Użycie listy typu generycznego
List<String> ls = new ArrayList<String>();
ls.add("hello");
String os1 = ls.get(0);

List<Integer> li = new ArrayList<>();
li.add(1);
Integer oi1 = li.get(0);
int i1 = li.get(0);
var li2 = new ArrayList<Integer>();// skrótowo
```

Przykłady typów generycznych

```
// sparametryzowana klasa
public class A<T> {
  // deklaracja pola z użyciem parametru klasy
  public T fieldT;

  // deklaracja metody z użyciem parametru klasy:
  // 1. typ atrybutu metody to parametr klasy
  // 2. typ wartości zwracanej to parametr klasy
  public T m01(T pT) {
    // nic nie wiemy o typie parametru, a więc:
    // 1. nie da się użyć konstruktora T:
    //     T t1 = new T(); // ŹLE
    // 2. można jednak wywołać metody
    //    odziedziczone z klasy Object:
    System.out.println(pT.toString());
    // 3. można też przekazać sparametryzowany
    //    atrybut pT do sparametryzowanych metod
    T out = m02(pT); // DOBRZE
    // 4. można zwracać wartość typu T
    return out; // DOBRZE
  }

  // sparametryzowana metoda innym typem niż klasa
  // 1. parametr pojawia się przed sygnaturą metody,
  //    za modyfikatorem dostępu
  public <U> U m02(U pU) {
    // reguły używania U wewnątrz m02 są takie,
    // jak reguły używania T wewnątrz m01
    System.out.println(pU.toString());
    return null;
  }

  // sparametryzowana metoda (parametrem V):
  // a. posiada atrybut sparametryzowano parametrem
  //    użytym do parametryzacji klasy (T),
  // b. żaden inny atrybut nie jest
  //    sparametryzowany parametrem metody (V)
  public <V> V m03(T pT) {
    // 0. można tak deklarować, ale trudno
    //    z takiej deklaracji skorzystać,
    //    bo wewnątrz metody nic nie wiadomo
    //    o typie V
    // 1. nie da się użyć konstruktora V:
    // V v1 = new V(); // ŹLE
    //    zawsze można zrobić przypisanie do null
    V v2 = null; // DOBRZE
    //    nie można przypisać obiektu klasy Object,
    //    bo V może okazać się jakąś klasą potomną
    // V v3 = new Object(); // ŹLE
    // 2. można uruchomić konstruktor
    //    klasy sparametryzowanej parametrem V:
    A<V> aV = new A<V>();  // DOBRZE
    // 3. typ A<V> jest różny od typu T,
    //    dlatego zmiennej typu A<V>
    //    nie da się użyć w miejscu
    //    przeznaczonym na zmienną typu T
    // Object out1 = aV.m01(pT); // ŹLE
    // 4. można jednak zmiennej typu V
    //    użyć tam, gdzie pozwala na to
    //    parametryzacja
    V out2 =  aV.m02(v2);  // DOBRZE
    // 5. można też użyć typu parametryzowanego
    //    przez V
    A<V> out3 = aV.m02(aV); // DOBRZE
    return null; // bez użycia rzutowania do typu V
                 // można zwrócić jedynie null
  }
}
```

Proszę zajrzeć do `ex01.A` w `Wyk04-2021.zip`

Do poczytania:

https://docs.oracle.com/javase/tutorial/java/generics/index.html
https://www.oracle.com/technetwork/java/javase/generics-tutorial-159168.pdf
https://bulldogjob.pl/articles/1294-poznaj-podstawy-javy-typy-generyczne
https://bulldogjob.pl/articles/1294-poznaj-podstawy-javy-typy-generyczne
https://www.studytrails.com/2016/09/10/java8_lambdas_functionalprogramming/

Typ wieloznaczny (ang. *wildcard type*)

Pozwala zawęzić zakres wartości dla parametru:

- przy parametryzacji typów (klas i interfejsów):
  - z ograniczeniem od góry `<U extends Type>`
    - podczas użycia można sparametryzować daną klasę lub interfejs typem potomnym dla `Type`,
- przy parametryzacji metod:
  - z ograniczeniem od góry `<V extends Type>`
    - podczas użycia można sparametryzować metodę typem potomnym dla `Type`,
- przy deklaracji pól, zmiennych i atrybutów o parametryzowanych typach:
  - bez ograniczeń dla parametru: `TTT<?>`
    - podczas użycia pola, zmiennej czy atrybutu wartość przypisywana może być dowolnego typu
  - z ograniczeniem parametru od góry: `TTT<? extends Type>`
    - podczas użycia pola, zmiennej czy atrybutu wartość przypisywana może być obiektem typu `TTT<X>`, którego parametr `X` musi być typem potomnym dla `Type`,
  - z ograniczeniem parametru od dołu: `TTT<? super Type>`
    - podczas użycia pola, zmiennej czy atrybutu wartość przypisywana może być obiektem typu `TTT<X>`, którego parametr `X` musi być typem bazowym dla `Type`.

Przykłady typów wieloznacznych

