Fundusze Europejskie
Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska
Europejski Fundusz Społeczny

JAVA FOUNDATIONS:
EXAM NUMBER: 1Z0-811

Fundusze Europejskie
Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska
Europejski Fundusz Społeczny

BARTOSZ ANDREATTO

Programista w Banku Pekao S.A z bogatym doświadczeniem w zakresie technologii back-end'owych wykorzystujących wirtualną maszynę Javy. Pasjonat rozwiązań opartych na ekosystemie Spring oraz rozwiązań Oracle. Certyfikowany programista Java. Uwielbia nauczać oraz dzielić się wiedzą.

LinkedIn: https://www.linkedin.com/in/bartosz-andreatto-02b03413a/

e-mail: bandreatto@gmail.com

Fundusze Europejskie
Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska
Europejski Fundusz Społeczny

ZREALIZOWANE SZKOLENIA:

- JDBC i Hibernate
- Testowanie oprogramowania – zaawansowane
- Spring
- Java podstawy: programowanie
- Java – zaawansowana
- Wzorce projektowe i dobre praktyki
- Relacyjne bazy danych
- Testowanie
- Usługi oparte o architekturę REST i wprowadzenie do Spring Security
- Java SE 8 Programmer II

ORGANIZACYJNIE

- Repozytorium GIT:

  https://github.com/bandreatto/1Z0-811_slupsk_2023_04
- 28 (**7 x 4**) godzin lekcyjnych
- Harmonogram:

  8.00- 9.30 (2h lekcyjne)

  9.30 - 9.40 – przerwa

  9.40 - 11.10 (2h lekcyjne)

  11.10 - 11.20 – przerwa

  11.20 - 12.50 (2h lekcyjne)

  12.50 - 13.20 (przerwa obiadowa)

  13.20 - 14.05 (1h lekcyjna)

Fundusze Europejskie
Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska
Europejski Fundusz Społeczny

ZAKRES SZKOLENIA

- Pojęcia związane z programowaniem obiektowym
- Zaawansowane aspekty klas
- Wyjątki i asercje, podstawy obsługi tekstu
- Typy generyczne w Javie
- Kolekcje w Javie
- Programowanie funkcyjne w Javie
- Wejście / wyjście (Java IO, NIO.2)
- Obsługa daty i czasu w Java SE 8
- Programowanie wielowątkowe
- Dostęp do baz danych za pomocą interfejsu JDBC

EGZAMIN

- **Java Foundations | Exam Number: 1Z0-811**

  https://education.oracle.com/java-foundations/pexam_1Z0-811

  - liczba pytań: **60**

  - czas trwania: **120 minut**

  - **pytania wielokrotnego wyboru**

  - próg zdawalności: **65%**

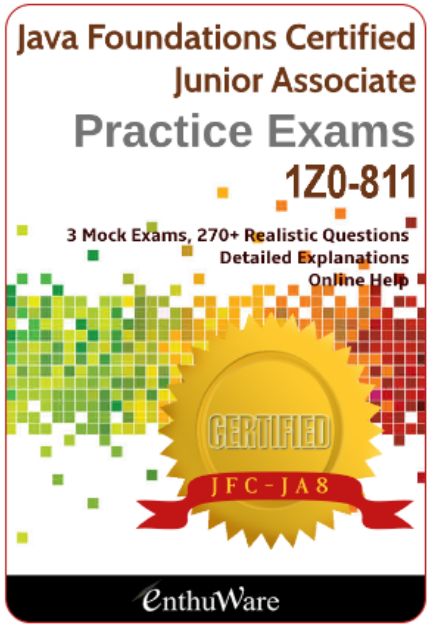
Fundusze Europejskie
Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska
Europejski Fundusz Społeczny

EGZAMIN - ZAKRES

- **Czym jest JAVA** (cechy języka i zastosowania)
- **Podstawy języka** (JDK i JRE, kompilacja, uruchamianie)
- **Podstawowe elementy języka** (konwencja, słowa kluczowe)
- **Typy danych**
- **Operatory**
- **Klasa String**
- **Klasy Random i Math**
- **Instrukcje warunkowe**
- **Pętle**
- **Debugowanie i przechwytywanie wyjątków**
- **Tablice i kolekcja ArrayList**
- **Klasy i metody**

PRZYGOTOWANIE DO EGZAMINU

- https://enthuware.com/java-certification-mock-exams/oracle-certified-associate/java-foundation-1z0-811
- https://www.whizlabs.com/ocfa-java-foundations-1z0-811/



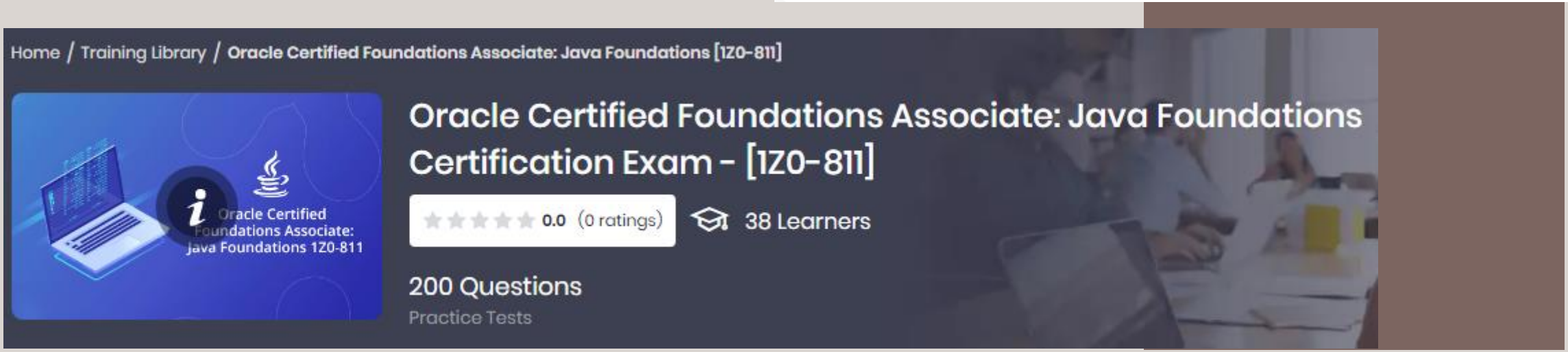
JF+ V8 for Java Certified Foundations Associate SE 8 1Z0-811

Price 9.99 USD

Updated Sep 2022 Reflects the new JCFA Java Foundations Certification Exam (JCFA - 1Z0-811) pattern!
4 Mock Exams - 270+ Questions - NO FLUFF*
Exam Code - **1Z0-811**

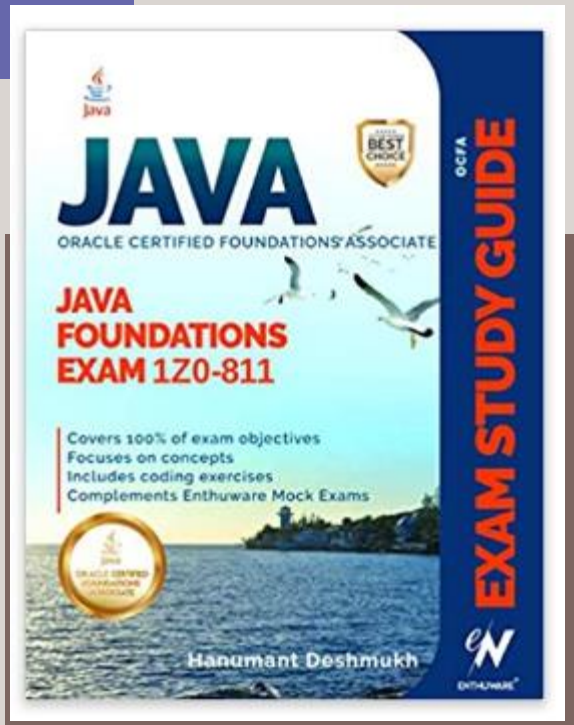
Enthuware JFC Java Practice Tests Features

1. 270+ Questions divided into 4 Mock Exams
2. **Detailed Explanations**
3. Questions covering all exam objectives
4. Highly Customizable - Standard tests, Sectionwise/Objectivewise tests, and Custom tests
5. Helpful URLs/Resources
6. Comparative Scores - See JFC Java Foundations Certification Test Scores



PRZYGOTOWANIE DO EGZAMINU

- https://coderanch.com/f/84/java-foundations
- **OCFA Java Foundations Exam Fundamentals 1Z0-811: Study guide for Oracle Certified Foundations Associate, Java Certification https://books.google.pl/books?id=_BcIEAAAQBAJ&printsec=copyright&redir_esc=y#v=onepage&q&f=false**



W zakresie szkolenia (wykraczające zakresem tematycznym w kontekście wymagań egzaminacyjnych):

- **OCA: Oracle Certified Associate Java SE 8 Programmer I Study Guide**
- **OCP: Oracle Certified Professional Java SE 8 Programmer II Study Guide**

ZAKRES SZKOLENIA – DZIEŃ 1

- **Pojęcia związane z programowaniem obiektowym**
- **Zaawansowane aspekty klas**

  enkapsulacja, dziedziczenie, polimorfizm, klasa Object, modyfikator static, final, klasy abstrakcyjne, enumeracje, interfejsy
- **Wyjątki i asercje, podstawy obsługi tekstu**

  try-catch, throw, finally, asercje

Fundusze Europejskie Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska Europejski Fundusz Społeczny

BUDOWA KLAS W JAVIE

Klasy w programowaniu obiektowym służą do opisywania otaczających nas przedmiotów, zdarzeń, czynności, stanów oraz relacji między opisywanymi przedmiotami. Takie typy zdefiniowane za pomocą klas nazywamy **typami złożonymi** lub też **typami referencyjnymi**.

Klasy posiadają dwie podstawowe składowe:

- **pola** – zmienne lub stałe opisujące obiekt danej klasy
- **metody** – operacje, które udostępnia nasza klasa

OOP

Koncepcja **O**bject **O**riented **P**rogramming (OOP) powiązana jest z językami programowania, które korzystają z obiektów. OOP wprowadza i łączy koncepcje, takie jak:

- **dziedziczenie**
- **polimorfizm**
- **enkapsulacja**
- **abstrakcja**
- **kompozycja**

ENKAPSULACJA

Enkapsulacja jest mechanizmem, który umożliwia **ukrywanie danych i metod przed „światem zewnętrznym”**. Enkapsulacja umożliwia udostępnianie tylko tych mechanizmów i danych, które mają być widoczne z zewnątrz klasy. Ukrywanie danych jest realizowane poprzez **modyfikatory dostępu**.

W przykładzie dostęp do danych o pracowniku **został ukryty** dla pól id oraz dateOfBirth. Został ograniczony tylko i wyłącznie do odczytu poprzez metody typu getter.

```java
public class Employee {
    private final String id;
    private final LocalDate dateOfBirth;
    private String firstName;
    private String lastName;

    public Employee(String id, String firstName, String lastName,
                    LocalDate dateOfBirth) {
        this.id = id;
        this.firstName = firstName;
        this.lastName = lastName;
        this.dateOfBirth = dateOfBirth;
    }

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public String getId() {
        return id;
    }

    public LocalDate getDateOfBirth() {
        return dateOfBirth;
    }
}
```

MODYFIKATORY DOSTĘPU

Can access	If that member is `private`?	If that member has default (package private) access?	If that member is `protected`?	If that member is `public`?
Member in the same class	yes	yes	yes	yes
Member in another class in the same package	no	yes	yes	yes
Member in a superclass in a different package	no	no	yes	yes
Method/field in a class (that is not a superclass) in a different package	no	no	no	yes

ENKAPSULACJA

UWAGA: Zawsze powinniśmy chować szczegóły implementacyjne danej klasy przed użytkownikiem końcowym klasy, wykorzystując enkapsulację.

