**1. Programowanie obiektowe**

**Wyzwania:**

* zrozumiesz dokładniej zasady programowania obiektowego,
* poznasz interfejsy i klasy abstrakcyjne,
* napiszesz dwa programy.

**Dodatkowe informacje**

W ramach kursu masz dostęp do kanałów:

* #help-java – problemy ze składnią Javy i jej zrozumieniem
* #help-java-spring – problemy z frameworkiem Spring
* #help-java-test – rozwiązywanie problemów związanych z pisaniem testów w Javie
* #help-java-tools – pomoc z narzędziami i bibliotekami Javy

Od teraz przesyłaj swoje zadania do sprawdzenia bezpośrednio do mentorów (miejsce na link jest pod treścią zadania). Jednak jeżeli chcesz coś skonsultować odnośnie kodu lub masz problem z wykonaniem zadania, to po pomoc zapraszamy właśnie na kanał #help-java. Pozostałe przydadzą Ci się później, gdy poznasz kolejne zagadnienia wraz z postępem kursu.

**1.1. Programy zorientowane obiektowo czy strukturalnie?**

Poznaliśmy już nieco programowanie obiektowe oraz strukturalne. Teraz pora poszerzyć naszą wiedzę i przekonać się, dlaczego programowanie obiektowe prowadzi do bardziej przejrzystego kodu i jest lepszą inwestycją w przyszłość naszego oprogramowania w porównaniu do pisania instrukcji programistycznych "jednym ciągiem".

**Podejście strukturalne**

Jak już wiesz z modułu zerowego, programowanie strukturalne polega na pisaniu kodu linijka po linijce – jest to swojego rodzaju potok myśli programisty, bądź potok wymagań jego przełożonego. Każde z zadań przebiega po kolei: od góry do dołu i jest to jak najbardziej poprawne.

Spójrz na poniższy kod napisany zgodnie z podejściem strukturalnym. Stworzyliśmy program, który wykonuje różne zadania, takie jak sprawdzenie wieku użytkownika, i wyświetla informację w zależności od wpisanych danych:

import java.util.\*;

import java.lang.\*;

import java.io.\*;

class Application

{

public static void main (String[] args) throws java.lang.Exception

{

System.out.println("Starting...");

int age = 15;

char sex = 'm';

if(age <= 15) {

System.out.println("Too young!");

} else {

System.out.println("User age is okey!");

if(sex == 'm') {

System.out.println("User is a male");

} else if(sex == 'f') {

System.out.println("User is a female");

} else {

System.out.println("Unknown sex!");

}

}

System.out.println("End of the program");

}

}

Pewnie zabolały Cię oczy od tego natłoku ifów :) Analizując kod zwróć uwagę, że obsługujemy różne przypadki, w jakich może znaleźć się nasz program:

1. Sprawdzenie wieku użytkownika – jeżeli użytkownik ma mniej bądź dokładnie 15 lat, to nasz program się zatrzyma i wyświetli informacje o zbyt niskim wieku użytkownika, następnie zakończy swoje działanie wyświetlając: "End of the program".
2. Jeżeli użytkownik ma więcej niż 15 lat, nasz program przechodzi do sprawdzania jego płci i w zależności od wybranej płci wyświetli tekst. Program też przewiduje błąd we wpisanej zmiennej i informuje o nieznanej płci.

Jak widzisz, wiele się tu dzieje. A co dopiero byłoby, gdybyśmy chcieli sprawdzić więcej różnych przypadków, lub musieli zmodyfikować kod po jakimś czasie!

Najpierw musielibyśmy usiąść i poważnie zastanowić się nad napisanym kodem, co się w nim dzieje, jak do niego podejść. Następnie musielibyśmy dodać zapewne kolejny if. Powoduje to niepotrzebną złożoność programu, co sprawia, że o wiele łatwiej jest popełnić błąd. Każde oprogramowanie z czasem się rozrasta, a wraz ze wzrostem tracimy szansę na bezpieczne wprowadzenie zmian.

Na szczęście na pomoc przychodzi **programowanie obiektowe**, które ma pomysł, jak rozwiązać tę nadmierną złożoność, i pozwala na obsługę zawiłych przypadków sprawdzania wieku, płci i wielu, wielu innych przypadków.

**Podejście obiektowe**

Uratujmy wcześniej napisany program przed tak zwaną falą ifów, która nie sprzyja przejrzystości naszego kodu!

Podejście obiektowe pozwala nam na stworzenie **odpowiedzialności klas i metod**. W Module 0 już wspominaliśmy o odpowiedzialności metody i klasy, kiedy robiliśmy nasz SimpleCalculator.

Aby tworzyć oprogramowanie czytelne i zrozumiałe, należy określać zakres czynności wykonywanych przez klasy i metody. Z tego powodu, aby usprawnić nasz strukturalny kod, stworzymy osobną klasę. Wiemy, że głównym zadaniem naszego programu jest walidacja danych użytkownika, dlatego mamy pełne prawo do przekazania tej odpowiedzialności klasie UserValidator i nadania jej dwóch metod:

1. validateAge(),
2. validateSex()

class UserValidator

{

public void validateAge() {

}

public void validateSex() {

}

}

Po stworzeniu takiego zestawu metod musimy zaimplementować instrukcje spełniające nasze wymagania, czyli sprawdzanie wieku oraz sprawdzanie płci. Zacznijmy budować nasze metody po kolei, czyli rozbudujmy je o odpowiednie argumenty:

import java.util.\*;

import java.lang.\*;

import java.io.\*;

class UserValidator

{

public void validateAge(int age) {

}

public void validateSex(char sex) {

}

}

Skoro obie metody zyskały już argumenty, które będą niezbędne do przeprowadzenia właściwych operacji, pora rozbudować metodę validateAge o odpowiednie instrukcje:

import java.util.\*;

import java.lang.\*;

import java.io.\*;

class UserValidator

{

public void validateAge(int age) {

if(age <= 15) {

System.out.println("Too young!");

} else {

System.out.println("User age is okey!");

}

}

public void validateSex(char sex) {

}

}

Jak widzisz, nasza metoda validateAge() otrzymała instrukcje potrzebne do sprawdzenia wieku naszego użytkownika, a co za tym idzie, uzyskała swoją odpowiedzialność za sprawdzenie wieku.

Przejdźmy teraz do naszego kodu i stwórzmy obiekt UserValidator, który wykorzysta zaimplementowaną metodę validateAge(). Zwróć uwagę na wynik wykonania programu wypisany w sekcji stdout:

<https://kodilla.com/pl/project-java/38568>

Udało się przeprowadzić nam walidację wieku użytkownika – pora zaimplementować metodę sprawdzającą płeć naszego użytkownika:

class UserValidator

{

public void validateAge(int age) {

if(age <= 15) {

System.out.println("Too young!");

} else {

System.out.println("User age is okey!");

}

}

public void validateSex(char sex) {

if(sex == 'm') {

System.out.println("User is a male");

} else if(sex == 'f') {

System.out.println("User is a female");

} else {

System.out.println("Unknown sex!");

}

}

}

Przenieśliśmy wcześniej napisany kod z kodu strukturalnego do naszej metody sprawdzającej płeć użytkownika. Dzięki tym zabiegom wydzieliliśmy zadania do metod, kod stał się czytelniejszy i podzielony na odpowiedzialności zawarte w metodach. Zobaczmy, jak będzie wyglądał kod po wywołaniu naszych metod:

import java.util.\*;

import java.lang.\*;

import java.io.\*;

class UserValidator

{

public void validateAge(int age) {

if(age <= 15) {

System.out.println("Too young!");

} else {

System.out.println("User age is okey!");

}

}

public void validateSex(char sex) {

if(sex == 'm') {

System.out.println("User is a male");

} else if(sex == 'f') {

System.out.println("User is a female");

} else {

System.out.println("Unknown sex!");

}

}

}

class Application

{

public static void main (String[] args) throws java.lang.Exception

{

System.out.println("Starting...");

int age = 15;

char sex = 'm';

UserValidator validator = new UserValidator();

validator.validateAge(age);

validator.validateSex(sex);

System.out.println("End of the program");

}

}

Na pierwszy rzut oka widać, że nasz kod stał się dużo bardziej przejrzysty. Jednakże nie działa on zgodnie z naszym założeniem, ponieważ chcemy, aby nasz program zakończył swoje działanie, gdy wiek użytkownika jest mniejszy bądź równy 15. W związku z tym musimy zmodyfikować naszą metodę, aby zwróciła nam informację o wyniku, najlepiej o typie boolean. Spójrz na linijkę 7:

import java.util.\*;

import java.lang.\*;

import java.io.\*;

class UserValidator

{

public boolean validateAge(int age) {

if(age <= 15) {

System.out.println("Too young!");

return false;

} else {

System.out.println("User age is okey!");

return true;

}

}

public void validateSex(char sex) {

if(sex == 'm') {

System.out.println("User is a male");

} else if(sex == 'f') {

System.out.println("User is a female");

} else {

System.out.println("Unknown sex!");

}

}

}

class Application

{

public static void main (String[] args) throws java.lang.Exception

{

System.out.println("Starting...");

int age = 15;

char sex = 'm';

UserValidator validator = new UserValidator();

boolean isHigherThanFifteen = validator.validateAge(age);

validator.validateSex(sex);

System.out.println("End of the program");

}

}

Nasza metoda została wzbogacona o zwracany typ boolean, który już wcześniej poznaliśmy. Następnie wykorzystaliśmy słowo kluczowe return do zwrócenia wyniku walidacji. Ten fragment kodu można opisać słowami "Jeżeli wiek użytkownika będzie mniejszy bądź równy 15, to zwróć fałsz, natomiast w innym przypadku zwróć prawdę".