```
public class B {
// *** Deklaracja typu sparametryzowanego typem rozszerzonym
// 1. Typ rozszerzony musi być taki, aby dało się wydedukować typ bazowy.
//    Dlatego na piewszym miejscu, w nawiasach <>, powinien pojawić się PARAMETR, tutaj U.
//    Nie może pojawić się tam ?.
// 2. Jeśli na zewnątrz zadeklarowano parametr U, to bieżący parametr U go nadpisze.
class D1<U extends Number>{}      // Dobrze (typ bazowy to Number)
// class D2<U super Number>{}     // Źle    (nieznany typ bazowy)
// class D3<?>{}                  // Źle    (nieznany typ bazowy)
// class D4<? extends Number>{}   // Źle    (nieznany typ bazowy)
// class D5<? super Number>{}     // Źle    (nieznany typ bazowy)
// *** Użycie typu sparametryzowanego typem rozszerzonym
D1<Integer> oid1 = new D1<>();    // Dobrze (Integer extends Number)
// D1<Object> ood1 = new D1<>(); // Źle    (Object !extends Number)
// Zakładając, że istnieje klasa generyczna class A<T>{}
// *** Deklaracja metod z atrybutem sparametryzowanym typem rozszerzonym
// 1. typ atrybutu musi być taki, aby dało się wydedukować jego typ bazowy.
void m05(A<? extends Number> oeN) { // Dobrze, oeN jest typu A,
   A<Integer> ai=null;              // sparametryzowanym typem dziedziczącym po Number
   m05(ai);                         // Dobre użycie (ai jest typu A<Integer>, Integer exteds Number)
}
void m06(A<? super Number> osN) { // Dobrze, atrybut oeN jest typu A,
   A<Object> ao = null;           // sparametryzowan typem będącym klasą potomną do Number
   m06(ao);                       // Dobrze (ao jest typu A<Object>, Number exteds Object)
}
// *** Deklaracja metody z atrybutem typu odpowiadającego
//       typowi rozszerzonemu, z jakim metodę sparamatryzowano,
// 1. typ atrybutu musi być taki, aby dało się wydedukować jego typ bazowy.
<V extends Number> void m07(V oV) {
   // m07(new Object()); // Źle (Object !exteds Number)
   m07(12);              // Dobrze (Integer extends Number)
}
// <U super A> void m07(U osN) {}    // Źle, nie można użyć super
// <U extends A> void m07(U osN) {}  // Dobrze, nadpisany jest parametr U
// <? extends A> void m07(U osN) {}  // Źle, ? to nieznany parametr
```

Proszę zajrzeć do `ex01.B` w `Wyk04-2021.zip`

Zasady ogólne parametryzacji typów

- Deklaracja parametryzowanych typów i metod odbywa się z wykorzystaniem nawiasów ostrych, w których pojawia się jeden lub, po przecinkach, więcej parametrów, z określeniem lub bez określenia zakresu stosowania

  ale nigdy w postaci `<? ... T>` (z `super` lub `extends` zamiast kropek)
- Parametr wcześniej zadeklarowany może wystąpić w deklaracjach wszędzie tam, gdzie oczekiwany jest typ, jednak wtedy nie można zawężać zakresu stosowania tego parametru (bo zakres określono już w deklaracji tegoż parametru), ani go parametryzować dalej.
- Stąd:
  - jeśli **zadeklarowany wcześniej parametr** ma posłużyć do zadeklarowania **parametryzowanego typu** pola, zmiennej czy atrybutu, to można go użyć na **dwa sposoby**:

    ```
    Type<T> fild;
    Type<? ... T> field;
    ```
    (z `super` lub `extends` zamiast kropek)

    ale nigdy `Type<T ... SomeType> field;` (z `super` lub `extends` zamiast kropek)

    (nie można zawężać zadeklarowanego wcześniej parametru `T`),
  - jeśli **zadeklarowany wcześniej parametr** ma posłużyć do zadeklarowania **typu** pola, zmiennej czy atrybutu, to można go użyć na **jeden tylko sposób**:

    ```
    T field;
    ```
    (parametru nie da się dalej parametryzować).

Zasady PECS (ang. *Producer Extends, Consumer Super*)

- *Producer Extends*: `<? extends Type >`
  - jeśli potrzebujesz listę, z której będziesz wyciągał elementy jakiegoś typu dziedziczącego po wskazanym typie `Type`, to użyj deklaracji: `List<? extends Type> l;`
  - wtedy jednak do listy nie będzie mógł nic dodać.
- *Consumer Super*: `<? super Type >`
  - jeśli chesz do listy dodać elementy jakiegoś typu `Type`, to użyj: `List<? super Type> l;`
  - ale nie masz gwarancji, jakie elementy z takiej listy będziesz pobierał
- General: `<Type >`
  - jeśli chcesz mieć pełen zakres możliwości w dodawaniu elementów do i pobieraniu elementów z listy, korzystaj z konkretnego typu `Type`, np. `List<Integer>`.

Przykłady zastosowania zasad *PECS*

```
class E1{ public int i;    }
class E2 extends E1 {}
class E3 extends E2 {}

void mRead(List<? extends E2> l) {
    // Elementami listy mogą być obiekty typu
    // jakaś klasa dziedzicząca po E2
    // Dlatego referencje do obiektów z tej listy
    // można przypisać jedynie do zmiennych
    // typu klasa E2 lub jej klasy bazowe.

    // E3 e3 = l.get(0); // Źle
    E2 e2 = l.get(0);    // Dobrze
    E1 e1 = l.get(0);    // Dobrze
    Object o = l.get(0); // Dobrze

    // Do listy da się nic wstawić. Nie wiadomo,
    // jaki będzie ostatecznie typ jej elementów
    // (gdy podczas wywołania metody wstawi się
    // do niej jakąś zadeklarowaną listę).
    // W szczególności mogłoby się okazać, że
    // typem tym jest jakaś klasa E4 (na razie
    // jeszcze nieznana) dziedzicząca po E3.
    // Zaś do zmiennej typu szczegółowego nie można
    // przypisać elementu typu ogólniejszego.

    // l.add(o);        // Źle
    // l.add(new E1()); // Źle
    // l.add(new E2()); // Źle
    // l.add(new E3()); // Źle
    }

void mWrite(List<? super E2> l) {
    // Elementami listy mogą być obiekty typu
    // jakaś klasa stojąca w drzewie dziedziczenia
    // powyżej E2. Nie wiadomo jednak, która.
    // Dlatego nie da się przypisać referencji
    // do obiektu z tej listy do żadnej zmiennej,
    // poza zmienną typu Object.