Fundusze Europejskie
Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska
Europejski Fundusz Społeczny

POLIMORFIZM

Mechanizm nazywany inaczej **wielopostaciowością**. Polega on głównie na tym, że programista korzystający z jakiegoś obiektu, **nie musi wiedzieć, czy jego funkcjonalność pochodzi z klasy będącej typem danego obiektu, czy jakiejś innej klasy, która implementuje lub dziedziczy po interfejsie lub klasie bazowej**. Innymi słowy, jest to zdolność obiektów do zwracania różnych odpowiedzi na to samo żądanie.

Polimorfizm jest bardzo często wykorzystywany **podczas pracy z kolekcjami**

```java
public class MainClass {
    public static void main(String[] args) {
        List<Integer> ints = new ArrayList<>(); // programista pracuje na
        // INTERFEJSIE, wykorzystując pewną implementację
        ints.add(1);
        ints.add(7);
        for (int i = 0; i < ints.size(); i++) {
            System.out.println(ints.get(i));
        }
    }
}
```

Fundusze Europejskie
Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska
Europejski Fundusz Społeczny

POLIMORFIZM

W kolejnym przykładzie definiujemy pewną hierarchię - **VodPlayer** i jej trzy klasy pochodne, **NetflixPlayer**, **HBOGoPlayer** i **DefaultPlayer**. W klasie **Android TV** wykorzystujemy polimorfizm pracując na abstrakcji, której implementacja jest wybierana na podstawie argumentów wejściowych aplikacji.

```java
public class VodPlayer {
    public void play(String title) {
        System.out.println("Playing " + title);
    }
}

public class NetflixPlayer extends VodPlayer {
    @Override
    public void play(final String title) {
        System.out.println("Playing " + title + " on Netflix");
    }
}

public class HBOGoPlayer extends VodPlayer {
    @Override
    public void play(final String title) {
        System.out.println("Playing " + title + " on HBO");
    }
}

public class DefaultPlayer extends VodPlayer {
    @Override
    public void play(final String title) {
        System.out.println("Playing " + title + " on default player");
    }
}

public class AndroidTV {
    public static void main(String[] args) {
        final String player = args[0];
        VodPlayer vodPlayer;
        if (player.equals("Netflix")) {
            vodPlayer = new NetflixPlayer();
        } else if (player.equals("HBO")) {
            vodPlayer = new HBOGoPlayer();
        } else {
            vodPlayer = new DefaultPlayer();
        }
        playEpisode(vodPlayer, "GOT_S1E1");
    }

    static void playEpisode(VodPlayer vodPlayer, String title) {
        // nie wiemy z jaką implementacją mamy do czynienia
        vodPlayer.play(title);
    }
}
```

INSTANCEOF

Jeżeli chcielibyśmy **sprawdzić, czy instancja danej klasy jest danego typu**, a następnie go wykorzystać możemy zrobić to w dwóch krokach, wykorzystując operator **instanceof**.

```
class HeavyAnimal { }
class Hippo extends HeavyAnimal { }
class Elephant extends HeavyAnimal { }
```

```
HeavyAnimal hippo = new Hippo();
boolean b1 = hippo instanceof Hippo;          // true
boolean b2 = hippo instanceof HeavyAnimal;    // true
boolean b3 = hippo instanceof Elephant;       // false
```

```
HeavyAnimal hippo = new Hippo();
boolean b4 = hippo instanceof Object;      // true
Hippo nullHippo = null;
boolean b5 = nullHippo instanceof Object;  // false
```

```
Hippo anotherHippo = new Hippo();
boolean b5 = anotherHippo instanceof Elephant; // DOES NOT COMPILE
```

```
public void feedAnimal(Animal animal) {
   if(animal instanceof Cow) {
      ((Cow)animal).addHay();
   } else if(animal instanceof Bird) {
      ((Bird)animal).addSeed();
   } else if(animal instanceof Lion) {
       ((Lion)animal).addMeat();
   } else {
      throw new RuntimeException("Unsupported animal");
   } }
```

KLASY WEWNĘTRZNE

W Javie **możliwe jest deklarowanie klas wewnątrz innych klas**. Klasy te nazywamy klasami zagnieżdżonymi. Mogą one być zdeklarowane jako:

- **klasy statyczne** (tzw. static nested), z wykorzystaniem słowa kluczowego static
- klasy niestatyczne (tzw. **non-static** lub **inner**)

UWAGA: Istnieją dwa specjalne rodzaje klas wewnętrznych: **klasy lokalne** i **klasy anonimowe**.

```
public class Outer {

    static class NestedStatic {
        // class body
    }

    class Inner {
        // class body
    }
}
```

Fundusze Europejskie
Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska
Europejski Fundusz Społeczny

KLASY WEWNĘTRZNE - WIDOCZNOŚĆ

Niestatyczne klasy zagnieżdżone mają bezpośredni dostęp do klasy nadrzędnej, tzn. z klasy nadrzędnej mogą wykorzystywać:

- **pola** (również statyczne i prywatne)
- **metody** (również statyczne i prywatne)

```
public class OuterClass {

  private static int outerClassStaticField;
  private int outerClassField;

  void outerClassMethod() {
    System.out.println("I am outer class method");
  }

  public class InnerClass {
    void useOuterClassField() {
      System.out.println(outerClassStaticField); // wykorzystanie statycznego pola
      outerClassMethod();                        // wykorzystanie metody
      System.out.println(outerClassField);       // wykorzystanie prywatnego pola
    }
  }
}
```

KLASY WEWNĘTRZNE - WIDOCZNOŚĆ

UWAGA: W przeciwieństwie do klas lokalnych, klasy wewnętrzne **mogą wykorzystywać modyfikatory dostępu**.

UWAGA: Niestatyczne klasy lokalne **nie mogą definiować pól i metod statycznych**.

Z kolei zagnieżdżone klasy statyczne z klasy nadrzędnej:

- mogą korzystać ze statycznych pól i metod klasy nadrzędnej
- nie mogą korzystać z niestatycznych pól i metod

```
public class OuterClass {

  private static int outerClassStaticField;
  private int outerClassField;

  void outerClassMethod() {
    System.out.println("I am outer class method");
  }

  protected static void outClassStaticMethod() {
    System.out.println("I am out class static method");
  }

  static class InnerStaticClass {

    void useOuterClassField() {
      System.out.println(outerClassStaticField);
      outClassStaticMethod();
      //outerClassMethod(); -> błąd kompilacji, NIE możemy korzystać z metod niestatycznych
    }
  }
}
```

KLASY WEWNĘTRZNE - KREACJA

Kolejne przykłady oprzemy o następujące klasy:

```
public class OuterClass {

  class InnerClass {
  }

  static class InnerStaticClass {
  }
}
```

Aby stworzyć **instancję wewnętrznej klasy niestatycznej**, musimy **najpierw stworzyć instancję klasy zewnętrznej**, np.:

```
OuterClass outerClass = new OuterClass();
final OuterClass.InnerClass innerClass = outerClass.new InnerClass();
```

```
OuterClass.InnerStaticClass innerStaticClass = new OuterClass.InnerStaticClass();
```

Fundusze Europejskie
Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska
Europejski Fundusz Społeczny

KLASY LOKALNE

Klasy lokalne są **klasami deklarowanymi w bloku kodu**, np. metodzie, pętli for, czy instrukcji if. Deklarując klasę lokalną **pomijamy informacje o modyfikatorze dostępu**. Obiekty klasy lokalnej mają dostęp do pól metod klas nadrzędnych. Kolejny przykład pokazuje wykorzystanie klasy lokalnej bezpośrednio w metodzie main.

```
public static void main(String[] args) {
  final List<String> names = List.of("Kasia", "Magda", "Gosia");
  final List<String> surnames = List.of("Piszczyk", "Olszańska", "Budrzeńska");
  final int someVariable = 3; // zmienna do zaprezentowania dostępu zmiennych z klas lokalnych

  class Name { // stworzenie definicji klasy lokalnej, bez modyfikatora dostępu
    private final String firstName;
    private final String lastName;

    public Name(final String firstName, final String lastName) {
      this.firstName = firstName;
      this.lastName = lastName;
    }

    public String getReadableName() {
      System.out.println("Hey I can use outer variable " + someVariable);
      return firstName + " " + lastName;
    }
  }

  for (int idx = 0; idx < names.size(); idx++) {
    final Name name = new Name(names.get(idx), surnames.get(idx)); // wykorzystanie klasy lokalnej
    System.out.println(name.getReadableName());
  }
}
```

KLASY LOKALNE

UWAGA: **W bloku kodu nie można deklarować interfejsów.**

Klasy lokalne nie są jednak tak elastyczne jak "zwykłe" klasy. Otóż:

- nie mogą definiować metod statycznych
- nie mogą zawierać pól statycznych
- ale mogą zawierać statyczne stałe, tzn. takie z modyfikatorami **static final**

```
class InnerClassExample {
  private static final String APP_NAME = "DummyApp"; // OK
  // Błąd kompilacji, brak modyfikatora final
  private static String INCORRECT_FIELD = "IAmMissingFinal";

  public void printAppName() {
    System.out.println(APP_NAME);
  }

  // Błąd kompilacji, metoda statyczna w klasie lokalnej
  public static void shouldNotBeDeclaredHere() {}
}
```

KLASY ANONIMOWE

Klasy anonimowe **działają tak samo, jak klasy lokalne**.

Różnią się tylko tym, że klasy anonimowe:

- nie mają nazwy
- powinny być deklarowane jeśli potrzebujemy ich tylko jeden raz

KLASY ANONIMOWE - SKŁADNIA

Tworzenie obiektu klasy anonimowej wygląda niemal **identycznie jak w przypadku tworzenia zwykłego obiektu**. Różnica polega na tym, że podczas tworzenia obiektu, implementuje się wszystkie wymagane metody. Wyrażenie składa się z:

- **operatora new** lub zastosowania **lambdy** w przypadku interfejsu funkcyjnego
- **nazwy interfejsu**, który implementujemy lub **klasy abstrakcyjnej**, którą dziedziczymy
- **parametrów konstruktora** (w przypadku interfejsu korzystamy z pustego konstruktora)
- **ciała klasy/interfejsu**

KLASY ANONIMOWE - SKŁADNIA

Tworzenie obiektu klasy anonimowej wygląda niemal **identycznie jak w przypadku tworzenia zwykłego obiektu**. Różnica polega na tym, że podczas tworzenia obiektu, implementuje się wszystkie wymagane metody. Wyrażenie składa się z:

- **operatora new** lub zastosowania **lambdy** w przypadku interfejsu funkcyjnego
- **nazwy interfejsu**, który implementujemy lub **klasy abstrakcyjnej**, którą dziedziczymy
- **parametrów konstruktora** (w przypadku interfejsu korzystamy z pustego konstruktora)
- **ciała klasy/interfejsu**

KLASY ANONIMOWE - SKŁADNIA

W ramach klasy anonimowej możemy deklarować:

- pola (w tym pola statyczne)
- metody (w tym metody statyczne)
- Stałe

Natomiast nie można deklarować:

- konstruktorów
- Interfejsów
- bloków statycznej inicjalizacji

```java
public interface ClickListener {
  void onClick();
}

public class UIComponents {
    void showComponents() {
        // implementacja klasy anonimowej z wykorzystaniem słowa kluczowego new
        ClickListener buttonClick = new ClickListener() {
            @Override
            public void onClick() {
                System.out.println("On Button click!");
            }
        };
        // koniec implementacji klasy anonimowej

        buttonClick.onClick();

        // implementacja klasy anonimowej z wykorzystaniem lambdy
        // jest to możliwe, ponieważ ClickListener jest interfejsem funkcyjnym,
        // tzn. ma jedną metodę do zaimplementowania
        ClickListener checkboxClick  = () -> System.out.println("On Checkbox click!");
        checkboxClick.onClick();
    }
}
```

KLASY ANONIMOWE - SKŁADNIA

Kolejny przykład pokazuje **dodatkowe możliwości klasy anonimowej** zaimplementowanej z wykorzystaniem słowa kluczowego new:

```
public interface ClickListener {
  void onClick();
}

void showComponentsV2() {
    // implementacja klasy anonimowej
    ClickListener buttonClick = new ClickListener() {

      private String name; // pole w klasie anonimowej
      // pole statyczne w klasie anonimowej
      private static final String BUTTON_CLICK_MESSAGE ="On Button click!";

      public void sayHello() { // implementacja metody w klasie anonimowej
        System.out.println("I am new method in anonymous class");
      }

      @Override
      public void onClick() {
        sayHello();
        System.out.println(BUTTON_CLICK_MESSAGE);
      }
    };

  }
```

DOSTĘP DO KLAS NADRZĘDNYCH

Klasy anonimowe **mogą odwoływać się do pól klasy nadrzędnej i zmiennych lokalnych**, pod warunkiem, że są **finalne**. Podobnie jak w przypadku klas zagnieżdżonych zmienne, o nazwach takich samych jak pola klasy nadrzędnej, przysłaniają ich właściwości.

KONTRAKT W JAVIE

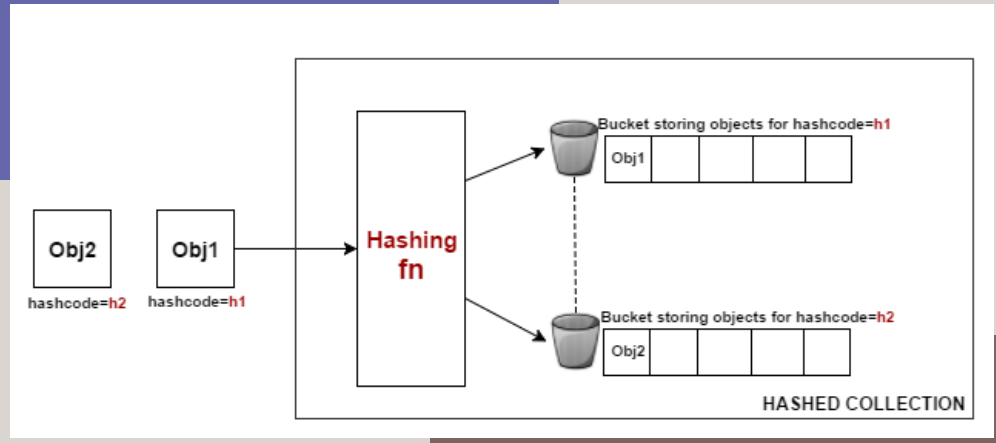
Dodając elementy do takich kolekcji, jak np. **Set**, na jakiejś podstawie musimy stwierdzić, **czy taki obiekt w takiej kolekcji się już znajduje**. Kontrakt w Javie jest ściśle powiązany z metodami **equals** i **hashCode**, które **deklarowane są w klasie Object**. Implementacje obydwu metod są często wykorzystywane łącznie.

Fundusze Europejskie
Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska
Europejski Fundusz Społeczny

KONTRAKT W JAVIE

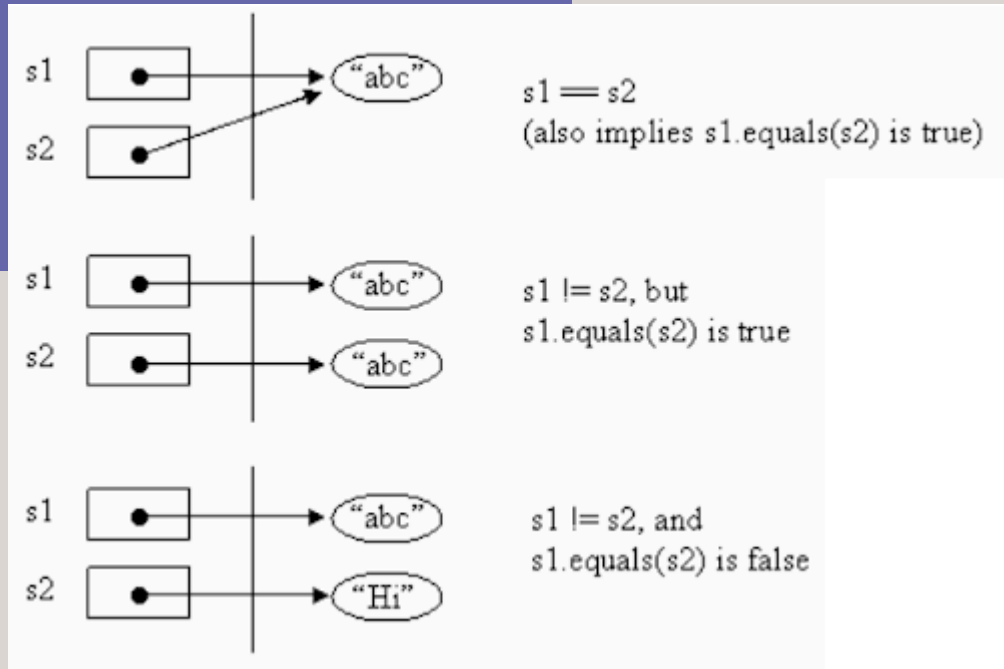
Kontrakt oparty jest o następujące zasady:

- **jeśli x.equals(y) == true**, to wówczas wymagane jest, aby **x.hashCode() == y.hashCode()**
- jeśli **x.hashCode() == y.hashCode()**, to nie jest wymagane, aby **x.equals(y) == true**
- **wielokrotne wywołanie metody hashCode** na tym samym obiekcie zawsze powinno dawać **taki sam wynik**.



EQUALS

Typy prymitywne w Javie porównywane są za pomocą operatora ==, natomiast ten sam operator logiczny, w przypadku porównywania obiektów, **porówna referencję do obiektów** (tzn. czy obie referencje wskazują ten sam adres), nie ich wartości. W celu porównania obiektów ze względu na wskazane wartości, wykorzystuje się nadpisaną metodę **equals**.



EQUALS

```java
public class Car {
    private String name;
    private String type;

    public Car(String name, String type) {
        this.name = name;
        this.type = type;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Car car = (Car) o;
        return Objects.equals(name, car.name) &&
                Objects.equals(type, car.type);
    }

    @Override
    public String toString() {
        return "Car{" +
                "name='" + name + '\'' +
                ", type='" + type + '\'' +
                '}';
    }
}
```

```java
int x = 10;
int y = 10;
boolean primitiveComparison = x == y;

Car car = new Car("BMW", "Sport");
Car car1 = new Car("BMW", "Sport");
// false, referencje są różne
boolean objectComparision = car == car1;
// true, jeżeli metoda equals jest zaimplementowana w intuicyjny sposób
boolean objectComparisionUsingEquals = car.equals(car1);
```

EQUALS

- jest dostępna w klasie Object
- można ją wywołać na każdym obiekcie
- metoda ta powinna być **zwrotna**, tzn. someObject.equals(someObject) powinna zwracać true
- powinna być metodą **symetryczną**, tzn. jeżeli x.equals(y) == true, to y.equals(x) == true
- metoda powinna być **przechodnia**, tzn. jeżeli x.equals(y) == true i y.equals(z) == true, to x.equals(z) == true
- metoda ta powinna byc´**spójna**, tzn. kilkukrotne wywołanie tej samej metody powinno dać ten sam wynik
- przy porównaniu z null metoda powinna zwracać false.

EQUALS

Metoda equals jest **wykorzystywana** podczas wywoływania, np. metody **contains z Collection API**, np.:

```
List<Car> cars = Arrays.asList(new Car("BMW", "Sport"), new Car("Volvo", "suv"));
boolean isVolvoInList = cars.contains(new Car("Volvo", "suv")); // true
```

HASHCODE

Metoda hashCode **zwraca typ prymitywny int** i docelowo ma oznaczać **pewnego rodzaju reprezentację liczbową obiektu**. Ponieważ typ ten ma ograniczony zakres, nie jest to unikalna wartość dla każdego obiektu, a co za tym idzie, **dwa różne obiekty mogą zwracać tę samą wartość hashCode**. To właśnie tę właściwość wykorzystuje wiele API Javy i **to ona jest podstawą kontraktu**. Metoda **hashCode** jest wykorzystywana do implementacji wielu struktur danych i algorytmów, np. **sortowania**.

HASHCODE

Metoda ta charakteryzuje się m.in.:

- metoda dostępna w klasie Object
- **implementacja metody hashCode powinna być powiązana z metodą equals**
- metoda powstaje najczęściej w oparciu o „**hashe" poszczególnych pól klasy**. Hashe atrybutów mnożymy najczęściej **przez liczby pierwsze** i sumujemy ze sobą.

UWAGA: W praktyce **rzadko ręcznie piszemy implementacje metod equals i hashcode**. Najczęściej je generujemy lub wykorzystujemy w tym celu zewnętrzne biblioteki.

```java
@Override
public int hashCode() {
  int result = name.hashCode();
  result = 31 * result + type.hashCode();
  return result;
}

Set<Car> cars = new HashSet<>();
cars.add(new Car("BMW", "Sport"));
cars.add(new Car("Volvo", "suv"));
boolean isVolvoInSet = cars.contains(new Car("Volvo", "suv"));
// true w przypadku, gdy metoda 'hashcode'
// jest zaimplementowana w klasie Car, false w przeciwnym wypadku
System.out.println(isVolvoInSet);
```

TOSTRING

W Javie **każda klasa dziedziczy po klasie Object**, w której znajduje się kilka publicznych metod, które również są automatycznie dziedziczone. Obok metody **equals()** i **hashCode()** do takich metod należy także metoda **toString()**, która zwraca opis obiektu w postaci Stringa.

```
public String toString()
```

```
public String toString() {
    return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```

Metodę toString() odziedziczoną z klasy Object warto **nadpisać własną implementacją**, aby wydruk był bardziej przyjazny i użyteczny

```
public static void main(String[] args) {
   System.out.println(new ArrayList());      // []
   System.out.println(new String[0]);        // [Ljava.lang.String;@65cc892e
}
```

```
public class Hippo {
   private String name;
   private double weight;
   public Hippo(String name, double weight) {
      this.name = name;
      this.weight = weight;
   }
   @Override
   public String toString() {
      return name;
   }
   public static void main(String[] args) {
      Hippo h1 = new Hippo("Harry", 3100);
      System.out.println(h1);     // Harry
   } }
```

Fundusze Europejskie
Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska
Europejski Fundusz Społeczny

BLOKI INICJALIZACYJNE

Bloki inicjalizacyjne występują w dwu rodzajach: jako **bloki instancyjne** i **bloki statyczne**. Blok inicjalizacyjny to fragment kodu, który wykonuje się – w przypadku **bloków statycznych** – **gdy ładowana jest klasa**, bądź – w przypadku **bloków instancyjnych** – gdy **tworzona jest instancja**.