Mamy już zaimplementowane zwracanie wartości w metodzie, zapisaliśmy również wynik do zmiennej w linijce 53 o nazwie isHigherThanOrEqualsFifteen:

import java.util.\*;

import java.lang.\*;

import java.io.\*;

class UserValidator

{

public boolean validateAge(int age) {

if(age <= 15) {

System.out.println("Too young!");

return false;

} else {

System.out.println("User age is okey!");

return true;

}

}

public void validateSex(char sex) {

if(sex == 'm') {

System.out.println("User is a male");

} else if(sex == 'f') {

System.out.println("User is a female");

} else {

System.out.println("Unknown sex!");

}

}

}

class Application

{

public static void main (String[] args) throws java.lang.Exception

{

System.out.println("Starting...");

int age = 15;

char sex = 'm';

UserValidator validator = new UserValidator();

boolean isHigherThanFifteen = validator.validateAge(age);

validator.validateSex(sex);

System.out.println("End of the program");

}

}

Teraz, aby móc poprawnie wykonać program, musimy wzbogacić go jeszcze jeden warunek – program musi zachować się odpowiednio do zwróconej wartości przez metodę validateAge. Popatrzmy teraz na linijkę 55:

import java.util.\*;

import java.lang.\*;

import java.io.\*;

class UserValidator

{

public boolean validateAge(int age) {

if(age <= 15) {

System.out.println("Too young!");

return false;

} else {

System.out.println("User age is okey!");

return true;

}

}

public void validateSex(char sex) {

if(sex == 'm') {

System.out.println("User is a male");

} else if(sex == 'f') {

System.out.println("User is a female");

} else {

System.out.println("Unknown sex!");

}

}

}

class Application

{

public static void main (String[] args) throws java.lang.Exception

{

System.out.println("Starting...");

int age = 15;

char sex = 'm';

UserValidator validator = new UserValidator();

boolean isHigherThanFifteen = validator.validateAge(age);

if(isHigherThanFifteen) {

validator.validateSex(sex);

}

System.out.println("End of the program");

}

}

Jeśli chcesz zobaczyć jak działa program możesz kliknąć w poniższy link:

<https://kodilla.com/pl/project-java/81677>

Dodaliśmy naszemu programowi warunek, który można wyjaśnić jako "Jeżeli wiek użytkownika jest większy od 15, to sprawdź jego płeć". W przeciwnym wypadku program po prostu zakończy swoje działanie.

Jak widzisz, musieliśmy się trochę namęczyć, aby rozłożyć nasz program na pojedyncze metody – należało stworzyć klasę walidującą i jej metody. Jednakże dzięki temu nasz kod stał się znacznie prostszy do zrozumienia, a jednocześnie gotowy na rozszerzenie jego funkcjonalności bez naruszania idei jego działania.

W przyszłości nauczymy się jeszcze bardziej upraszczać kod. Aby móc pozwolić sobie na dalsze operacje, musimy jeszcze trochę się nauczyć, ale to już wkrótce!

**Wniosek**

Omówiliśmy dwa sposoby pisania kodu. W pierwszym przypadku (strukturalnym) mogliśmy zaobserwować zawiłości i dużą złożoność kodu, która nie sprzyjała jego zrozumieniu. Powodowało to, że każdy pomysł na małą zmianę zachowania oprogramowania wymagałby pełnego zrozumienia działania programu i wszystkich jego elementów. W drugim przypadku został stworzony program w podejściu obiektowym, który wymusił na nas stworzenie nowej klasy z 2 metodami odpowiedzialnymi za konkretne zachowania.

Kod obiektowy jest bardziej przejrzysty, ale jednocześnie wielokrotnego użytku. Gdy wpadniemy na pomysł na stworzenie nowego programu, który również będzie zawierać walidację użytkownika, będziemy mogli użyć wcześniej stworzonej już klasy i ewentualnie dostosować ją do naszych aktualnych potrzeb. Taka praktyka znacznie przyspiesza pracę z kodem i pozwala nam być bardziej wydajnymi programistami.

Podejście obiektowe jest dla nas dużo bardziej namacalne i zrozumiałe, ponieważ wszystko możemy przedstawić właśnie w takiej formie. Czasem możesz stwierdzić, że coś na pozór lepiej pisze się proceduralnie, jednakże z czasem głębszego poznawania obiektowości, będziesz w stanie zauważyć, że taki sposób tworzenia programów jest lepszy i bardzo przejrzysty!

**Zadanie: Kod obiektowy to jest to!**

W linku poniżej umieściliśmy kod napisany strukturalnie. Zastanów się i przepisz ten kod na obiekt wraz z metodami, który nada naszemu programowi cechy podejścia obiektowego:

<https://kodilla.com/pl/project-java/38592>

Od teraz przesyłaj swoje zadania do sprawdzenia bezpośrednio do mentorów (przy pomocy przycisku "Wyślij do sprawdzenia" w górnej części edytora kodu – wewnątrz projektu zadania). Jednak jeżeli chcesz coś skonsultować odnośnie kodu lub masz problem z wykonaniem zadania, to po pomoc zapraszamy właśnie na kanał help-java.

[**Podgląd zadania**](https://kodilla.com/pl/bootcamp-task/310/476569)[**Przejdź do projektu**](https://kodilla.com/pl/project-java/162050)

**1.2. Interfejs**

**Czym jest interfejs?**

Zanim zaczniemy dokładnie opisywać, czym tak naprawdę jest interfejs oraz jak z niego skorzystać, zerknij na przykładowy kod zawierający interfejs:

interface Telephone

{

public void callTo(String name);

void sendMessageTo(String name);

}

Na pierwszy rzut oka widać, że jedna metoda jest **publiczna**, a druga jest pozbawiona określenia dostępu. Wynika to z faktu, że metody w interfejsie są publiczne i... **tylko publiczne**! To od Ciebie zależy, jak zapiszesz metodę w interfejsie. Możesz pominąć modyfikator dostępu, ale nie musisz :) Warto też zwrócić uwagę na fakt, że metody w interfejsie nie mają ciała – są jedynie pustymi deklaracjami.

Z racji tego, że już mniej więcej znamy programistyczny "wygląd" interfejsu, musimy teraz zrozumieć, po co on w ogóle jest i jakie daje nam korzyści w kontekście tworzenia programów.

Interfejs można opisać jako **zestaw wymagań**, które stawiamy jakiejś klasie odnośnie jej metod, które powinna zawierać.

Przykładowo, jeśli mamy w swoim kodzie kilka klas, z których każda powinna zawierać metody sayHello() oraz sayGoodbye(), moglibyśmy wykorzystać interfejs do narzucenia tego wymagania na poszczególne klasy. Uodparnia to nasz kod na działania innych programistów, którzy mogliby chcieć usunąć te metody lub zaimplementować klasę bez nich.

Należy pamiętać, że interfejs **nie jest** klasą, w związku z tym nie możemy po nim dziedziczyć ani używać na nim operatora new. Oprócz tego, kod zawierający interfejsy jest bardziej czytelny: z góry wiemy, jakich metod możemy się spodziewać w kodzie, a także zgadzamy się na to, by nasza klasa przyjęła jakiś określony "kształt" współdzielony z innymi klasami.

Sam koncept interfejsu pewnie brzmi na razie nieco groźnie, ale nie przejmuj się – wykorzystanie go w praktyce jest o wiele łatwiejsze niż myślisz. Opiszemy to sobie bardziej szczegółowo już za chwilę.