    // E3 e3 = l.get(0); // Źle,
    // E2 e2 = l.get(0); // Źle,
    // E1 e1 = l.get(0); // Źle,
    Object o = l.get(0); // Dobrze,

    // Do listy za to można wstawić
    // obiekty typów dziedziczących po E2
    // l.add(o);        // Źle
    // l.add(new E1()); // Źle
    l.add(new E2());    // Dobrze
    l.add(new E3());    // Dobrze
    }

void m(List<E2> l) {
    // Z parametru wynika, że elementami
    // listy są obiekty typu E2.
    // Dlatego referencje do obiektów pozyskanych
    // z listy można przypisać do zmiennych typu E2
    // lub któraś z klas bazowych.
    // E3 e3 = l.get(0); // Źle,
    E2 e2 = l.get(0);    // Dobrze,
    E1 e1 = l.get(0);    // Dobrze,
    Object o = l.get(0); // Dobrze,

    // Do listy dodawać można elementy typu
    // E2 oraz jej klas pochodnych
    // l.add(o);        // Źle
    // l.add(new E1()); // Źle
    l.add(new E2());    // Dobrze
    l.add(new E3());    // Dobrze
    }
}
```

Proszę zajrzeć do `ex01.PECS` w `Wyk04-2021.zip`

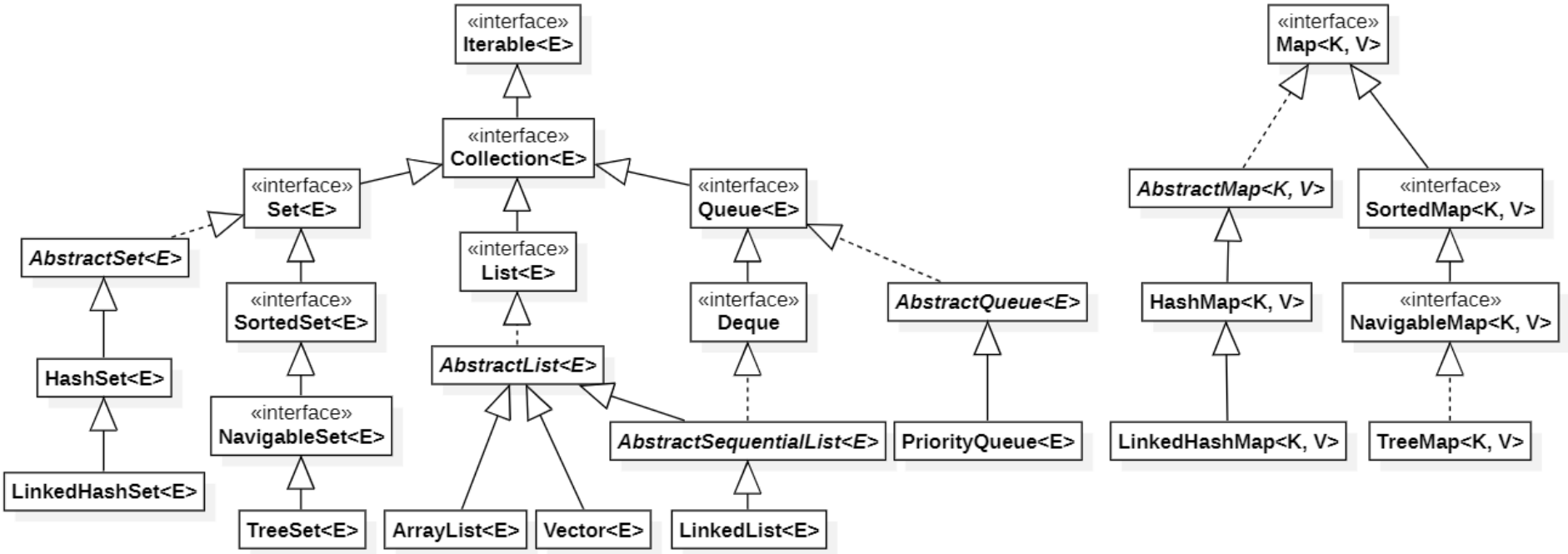
Przykłady dedukcji wartości parametru

```
import java.util.ArrayList;
import java.util.Collection;

public class GenericsTest {
// Po deklaracji metody z atrybutami o typach
// parametryzowanych tym samym parametrem
// przy użyciu tej metody z jakimiś konkretnymi
// atrybutami wartość parametru jest dedukowana.
// Wartością tą staje się typ "minimalny"
// tj. typ atrybutu, który jest typem stojącym
// najwyżej wspólnego drzewa dziedziczenia
// wszystkich atrybutów. Brak "minimalnego" to błąd.

// Użycie parametru w metodzie musi być poprawne,
// a więc używane typy muszą się zgadzać.
  static void fromArrayToCollection1(Object[] a,
Collection<?> c) {
    // przykład użycia typu sparametryzowanego,
    for (Object o : a) {
      //  c.add(o); // Źle, (? != Object)
    }
  }
  static <T> void fromArrayToCollection2(T[] a,
Collection<T> c) {
    // przykład użycia typu sparametryzowanego,
      for (T o : a) {
          c.add(o); // Dobrze, T == T
      }
  }

  public static void main(String[] args) {
    Object[] ao = new Object[100];
    Collection<Object> co = new ArrayList<Object>();

    fromArrayToCollection2(ao, co); // Dobrze, T = Object

    String[] as = new String[100];
    Collection<String> cs = new ArrayList<String>();

    fromArrayToCollection2(as, cs); // Dobrze, T = String

    fromArrayToCollection2(as, co); // Dobrze, T = Object

    Integer[] ai = new Integer[100];
    Float[] af = new Float[100];
    Number[] an = new Number[100];
    Collection<Number> cn = new ArrayList<Number>();

    fromArrayToCollection2(ai, cn); // Dobrze, T = Number

    // T inferred to be Number
    fromArrayToCollection2(af, cn); // Dobrze, T = Number

    fromArrayToCollection2(an, cn); // Dobrze, T = Number

    // T inferred to be Object
    fromArrayToCollection2(an, co); // Dobrze, T = Object