```
public class Test {
    public Test() {
        System.out.println("constructor");
    }

    static {
        System.out.println("static init block");
    }

    {
        System.out.println("instance init block");
    }

    public static void main(String[] args) {
        System.out.println("start main");

        Test test = new Test();

        System.out.println("end main");
    }
}
```

BLOKI INICJALIZACYJNE

```
public class Counter {
        private static int count;
        static {
                // code in this static block will be executed when JVM loads the class into memory
                count = 1;
        }
        public Counter() {
                count++;
        }
        public static void printCount() {
                System.out.println("Number of instances created so far is: " + count);
        }
        public static void main(String []args) {
                Counter anInstance = new Counter();
                Counter.printCount();
                Counter anotherInstance = new Counter();
                Counter.printCount();
        }
}
```

Fundusze Europejskie
Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska
Europejski Fundusz Społeczny

ENUMERACJE

Typy wyliczeniowe, zwane częściej **enumami**, to specjalne typy obiektów w języku Java. Ich główną charakterystyką jest z góry znana ilość instancji. W celu stworzenia typu wyliczeniowego powinniśmy postąpić podobnie jak w przypadku tworzenia zwykłej klasy, tzn. stworzyć plik o dowolnej nazwie z rozszerzeniem java, a następnie wewnątrz pliku zdefiniować **typ wyliczeniowy o takiej samej nazwie jak plik.** Zamiast słowa kluczowego class wykorzystujemy **słowo kluczowe enum**.

```
public enum ExampleEnum {
 // zawartość
}
```

DEFINIOWANIE TYPÓW WYLICZENIOWYCH

Typ wyliczeniowy może posiadać **dowolną ilość predefiniowanych wartości** (instancji). Definiujemy je na początku ciała enuma. Wartości te najczęściej piszemy wielkimi literami i oddzielamy od siebie przecinkami, np.:

```
public enum JsonSerializationStrategy {
  SNAKE_CASE,
  CAMEL_CASE,
  KEBAB_CASE
}
```

ENUMERACJE - POBIERANIE INSTANCJI

W celu odwołania się do konkretnej wartości wykorzystujemy notację z kropką, np.:

```
JsonSerializationStrategy strategy = JsonSerializationStrategy.SNAKE_CASE;
```

Nie jest z kolei możliwe stworzenie nowej instancji takiego obiektu **przy pomocy słowa kluczowej new**:

```
// niemożliwe dla typu wyliczeniowego
JsonSerializationStrategy jsonSerializationStrategy = new JsonSerializationStrategy();
```

ENUMERACJE - PORÓWNYWANIE

Instancje enumów są **tworzone na starcie aplikacji**, każda z nich tworzona jest dokładnie raz. Stąd też chcąc porównywać instancje enumów możemy to zrobić przy pomocy metody **equals** ale również **==**, np.:

```
JsonSerializationStrategy strategyA = JsonSerializationStrategy.CAMEL_CASE;
JsonSerializationStrategy strategyB = JsonSerializationStrategy.CAMEL_CASE;
System.out.println(strategyA == strategyB); // true
System.out.println(strategyA.equals(strategyB)); // true
```

ENUMERACJE - PODOBIEŃSTWA

Typ wyliczeniowy pod względem mechanizmów jest bardzo podobny do klasy. Oprócz listy możliwych wartości enum może:

- zdefiniować konstruktory
- zdefiniować pola
- zdefiniować metody
- implementować interfejsy
- wykorzystywać modyfikatory dostępu

Enum ma jednak pewne ograniczenia, w przeciwieństwie do klasy **nie może**:

- **dziedziczyć po klasie**
- **nie może być wykorzystany jako klasa**

Fundusze Europejskie
Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska
Europejski Fundusz Społeczny

ENUMERACJE - KONSTRUKTORY, POLA

Jeżeli typ wyliczeniowy, oprócz predefiniowanych wartości, definiuje jakiekolwiek dodatkowe elementy (metody czy konstruktory) to **po liście wartości zawsze stawiamy średnik**.

Każdy enum może definiować wiele konstruktorów i wszystkie z nich muszą być **prywatne** (a najlepiej nie wykorzystywać żadnego modyfikatora dostępu).
W przypadku gdy konstruktor posiada parametry, chcąc go wykorzystać, musimy przekazać wartości argumentów bezpośrednio przy predefiniowanej wartości typu wyliczeniowego.

```
public enum JsonSerializationStrategy {
  SNAKE_CASE("snake case"),
  CAMEL_CASE("camel case"),
  KEBAB_CASE("kebab case");

  private final String readableName;

  JsonSerializationStrategy(final String readableName) {
    this.readableName = readableName;
  }
}
```

ENUMERACJE - METODY, INTERFEJSY

Każdy typ wyliczeniowy może również **tak jak klasa**, **implementować interfejsy i definiować metody**:

```
JsonSerializationStrategy strategy = JsonSerializationStrategy.CAMEL_CASE;
System.out.println(strategy.getId() + " " + strategy.getReadableName());
```

```
public interface IdProvider {
  String getId();
}

public enum JsonSerializationStrategy implements IdProvider {
  SNAKE_CASE("snake case"),
  CAMEL_CASE("camel case", "1"),
  KEBAB_CASE("kebab case", "2");

  private final String readableName;
  private final String id;

  JsonSerializationStrategy(final String readableName) {
    this.readableName = readableName;
    this.id = "0";
  }

  JsonSerializationStrategy(final String readableName, final String id) {
    this.readableName = readableName;
    this.id = id;
  }

  public String getReadableName() {
    return readableName;
  }

  @Override
  public String getId() {
    return id;
  }
}
```

ENUMERACJE - DZIEDZICZENIE

Typy wyliczeniowe nie mogą dziedziczyć, tzn. poniższy kod nie skompiluje się poprawnie:

```
public class SomeBaseClass {
}

public enum EnumExample extends SomeBaseClass {
}
```

Również inne klasy nie mogą dziedziczyć po typie wyliczeniowym. Ponownie, **poniższy kod nie skompiluje się**:

```
public enum EnumExample {
}

public class SomeBaseClass extends EnumExample {
}
```

ENUMERACJE - METODY WBUDOWANE

Każdy typ wyliczeniowy ma dostęp do metod statycznych:

- **values** - zwraca tablicę dostępnych wartości
- **valueOf** - zwraca predefiniowaną wartość na podstawie nazwy

Ponadto każda instancja enuma posiada metodę name(), która zwraca nazwę instancji jako String

```
public enum Difficulty {
  EASY,
  MEDIUM,
  HARD
}

Stream.of(Difficulty.values()).forEach(System.out::println); // EASY MEDIUM HARD
System.out.println(Difficulty.valueOf("EASY") == Difficulty.EASY); // true
System.out.println(Difficulty.MEDIUM.name()); // MEDIUM
```

Fundusze Europejskie
Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska
Europejski Fundusz Społeczny

ZAAWANSOWANE ASPEKTY BUDOWY KLAS W JAVIE

Fundusze Europejskie
Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska
Europejski Fundusz Społeczny

KOMPOZYCJA

Implementowanie **re-używalnych komponentów**, które skupia się **na składaniu obiektów** nazywa się **kompozycja**. Klasa **agreguje** w sobie składniki innych klas.

```
public class Computer {
    private Processor processor;
    private Ram ram;

    public Computer(Processor processor, Ram ram) {
        this.processor = processor;
        this.ram = ram;
    }

    public void run() {
        // usage of processor and ram object
    }
}

class Processor {
    private String name;
    private int numberOfCores;

    public Processor(String name, int numberOfCores) {
        this.name = name;
        this.numberOfCores = numberOfCores;
    }
}

class Ram {
    private String name;
    private int size;

    public Ram(String name, int size) {
        this.name = name;
        this.size = size;
    }
}
```

Fundusze Europejskie
Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska
Europejski Fundusz Społeczny

DZIEDZICZENIE

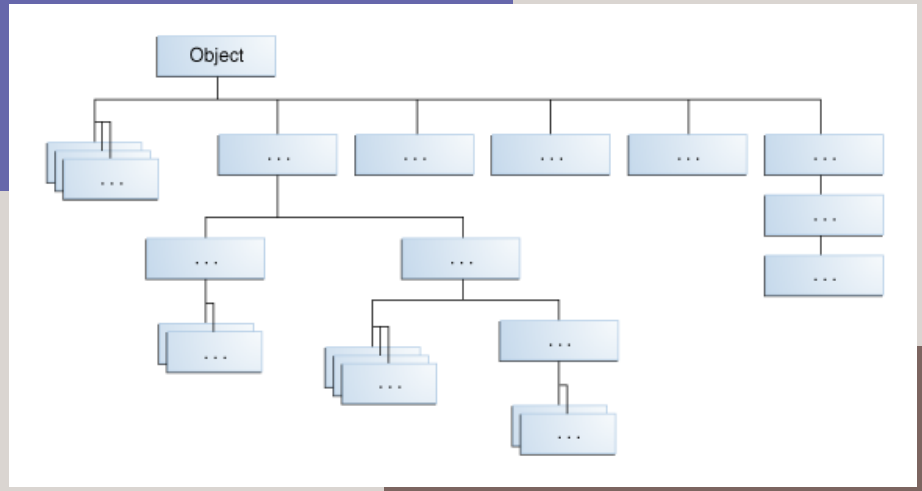
Jednym z najważniejszych **filarów OOP** jest **dziedziczenie klas**. Mechanizm ten umożliwia **współdzielenie zachowania pomiędzy klasami**. Klasa może dziedziczyć po innej klasie, co oznacza, że oprócz dotychczasowej funkcjonalności, może definiować nowe zachowania i właściwości. Dziedziczenie w Javie jest realizowane w oparciu o słówko kluczowe **extends**, które umieszczamy po nazwie deklarowanej klasy.

```java
public class Car {
    public void turnOnEngine(){
        System.out.println("Turn on engine!");
    }
}

public class SportCar extends Car {
    // klasa SportCar dziedziczy po klasie Car
    public void drive() {
        turnOnEngine();
        System.out.println("I'm driving!");
    }
}
```

Fundusze Europejskie
Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska
Europejski Fundusz Społeczny

HIERARCHICZNOŚĆ DZIEDZICZENIA

W Javie **wszystkie klasy dziedziczą po klasie Object**, pochodzącej z pakietu **java.lang**. Zawiera ona wspólną logikę i implementację **dla każdej klasy pochodnej**, która istnieje w ramach platformy, bądź będzie dopiero powstawać.



HIERARCHICZNOŚĆ DZIEDZICZENIA

Właściwości klasy pochodnej:

- dziedziczy **wszystkie metody i pola klasy bazowej**
- może rozszerzać **tylko co najwyżej jedną klasę**
- ma dostęp do wszystkich pól i metod, które zostały zdefiniowane **z wykorzystaniem modyfikatorów dostępu public i protected**
- ma dostęp do elementów z domyślnym modyfikatorem dostępu (**package-private**), jeżeli klasa bazowa znajduje się w tym samym pakiecie

HIERARCHICZNOŚĆ DZIEDZICZENIA

Klasa taka może bezpośrednio korzystać z możliwości klasy bazowej, tzn. **może np. wywoływać metody, do których ma dostęp**. Ponadto nadal może „zachowywać" się, jak zwykły obiekt, tzn. **może definiować nowe metody, pola czy stałe**.

Oprócz tego ma możliwość **nadpisania lub rozszerzenia** metody z klasy bazowej, wykorzystując **odpowiednie adnotacje**. W celu rozszerzenia możliwości konstruktora lub metody bazowej, musimy wykorzystać słowo kluczowe **super**, które umożliwia wywołanie logiki z klasy bazowej.

Fundusze Europejskie
Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska
Europejski Fundusz Społeczny

HIERARCHICZNOŚĆ DZIEDZICZENIA