Oczywiście póki co, cały ten interfejs brzmi strasznie abstrakcyjnie – opiszmy sobie to zagadnienie na podstawie przykładu z życia codziennego. Wyobraź sobie klasę odwzorowującą blender kuchenny, który ma gniazdo (nasz interfejs) przyjmujące różne końcówki, które będą wykonywać zadania takie jak miksowanie, siekanie i mieszanie. Każda z tych końcówek, czyli klas implementujących interfejs gniazda (będących zgodnymi z tym gniazdem), musi posiadać ten sam kształt wejścia tak, aby pasowały do naszego gniazda (innymi słowy, spełniały wymagania interfejsu), a poszczególne końcówki odróżnia od siebie sposób wykonania zadania i efekt końcowy. Dzięki zastosowaniu gniazda na różne końcówki w blenderze, staje się on bardziej wielofunkcyjny, a dodawanie do niego nowych możliwości jest prostsze.

Tak właśnie sprawa ma się również z interfejsem w programowaniu.

Co więcej, mówi się także, że interfejs jest **gwarantem implementacji** – co to znaczy? Otóż, jak wspomnieliśmy wcześniej, każda klasa, która *zaimplementuje* interfejs, jest zmuszona mieć jego metody w sobie. Aby przełożyć to na ludzki język, możemy ponownie odwołać się do przykładu blendera. Gniazdo na końcówki samo w sobie nic nie robi, jednak gwarantuje nam, że po włożeniu końcówki coś się wydarzy (miksowanie, siekanie, mieszanie). Tak samo interfejs sam w sobie nie wykonuje żadnego działania oraz nie dostarcza nam metod, ale zapewnia nas, że wszystkie klasy, które go zaimplementują, będą zawierały określone w nim metody.

Uzyskanie takiego **gwarantu implementacji** jest bardzo pomocne w codziennym tworzeniu oprogramowania, chociażby w przypadku, gdy tworzymy aplikację, która posiada wybór różnych figur geometrycznych do narysowania w przeglądarce.

**Wykorzystanie interfejsu do projektu klasy**

Aplikacja, o której przed chwilą wspomnieliśmy, jest świetnym przykładem zastosowania funkcjonalności interfejsu. Wyobraźmy sobie trzy różne figury geometryczne – koło, kwadrat oraz trójkąt. Wiemy, że nasza aplikacja ma za zadanie narysować jedno z nich w zależności od tego, którą figurę wybierze użytkownik naszego programu.

Przed projektem interfejsu zbudujmy jedną klasę reprezentującą nasze koło:

class Circle

{

public void draw() {

System.out.println("I am drawing a circle!");

}

}

Klasa Circle posiada metodę draw(), która po wywołaniu wyświetli informację o rysowaniu koła. Możemy też spokojnie przyjąć, że każda z klas reprezentujących nasze figury będzie miała dokładnie taką samą metodę.

Dokładnie w tym tkwi cały sens interfejsu! Jeżeli klasy są podobne do siebie i wiemy, że w przyszłości na pewno będą zawierały w sobie metody o tych samych nazwach, jednakże sposób, w jaki będą wykonywać metody będzie różny od siebie, stosujemy interfejs. Zbudujmy go zatem dla wyjaśnienia tej zawiłości programistycznej :)

interface Shape {

void draw();

}

W powyższym kodzie interfejs Shape zawiera w sobie jedną metodę (**publiczną**, bo w interfejsie wszystkie metody są domyślne publiczne) o nazwie draw(), która nie zwraca żadnej wartości – dokładnie tak jak nasza klasa Circle.

Interfejsy są implementowane przez klasy. Z tego powodu po prawej stronie od nazwy klasy, zawsze przed podaniem nazwy interfejsu, musimy użyć słowa kluczowego implements (w dziedziczeniu korzystaliśmy z extends).

Spójrz na linijkę 7:

interface Shape {

void draw();

}

class Circle implements Shape {

public void draw() {

}

}

Nasz interfejs został zaimplementowany do klasy i jednocześnie **wymusza utworzenie metody draw() w naszej klasie**. Teraz jest to niewidoczne, jednak program nie skompilowałby się poprawnie, gdyby nasza klasa nie zawierała metody, której wymaga interfejs.

Stworzymy teraz kolejne klasy implementujące interfejs Shape:

interface Shape {

void draw();

}

class Circle implements Shape {

public void draw() {

}

}

class Rectangle implements Shape {

public void draw() {

}

}

class Square implements Shape {

public void draw() {

}

}

Świetnie! Jak widzisz, rozbudowaliśmy nasz projekt o klasy Square i Rectangle, które implementują interfejs Shape. Wymusza to na nich posiadanie metody draw(). Każda z klas zawiera dokładnie tę samą nazwę metody, co bardzo nam się za chwilę przyda!!

**Implementacja**

Aby zobaczyć, jaką dodatkową korzyść daje nam korzystanie z interfejsu, zaimplementujemy najpierw jakieś zadania naszym metodom draw():

interface Shape {

void draw();

}

class Circle implements Shape {

public void draw() {

System.out.println("I am drawing a Circle!");

}

}

class Rectangle implements Shape {

public void draw() {

System.out.println("I am drawing a Rectangle!");

}

}

class Square implements Shape {

public void draw() {

System.out.println("I am drawing a Square!");

}

}

Nasze metody uzyskały swoje własne zachowania... ale interfejs może dużo więcej. Zastanówmy się, jak wywołalibyśmy metody naszych obiektów. Na podstawie aktualnej wiedzy, w klasie Application stworzylibyśmy trzy nowe obiekty i wywołali ich metody. Jest to jak najbardziej poprawny sposób, ale czy możemy napisać klasę, która sama wywoła metody rysowania poszczególnych klas, jeżeli tylko podamy jej, jakiej klasy ma użyć? Oczywiście! Zbudujmy taką klasę, bo najlepszym tłumaczeniem jest kod!

interface Shape {

void draw();

}

class Circle implements Shape {

public void draw() {

System.out.println("I'm drawing a circle for you!");

}

}

class ShapeDrawer {

private Shape shape;

public ShapeDrawer(Shape shape) {

this.shape = shape;

}

public void process() {

System.out.println("Starting to draw the chosen shape...");

this.shape.draw();

System.out.println("Shape has been drawn.");

}

}

class App {

public static void main (String[] args) throws java.lang.Exception {

ShapeDrawer drawer = new ShapeDrawer(new Circle());

drawer.process();

}

}

Zobacz wynik kompilacji kodu:

<https://kodilla.com/pl/project-java/38578>

Odeszliśmy na chwilę od naszych wcześniej stworzonych klas, żeby nie mieszać w głowie. Skupmy się na właśnie utworzonej klasie ShapeDrawer.

Nasza klasa zawiera konstruktor (linia 15), w którego argumencie znajduje się **interfejs**. Jak pewnie się domyślasz, interfejs jest typem danych tak samo jak int, double czy String. W konstruktorze spełnia on bardzo ciekawe zadanie, do którego przejdziemy już za chwilę – na razie skupimy się na omówieniu naszej klasy.

Poza konstruktorem, w naszej klasie jest również atrybut klasy (linia 13) o typie naszego interfejsu (Shape), do którego przypiszemy wartość podaną w trakcie tworzenia nowego obiektu ShapeDrawer.

Wewnątrz klasy znajduje się również metoda process() (linia 19), która spełnia następujące instrukcje:

1. Wyświetla informację o rozpoczęciu rysowania,
2. Wywołuje metodę interfejsu – draw(),
3. Wyświetla informację o zakończeniu rysowania.

Klasa powinna być dla nas jasna i przejrzysta, ale na pewno zastanawiasz się, o co chodzi dokładnie z interfejsem wpisanym w atrybucie i polu klasy. Aby to dokładnie zrozumieć, należy po prostu z niej skorzystać.

interface Shape {

void draw();

}

class Circle implements Shape {

public void draw() {

System.out.println("I am drawing a Circle!");

}

}

class Rectangle implements Shape {

public void draw() {

System.out.println("I am drawing a Rectangle!");

}

}

class Square implements Shape {

public void draw() {

System.out.println("I am drawing a Square!");

}

}

class ShapeDrawer {

private Shape shape;

public ShapeDrawer(Shape shape) {

this.shape = shape;

}

public void process() {

System.out.println("Starting drawing choosen shape...");

this.shape.draw();

System.out.println("Shape has been drawed");

}

}

class Application {

public static void main(String args[]) {

Shape circle = new Circle();

ShapeDrawer shapeDrawer = new ShapeDrawer(circle);

shapeDrawer.process();

}

}

Tworzymy obiekt ShapeDrawer, następnie w argumencie konstruktora ShapeDrawer podajemy nowo utworzony obiekt Circle.