   // fromArrayToCollection2(an, cs); // Źle,
   // wśród atrybutów nie da się wskazać
   // atrybutu "minimalnego"
   // (Number nie stoi powyżej String,
   //  String nie stoi powyżej Number)
  }
}
```

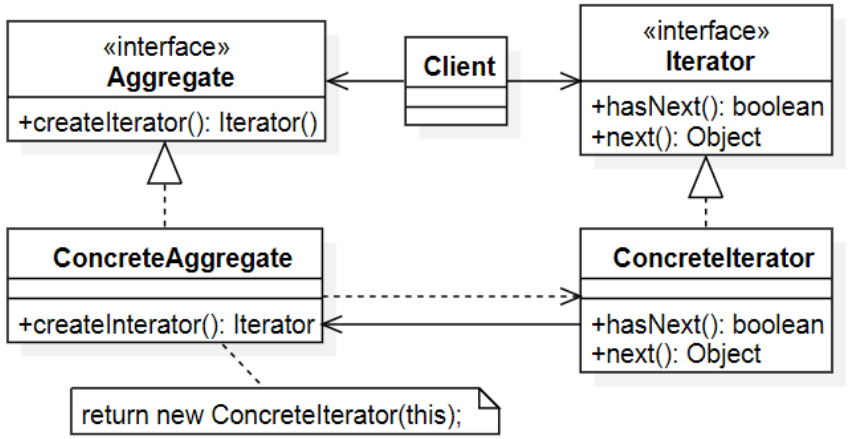
Proszę zajrzeć do `ex01.GenericsTest` w `Wyk04-2021.zip`

Mapy i kolekcje

- Powstały, aby ułatwiać organizowanie danych w złożone struktury.

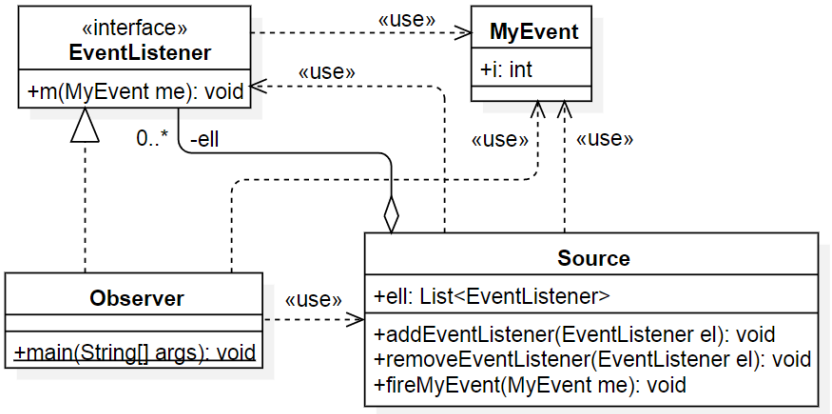


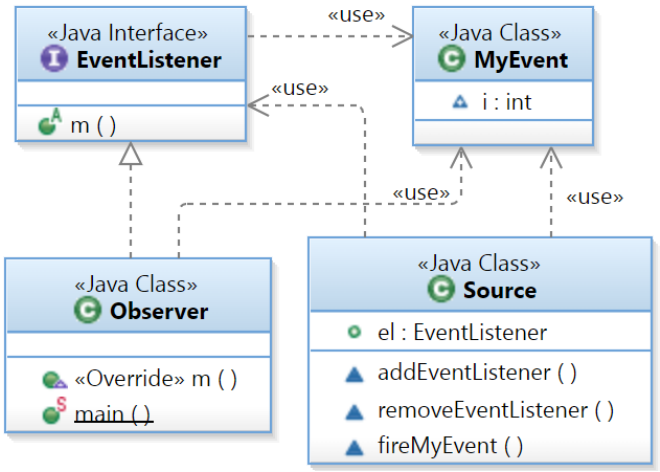
- Można je przeglądać: iteratorem (wg wzorca ***iteratora***) lub stosując *foreach*.



Do poczytania:
https://www.javabrahman.com/design-patterns/iterator-design-pattern-in-java/
https://www.tutorialspoint.com/design_pattern/iterator_pattern.htm
https://www.dineshonjava.com/iterator-pattern-design-patterns-java/

Wzorzec ***obserwatora*** (ang. *observer*)





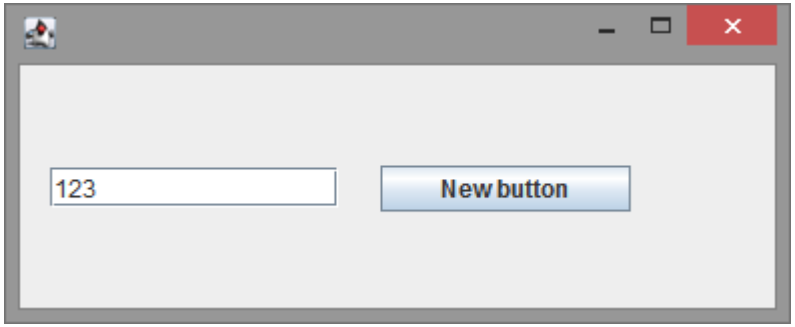