```java
public class Computer {
    private String cpu;
    private String ram;
    private String gpu;

    public Computer(String cpu, String ram, String gpu) {
        this.cpu = cpu;
        this.ram = ram;
        this.gpu = gpu;
    }

    public void configure() {
        System.out.println("Booting ... ");
        System.out.println("Configure cpu: " + cpu);
        System.out.println("Configure ram: " + ram);
        System.out.println("Configure gpu: " + gpu);
    }
}

public class Laptop extends Computer {
    // Computer jest klasą bazową klasy Laptop
    private int battery;

    public Laptop(String cpu, String ram, String gpu, int battery) {
        // w przypadku klasy pochodnej, wywołanie domyślnego konstruktora
        // klasy bazowej jest wymagane
        super(cpu, ram, gpu);
        this.battery = battery;
    }

    @Override // (1)
    public void configure() {
        // wywołujemy funkcjonalność z klasy bazowej
        super.configure();
        System.out.println("Configure battery: " + battery);
    }
}
```

KOMPOZYCJA A DZIEDZICZENIE

Dziedziczenie klas **definiowane jest statycznie**, co skutkuje tym, że nie ma **możliwości zmiany implementacji w czasie wykonywania aplikacji**. Implementacja podklas zależna jest od implementacji klasy bazowej, więc zmiany w klasie bazowej często wymuszają również zmiany w podklasach. **Rozbudowane hierarchie dziedziczenia wpływają także negatywnie na testowanie kodu** oraz analizę struktury implementacyjnej. Dzięki wykorzystaniu kompozycji możemy dynamicznie zmieniać implementację (m.in. dzięki poliformizmie), bez potrzeby modyfikacji komponentów zagregowanych i na odwrót.

DEFINICJE

Klasa abstrakcyjna jest definicją klasy, której instancji (podobnie jak interfejsu) **nie możemy stworzyć**. Aby zadeklarować taką klasę, pomiędzy modyfikatorem dostępu a nazwą klasy, powinniśmy wykorzystać słowo kluczowe **abstract**. W przeciwieństwie do zwykłych klas, **klasa abstrakcyjna może dodatkowo posiadać metody abstrakcyjne.**

Fundusze Europejskie
Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska
Europejski Fundusz Społeczny

DEFINICJE

Metody abstrakcyjne to takie metody, które:

- **nie mogą posiadać ciała w definicji**
- ciało musi być zaimplementowane **w klasie, która po takiej klasie abstrakcyjnej dziedziczy**
- mogą być deklarowane **tylko i wyłącznie w klasach abstrakcyjnych oraz interfejsach**
- wymagają wykorzystania **słowa kluczowego abstract** przed typem zwracanym w jej sygnaturze

UWAGA: Metody abstrakcyjne nie mogą być deklarowane jako prywatne.

```
public abstract class Button { // klasa zdefiniowana jako abstrakcyjna

    public String getComponentName() { // zwykła, nieabstrakcyjna metoda
        return "Button";
    }

    public abstract void onClick(); // metoda abstrakcyjna, NIE posiada ciała
}
```

TWORZENIE INSTANCJI

Próba stworzenia instancji klasy zdefiniowanej na poprzednim slajdzie, kończy się **błędem kompilacji**:

```
public static void main(String[] args) {
  Button button = new Button(); // BŁĄD
}
```

Aby stworzyć instancję takiej klasy, **musimy najpierw ją rozszerzyć**, np.

```
public class SimpleButton extends Button {
  // dziedziczymy klasę abstrakcyjną tak, jak każdą inną klasę

  // KONIECZNIE musimy zaimplementować WSZYSTKIE metody abstrakcyjne
  @Override
  void onClick() {
    System.out.println("Simple Button was clicked");
  }
}
```

PORÓWNANIE Z INTERFEJSAMI

W odróżnieniu od interfejsów, **w klasach abstrakcyjnych możemy deklarować pola**. Ponadto, zdefiniowane w tej klasie metody, nie posiadają żadnych domyślnych słów kluczowych.

Jeśli chcemy współdzielić część kodu pomiędzy wieloma powiązanymi klasami, wtedy warto zastanowić się nad użyciem klasy abstrakcyjnej. Jeśli chcemy tylko wydzielić pewną abstrakcję, która może zostać wykorzystana w przypadku wielu niepowiązanych ze sobą klas, wtedy powinniśmy skorzystać z interfejsu.

Fundusze Europejskie
Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska
Europejski Fundusz Społeczny

PORÓWNANIE Z INTERFEJSAMI

Klasa abstrakcyjna może implementować interfejsy bez deklarowania ciał ich metod. Nie zmienia to faktu, że tworząc klasę pochodną, należy nadać ciało wszystkim metodom abstrakcyjnym, bez względu na to, czy są one przynależne do interfejsu, czy do klasy abstrakcyjnej.

```java
public interface ComponentClickListener {

    void onClick();
}

public abstract class AbstractButton implements ComponentClickListener {
    public static final String TAG = "Button";

    String componentName;

    String getComponentName() {
        return componentName;
    }

    abstract void click();
}

public class ButtonComponent extends AbstractButton {
  @Override
  void click() {
    // implementacja wymagana -> metoda abstrakcyjna klasy abstrakcyjnej
    System.out.println("I just clicked a button! Amazing");
  }

  @Override
  public void onClick() {
    // implementacja wymagana -> metoda abstrakcyjna interfejsu ComponentClickListener
    System.out.println("I am an onClick handler");
  }
}
```

Fundusze Europejskie
Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska
Europejski Fundusz Społeczny

PORÓWNANIE Z INTERFEJSAMI

Ponadto klasy abstrakcyjne **mogą dziedziczyć inne klasy abstrakcyjne**. Definiując klasę abstrakcyjną, która dziedziczy po klasie abstrakcyjnej, **możemy, ale nie musimy implementować dziedziczone metody abstrakcyjne**. Poniższy przykład pokazuje taką definicję:

```java
public abstract class TestTemplate {
  abstract public void run();
  protected abstract int getNumberOfIterations();
}

public abstract class PerformanceTestTemplate extends TestTemplate {
  // klasa abstrakcyjna dziedzicząca po klasie abstrakcyjnej
  @Override
  public void run() { // opcjonalnie decydujemy się na definicję metody abstrakcyjnej
    System.out.println("I should run test here");
  }
  // w tej klasie NIE implementujemy getNumberOfIterations (ale możemy to zrobić)

  public abstract double getAverageExecutionTime();
}

public class SortListPerformanceTest extends PerformanceTestTemplate {
  // definicja klasy nieabstrakcyjnej

  @Override // wymagane - metoda abstrakcyjna nieposiadająca implementacji w hierarchii dziedziczenia
  public double getAverageExecutionTime() {
    // calculate value based on real scenario
    return 5.0;
  }

  @Override // wymagane - metoda abstrakcyjna nieposiadająca implementacji w hierarchii dziedziczenia
  protected int getNumberOfIterations() {
    return 3;
  }
}
```

INTERFEJS

Interfejs jest, podobnie jak klasy, typem referencyjnym.

Może zawierać tylko:

- **stałe**
- **sygnatury metod**
- **metody typu default**
- **metody statyczne**
- **definicje typów zagnieżdżonych**

Nie można stworzyć instancji interfejsu. Interfejsy mogą być:

- implementowane przez klasy (keyword: **implements**)
- rozszerzane przez inne interfejsy (keyword: **extends**)

Fundusze Europejskie
Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska
Europejski Fundusz Społeczny

DEFINIOWANIE INTERFEJSU

Podczas deklarowania interfejsu definiuje się:

- **modyfikator dostępu**
- słowo kluczowe **interface**
- **nazwę interfejsu**
- opcjonalnie, **listę interfejsów** oddzielonych przecinkami
- **ciało interfejsu**, które może być puste

```
public interface MediaPlayer {
    void stop();
    void play();
}
```

Fundusze Europejskie
Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska
Europejski Fundusz Społeczny

CIAŁO INTERFEJSU

Ciało interfejsu może zawierać tylko metody typu:

- **abstract**
- **default**
- **static**

Pewne słowa kluczowe są domyślne:

- metody niestatyczne, w których nie zdefiniowaliśmy ciała, **są domyślnie abstrakcyjne i publiczne**, tzn. muszą zostać zaimplementowane w implementacjach
- stałe są domyślnie publiczne, finalne i statyczne, tzn. są poprzedzone słowami kluczowymi **public static final**
- metody **statyczne są domyślnie publiczne**, tzn. są poprzedzone słowem kluczowym public

```
public interface MediaPlayer {

    String TAG = "MediaPlayer";
    // domyślnie zawiera słowa kluczowe - public static final

    void stop();
    // metoda bez zdefiniowanego ciała - domyślnie abstrakcyjna

    default void next() {
        // metoda z domyślnym ciałem, można, ale nie trzeba, nadpisywać w implementacji interfejsu
        throw new NoSuchMechanismException("not supported by default");
    }

    static String getName() {
        // metoda statyczna, domyślnie posiada modyfikator public
        return "MediaPlayer Interface";
    }
}
```

Fundusze Europejskie
Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska
Europejski Fundusz Społeczny

IMPLEMENTOWANIE INTERFEJSU

W celu **stworzenia implementacji interfejsu** należy użyć słowa kluczowego **implements** w deklaracji klasy i zaimplementować wszystkie metody abstrakcyjne danego interfejsu, np.:

```
// deklaracja interfejsu
public interface SomeInterface {
  void someMethod();
}

// deklaracja klasy implementującej interfejs
public class SomeInterfaceImpl implements SomeInterface {
  @Override
  public void someMethod() {
    System.out.println("methodImplementation");
  }
}

// stworzenie instancji klasy implementującej interfejs
SomeInterface someInterface = new SomeInterfaceImpl();
```

Fundusze Europejskie
Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska
Europejski Fundusz Społeczny

IMPLEMENTOWANIE INTERFEJSU

Klasa może **implementować więcej niż jeden interfejs** (oddzielając ich nazwy od siebie przecinkiem), np.:

```java
// deklaracja pierwszego interfejsu
public interface SomeInterface {
  void someMethod();
}

// deklaracje drugiego interfejsu, tym razem bez żadnych metod abstrakcyjnych
public interface SomeOtherInterface {
}

// definicja klasy implementującej oba powyższe interfejsy
public class ClassImplementingInterfaces implements SomeInterface, SomeOtherInterface {
  @Override
  public void someMethod() {
    System.out.println("I am interface method implementation");
  }
}
```

Fundusze Europejskie
Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska
Europejski Fundusz Społeczny

IMPLEMENTOWANIE INTERFEJSU

Pojedyncza klasa **oprócz implementowania wielu interfejsów**, może również rozszerzać **co najwyżej jedną klasę**, np.:

```
// deklaracja pierwszego interfejsu
public interface SomeInterface {
  void someMethod();
}

// deklaracja drugiego interfejsu, tym razem bez żadnych metod abstrakcyjnych
public interface SomeOtherInterface {
}

// definicja klasy, która rozszerza klasę SomeClass i implementuje oba powyższe interfejsy
public class ClassImplementingInterfacesAndExtendingClass
    extends SomeClass implements SomeInterface, SomeOtherInterface {
  @Override
  public void someMethod() {
    System.out.println("I am interface method implementation");
  }
}
```

Fundusze Europejskie Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska Europejski Fundusz Społeczny

MODYFIKOWANIE INTERFEJSU

Wykonanie dowolnej modyfikacji na zaimplementowanym interfejsie, **będzie skutkować błędami na poziomie kompilacji, ze względu na brak implementacji nowego (zmodyfikowanego) interfejsu**.

UWAGA: Zmieniając definicję interfejsu, musimy pamiętać o koniecznych zmianach we wszystkich jego implementacjach.

UWAGA: Podobnych błędów kompilacji można uniknąć, wykorzystując opcję refactor w IntelliJ IDEA używając skrótu **Shift + F6**.