Następnie wywołujemy metodę process(), która należy do ShapeDrawer. Kliknij w poniższy link i zobacz wynik tej operacji w stdout:

<https://kodilla.com/pl/project-java/38583>

Na pewno wydarzyło się tutaj dla Ciebie wiele, i nie do końca rozumiesz zastosowanie tego całego ShapeDrawer. Przecież wystarczyłoby wywołać metodę draw() dla obiektu Circle i wszystko działałoby tak samo. To prawda, ale jednakże gdyśmy faktycznie tworzyli aplikację, w której użytkownik ma wybór między różnymi kształtami, wtedy kształty byłyby wskazywane dynamicznie – oznacza to tyle, że mógłby to być po prostu inny kształt niż Circle, a nasz kod przestałby działać.

**Notatka:** W programowaniu, jeżeli stosujemy słowo **dynamicznie**, oznacza to, że bezpośrednio nie mamy wpływu na to, co się wydarzy. W kontekście kształtów, to użytkownik zdecyduje, jaki kształt ma zostać narysowany.

Jeżeli nasz użytkownik ma możliwość dynamicznego wyboru kształtów, to rozwiązanie przedstawione powyżej będzie najlepszym sposobem na przyszłość. Dlaczego? Otóż wyobraź sobie, że użytkownik może wybrać w naszej aplikacji 50 różnych kształtów. Każdy obiekt jakiegoś kształtu będzie korzystać właśnie z klasy ShapeDrawer, która wywoła metodę rysowania tak jak należy.

**Interfejsy w Javie 8**

Java 8 urozmaiciła interfejsy o kilka nowych funkcjonalności, które je zrewolucjonizowały i nadały im jeszcze szersze pole do popisu.

**Metody domyślne**

Metody te dokonały niemożliwego! Metody domyślne mogą mieć swoje ciało i wykonywać instrukcje w nich zawarte. Interfejs może wymuszać implementację poszczególnych metod, bądź od razu ją w sobie zawierać. Spójrz na przykład:

interface Telephone {

void callTo();

default void callToMom() {

System.out.println("I am calling to my mom!");

}

}

Pierwsza metoda (callTo()) wymaga implementacji po stronie klasy, natomiast druga (callToMom()) czeka na wywołanie tylko wtedy, kiedy będzie potrzebna. Aby to zwizualizować, stworzyliśmy taki program:

interface Telephone {

void callTo(String name);

default void callToMom() {

System.out.println("I am calling to my mom!");

}

}

class Mobile implements Telephone {

public void callTo(String name) {

System.out.println("Calling to " + name);

}

}

class Application

{

public static void main (String[] args) throws java.lang.Exception

{

Mobile mobile = new Mobile();

mobile.callTo("Adam");

mobile.callToMom();

}

}

Jak widzisz, sytuacja wygląda podobnie jak w dziedziczeniu, jednakże klasa dziedziczona nie wymagała od nas implementacji jakiejś konkretnej metody. Interfejs w Javie 8 wymaga, ale również pozwala nam korzystać ze swoich już wcześniej stworzonych metod.

**Metody statyczne**

Na temat metod statycznych jeszcze wcześniej nic nie wspominaliśmy, jednak spotkaliśmy się już z taką metodą, która nazywa się main – na pewno ją kojarzysz! Otóż w Javie można tworzyć też metody statyczne nie tylko o takiej nazwie ;) Ale czym dokładnie jest ta metoda statyczna?

Metoda statyczna jest metodą **przypisaną do klasy**, czyli inaczej mówiąc, nie musimy tworzyć obiektu, żeby z niej skorzystać. Niestety to, że łatwo się ją wywołuje i nie trzeba tworzyć obiektu, wcale nie znaczy, że jest lepsza. Ma sporo ograniczeń i minusów, lecz o tym powiemy więcej w temacie związanym z testowaniem naszego oprogramowania.

Na ten moment wzbogacimy nasz interfejs o taką metodę:

interface Telephone {

void callTo(String name);

default void callToMom() {

System.out.println("I am calling to my mom!");

}

static String getMyNumber() {

return "+48 909 901 928";

}

}

class Application

{

public static void main (String[] args) throws java.lang.Exception

{

System.out.println(Telephone.getMyNumber());

}

}

Sprawdź działanie powyższego kodu:

<https://kodilla.com/pl/project-java/38584>

Jak widzisz, metodę możemy wywołać bezpośrednio z samego interfejsu!

**Zadanie: Bankomat**

Stwórz interfejs obsługujący bankomat (eng. ATM) zawierający:

* Dwie metody wymagające implementacji – wpłata i wypłata
* Metodę domyślną – informację o połączeniu z bankiem
* Metodę statyczną – informację o zakończeniu działania bankomatu

Następnie zaimplementuj taki interfejs do klasy o nazwie ATMImpl (ATM Implementation) i stwórz ciała dla metod, które tego wymagają. W klasie Application w metodzie main wywołaj każdą metodę.

[**Podgląd zadania**](https://kodilla.com/pl/bootcamp-task/310/476570)[**Przejdź do projektu**](https://kodilla.com/pl/project-java/162051)

**1.3. Wykorzystanie interfejsu w programach**

**Wstępny projekt aplikacji**

Wykorzystajmy naszą wiedzę o interfejsach i obiektowości do stworzenia aplikacji, która rozpozna zwierzę, które zostanie jej podane w konstruktorze! Dzięki takiemu programowi lepiej zrozumiesz, jak działają interfejsy.

* Nasza aplikacja powinna zawierać interfejs Animal, który dostarczy każdej klasie, takiej jak Dog, Cat i Mouse, odpowiednie metody.
* Dog, Cat i Mouse powinny mieć własne implementacje metod wymaganych przez interfejs.
* Następnie musimy wzbogacić aplikację o klasę AnimalRecognizer, która będzie odpowiedzialna za rozpoznawanie przekazanego do niej zwierzęcia.

Takie podpunkty pomogą nam dobrze i schematycznie zbudować wymyśloną przez nas aplikację. Zawsze dobrą praktyką przed rozpoczęciem jakiegoś zadania jest przygotowanie sobie wstępnego planu kodowania. Do tego rodzaju projektów można podejść kilka sposobów:

* projektowanie za pomocą tekstu (tylko bardzo małe aplikacje),
* projektowanie za pomocą UML, o którym więcej możesz przeczytać na tej stronie: [https://www.tutorialspoint.com/uml/](https://www.tutorialspoint.com/uml). Do UML wrócimy w dalszej części tego kursu, kiedy będziemy projektować większą aplikację.

**Tworzenie interfejsu dla aplikacji**

Jak wspomnieliśmy wcześniej, musimy stworzyć interfejs Animal, który dostarczy niezbędnych metod naszym klasom reprezentującym konkretne zwierzęta. Do zaspokojenia naszych potrzeb interfejs będzie zawierał 2 metody zwracające dane typu String:

* String voice(),
* String name()

Przejdźmy do stworzenia takiego interfejsu:

interface Animal

{

/\*\*

\* Retrieves voice of an animal

\*/

String voice();

/\*\*

\* Retrieves name of an animal

\*/

String name();

}

**Notatka**: Jako mały dodatek do naszego interfejsu dodaliśmy komentarz, którego zadaniem jest objaśnienie celu metody w interfejsie. Nie powinna być to twoja stała praktyka, jednakże warto mieć na uwadze taką możliwość, jeżeli w przyszłości zechcesz poinformować innego programistę o ścisłym celu konkretnej metody. Komentarze wspominaliśmy w Module 0, a to przykład jednego z nich. Pamiętaj, że komentarze są tylko dla programistów, kompilator nie będzie zwracał na nie uwagi :)

Nasz interfejs jest już wzbogacony o metody. W następnej sekcji będziemy implementować je w klasach reprezentujących konkretne zwierzęta.

**Implementacja**

Zgodnie z naszym planem musimy stworzyć 3 nowe klasy implementujące interfejs Animal oraz stworzyć unikalne instrukcje zachowania tych zwierząt:

interface Animal

{

/\*\*

\* Retrieves voice of an animal

\*/

String voice();

/\*\*

\* Retrieves name of an animal

\*/

String name();

}

class Dog implements Animal {

public String voice() {

return "Hau hau!";

}

public String name() {

return "Dog";

}

}

class Cat implements Animal {

public String voice() {

return "miau!";

}

public String name() {

return "Cat";

}

}

class Mouse implements Animal {

public String voice() {

return "pi!";

}

public String name() {

return "Mouse";

}

}

Według planu, rozbudowaliśmy nasz program o trzy nowe klasy implementujące interfejs Animal – każda z tych klas nadała własne zachowania metodom interfejsu. Dzięki temu możemy przystąpić do realizowania naszej klasy AnimalRecognizer, która rozpozna przekazane do niej obiekty w postaci zwierząt. W tym celu na ten moment pominiemy wcześniej napisany kod i zajmiemy się samą klasą AnimalRecognizer:

class AnimalRecognizer

{

public String recognize(Animal animal) {

return "I am sure that this is: " + animal.name()

+ "and sounds like: " + animal.voice();

}

}

Nasza klasa AnimalRecognizer ma jedną metodę publiczną recognize(), przyjmującą w argumencie obiekty implementujące interfejs Animal.