```java
interface EventListener{
    void m(MyEvent me);
}
class MyEvent{
    int i;
}
class Source{
  private List<EventListener> ell = new ArrayList<>();
  void addEventListener(EventListener el) {
        ell.add(el); }
  void removeEventListener(EventListener el) {
        ell.remove(el); }
  void fireMyEvent(MyEvent me) {
        for(EventListener el: ell) {
            el.m(me);      }
  }
}

public class Observer implements EventListener{
  public static void main(String[] args) {
    Source s = new Source();
    Observer o = new Observer();
    s.addEventListener(o);
    s.addEventListener(e -> o.m(e));
    s.fireMyEvent(new MyEvent());
    s.removeEventListener(o);
    s.fireMyEvent(new MyEvent());
  }
  @Override
  public void m(MyEvent me) {
     System.out.println(me.i);
  }
}
```

Proszę zajrzeć do `ex01.Observer` w `Wyk04-2021.zip`
Proszę zajrzeć do `ex02.Event` w `Wyk04-2021.zip`

Implementacja graficznego interfejsu użytkownia

- Zwykle odbywa się na dwa sposoby:

z wykorzystaniem klas Swing,



z wykorzystaniem środowiska JavaFX.

- Projekty modułowe wymagają ustawienia odpowiednich zależności:

dla klas SWING,

```
module wyk04 {
  requires java.desktop;
}
```

dla środowiska JavaFX.

```
module SimpleFX {
  requires javafx.controls;
  requires javafx.fxml;
  requires javafx.base;
  requires javafx.graphics;

  opens application to javafx.graphics, javafx.fxml;
}
```

Proszę zajrzeć do `SimpleFX.zip`
Proszę zajrzeć do `ex02` w `Wyk04-2021.zip`

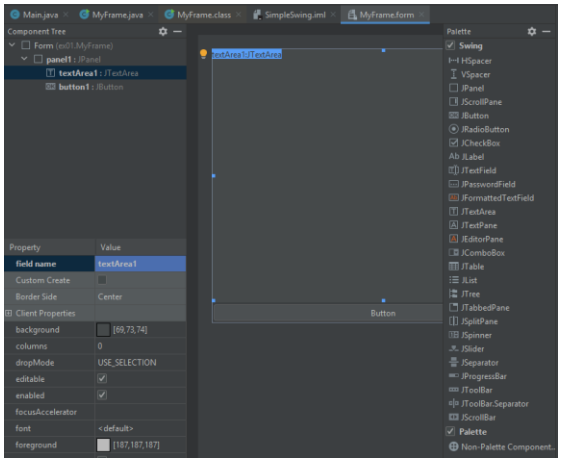
Implementacja graficznego interfejsu użytkownia w Java

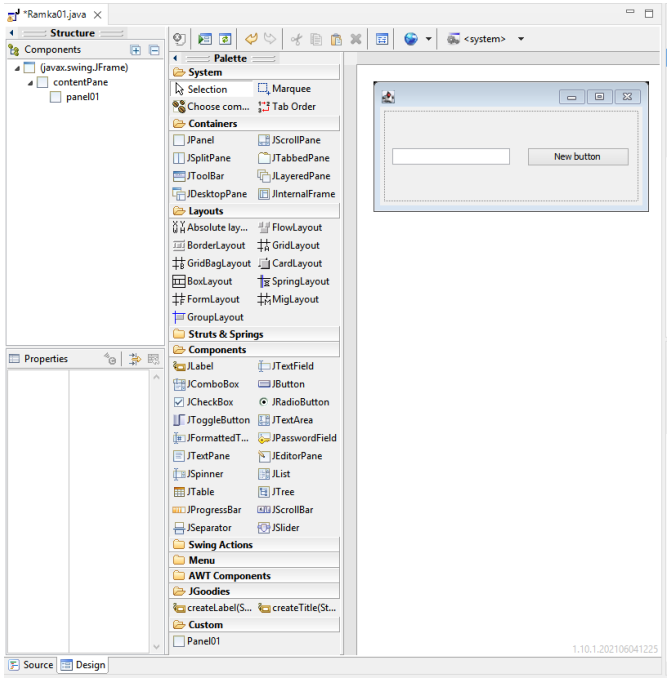
- Zintegrowane środowiska programowania zwykle oferują narzędzia do realizacji tego zadania.

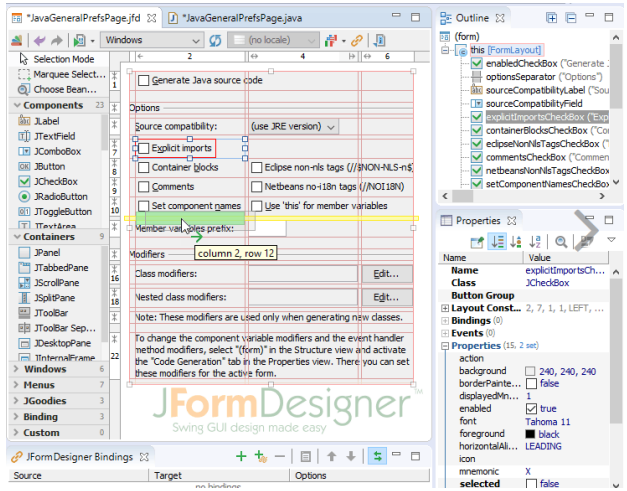
W IntelliJ IDEA istnieje Swing UI Designer (z GUI Designer).

W eclipse narzędzie dostarczane jest w postaci rozszerzenia WindowBuilder (zawiera SwingDesiger).

W obu środowiskach można używać komercyjnego rozszerzenia JFormDesigner.

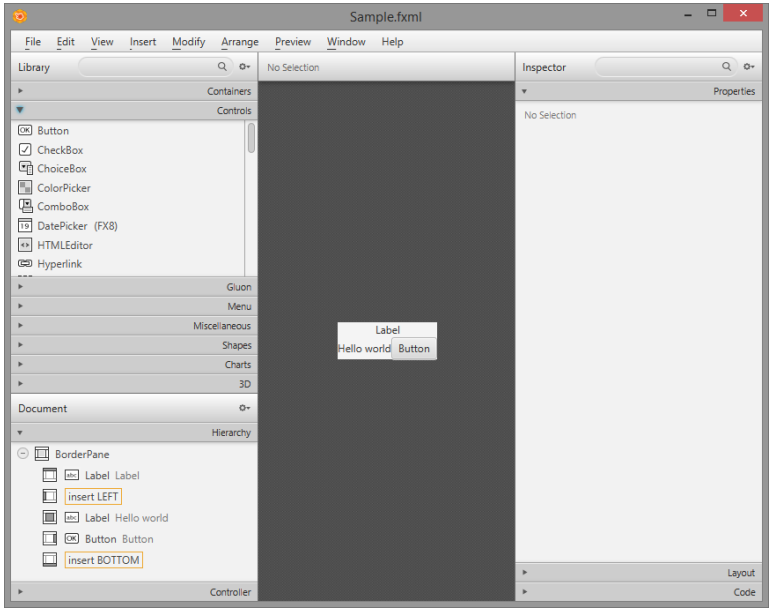






Implementacja graficznego interfejsu użytkownika w JavaFX

- Zwykle odbywa się z wykorzystaniem narzędzia SceneBuilder
  - narzędzie to generuje opis interfejsu z języka fxml,
  - na podstawie tego opisu podczas inicjalizacji aplikacji interfejs zostanie utworzony w dynamiczny sposób.



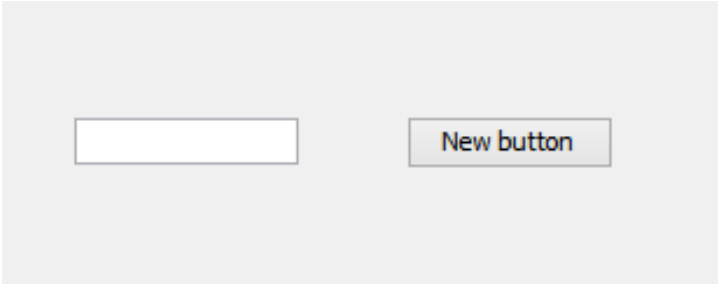

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.Button?>
<?import javafx.scene.control.Label?>
<?import javafx.scene.layout.BorderPane?>

<BorderPane xmlns="http://javafx.com/javafx/17"
xmlns:fx="http://javafx.com/fxml/1" fx:controller="application.SampleController">
   <right>
      <Button fx:id="myButton" mnemonicParsing="false" onAction="#clicked"
text="Button" BorderPane.alignment="CENTER" />
   </right>
   <center>
      <Label fx:id="myLabel" text="Hello world" BorderPane.alignment="CENTER" />
   </center>
   <top>
      <Label text="Label" BorderPane.alignment="CENTER" />
   </top>
</BorderPane>
```