```java
public interface NewsletterAPI {

    void subscribe();

    void unsubscribe();
}
```

```java
public interface NewsletterAPI {

    void subscribe(String type);

    void unsubscribe();
}
```

```java
// Class must either be declared abstract or implement abstract method error
public class ShopNewsletterAPI implements NewsletterAPI {
    @Override
    public void subscribe() {
        System.out.println("Subscribe to shop newsletter!");
    }

    @Override
    public void unsubscribe() {
        System.out.println("Unsubscribe from shop newsletter!");
    }
}
```

METODY DOMYŚLNE

Metody domyślne zostały wprowadzone w wersji Javy 1.8. Ich główne zastosowanie to **wprowadzenie modyfikacji w istniejącym już interfejsie**, bez prowokowania błędów kompilacji związanych ze zmianą ciała interfejsu. Domyślne metody **pozwalają dodawać nowe funkcje do interfejsów bibliotek** i zapewniają zgodność binarną z kodem napisanym dla starszych wersji tych interfejsów. W przypadku implementacji interfejsów z metodami domyślnymi, nie ma potrzeby, by nadpisywać te metody, tak jak w poniższym przykładzie.

```
public interface Encoder {

    default String encode(String encodeText) {
        return encodeText;
    }

    default String decode(String decodeText) {
        return decodeText;
    }
}

public class SampleEncoder implements Encoder { }
```

Fundusze Europejskie
Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska
Europejski Fundusz Społeczny

WZORCE PROJEKTOWE

Wzorce projektowe są zestawem gotowych szkieletów rozwiązań, które powinniśmy wykorzystywać w naszych aplikacjach. Wyróżniamy wzorce:

- konstrukcyjne
- strukturalne
- czynnościowe (behawioralne)

Fundusze Europejskie
Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska
Europejski Fundusz Społeczny

WZORCE STRUKTURALNE

Wzorce konstrukcyjne są podgrupą, które opisują sposoby tworzenia obiektów. **Obiekty możemy tworzyć za pomocą konstruktorów**, ale ograniczając się tylko do tej metody, w przypadku bardziej skomplikowanych obiektów, nasz kod **może stać się bardzo nieczytelny**.

Wyróżniamy m.in.:

- **Singleton** - opisuje, w jaki sposób stworzyć pojedynczą instancję obiektu w aplikacji.
- **Builder** - wykorzystywany do tworzenia skomplikowanych obiektów.

Fundusze Europejskie
Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska
Europejski Fundusz Społeczny

SINGLETON

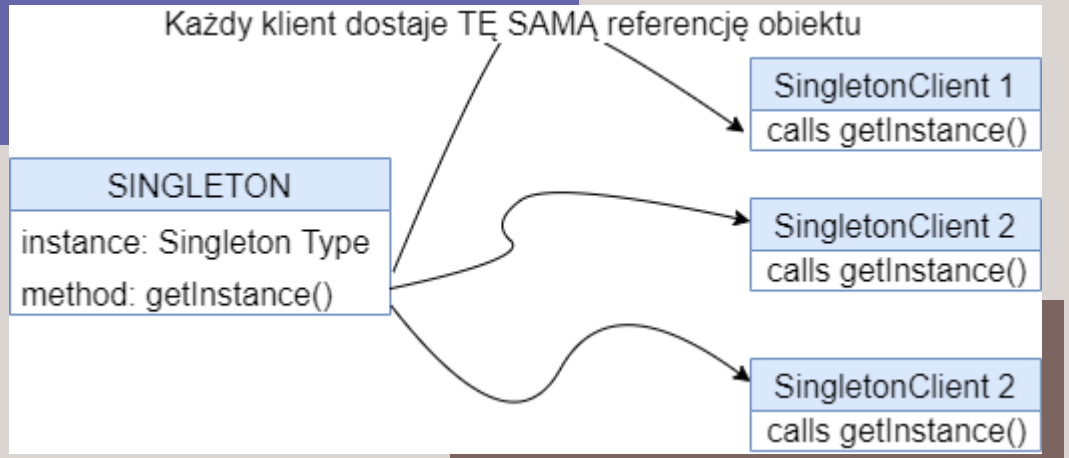
Singleton jest wzorcem projektowym, który opisuje w jaki sposób stworzyć obiekt, który w danej aplikacji **może być skonstruowany co najwyżej raz**.

Singletony możemy podzielić na dwie główne grupy:

- **lazy**
- **Eager**

Z kolei singleton typu lazy również możemy stworzyć na dwa sposoby:

- **lazy, który nie nadaje się do wykorzystania w aplikacjach wielowątkowych**
- **lazy double checked** , który można wykorzystać w aplikacjach wielowątkowych.



SINGLETON - EAGER

```java
public class SimpleCounter {

  // pole statyczne, w którym przetrzymujemy referencję singletonu
  // jest to singleton typu eager więc instancję tworzymy od razu,
  // przypisując ją do pola
  private static final SimpleCounter INSTANCE = new SimpleCounter();

  // getter dla referencji singletonu
  public static SimpleCounter getInstance() {
    return INSTANCE;
  }

  // ukryty konstruktor
  private SimpleCounter() {}

  private int currentCount = 0;

  public int getCurrentCount() {
    return currentCount;
  }

  public void increment() {
    currentCount++;
  }
}
```

SINGLETON - LAZY

```java
import java.util.ArrayList;
import java.util.List;

public class CommonStorage {
  private static CommonStorage instance;

  public static CommonStorage getInstance() {
    if (instance == null) { // (1)
      instance = new CommonStorage(); // (2)
    }
    return instance;
  }

  private List<Integer> values = new ArrayList<>();

  private CommonStorage() {
  }

  public void addValue(final int value) {
    values.add(value);
  }

  public List<Integer> getValues() {
    return values;
  }
}
```

SINGLETON
- DOUBLE CHECKED

```
import java.util.ArrayList;
import java.util.List;

public class CommonStorage {
  private static CommonStorage instance;

  public static CommonStorage getInstance() {
    if (instance == null) { // (1)
      synchronized (CommonStorage.class) {
        if (instance == null) { // (2)
          instance = new CommonStorage();
        }
      }
    }
    return instance;
  }

  private List<Integer> values = new ArrayList<>();

  private CommonStorage() {
  }

  public void addValue(final int value) {
    values.add(value);
  }

  public List<Integer> getValues() {
    return values;
  }
}
```

IMMUTABLE PATTERN

Obiekt niezmienny (**immutable**) to taki, który po utworzeniu i inicjalizacji **pozostaje niezmienny** i co ważne, **nie ma możliwości jego zmiany** (oczywiście w konwencjonalny sposób). Czyli taki obiekt nie udostępnia metod, które pozwalają **na zmianę jego stanu**. A jego wszystkie pola są prywatne.

IMMUTABLE PATTERN

Wymagania:

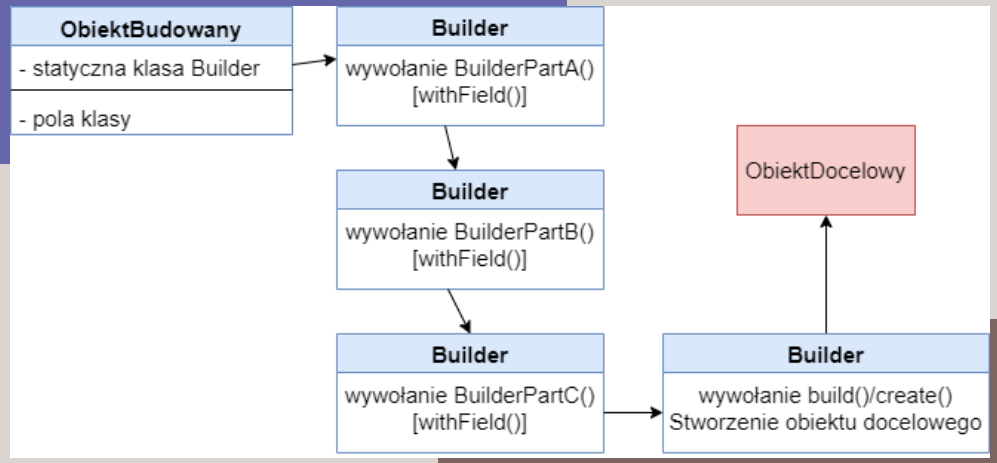
- Użycie konstruktora ustawiającego wszystkie wartości parametrów danej klasy (ewentualnie wykorzystanie wzorców konstrukcyjnych)
- Oznaczenie wszystkich pól klasy jako **private final**
- Brak metod typu seter
- Brak bezpośredniego dostępu do obiektów „mutowalnych”
- Zabezpieczenie przed możliwością nadpisania metody

```
    private final String species;
    private final int age;
    private final List<String> favoriteFoods;

    public Animal(String species, int age, List<String> favoriteFoods) {
        this.species = species;
        this.age = age;
        if(favoriteFoods == null) {
            throw new RuntimeException("favoriteFoods is required");
        }
        this.favoriteFoods = new ArrayList<String>(favoriteFoods);
    }

    public String getSpecies() {
        return species;
    }

    public int getAge() {
        return age;
    }

    public int getFavoriteFoodsCount() {
        return favoriteFoods.size();
    }

    public String getFavoriteFood(int index) {
        return favoriteFoods.get(index);
    }
}
```

BUILDER

Wzorzec **Builder** jest kolejnym sposobem tworzenia obiektów. **Wykorzystywany jest najczęściej do tworzenia obiektów, które składają się z wielu pól**. Obiekty, które składają się z wielu pól (np. 4 lub więcej) W celu stworzenia klasy będącej builderem dla pewnej klasy, musimy stworzyć:

- klasę statyczną w klasie
- lub osobną klasę



BUILDER - PRZYKŁAD

```java
import java.util.ArrayList;
import java.util.List;

public class Weapon {

  private String type;
  private String name;
  private Integer damage;
  private Long durability;
  private List<String> perks;

  private Weapon(final String type, final String name, final Integer damage,
      final Long durability, final List<String> perks) {
    this.type = type;
    this.name = name;
    this.damage = damage;
    this.durability = durability;
    this.perks = perks;
  }

  // getters + setters
}
```

```java
  public static class Builder {
    private String type;
    private String name;
    private Integer damage;
    private Long durability;
    private List<String> perks = new ArrayList<>();

    // metody budownicze
    public Builder withType(final String type) {
      this.type = type;
      return this;
    }

    public Builder withName(final String name) {
      this.name = name;
      return this;
    }

    public Builder withDamage(final Integer damage) {
      this.damage = damage;
      return this;
    }

    // ...

