Java

Tartalom

[Felépítés 5](#_Toc132066104)

[Kiíratás 5](#_Toc132066105)

[Változók 5](#_Toc132066106)

[Operátorok 6](#_Toc132066107)

[Feltételes utasítások 7](#_Toc132066108)

[Keywords 7](#_Toc132066109)

[Metódusok 8](#_Toc132066110)

[Switch 8](#_Toc132066111)

[For ciklus 8](#_Toc132066112)

[While és do while ciklus: 9](#_Toc132066113)

[Beolvasás billentyűzetről 10](#_Toc132066114)

[OOP 11](#_Toc132066115)

[Belső változók 11](#_Toc132066116)

[Getters and Setters 11](#_Toc132066117)

[Constructors 11](#_Toc132066118)

[Meghívások 11](#_Toc132066119)

[Öröklődés (inheritance) 12](#_Toc132066120)

[Reference vs Object vs Instance vs Class 12](#_Toc132066121)

[Static metódus 13](#_Toc132066122)

[Composition 13](#_Toc132066123)

[Encapsulation 13](#_Toc132066124)

[Polymorphism 14](#_Toc132066125)

[Tömbök 16](#_Toc132066126)

[Szöveges tömbök 17](#_Toc132066127)

[Kétdimenziós tömbök 17](#_Toc132066128)

[Tömblisták (ArrayList) 18](#_Toc132066129)

[Lista készítése a String beépített típusból 18](#_Toc132066130)

[LinkedList 21](#_Toc132066131)

[Iteratorok 22](#_Toc132066132)

[Autoboxing and Unboxing 24](#_Toc132066133)

[Enumeration 25](#_Toc132066134)

[Absztrakció a Javaban 26](#_Toc132066135)

[Generics 32](#_Toc132066136)

[Naming Conventions 32](#_Toc132066137)

[Packages 32](#_Toc132066138)

[Scope 33](#_Toc132066139)

[Access Modifiers 33](#_Toc132066140)

[The static statement 33](#_Toc132066141)

[The final statement 33](#_Toc132066142)

[Java Collections 34](#_Toc132066143)

[Bináris keresés 35](#_Toc132066144)

[Sort rendezés 35](#_Toc132066145)

[Comparable and Comparator 35](#_Toc132066146)

[Az inner és nested, anonim class 36](#_Toc132066147)

[Map 36](#_Toc132066148)

[Immutable classes 37](#_Toc132066149)

[Set és HashSet 37](#_Toc132066150)

[Equals() és hashCode() 37](#_Toc132066151)

[Unió 38](#_Toc132066152)

[Metszet 38](#_Toc132066153)

[Különbség 38](#_Toc132066154)

[LinkedHashMap 39](#_Toc132066155)

[TreeMap 39](#_Toc132066156)

[UnmodifiableMap 39](#_Toc132066157)

[Streams 40](#_Toc132066158)

[JavaFX 42](#_Toc132066159)

[Elemei 42](#_Toc132066160)

[.iml 42](#_Toc132066161)

[FXML 42](#_Toc132066162)

[Controller.java 42](#_Toc132066163)

[Start metódus fxml nélkül 43](#_Toc132066164)

[Új JavaFX készítése 43](#_Toc132066165)

[Layouts 43](#_Toc132066166)

[GridPane Layout 43](#_Toc132066167)

[HBox Layout 44](#_Toc132066168)

[BorderPane Layout 44](#_Toc132066169)

[AnchorPane 45](#_Toc132066170)

[FlowPane 45](#_Toc132066171)

[TilPane 45](#_Toc132066172)

[StackPane 45](#_Toc132066173)

[Controls 46](#_Toc132066174)

[Imports 46](#_Toc132066175)

[RadioButton and CheckBox 47](#_Toc132066176)

[TextField, PasswordField, ComboBox and ChoiceBox 47](#_Toc132066177)

[Slider, Spiner, ColorPicker and DatePicker Controls 47](#_Toc132066178)

[Titled Pane 48](#_Toc132066179)

[Events and Event Handlers 48](#_Toc132066180)

[Sample.fxml 48](#_Toc132066181)

[Controller.java 48](#_Toc132066182)

[UI Thread, Runnable 49](#_Toc132066183)

[Singleton 50](#_Toc132066184)

[Kivételkezelés 51](#_Toc132066185)

[Több kivétel kezelése 51](#_Toc132066186)

[I/O 52](#_Toc132066187)

[FileWriter class and Finally block 52](#_Toc132066188)

[Throw 52](#_Toc132066189)

[Reader 53](#_Toc132066190)

[Buffered reader 53](#_Toc132066191)

[Bufferred Writer 54](#_Toc132066192)

[Byte Streams 54](#_Toc132066193)

[Object I/O including Serialization 56](#_Toc132066194)

[Random Access File 57](#_Toc132066195)

[Java NIO 60](#_Toc132066196)

# Felépítés

package academy.learnprogrammig;  
  
public class Main {  
 public static void main(String[] args) {

}  
}

# Kiíratás

System.*out*.println("Hello world!");

Egy szöveg kiíratása meghatározott mennyiségben.