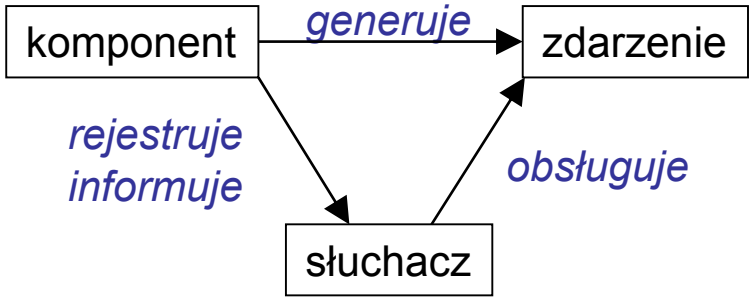
```
public class Main extends Application {
  @Override
  public void start(Stage primaryStage) {
    try {
      BorderPane root = (BorderPane)FXMLLoader.load(getClass().getResource("Sample.fxml"));
      Scene scene = new Scene(root,400,400);
      scene.getStylesheets().add(getClass().getResource("application.css").toExternalForm());
      primaryStage.setScene(scene);
      primaryStage.sizeToScene();
      primaryStage.show();
    } catch(Exception e) {  e.printStackTrace(); }
  }

  public static void main(String[] args) {
    launch(args);
  }
}
```

Zdarzenia komponentów Swing

- Model obsługi zdarzeń działa zgodnie z wzorcem ***obserwatora***

```
public class Panel01 extends JPanel {
  private JTextField textField;
  public Panel01() {
    setLayout(null);
    textField = new JTextField();
    textField.setBounds(45, 87, 96, 20);
    add(textField);
    textField.setColumns(10);

    Button btnNewButton = new JButton("New button");
    btnNewButton.addActionListener(
          e->textField.setText("AAA"));
    btnNewButton.setBounds(187, 86, 89, 23);
    add(btnNewButton);
  }
}
```