    // metoda budująca obiekt docelowy
    public Weapon build() {
      return new Weapon(type, name, damage, durability, perks);
    }
  }
}
```

Fundusze Europejskie
Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska
Europejski Fundusz Społeczny

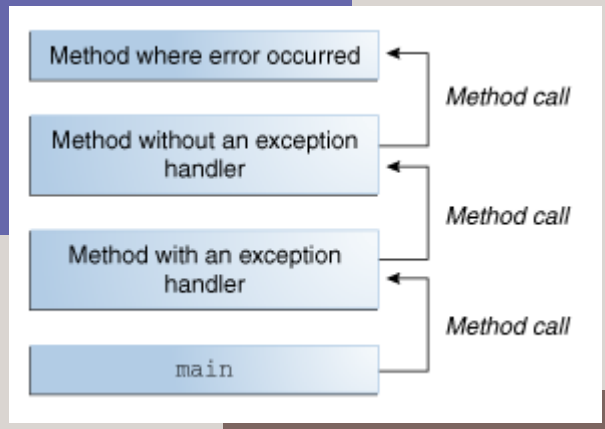
WYJĄTKI I ASERCJE

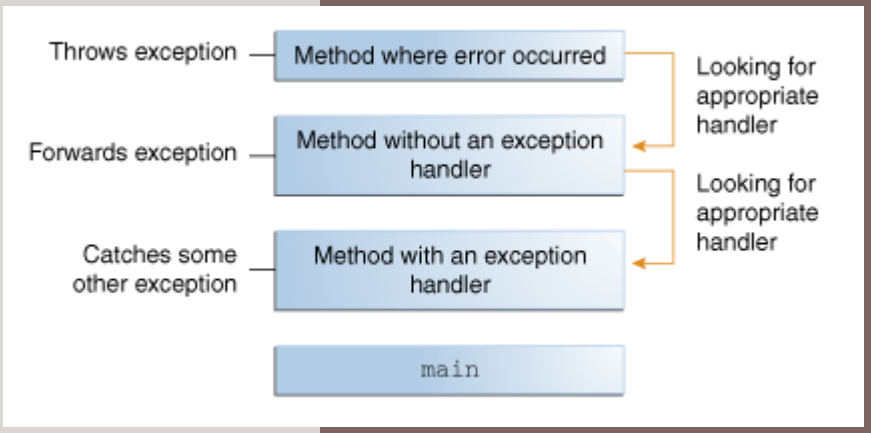
Wyjątek jest zdarzeniem, które następuje podczas wykonywania programu i **łamie normalny proces jego działania**. Po tym, jak wyjątek zostanie wyrzucony, system wykonawczy próbuje znaleźć fragment kodu, który sobie z tym poradzi. Zestawem możliwych elementów do obsługi wyjątku jest uporządkowana lista metod, które zostały wywołane, aby dostać się do metody, w której wystąpił błąd. **Lista metod jest znana jako stos wywołań** (ang. stacktrace).

Fundusze Europejskie
Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska
Europejski Fundusz Społeczny

WYJĄTKI

System w trakcie wykonywania **przeszukuje stos wywołań** w celu znalezienia metody, która obsłuży wyjątek. Wyszukiwanie rozpoczyna się od metody, w której wyrzucony został wyjątek, a kończy się na metodzie, która jest odpowiedzialna za obsługę zdarzenia.



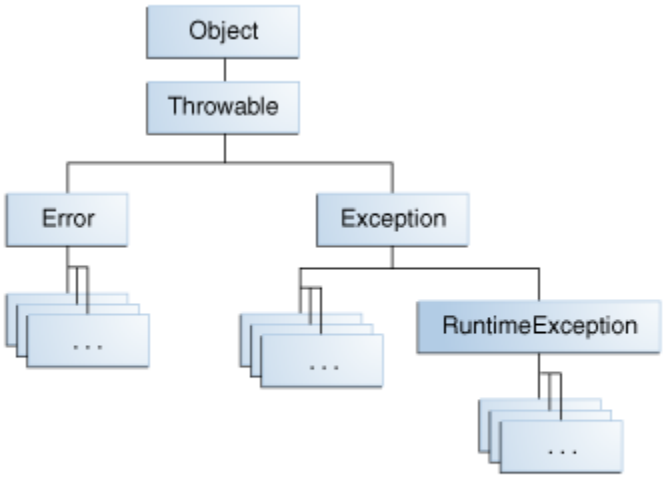


Fundusze Europejskie
Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska
Europejski Fundusz Społeczny

TYPY WYJĄTKÓW

W Javie wszystkie wyjątki możemy podzielić na dwie główne grupy. Grupy te zawierają wyjątki, które:

- dziedziczą po klasie **RuntimeException**, tzw. **wyjątki unchecked**
- dziedziczą po klasie **Exception**, tzw. **wyjątki checked**



> *UWAGA: Każdy* wyjątek dziedziczy po klasie `Throwable`.

TYPY WYJĄTKÓW

Type	How to recognize	Recommended for program to catch?	Is program required to catch or declare?
Runtime exception	`RuntimeException` or its subclasses	Yes	No
Checked exception	`Exception` or its subclasses but not `RuntimeException` or its subclasses	Yes	Yes
Error	`Error` or its subclasses	No	No

PODSTAWOWE WYJĄTKI

- **ArithmeticException** – dzielenie przez zero
- **ArrayIndexOutOfBoundsException** – nieprawidłowy indeks tablicy
- **ClassCastException** – nieprawidłowe rzutowanie klasy
- **IllegalArgumentException** – do metody przekazano nieprawdiłowy argument
- **NullPointerException** – otrzymano wartość nieokreśloną **null** podczas, gdy oczekiwano obiektu
- **NumberFormatException** – błąd konwersji łańcucha znaków do liczby (String -> numeric type)

CHECKED EXCEPTIONS

Exception	Used when	Checked or unchecked?
`java.text.ParseException`	Converting a `String` to a number.	Checked

`java.io.IOException`
`java.io.FileNotFoundException`
`java.io.NotSerializableException`
	Dealing with IO and NIO.2 issues. `IOException` is the parent class. There are a number of subclasses. You can assume any `java.io` exception is checked.	Checked

`java.sql.SQLException`	Dealing with database issues. `SQLException` is the parent class. Again, you can assume any `java.sql` exception is checked.	Checked

RUNTIME EXCEPTIONS

Exception	Used when	Checked or unchecked?
`java.lang.ArrayStoreException`	Trying to store the wrong data type in an array.	Unchecked
`java.time.DateTimeException`	Receiving an invalid format string for a date.	Unchecked
`java.util.MissingResourceException`	Trying to access a key or resource bundle that does not exist.	Unchecked
`java.lang.IllegalStateException` `java.lang.UnsupportedOperationException`	Attempting to run an invalid operation in collections and concurrency.	Unchecked

Fundusze Europejskie
Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska
Europejski Fundusz Społeczny

PRZECHWYTYWANIE I OBSŁUGA WYJĄTKÓW

Jeżeli wywołanie pewnej metody wyrzuca wyjątek **typu checked**, możemy taki wyjątek:

- nie obsługiwać i przekazać „wyżej” (tzn. **spropagować wyjątek w górę**, zgodnie ze stosem wywołań)
- obsłużyć w obrębie metody.

Aby przekazać konieczność obsługi wyjątków, musimy w sygnaturze metody dodać **słowo kluczowe throws**, a po nim listę typów wyjątków, jakie ta metoda może wyrzucić.

```
public void sleepAndOpenFile(final String filePath) throws InterruptedException, FileNotFoundException {
    Thread.sleep(10);
    new FileReader(filePath);
}
```

PRZECHWYTYWANIE I OBSŁUGA WYJĄTKÓW

W **celu obsłużenia wyjątku**, tzn. zrobienia czegoś z faktem jego wystąpienia, musimy wykorzystać pewne słowa kluczowe:

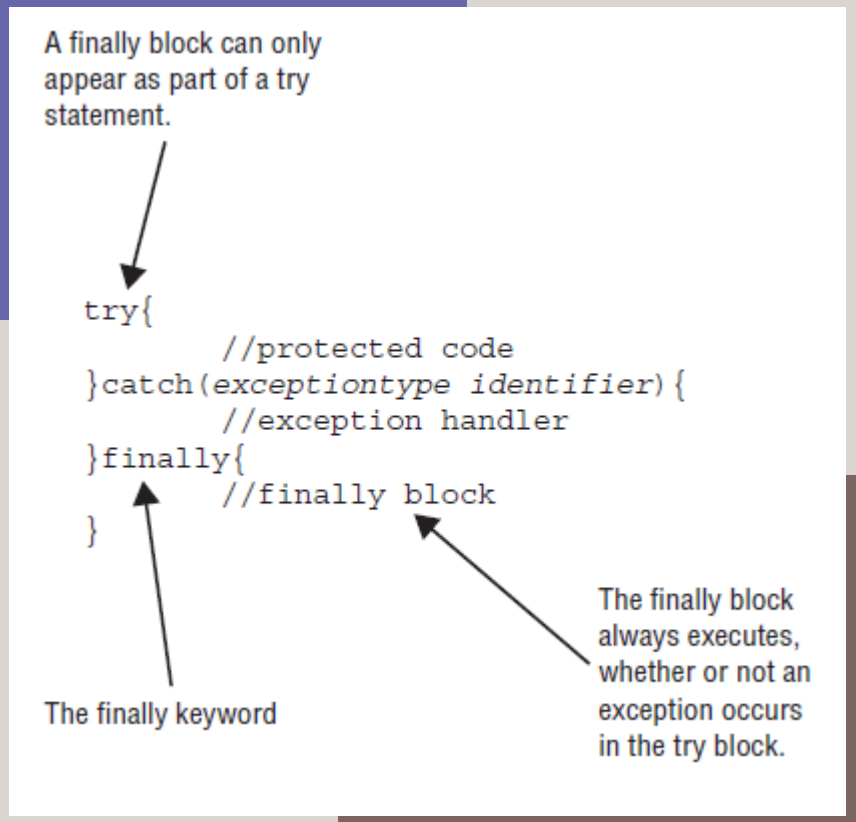
- **try** wraz z **catch** i opcjonalnie **finally**
- **try** wraz z przekazaniem zasobów (**try-with-resources**)

Fundusze Europejskie
Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska
Europejski Fundusz Społeczny

TRY I CATCH

Pierwszym krokiem w kierunku obsługi wyjątków jest zdeklarowanie bloku **try** wraz z blokiem **catch**. W bloku **try** definiujemy **kod, który może wyrzucić wyjątki pewnego typu**, następnie w bloku **catch**, który następuje zaraz po bloku try, definiujemy **kod, który powinien zostać wykonany w przypadku wystąpienia wyjątku**. Typ wyjątku i jego instancję podajemy wewnątrz nawiasów, zaraz po słowie kluczowym catch, np.:

```
try {
    Thread.sleep(100L);
} catch (InterruptedException e) {
    e.printStackTrace();
}
```



Fundusze Europejskie
Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska
Europejski Fundusz Społeczny

TRY I CATCH

Do pojedynczego bloku try, możemy dodać **wiele bloków catch**. Każdy z bloków **catch może obsługiwać inny typ wyjątku**. Tworząc bloki catch musimy pamiętać o kilku faktach:

- możemy złapać **dowolny wyjątek typu unchecked**
- możemy złapać tylko te **wyjątki typu checked, które mogą zostać wyrzucone w obrębie bloku try**
- możemy złapać typ będący podklasą typu wyjątku
- możemy łapać wyjątki, które po sobie dziedziczą, ale **bloki catch z bardziej specyficznym wyjątkiem muszą pojawić się jako pierwsze**