Metoda recognize() zwraca dane typu String, a dokładnie tekst, który odpowiada na pytanie: "Jakie zwierzę zostało przekazane do AnimalRecognizer?"

Wszystko mamy już prawie gotowe, pora stworzyć klasę, która będzie odpowiedzialna za uruchomienie programu...

**Uruchomienie programu**

Aby uruchomić nasz prawie gotowy program, będziemy potrzebować klasy Application z metodą main. Utworzy ona nowy obiekt AnimalRecognizer i pozwoli nam na przekazanie obiektów reprezentujących zwierzęta:

interface Animal

{

/\*\*

\* Retrieves voice of an animal

\*/

String voice();

/\*\*

\* Retrieves name of an animal

\*/

String name();

}

class Dog implements Animal {

public String voice() {

return "Hau hau!";

}

public String name() {

return "Dog";

}

}

class Cat implements Animal {

public String voice() {

return "miau!";

}

public String name() {

return "Cat";

}

}

class Mouse implements Animal {

public String voice() {

return "pi!";

}

public String name() {

return "Mouse";

}

}

class AnimalRecognizer {

public String recognize(Animal animal) {

return "I am sure that this is: " + animal.name()

+ " and sounds like: " + animal.voice();

}

}

class Application {

public static void main(String args[]) {

AnimalRecognizer recogonizer = new AnimalRecognizer();

String recognizedAnimal = recogonizer.recognize(new Cat());

System.out.println(recognizedAnimal);

}

}

Sprawdź działanie programu:

<https://kodilla.com/pl/project-java/38586>

To co nowe, to stworzenie obiektu Cat za pomocą operatora new wewnątrz metody recognize(). Jest to w pełni dopuszczalne w Javie – nie musimy za każdym razem tworzyć referencji obiektu, możemy stworzyć obiekt tam, gdzie go potrzebujemy. Zerknij na stdout – jak widzisz, udało się rozpoznać przekazane zwierze w argumencie metody recognize(). Dzięki temu widzimy, jak wiele może dać nam korzystanie z interfejsów i jak bardzo praktyczne jest jego zastosowanie!

**Zadanie: Odważny rycerz**

W swojej aplikacji stwórz:

* Interfejs Quest, który będzie zawierał w metodę process().
* Dwie klasy DeadIslandQuest i EliteKnightQuest, które implementują interfejs Quest – wykorzystaj polecenie System.out.println().
* Stwórz klasę Knight, która w konstruktorze przyjmie różne zadania implementujące interfejs Quest, oraz dowolną metodę, która wywoła metodę process() interfejsu Quest.
* Główny program powinien wyświetlić informacje o zakończeniu zadania razem z jego nazwą.

[**Podgląd zadania**](https://kodilla.com/pl/bootcamp-task/310/476571)[**Przejdź do projektu**](https://kodilla.com/pl/project-java/162052)

**1.4. Klasa abstrakcyjna**

**Czym jest klasa abstrakcyjna?**

Klasa abstrakcyjna jest klasą, która bardzo mocno przypomina interfejs z Javy 8. Przyczyniło się to do wyparcia klas abstrakcyjnych właśnie przez interfejs, w którym dodano metody statyczne i domyślne, wspomniane w submodule 1.2.

Warto powiedzieć tutaj, dlaczego klasy abstrakcyjne nie są aż tak fajne jak interfejsy: **klasy abstrakcyjne są dziedziczone, a interfejsy implementowane**. Java jest językiem programowania, który pozwala na dziedziczenie tylko jednej klasy, a implementowanie wielu interfejsów, co czyni interfejsy bardziej przydatnymi narzędziami w trakcie budowania aplikacji. Jednak, jak wiadomo, ze wszystkim można przesadzić – nie jest dobrą praktyką implementowanie 3-4 interfejsów w jednej klasie, gdyż trudno jest wtedy doprowadzić do pojedynczej odpowiedzialności klasy.

Klasy abstrakcyjne są klasami, których, tak jak interfejsów, nie da się utworzyć za pomocą operatora new. Można to tłumaczyć w ten sposób, że nie są one pełnoprawnymi klasami, na podstawie których można stworzyć obiekt. Są to "niekompletne" klasy, definiujące jedynie pewien schemat dla klas, które te schematy odziedziczą, ale jednocześnie wymagające od programisty zaprogramowania całej reszty w klasie dziedziczącej tak, aby całość miała sens. Spójrzmy, jak wygląda przykładowa klasa abstrakcyjna:

abstract class Person {

/\*\*

\* Attribute same as basic class

\*/

private String name;

/\*\*

\* Constructor same as basic class

\*/

public Person(String name) {

this.name = name;

}

/\*\*

\* Public method same as basic class

\*/

public String getName() {

return this.name;

}

/\*\*

\* Abstract method which requires implementation in class that

\* extends abstract class Person.

\*/

abstract void voice();

}

Jak widzisz, klasa abstrakcyjna może mieć konstruktor i atrybuty klasy, co pozwala na tworzenie *getterów* i jednocześnie sprzyja to czystości klas, które będą ją dziedziczyć. Jednakże klasy abstrakcyjnej, tak jak wcześniej wspominaliśmy, nie można tworzyć za pomocą operatora new. Jedyną szansą na przekazanie wartości do konstruktora klasy abstrakcyjnej jest poznana wcześniej metoda super() wywołana w konstruktorze klasy, która dziedziczy klasę abstrakcyjną. Jak zapewne pamiętasz, metoda super() wywołuje konstruktor klasy, po której dziedziczymy – dzięki temu możemy wywołać konstruktor klasy abstrakcyjnej bez używania słowa kluczowego new. Możesz zaobserwować to w poniższym przykładzie:

abstract class Person {

/\*\*

\* Attribute same as basic class

\*/

private String name;

/\*\*

\* Constructor same as basic class

\*/

public Person(String name) {

this.name = name;

}

/\*\*

\* Public method same as basic class

\*/

public String getName() {

return this.name;

}

/\*\*

\* Abstract method which requires implementation in class that

\* extends abstract class Person.

\*/

abstract public void voice();

}

class Male extends Person {

public Male(String name) {

super(name);

}

public void voice() {

System.out.println("Hey there!");

}

}

class Application {

public static void main(String args[]) {

Male male = new Male("John");

String name = male.getName();

System.out.println(name);

male.voice();

}

}

Podsumowując, klasy abstrakcyjne można krótko opisać jako połączenie interfejsów i dziedziczenia, przedstawionego w module 0.

**Zastosowanie**

Przede wszystkim klasy abstrakcyjnej możemy użyć do zaimplementowania podstawowej funkcjonalności klasom, które będą po niej dziedziczyć, lecz samo utworzenie jej obiektu byłoby zbyt abstrakcyjne i nie wnosiłoby zbyt wiele do programu.

Dobrym zastosowaniem klasy abstrakcyjnej jest również wydzielenie w niej kodu wspólnego dla każdej z klas, które będą po niej dziedziczyć, jednocześnie wymagając implementacji innych metod specyficznych dla klas dziedziczących, dokładnie tak jak w interfejsie. Aby to sobie zobrazować, stwórzmy przykładową klasę abstrakcyjną:

abstract class ComplicatedAlgorithm {

abstract protected void firstMethodToExecute();

abstract protected void secondMethodToExecute();

private void thirdMethodToExecute() {

System.out.println("This is third method to execute!");

}

private void fourthMethodToExecute() {

System.out.println("This is fourth method to execute.");

}

public void run() {

this.firstMethodToExecute();

this.secondMethodToExecute();

this.thirdMethodToExecute();

this.fourthMethodToExecute();

}

}

Dwie metody klasy abstrakcyjnej (firstMethodToExecute() i secondMethodToExecute()) wymagają implementacji. W innym wypadku nasz program nie skompiluje się i wyświetli nam się błąd informujący, że mimo obowiązku zaimplementowania metod nie zostały one utworzone w klasie dziedziczącej klasę abstrakcyjną. Pozostałe trzy metody mają już zawarte w sobie instrukcje. Dwie z nich odpowiedzialne są za poinformowanie o tym, która metoda jest wywoływana. Jednakże w metodzie run() możemy zauważyć wywołanie wszystkich metod zawartych w klasie, razem z tymi metodami, które wymagają własnej implementacji.