System.*out*.println("\_".repeat(20));

# Változók

int myFirstNumber = 10 + 5;  
System.*out*.println(myFirstNumber);

Integer (32bit, -2147483648, 2147483647):

int myIntMinValue = Integer.*MIN\_VALUE*;  
int myIntMaxValue = Integer.*MAX\_VALUE*;  
System.*out*.println("Integer Minimum Value = " + myIntMinValue);  
System.*out*.println("Integer Maximum Value = " + myIntMaxValue);  
System.*out*.println("Busted Max value = " + (myIntMaxValue + 1));  
System.*out*.println("Busted Min value = " + (myIntMinValue -1));  
  
int myMaxIntTest = 2\_147\_483\_647;  
System.*out*.println(myMaxIntTest);

Byte (8bit, -128, 127):

byte myByteMinValue = Byte.*MIN\_VALUE*;  
byte myByteMaxValue = Byte.*MAX\_VALUE*;  
System.*out*.println("Byte Minimum Value = " + myByteMinValue);  
System.*out*.println("Byte Maximum Value = " + myByteMaxValue);

Short (16bit, -32768, 32767):

short myShortMinValue = Short.*MIN\_VALUE*;  
short myShortMaxValue = Short.*MAX\_VALUE*;  
System.*out*.println("Short Minimum Value = " + myShortMinValue);  
System.*out*.println("Short Maximum Value = " + myShortMaxValue);

Long (32bit, -9223372036854775808, 9223372036854775807):

long myLongValue = 100L;  
System.*out*.println(myLongValue);

long myLongMinValue = Long.*MIN\_VALUE*;  
long myLongMaxValue = Long.*MAX\_VALUE*;  
System.*out*.println("Long Minimum Value = " + myLongMinValue);  
System.*out*.println("Long Maximum Value = " + myLongMaxValue);

Típuskonverzió (casting):

byte byteValue = 10;  
short shortValue = 20;  
int intValue = 50;  
  
long longTotal = 50000L + 10L \* (byteValue + shortValue + intValue);  
System.*out*.println(longTotal);

short shortTotal = (short) (1000 + 10 \* (byteValue + shortValue + intValue));

Float (32bit, 1.4E-45 to 3,4E+38)

Double (64bit, 4.9E-324 to 1,79E+308)

Alapértelmezett típusok: integer, double.

int myIntValue = 5;  
float myFloatValue = 5.25f;  
double myDoubleValue = 5d;

Char (<https://unicode-table.com/>):

char myChar = 'D';  
char myUnicodeChar = '\u0044';

Boolean

boolean myTrueBooleanValue = true;  
boolean myFalseBooleanValue = false;

String (2,4 milliárd karakter max.)

String myString = "This is a string";  
System.*out*.println("myString is equal to: " + myString);

String lastString = "10";  
int myInt = 50;  
lastString = lastString + myInt;  
System.*out*.println("LastString is equal to " + lastString);

# Operátorok

% - modulus

result--;

result /= 3;

# Feltételes utasítások

int topScore = 80;  
if (topScore < 100) {  
 System.*out*.println("You got the high score!");  
}  
  
int secondTopScore = 60;  
if ((topScore > secondTopScore) && (topScore < 100)) {  
 System.*out*.println("Greater than second top score and less than 100");  
}

boolean isCar = false;  
if (!isCar) {  
 System.*out*.println("This is not supposed to happen!");  
}

Ternary operator (hármas operátor):

int ageOfClient = 20;  
boolean isEighteenOrOver = (ageOfClient == 20) ? true : false;  
System.*out*.println(isEighteenOrOver);

<https://docs.oracle.com/javase/tutorial/java/nutsandbolts/opsummary.html>

<https://www.cs.bilkent.edu.tr/~guvenir/courses/CS101/op_precedence.html>

# Keywords

<https://en.wikipedia.org/wiki/List_of_Java_keywords>

Expression

int myVariable = 50;

Statement

int myVariable = 50;

Blokk

A blokkban létrehozott változó blokkon kívül nem létezik.