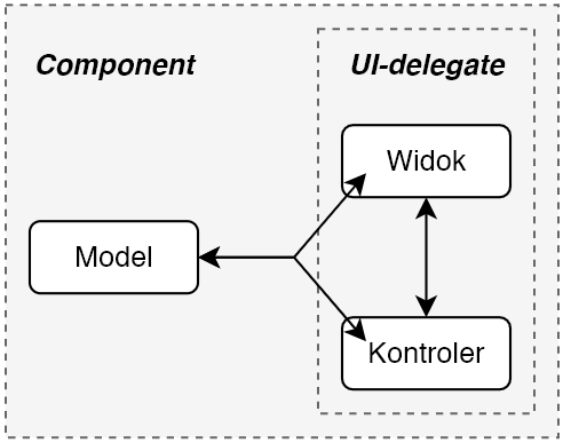




Proszę zajrzeć do `ex02` w `Wyk04-2021.zip`

Wzorzec ***model-delegat***

- Komponenty Swing korzystają z wzorca ***model-delegat*** (ang. *model-delegate*), który jest uproszczonym wariantem wzorca MVC (ang. *model-view-controller*).
- We wzorcu tym obiekty widoku (ang. *view*) oraz kontrolera (ang. *controller*) są zagregowane w pojedynczym elemencie, zwanym ***delegatem*** *interfejsu użytkownika* (ang. *UI-delegate*), który prezentuje komponent na ekranie oraz obsługuje zdarzenia GUI.
- ***delegat*** współpracuje z **modelem**, przy czym sposób interakcji **delegata** z **użytkownikiem** można kontrolować poprzez ustawienie w nim odpowiedniego **edytora** (ang. *editor*) i **renderera** (ang. *render*) dla każdego obsługiwanego typu danych
  - **model** zapewnia dostęp do danych,
  - **edytor** umożliwia edycję danych na interfejsie gragicznym,
  - **renderer** służy do graficznej prezentacji danych.
- Choć w Java API dostarczono **domyślne implementacje** modeli, edytorów i rendererów, dobrze jest zaimplementować ich **własne wersje** (implementując odpowiednie interfejsy z Java API).




```
table = new JTable();
table.setModel(new
DefaultTableModel(
  new Object[][] {
    { "some", "text" },
    { "any", "text" },
    { "even", "more" },
    { "text", "values" } },
      new Object[] {
       "Column 1",
       "Column 2"}
));
scrollPane.setViewportView(table);
```

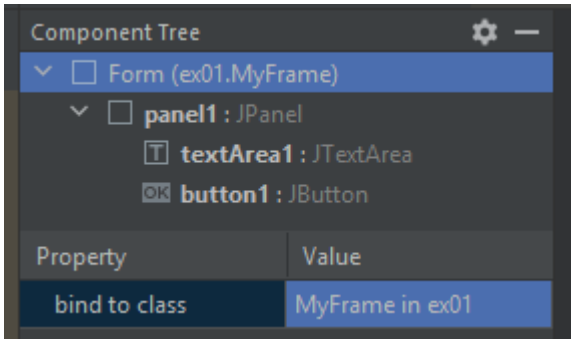
Column 1	Column 2
some	text
any	text
even	more
text	values

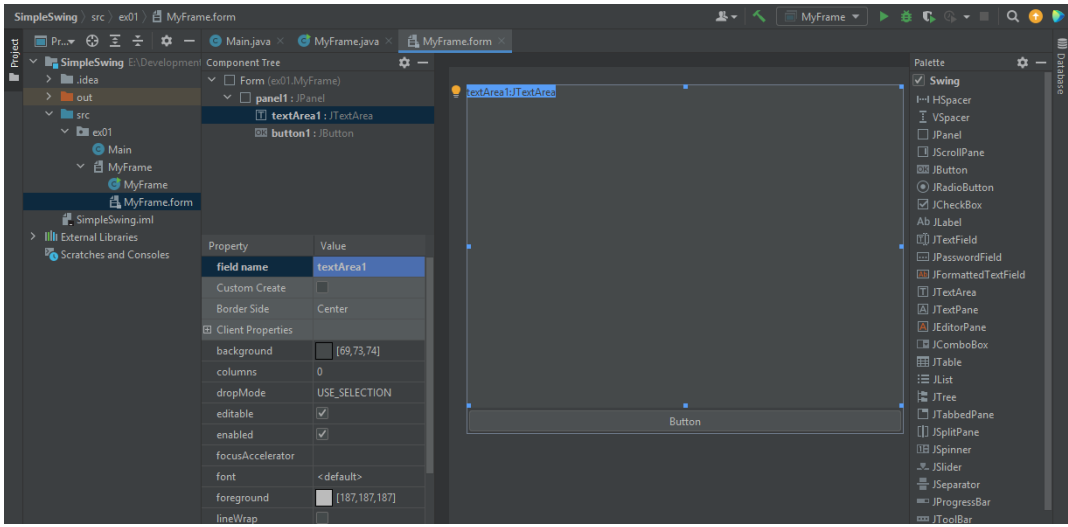
Do poczytania:
https://docs.oracle.com/javase/tutorial/uiswing/components/index.html

Uwagi do GUI Designer (IntelliJ IDEA)

- W IntelliJ IDEA graficzny interfejs użytkownika można zaprojektować korzystając z narzędzia *GUI Designer*.
  - narzędzie to tworzy opis interfejsu w postaci pliku xml z rozszerzeniem `.form`
  - na bazie tak przygotowanego opisu podczas budowania projektu dokładane są do niego dodatkowe klasy Java odpowiedzialne za wygląd i częściowe zachowania używanych komponentów graficznych,
  - do zaprojektowanej formy można dowiązać wybraną klasę
- Przyjęte podejście jest więc trochę podobne do podejścia zastosowanego w JavaFX (graficzny interfejs opisuje się w jakimś języku, a potem, na podstawie tego opisu, tworzone są instancje odpowiednich klas).

```
<?xml version="1.0" encoding="UTF-8"?>
<form xmlns="http://www.intellij.com/uidesigner/form/" version="1" bind-to-
class="ex01.MyFrame">
  <grid id="27dc6" binding="panel1" default-binding="true" layout-
manager="BorderLayout" hgap="0" vgap="0">
    <constraints>
      <xy x="20" y="20" width="500" height="400"/>
    </constraints>
    <properties/>
    <border type="none"/>
    <children>
      <component id="7072" class="javax.swing.JTextArea" binding="textArea1" default-
binding="true">
        <constraints border-constraint="Center"/>
        <properties/>
      </component>
      <component id="be29d" class="javax.swing.JButton" binding="button1" default-
binding="true">
        <constraints border-constraint="South"/>
        <properties>
          <text value="Button"/>
        </properties>
      </component>
    </children>
  </grid>
</form>
```