**Metody o dostępności protected**

Nowym terminem, który pojawił się w kodzie klasy abstrakcyjnej (termin ten jest dostępny również dla wszystkich klas, poza interfejsem), jest **protected** – co oznacza? Otóż jest to ograniczenie dostępności metody dla klasy dziedziczącej i wszystkich klas w danym pakiecie. Innymi słowy, klasa abstrakcyjna mówi klasie, która po niej dziedziczy, że metody oznaczone jako **protected** będą dostępne w jej obrębie i we wszystkich klasach w tym samym pakiecie.

Do czego służy taka operacja? Jest ona bardzo przydatna, jeżeli znamy konkretne zachowanie stworzonego algorytmu (wiemy co po sobie dokładnie następuje), jednak każda klasa dziedzicząca klasę abstrakcyjną ma obowiązek stworzyć instrukcje dla pierwszych dwóch metod dla sprawnego wywołania algorytmu.

Jest to nowy termin i warto zaznaczyć, że możemy wykorzystywać go w każdej klasie, nie tylko w klasach abstrakcyjnych.

Zobaczmy, jak będzie wyglądała taka implementacja:

abstract class ComplicatedAlgorithm {

abstract protected void firstMethodToExecute();

abstract protected void secondMethodToExecute();

private void thirdMethodToExecute() {

System.out.println("This is third method to execute!");

}

private void fourthMethodToExecute() {

System.out.println("This is fourth method to execute.");

}

public void run() {

this.firstMethodToExecute();

this.secondMethodToExecute();

this.thirdMethodToExecute();

this.fourthMethodToExecute();

}

}

class ComplicatedAlgorithmRunner extends ComplicatedAlgorithm {

protected void firstMethodToExecute() {

System.out.println("This is first method to execute");

}

protected void secondMethodToExecute() {

System.out.println("This is second method to execute");

}

}

Jak widzisz, w linijce 24 w klasie ComplicatedAlgorithmRunner utworzone zostały implementacje dwóch metod abstrakcyjnych, wykonujące wyświetlanie tekstu przy pomocy polecenia System.out.println(). Reszta metod została nam dostarczona przez klasę abstrakcyjną.

Wywołajmy kod przygotowany do tej pory w klasie Application zawierającej metodę main():

abstract class ComplicatedAlgorithm {

abstract protected void firstMethodToExecute();

abstract protected void secondMethodToExecute();

private void thirdMethodToExecute() {

System.out.println("This is third method to execute!");

}

private void fourthMethodToExecute() {

System.out.println("This is fourth method to execute.");

}

public void run() {

this.firstMethodToExecute();

this.secondMethodToExecute();

this.thirdMethodToExecute();

this.fourthMethodToExecute();

}

}

class ComplicatedAlgorithmRunner extends ComplicatedAlgorithm {

protected void firstMethodToExecute() {

System.out.println("This is first method to execute");

}

protected void secondMethodToExecute() {

System.out.println("This is second method to execute");

}

}

class Application {

public static void main(String args[]) {

ComplicatedAlgorithmRunner runner = new ComplicatedAlgorithmRunner();

runner.run();

}

}

[Spójrz na wynik w *stdout*](https://kodilla.com/pl/project-java/38587) – każda z naszych wcześniej zaprogramowanych metod wykorzystała kod zgodnie z ustaloną przez nas kolejnością. Jednocześnie kod jest oddzielony do klas, co sprzyja przejrzystości i czytelności naszego programu. Klasa Application zawiera dzięki tej operacji tylko i wyłącznie utworzenie obiektu ComplicatedAlgorithmRunner i wywołanie metody run(), która została odziedziczona po klasie abstrakcyjnej ComplicatedAlgorithm.

Taki sposób tworzenia kodu nazywamy **Metodą szablonową (Template method)**: klasa abstrakcyjna gwarantuje szablon wykonania jakiegoś określonego przez programistę zadania. W programowaniu znajduje się wiele sposobów na tworzenie kodu i są one nazywane **wzorcami projektowymi**, które poznamy i wykorzystamy w praktyce w dalszej części kursu.

**Nadpisywanie**

Wyobraźmy sobie sytuację, w której napisana wcześniej przez jakiegoś programistę metoda nie spełnia naszych oczekiwań, ponieważ akurat dla tej klasy, którą tworzymy, musimy wykonać całkowicie inne operacje. Dziwną sytuacją byłoby tworzenie nowej klasy, która zawierałaby tą samą nazwę metody i wykonywałaby inne zadanie.

Na szczęście Java posiada w sobie **adnotacje**, które mówią Javie, co dokładnie należy zrobić przed wykonaniem jakiegokolwiek kodu. **Adnotacje** oznacza się za pomocą znaku @ (ang. *at*), a następnie podaje się nazwę adnotacji. Żeby dokładnie zrozumieć, jak działa taka adnotacja, użyjemy przykładu opartego na wcześniejszym przykładzie klasy abstrakcyjnej. Spójrz na linjkę 34:

abstract class ComplicatedAlgorithm {

abstract protected void firstMethodToExecute();

abstract protected void secondMethodToExecute();

private void thirdMethodToExecute() {

System.out.println("This is third method to execute!");

}

private void fourthMethodToExecute() {

System.out.println("This is fourth method to execute.");

}

public void run() {

this.firstMethodToExecute();

this.secondMethodToExecute();

this.thirdMethodToExecute();

this.fourthMethodToExecute();

}

}

class ComplicatedAlgorithmRunner extends ComplicatedAlgorithm {

protected void firstMethodToExecute() {

System.out.println("This is first method to execute");

}

protected void secondMethodToExecute() {

System.out.println("This is second method to execute");

}

@Override

public void run() {

//Implementation

}

}

class Application {

public static void main(String args[]) {

ComplicatedAlgorithmRunner runner = new ComplicatedAlgorithmRunner();

runner.run();

}

}

Użyliśmy właśnie adnotacji @Override dla metody run(), czyli powiedzieliśmy naszemu programowi, że chcemy **nadpisać** instrukcje w niej zawarte i zastąpić je swoimi własnymi. Można zaobserwować również działanie tej adnotacji [w postaci pustego wyniku w *stdout.*](https://kodilla.com/pl/project-java/38589)

Zwróć uwagę, że nie musieliśmy nadpisywać metod implementowanych na podstawie interfejsu czy "wymaganych" przez klasę abstrakcyjną. Wniosek jest taki, że musimy użyć adnotacji @Override do nadpisywania metod, które faktycznie mają już jakieś ciało w klasie z której dziedziczymy, a nie są jedynie "wymaganiem" zaimplementowania danej metody.

Wracając do @Override: adnotacja ta pozwala nam całkowicie zmienić nadpisywaną metodę zgodnie z naszymi potrzebami. Dla eksperymentu wywołajmy w nadpisanej metodzie run() tylko metody zaimplementowane przez nas, czyli firstMethodToExecute() i secondMethodToExecute() w linijce 36 i 37:

abstract class ComplicatedAlgorithm {

abstract protected void firstMethodToExecute();

abstract protected void secondMethodToExecute();

private void thirdMethodToExecute() {

System.out.println("This is third method to execute!");

}

private void fourthMethodToExecute() {

System.out.println("This is fourth method to execute.");

}

public void run() {

this.firstMethodToExecute();

this.secondMethodToExecute();

this.thirdMethodToExecute();

this.fourthMethodToExecute();

}

}

class ComplicatedAlgorithmRunner extends ComplicatedAlgorithm {

protected void firstMethodToExecute() {

System.out.println("This is first method to execute");

}

protected void secondMethodToExecute() {

System.out.println("This is second method to execute");

}

@Override

public void run() {

this.firstMethodToExecute();

this.secondMethodToExecute();

}

}

class Application {

public static void main(String args[]) {

ComplicatedAlgorithmRunner runner = new ComplicatedAlgorithmRunner();

runner.run();

}

}

[Zerknij na wynik w *stdout*](https://kodilla.com/pl/project-java/38590) – udało nam się zmienić działanie metody run(), nadając jej pożądane przez nas działanie. Jak widzisz, adnotacja @Override może się okazać bardzo przydatna w pracy z kodem napisanym zgodnie z podejściem obiektowym.

**Zadanie: Transakcja bankowa**

Aby przećwiczyć wykorzystywanie klas abstrakcyjnych w praktyce, stwórz swoje własne zachowanie dla transakcji bankowej.

Załóżmy, że transakcja bankowa zawiera się w 8 metodach, z czego 3 są zawsze po stronie banku (w klasie abstrakcyjnej).