# Metódusok

*calculateScore*(true, 10000, 8, 200);

public static void calculateScore(boolean gameOver, int score, int levelCompleted, int bonus)

int highScore = *calculateScore*(true, 800, 5,100);  
System.*out*.println("Your final score was " + highScore);

public static int calculateScore(boolean gameOver, int score, int levelCompleted, int bonus)

# Switch

String month = "JANUARy";  
switch (month.toLowerCase()) {  
 case "january":  
 System.*out*.println("Jan");  
 break;  
 case "june":  
 System.*out*.println("Jun");  
 break;  
 default:  
 System.*out*.println("Not sure");  
 break;  
}

# For ciklus

for(int i=2; i<=(long)Math.*sqrt*(n); i++) {  
 if (n % i == 0) {  
 return false;  
 }  
}

# While és do while ciklus:

do {  
 System.*out*.println("Count value is " + count);  
 count++;  
} while (count != 6);

while (number <= finishNumber) {  
 number++;  
 if (!*isEvenNumber*(number)) {  
 continue;  
 }  
 evenNumbersFound++;  
 if (evenNumbersFound >= 5) {  
 break;  
 }  
 System.*out*.println("Even number " + number);  
}

# Beolvasás billentyűzetről

Scanner scanner = new Scanner(System.*in*);  
  
System.*out*.println("Enter your year of birth:");  
  
boolean hasNextInt = scanner.hasNextInt();  
if (hasNextInt) {  
 int yearOfBirth = scanner.nextInt();  
 scanner.nextLine();  
  
 System.*out*.println("Enter your name:");  
 String name = scanner.nextLine();  
 int age = 2022 - yearOfBirth;  
  
 if (age >= 0 && age <= 100) {  
 System.*out*.println("Your name is " + name + " and you are " + age + " years old.");  
 } else {  
 System.*out*.println("Invalid year of birth.");  
 }  
} else {  
 System.*out*.println("Unable to parse year of birth.");  
}  
  
scanner.close();

Scanner scanner = new Scanner(System.*in*);  
  
 int counter = 0;  
 int sum = 0;  
  
 while (true) {  
 int order = counter +1;  
 System.*out*.println("Enter number #" + order + ":");  
  
 boolean isAnInt = scanner.hasNextInt();  
  
 if (isAnInt) {  
 int number = scanner.nextInt();  
 counter++;  
 sum += number;  
 if (counter == 10) {  
 break;  
 }  
 } else {  
 System.*out*.println("Invalid number");  
 }  
 System.*out*.println("sum =" + sum);  
 scanner.nextLine();  
 }  
  
 scanner.close();  
}

Többsoros string kiíratáshoz:

String textBlock = """

”””;

# OOP

Az osztályok neve nagy betűvel kezdődhet.

## Belső változók

private String number;  
private double balance;

## Getters and Setters

Code/Generate/ Getters and Setters

public double getFirstNumber() {  
 return this.firstNumber;  
}

public void setFirstNumber(double firstNumber) {  
 this.firstNumber = firstNumber;  
}

## Constructors

Code/Generate/ Getters and Setters

public VipPerson() {  
 this("Default name", 50000.00, "deafult@email.com");  
}  
  
public VipPerson(String name, double creditLimit) {  
 this(name, creditLimit, "unknown@email.com");  
}  
  
public VipPerson(String name, double creditLimit, String emailAddress) {  
 this.name = name;  
 this.creditLimit = creditLimit;  
 this.emailAddress = emailAddress;  
}

## Meghívások

VipPerson person1 = new VipPerson();  
  
VipPerson person2 = new VipPerson("Bob", 25000.00);  
  
VipPerson person3 = new VipPerson("Tim", 100,"tim@email.com");

A metódusok static nélkül vannak.

## Öröklődés (inheritance)

Akkor használom, ha van egy szülő osztály, aminek vannak gyerek osztályai. A szülő osztály változói használhatók a gyerek osztályokban, azoknak további változóik vannak, amelyek speciálisak és a szülő osztályra nem alkalmazható, pl. Állatok – Halak, Autó – Ford. A gyermekosztályoknál az extends szó után jelöljük a szülő osztályt.

public Fish(String name, int size, int weight,  
 int gills, int eyes, int fins) {  
 super(name, 1, 1, size, weight);  
 this.gills = gills;  
 this.eyes = eyes;  
 this.fins = fins;  
}