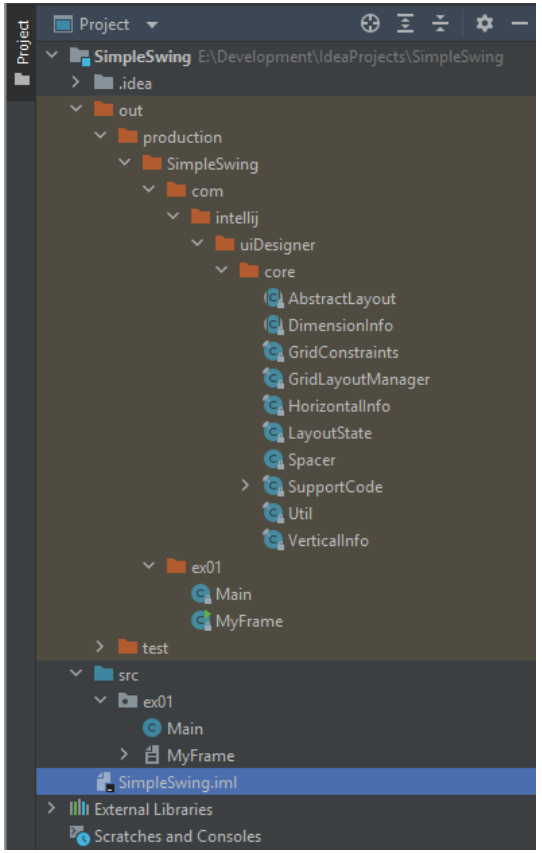


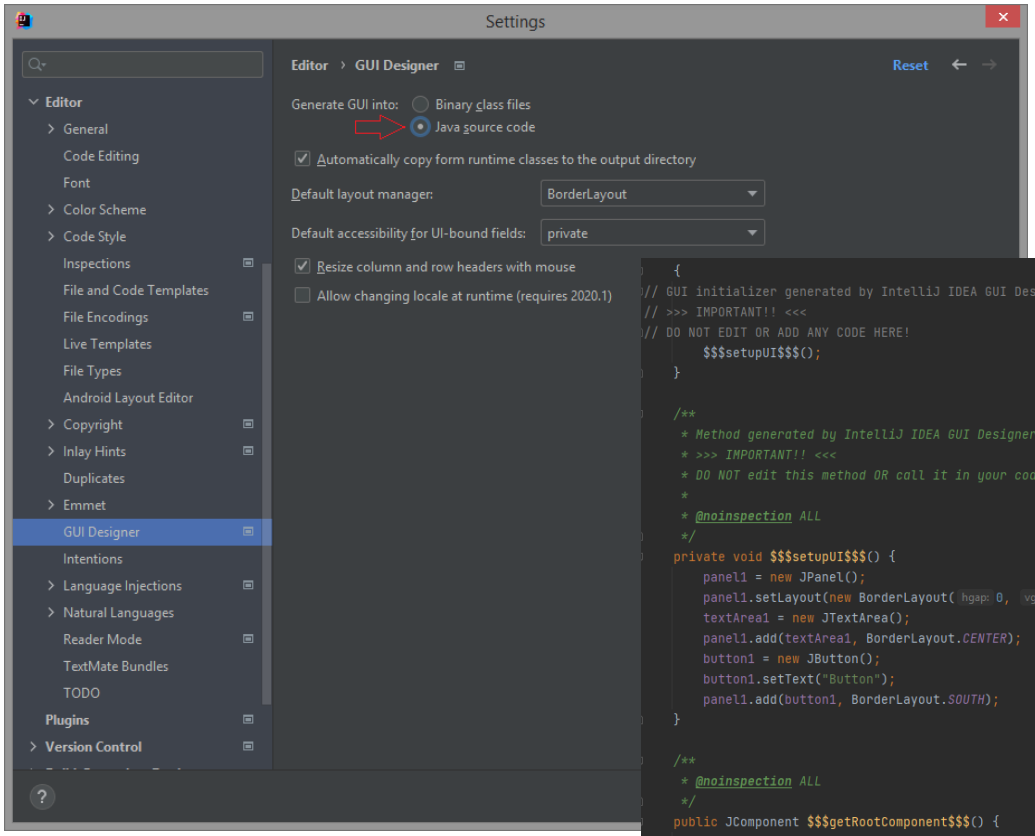


Do poczytania:
https://www.jetbrains.com/help/idea/gui-designer.html

Uwagi do GUI Designer (IntelliJ IDEA)

- Źródła kodu dodatkowych klas są niewidocze, jednak w wynikach kompilacji można znaleźć ich kod bajtowy.
- O tym, jaka „magia” dzieje się w kodzie można się przekonać po wybraniu opcji *Generate GUI info/Java source code* i przebudowaniu projektu:
  - w klasie dowiązanej do formy pojawi się dodatkowa metoda oraz blok inicjalizacji instancyjnej.






```
{
// GUI initializer generated by IntelliJ IDEA GUI Designer
// >>> IMPORTANT!! <<<
// DO NOT EDIT OR ADD ANY CODE HERE!
        $$$setupUI$$$();
    }

    /**
     * Method generated by IntelliJ IDEA GUI Designer
     * >>> IMPORTANT!! <<<
     * DO NOT edit this method OR call it in your code!
     *
     * @noinspection ALL
     */
    private void $$$setupUI$$$() {
        panel1 = new JPanel();
        panel1.setLayout(new BorderLayout( hgap: 0, vgap: 0));
        textArea1 = new JTextArea();
        panel1.add(textArea1, BorderLayout.CENTER);
        button1 = new JButton();
        button1.setText("Button");
        panel1.add(button1, BorderLayout.SOUTH);
    }

    /**
     * @noinspection ALL
     */
    public JComponent $$$getRootComponent$$$() {
        return panel1;
    }
}
```