Schemat transakcji bankowej do zaimplementowania:

1. Prośba o wprowadzenie karty (Bankomat)
2. Prośba o wprowadzenie kodu PIN (Bankomat)
3. Prośba o wprowadzenie kwoty do wypłaty (Bankomat)
4. Informacja o przyjęciu prośby wypłaty przez Bank (Bank)
5. Sprawdzenie stanu konta (Bank)
6. Potwierdzenie możliwości wypłaty (Bank)
7. Wypłacanie gotówki (Bankomat)
8. Podziękowanie za skorzystanie z usług Banku (Bankomat)

Załóż, że każde z tych działań sygnalizujemy zwykłym println. Tak jak wspominaliśmy wcześniej, póki co zajmujemy się głównie poznawaniem samego języka Java – praktyczne zadania przyjdą niebawem :)

Posługuj się przypadkiem przedstawionym w tym submodule: <https://kodilla.com/pl/project-java/38591>

Spróbuj po prostu napisać kod, który będzie imitował transakcję bankową. W tym zadaniu najważniejsze jest to, żeby samodzielnie spróbować podejść do definiowania metod i pisania kodu.

[**Podgląd zadania**](https://kodilla.com/pl/bootcamp-task/310/476572)[**Przejdź do projektu**](https://kodilla.com/pl/project-java/162053)

**1.5. Wykorzystanie klasy abstrakcyjnej w programach**

**Wstępny projekt aplikacji**

Zaprojektujemy aplikację do tworzenia wypłat dla pracowników w firmie. W tym celu wykorzystamy klasę abstrakcyjną, która pozwoli nam przeprowadzić tę operację sprawnie. Wykorzystamy również poznany wcześniej wzorzec projektowy Template.

Aby sprawnie zrealizować projektowanie aplikacji, zastanówmy się najpierw, co dokładnie musi w sobie zawierać taki program:

1. **Klasę abstrakcyjną** – SalaryPayout, która będzie zawierać metody odpowiedzialne za wykonanie przelewu i wyświetlanie informacji o zrealizowanym przelewie.
2. **Klasę dziedziczącą klasę abstrakcyjną** – SalaryPayoutProcessor, która zadba o implementację metod klasy abstrakcyjnej.
3. **Dwie klasy pracowników**, które będą reprezentować różnego rodzaje pracowników – takich, którzy mają stałe wynagrodzenie i pracowników, którzy rozliczani są z przepracowanych godzin.

Projekt nabrał już jakiegoś kształtu, co pozwoli nam na zaprojektowanie wstępnych klas z metodami – póki co bez implementacji. W tym projekcie zaczniemy od utworzenia klas reprezentujących dwa rodzaje pracowników: FixedSalaryEmployee i HourlySalaryEmployee, którym będzie towarzyszył wcześniej poznany interfejs:

interface Employee {

double calculateSalary();

}

class FixedSalaryEmployee implements Employee {

private double salary;

public FixedSalaryEmployee(double salary) {

this.salary = salary;

}

public double calculateSalary() {

return this.salary;

}

}

class HourlySalaryEmployee implements Employee {

private double hours;

private double hourlyPay;

public HourlySalaryEmployee(double hours, double hourlyPay) {

this.hours = hours;

this.hourlyPay = hourlyPay;

}

public double calculateSalary() {

return this.hourlyPay \* this.hours;

}

}

Na samym początku kodu widzimy interfejs Employee, który zawiera tylko jedną metodę (calculateSalary()). Dzięki zastosowaniu interfejsu, wymusimy zaimplementowanie obliczania pensji w zależności od rodzaju pracownika. Każdy z utworzonych pracowników posiada tę metodę i implementuje inne zachowanie. Pierwszy typ pracownika – FixedSalaryEmployee – zwraca po prostu przekazaną w konstruktorze pensję, natomiast dla HourlySalaryEmployee mnoży przepracowane godziny i stawkę godzinową pracownika, co w rezultacie daje nam jego pensję. Tak zaimplementowane typy pracowników będą dla nas świetnym początkiem do tworzenia aplikacji do wypłat pensji.

Następnym krokiem będzie przygotowanie szablonu klasy abstrakcyjnej SalaryPayout i klasy dziedziczącej SalaryPayoutProcessor:

interface Employee {

double calculateSalary();

}

class FixedSalaryEmployee implements Employee {

private double salary;

public FixedSalaryEmployee(double salary) {

this.salary = salary;

}

public double calculateSalary() {

return this.salary;

}

}

class HourlySalaryEmployee implements Employee {

private double hours;

private double hourlyPay;

public HourlySalaryEmployee(double hours, double hourlyPay) {

this.hours = hours;

this.hourlyPay = hourlyPay;

}

public double calculateSalary() {

return this.hourlyPay \* this.hours;

}

}

// Payout

abstract class SalaryPayout {

private Employee employee;

public SalaryPayout(Employee employee) {

this.employee = employee;

}

protected abstract void payout();

public void processPayout() {

//Implementation

}

}

class SalaryPayoutProcessor extends SalaryPayout {

public SalaryPayoutProcessor(Employee employee) {

super(employee);

}

protected void payout() {

//Implementation

}

}

**Klasa abstrakcyjna SalaryPayout**

Skupmy się teraz na kodzie rozpoczynającym się od linijki 37. Stworzyliśmy klasę abstrakcyjną (linia 39), wymagającą implementacji metody payout() (linia 47) przez klasę dziedziczącą, oraz metodę processPayout(), którą zaimplementujemy jeszcze wewnątrz klasy abstrakcyjnej.

**Klasa SalaryPayoutProcessor**

W dalszej części kodu, a mianowicie w linijce 55, została utworzona klasa SalaryPayoutProcessor, która ma już zawartą w sobie metodę do implementacji klasy abstrakcyjnej oraz konstruktor przyjmujący obiekty implementujące interfejs Employee, który posłuży do poprawnego zaimplementowania metody payout(). Pamiętaj, aby dokładnie przeanalizować kod powyżej i zrozumieć jego każdą linijkę – dzięki temu późniejsze korzystanie z klasy abstrakcyjnej bądź interfejsu będzie dla Ciebie bardzo proste!

Robimy aktualnie krok dalej i rozpoczynamy implementacje metod, które na ten moment są puste. Sprawmy, by nasz program zaczął działać!

**Implementacja**

Nasi użytkownicy mają już zaimplementowaną funkcjonalność obliczania pensji, więc pierwszy krok za nami. Zadbajmy teraz o wewnętrzną metodę klasy abstrakcyjnej processPayout(). Spójrz na linię 49:

interface Employee {

double calculateSalary();

}

class FixedSalaryEmployee implements Employee {

private double salary;

public FixedSalaryEmployee(double salary) {

this.salary = salary;

}

public double calculateSalary() {

return this.salary;

}

}

class HourlySalaryEmployee implements Employee {

private double hours;

private double hourlyPay;

public HourlySalaryEmployee(double hours, double hourlyPay) {

this.hours = hours;

this.hourlyPay = hourlyPay;

}

public double calculateSalary() {

return this.hourlyPay \* this.hours;

}

}

// Payout

abstract class SalaryPayout {

private Employee employee;

public SalaryPayout(Employee employee) {

this.employee = employee;

}

protected abstract void payout();

public void processPayout() {

System.out.println("Creating payout for: " + this.employee.calculateSalary() + " PLN");

this.payout();

System.out.println("Payout has been completed!");

}

}

class SalaryPayoutProcessor extends SalaryPayout {

public SalaryPayoutProcessor(Employee employee) {

super(employee);

}

protected void payout() {

//Implementation

}

}

Nasza metoda processPayout() informuje o tworzeniu wypłaty na określoną kwotę, a następnie wywołuje metodę payout(), która zostanie zaimplementowana przez SalaryPayoutProcessor. Po udanym procesie wykonywania wypłaty, program informuje o udanej transakcji.

W dalszej części naszego programu musimy skupić się na klasie SalaryPayoutProcessor, która jest odpowiedzialna za realizację wypłaty dla naszych pracowników.