```
try {
    Thread.sleep(100L);
    new RestTemplate().getForEntity("http://localhost:8080", Object.class);
} catch (RestClientException e) {
    System.err.println("Error occurred from RestTemplate object");
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

```
public class CatchHierarchy {
  public static void main(String[] args) {
    try {
      final KeyGenerator keyGenerator = KeyGenerator.getInstance("AES");
      final Cipher cipher = Cipher.getInstance("AES");
    } catch (NoSuchAlgorithmException e) {
      System.err.println("Incorrect algorithm name");
    } catch (NoSuchPaddingException e) {
      System.err.println("Oops");
    } catch (Exception e) {
      System.err.println("Generic exception occurred");
    }
  }
}
```

TRY I CATCH

Co jeżeli chcemy wiele typów wyjątków **obsłużyć w dokładnie taki sam sposób**? Nie musimy w tym celu definiować wielu bloków catch. Wystarczy jeden, a **listę wyjątków które chcemy obsłużyć za pomocą tego samego kodu, oddzielamy za pomocą znaku |**

UWAGA: Wykorzystując znak | w bloku catch, **nie możemy podać dwóch klas z jednej hierarchii** (tzn. jedna z nich jest pochodną drugiej). Skończy się to błędem kompilacji, np.: catch (Exception | RuntimeException e).

```
public class MultiExceptionsSingleCatch {
  public static void main(String[] args) {
    try {
      final KeyGenerator keyGenerator = KeyGenerator.getInstance("AES");
      final Cipher cipher = Cipher.getInstance("AES");
    } catch (NoSuchPaddingException | NoSuchAlgorithmException e) {
      e.printStackTrace();
    }
  }
}
```

TRY I CATCH

```
try {
   // do some work
} catch(RuntimeException e) {
   e = new RuntimeException();
}
```

```
try {
   throw new IOException();
} catch(IOException | RuntimeException e) {
   e = new RuntimeException();              // DOES NOT COMPILE
}
```

```
public static void main(String[] args) {
   try {
      Path path = Paths.get("dolphinsBorn.txt");
      String text = new String(Files.readAllBytes(path));
      LocalDate date = LocalDate.parse(text);
      System.out.println(date);
   } catch (DateTimeParseException | IOException e) {
      e.printStackTrace();
      throw new RuntimeException(e);
   } }
```

TRY I CATCH

Dlaczego poniższy kod nie kompiluje się?

```
11:   public void doesNotCompile() {  // METHOD DOES NOT COMPILE
12:     try {
13:         mightThrow();
14:      } catch (FileNotFoundException | IllegalStateException e) {
15:      } catch (InputMismatchException e | MissingResourceException e) {
16:      } catch (SQLException | ArrayIndexOutOfBoundsException e) {
17:      } catch (FileNotFoundException | IllegalArgumentException e) {
18:      } catch (Exception e) {
19:      } catch (IOException e) {
20:      }
21:  }
22:  private void mightThrow() throws DateTimeParseException, IOException { }
```

FINALLY

Aplikacje operują skończoną ilością zasobów (np. ilość pamięci, maksymalna ilość plików, czy połączeń do bazy danych). Jeżeli w trakcie wykonywania kodu pewne **zasoby zostały otwarte i następnie został wyrzucony wyjątek, mimo to powinniśmy je zawsze zamknąć**. Mechanizm ten zapewnia blok **finally**, który wywołuje się **zawsze, gdy istnieje blok try**, niezależnie od tego, czy został wyrzucony wyjątek czy nie. **Block finally znajduje się zawsze na końcu bloków try/catch**.

- Blok **try wymaga bloku catch i/lub bloku finally**.
- Blok finally wykona się **nawet, jeżeli w bloku try wyjdziemy z metody za pomocą return**.

```
public class FinallyExample {
  public static void main(String[] args) throws IOException {
    BufferedReader bufferedReader = null;
    try {
      bufferedReader = new BufferedReader(new FileReader("/tmp/java_course.txt"));
    } catch (FileNotFoundException e) {
      System.exit(1);
    } finally {
      if (bufferedReader != null) {
        bufferedReader.close();
      }
    }
  }
}
```

TRY Z ZASOBEM

Istnieje również możliwość przekazania do bloku try **listy zasobów**. Wykorzystanie bloku try w taki sposób **zwalnia nas z ręcznego zamykania zasobów w bloku finally**. Zasobem takim może być dowolny obiekt, który implementuje **java.lang.AutoCloseable**, bądź realizuje **java.io.Closeable**.

```
public String readFirstLineFromFile(String path) {
    try (BufferedReader br = new BufferedReader(new FileReader(path))) {
        return br.readLine();
    } catch (IOException e) {
        return "FAILED";
    }
}
```

```
public class Turkey {
   public static void main(String[] args) {
      try (Turkey t = new Turkey()) {  // DOES NOT COMPILE
         System.out.println(t);
      }
   }
}
```

TRY Z ZASOBEM

```
try (BufferedReader r = Files.newBufferedReader(path1);
     BufferedWriter w = Files.newBufferedWriter(path2)) {
   //protected code

} catch (IOException e) {
   // exeption handler

} finally {
   // finally block

}
```

Optional clauses; resources still closed automatically

TRY-CATCH-FINALLY - KONFIGURACJE

traditional try statement

	0 finally blocks	1 finally block	2 or more finally blocks
0 catch blocks	Not legal	Legal	Not legal
1 or more catch blocks	Legal	Legal	Not legal

try-with-resources statement

	0 finally blocks	1 finally block	2 or more finally blocks
0 catch blocks	Legal	Legal	Not legal
1 or more catch blocks	Legal	Legal	Not legal

RZUCANIE WYJĄTKÓW

W kodzie źródłowym, oprócz przechwytywania zgłoszonych już wyjątków, **istnieje możliwość rzucania wyjątków**. Wyjątek taki musi być **pochodną klasy Throwable**. Do rzucania wyjątków wykorzystujemy klauzulę **throw**

```
public void validateAddress(String address) {
  if (address == null || address.isEmpty()){
    throw new IllegalArgumentException("illegal address");
  }
  // do some operations
}
```

THROW VS THROWS

```
public String getDataFromDatabase() throws SQLException {
    throw new UnsupportedOperationException();
}
```

STŁUMIONE WYJĄTKI (SUPPRESSED EXCEPTIONS)

```
public class JammedTurkeyCage implements AutoCloseable {
   public void close() throws IllegalStateException {
      throw new IllegalStateException("Cage door does not close");
   }
   public static void main(String[] args) {
      try (JammedTurkeyCage t = new JammedTurkeyCage()) {
         System.out.println("put turkeys in");
      } catch (IllegalStateException e) {
         System.out.println("caught: " + e.getMessage());
      }
   }
}
```

```
try (JammedTurkeyCage t = new JammedTurkeyCage()) {
   throw new IllegalStateException("turkeys ran off");
} catch (IllegalStateException e) {
   System.out.println("caught: " + e.getMessage());
   for (Throwable t: e.getSuppressed())
      System.out.println(t.getMessage());
}
```

Fundusze Europejskie
Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska
Europejski Fundusz Społeczny

TWORZENIE WŁASNYCH WYJĄTKÓW

Java zapewnia wiele klas wyjątków, z których można skorzystać. **Można też tworzyć własne wyjątki**. Jeśli któreś z poniższych kryteriów zostanie spełnione, warto jest rozważyć przygotowanie własnego typu:

- istnieje potrzeba posiadania wyjątku, który **nie jest reprezentowany przez żaden dostępny typ wyjątku**
- wyjątek wykorzystywany w kodzie powinien **różnić się od wyjątków innych dostawców**.

```
public class JavaCourseException extends RuntimeException {
  public JavaCourseException(final String message, final Throwable cause) {
    super(message, cause);
  }
}

public class IllegalAddressException extends IllegalArgumentException {

    public IllegalAddressException(final String address) {
        super(String.format("Provided address %s is not valid!", address));
    }
}
```

ERROR

Instancje klasy **Error** są wyjątkami, których **nie powinniśmy obsługiwać przy pomocy bloku try i catch**. Nie musimy też deklarować informacji o takim błędzie przy pomocy throws. Oznaczają one błędy, z którymi **nie poradzimy sobie z poziomu aplikacji i najczęściej ich efektem jest zakończenie działania danej aplikacji**. Przykładem takiego typu błędu jest wyjątek **OutOfMemoryError**.

ASERCJE

W najbardziej ścisłym ujęciu można powiedzieć, że asercja jest **predykatem**, czyli **stwierdzeniem, które może być prawdziwe, lub fałszywe**. W języku JAVA istnieje **słowo kluczowe assert**, które pozwala nam stwierdzić, że pewne wyrażenie jest **prawdziwe, lub wygenerować błąd w przeciwnym wypadku**.

Głównym zastosowaniem asercji jest **wykonywanie rozmaitych testów**.

ASERCJE

Domyślnie sprawdzanie asercji **jest wyłączone** w języku JAVA i nie zaleca się używania ich poza etapem wytwarzania oprogramowania. W celu włączenia asercji możemy posłużyć się następującymi poleceniami:

```
java -enableassertions Rectangle
```

```
java -ea Rectangle
```

Użycie flagi **–ea** bez dodatkowych argumentów powoduje odblokowanie wszystkich asercji (za wyjątkiem klas systemowych javy).

ASERCJE

- włączenie asercji dla **pakietu com.wiley.demos** oraz odpowiednich **podpakietów**:

```
java -ea:com.wiley.demos... my.programs.Main
```

- włączenie asercji dla klasy com.wiley.demos.TestColors:

```
java -ea:com.wiley.demos.TestColors my.programs.Main
```

- włączenie asercji dla pakietu com.wiley.demos oraz wyłączenie asercji dla klasy TestColors:

```
java -ea:com.wiley.demos... -da:com.wiley.demos.TestColors my.programs.Main
```

Fundusze Europejskie
Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska
Europejski Fundusz Społeczny

POLECENIE ASSERT

```
assert boolean_expression;
assert boolean_expression: error_message;
```

- **asercje wyłączone** – instrukcje **assert** są pomijane w trakcie wykonywania kodu
- **asercje włączone** – wyrażenie **bolean_expression** ewaluuje do **true** – aplikacja wykonuje się bez zmian
- **asercje włączone** - wyrażenie **bolean_expression** ewaluuje do **false** – aplikacja rzuca wyjątek **AssertionError**

```
if (!boolean_expression) throw new AssertionError();
```

```
public class Assertions {
    public static void main(String[] args) {
        int numGuests = -5;
        assert numGuests > 0;
        System.out.println(numGuests);
    }
}
```

ASERCJE - UŻYCIE

```
public enum Seasons {
  SPRING, SUMMER, FALL
}
```

```
public class TestSeasons {
  public static void test(Seasons s) {
      switch (s) {
      case SPRING:
      case FALL:
         System.out.println("Shorter hours");
         break;
      case SUMMER:
         System.out.println("Longer hours");
         break;
      default:
         assert false: "Invalid season";
      }}}
```

```
Exception in thread "main" java.lang.AssertionError: Invalid season
        at TestSeason.main(Test.java:12)
        at TestSeason.main(Test.java:18)
```

ASERCJE - UŻYCIE

- Asercje nie powinny modyfikować stanu zmiennych oraz obiektów

```
int x = 10;
assert ++x > 10;  // Not a good design!
```

- Asercje wykorzystujemy do debugowania
- Nie powinniśmy używa asercji do walidowania parametrów wejściowych

```
public Rectangle(int width, int height) {
    if(width < 0 || height < 0) {
        throw new IllegalArgumentException();
    }
    this.width = width;
    this.height = height;
}
```

Fundusze Europejskie Wiedza Edukacja Rozwój

Rzeczpospolita Polska

Unia Europejska Europejski Fundusz Społeczny

DZIĘKUJĘ ZA UWAGĘ