W naszym przykładzie nie będzie ona spełniała żadnej skomplikowanej funkcji, jak np. znajdowanie numeru konta czy łączenie się z systemem bankowości internetowej, ponieważ zajęłoby nam to wiele pracy i musielibyśmy rozbudować nasz projekt **znacznie** bardziej. Metoda ta najzwyczajniej wyświetli informację o tym, że wysyła pieniądze do pracownika. Wracając do naszej implementacji, spójrz na linię 64 – znajduje się tam wywołanie polecenia System.out.println();:

interface Employee {

double calculateSalary();

}

class FixedSalaryEmployee implements Employee {

private double salary;

public FixedSalaryEmployee(double salary) {

this.salary = salary;

}

public double calculateSalary() {

return this.salary;

}

}

class HourlySalaryEmployee implements Employee {

private double hours;

private double hourlyPay;

public HourlySalaryEmployee(double hours, double hourlyPay) {

this.hours = hours;

this.hourlyPay = hourlyPay;

}

public double calculateSalary() {

return this.hourlyPay \* this.hours;

}

}

// Payout

abstract class SalaryPayout {

private Employee employee;

public SalaryPayout(Employee employee) {

this.employee = employee;

}

protected abstract void payout();

public void processPayout() {

System.out.println("Creating payout for: " + this.employee.calculateSalary() + " PLN");

this.payout();

System.out.println("Payout has been completed!");

}

}

class SalaryPayoutProcessor extends SalaryPayout {

public SalaryPayoutProcessor(Employee employee) {

super(employee);

}

protected void payout() {

System.out.println("Sending money to employee");

}

}

Jak widzisz, na razie nasze programy opierają się na wyświetlaniu danych w [stdout](https://kodilla.com/pl/project-java/38593) – wynika to z tego, że chcemy najpierw poznać dokładnie tajniki programowania obiektowego oraz samej Javy, a następnie przejść do bardziej "praktycznego" programowania :)

**Uruchomienie programu**

Sprawdźmy, jak działa nasz program w zależności od użytego obiektu pracownika. Dla każdego obiektu pracownika powinniśmy otrzymać różne wyliczenia pensji. Uruchomimy nasz program na początku z wykorzystaniem pracownika FixedSalaryEmployee – spójrz na linijkę 68, gdzie znajduje się klasa Application:

interface Employee {

double calculateSalary();

}

class FixedSalaryEmployee implements Employee {

private double salary;

public FixedSalaryEmployee(double salary) {

this.salary = salary;

}

public double calculateSalary() {

return this.salary;

}

}

class HourlySalaryEmployee implements Employee {

private double hours;

private double hourlyPay;

public HourlySalaryEmployee(double hours, double hourlyPay) {

this.hours = hours;

this.hourlyPay = hourlyPay;

}

public double calculateSalary() {

return this.hourlyPay \* this.hours;

}

}

// Payout

abstract class SalaryPayout {

private Employee employee;

public SalaryPayout(Employee employee) {

this.employee = employee;

}

protected abstract void payout();

public void processPayout() {

System.out.println("Creating payout for: " + this.employee.calculateSalary() + " PLN");

this.payout();

System.out.println("Payout has been completed!");

}

}

class SalaryPayoutProcessor extends SalaryPayout {

public SalaryPayoutProcessor(Employee employee) {

super(employee);

}

protected void payout() {

System.out.println("Sending money to employee");

}

}

class Application {

public static void main(String args[]) {

FixedSalaryEmployee employee = new FixedSalaryEmployee(2000.0);

SalaryPayoutProcessor processor = new SalaryPayoutProcessor(employee);

processor.processPayout();

}

}

Opiszmy sobie po kolei, co dzieje się w tej metodzie:

1. Wewnątrz metody main został stworzony obiekt na podstawie klasy FixedSalaryEmployee, w którego konstruktorze podaliśmy wartość stałego wynagrodzenia pracownika.
2. W następnym kroku tworzymy obiekt SalaryPayoutProcessor, który jest odpowiedzialny za realizowanie wypłat dla wcześniej stworzonych pracowników. W trakcie tworzenia tego obiektu przekazujemy mu także obiekt odpowiadający za pracownika, na podstawie którego będzie wykonywał określone działania.
3. Na końcu wywołujemy metodę processPayout(), która wypłaca pensję pracownikowi.

Utwórzmy nowego pracownika, którego sposób wyliczania pensji jest zależny od czasu pracy i stawki godzinowej. Aby zrealizować płatność dla tego pracownika, musimy stworzyć nowy obiekt klasy HourlySalaryEmployee w linijce 73:

interface Employee {

double calculateSalary();

}

class FixedSalaryEmployee implements Employee {

private double salary;

public FixedSalaryEmployee(double salary) {

this.salary = salary;

}

public double calculateSalary() {

return this.salary;

}

}

class HourlySalaryEmployee implements Employee {

private double hours;

private double hourlyPay;

public HourlySalaryEmployee(double hours, double hourlyPay) {

this.hours = hours;

this.hourlyPay = hourlyPay;

}

public double calculateSalary() {

return this.hourlyPay \* this.hours;

}

}

// Payout

abstract class SalaryPayout {

private Employee employee;

public SalaryPayout(Employee employee) {

this.employee = employee;

}

protected abstract void payout();

public void processPayout() {

System.out.println("Creating payout for: " + this.employee.calculateSalary() + " PLN");

this.payout();

System.out.println("Payout has been completed!");

}

}

class SalaryPayoutProcessor extends SalaryPayout {

public SalaryPayoutProcessor(Employee employee) {

super(employee);

}

protected void payout() {

System.out.println("Sending money to employee");

}

}

class Application {

public static void main(String args[]) {

FixedSalaryEmployee employee = new FixedSalaryEmployee(2000);

HourlySalaryEmployee employee1 = new HourlySalaryEmployee(40, 19);

SalaryPayoutProcessor processor = new SalaryPayoutProcessor(employee1);

processor.processPayout();

}

}

[Spójrz na stdout](https://kodilla.com/pl/project-java/38594) – można zauważyć, że program wyliczył pensje dla naszego pracownika sam, na podstawie podanych danych w konstruktorze, a następnie została zrealizowana wypłata na konto pracownika.

**Zadanie: Tworzenie obiektów z wykorzystaniem klasy abstrakcyjnej**

Wykorzystując kod stworzony przez nas w tym submodule: <https://kodilla.com/pl/project-java/38595>, utwórz nowe obiekty pracowników:

* pracownika zarabiającego 5 zł za stworzenie jednego produktu,
* pracownika, który zarabia pensję podstawową 2000 zł i 20% tej pensji, jeżeli zrealizuje dodatkową sprzedaż dowolnego produktu.

**Podpowiedź:** w tym zadaniu będziesz tworzyć tylko obiekty na podstawie naszego kodu, który używa klas abstrakcyjnych. Nie musisz tworzyć nowych klas abstrakcyjnych.

[**Podgląd zadania**](https://kodilla.com/pl/bootcamp-task/310/476573)[**Przejdź do projektu**](https://kodilla.com/pl/project-java/162054)

**1.6. Klasa Object**

Żeby w pełni zrozumieć, jak działają obiekty w Javie, należy pojąć, czym i po co została stworzona i udostępniona klasa o nazwie Object. Klasa Object jest klasą-matką, po której w niewidzialny sposób dziedziczy każda istniejąca klasa. Object udostępnia każdej klasie kilka bardzo ważnych i przydatnych metod, które są już napisane przez twórców Javy, tzn. są w nią wbudowane i gotowe do wykorzystania.

Oto kilka istniejących metod w klasie Object:

* hashCode() – metoda zwraca liczbę całkowitą, która jest przypisana dla tego konkretnego obiektu, jednakże liczba ta może się powtórzyć. Wynika to z tego, że HashCode jest liczony na podstawie wzoru matematycznego, który nie musi generować unikalnych liczb dla każdego obiektu. hashCode jest swojego rodzaju liczbową reprezentacją klasy – program nadaje mu swoją nazwę w postaci liczby, ale sama liczba nie musi być unikalna. W module 2 poznasz realne zastosowanie tej liczby. Jej główną zaletą jest fakt, że umożliwia grupowanie klas w kolekcjach – o nich i o tej metodzie będziemy jeszcze wspominać w trakcie zdobywania wiedzy o kolekcjach.
* equals() – metoda sprawdza, czy jakiś obiekt jest równy drugiemu obiektowi. Przypomnij sobie przyrównanie == – w Javie każdy obiekt ma taką metodę w sobie!

Po co właściwie są te metody? Otóż Java potrzebuje ich do wykonywania niezbędnego zarządzania pamięcią Twojego programu w trakcie jego działania i do innego rodzaju zadań, które będziemy zgłębiać z czasem. Ty też będziesz potrzebować ich w trakcie trwania tego kursu i przyszłej pracy jako programista.

Dziedziczenie po klasie Object jest niewidoczne i nie musimy o nim nigdy pamiętać – dzieje się to automatycznie. Każda nowo utworzona klasa dziedziczy metody z klasy Object.

Co z tego wynika? Za każdym razem, jeśli chcesz skorzystać z metod equals() lub hashCode(), możesz to robić bez jawnego ich deklarowania. Przyda Ci się to szybciej niż myślisz, bo już w drugim module bootcampa :)

**Zadanie: Użycie metody hashCode()**

Przeanalizuj poniższy kod: <https://kodilla.com/pl/project-java/38596> . Zobaczysz w nim wywołanie metody hashCode() na obiekcie SampleObject, który przedstawia przykładowy obiekt. Jak widzisz, nie dziedziczy on żadnej innej klasy, a jednak posiada metodę hashCode(). Stwórz swoją własną klasę i obiekt, a następnie wywołaj na nim metodę hashCode().