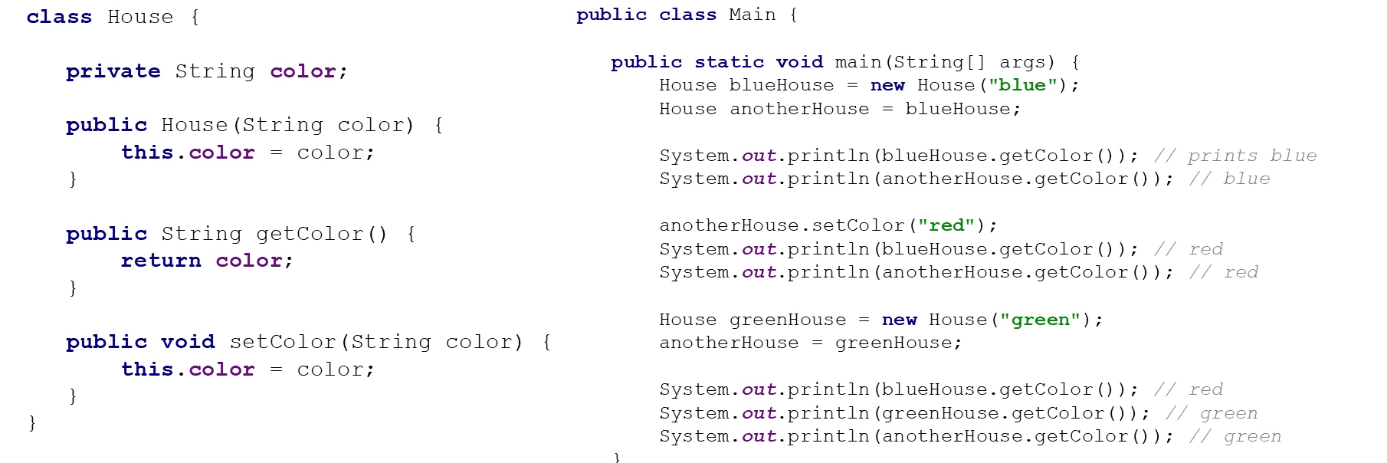
private void swim(int speed) {  
 moveMuscles();  
 moveBackFin();  
 super.move(speed);  
}

Code / Generate / Override methods

Az overloading kiegészíti, az overriding felülírja a szülő metódust.

@Override  
public void move(int speed) {  
 System.*out*.println("Dog.move() called.");  
 moveLegs();  
 super.move(speed);  
}

## Reference vs Object vs Instance vs Class



Az class-ban van az object és a method. A reference az objektumnak ugyanaz a megvalósulása más néven, együtt változnak, az instanse az objektumnak más megvalósulása, más néven.

A felülírt metódusnak a szülő osztályban kell lennie, publicon és a felülírt is public.

## Static metódus

Nem használhatom a this-t, instance változókat. Az objektumok példányaiban nem léteznek, az osztály nevével lehet elérni.

## Composition

A composition alkalmazása során van egy osztály, amelyhez olyan osztályok kapcsolódnak, melyek nem tartalmazzák az eredeti osztály változóit, mégis annak tartalmi részei. Pl. Hálószoba, ágy, lámpa. Nincs extends.

public Lamp getLamp() {  
 return this.lamp;  
}

public void turnOn() {  
 System.*out*.println("Lamp --> Turning on");  
}

Lamp lamp = new Lamp("Classic", false,75);  
Bedroom bedRoom = new Bedroom("Tims", wall1, wall2, wall3, wall4,  
 ceiling, bed, lamp);

bedRoom.getLamp().turnOn();

public void makeBed() {  
 System.*out*.println("Bedroom --> Making bed");  
 bed.make();  
}

public void make() {  
 System.*out*.println("Bed --> Making");  
}

Bedroom bedRoom = new Bedroom("Tims", wall1, wall2, wall3, wall4,  
 ceiling, bed, lamp);  
bedRoom.makeBed();

## Encapsulation

A célja, hogy elrejtsük a metódusok működését mások elől.

public int printPages(int pages) {  
 int pagesToPrint = pages;  
 if(this.duplex) {  
 pagesToPrint = (pages / 2) + (pages % 2);  
 System.*out*.println("Printing in duplex mode");  
 }  
 this.pagesPrinted += pagesToPrint;  
 return pagesToPrint;  
}  
  
public int getPagesPrinted() {  
 return pagesPrinted;  
}

Printer printer = new Printer(50, false);  
System.*out*.println("initial page count = " +printer.getPagesPrinted());  
int pagesPrinted = printer.printPages(4);  
System.*out*.println("Pages printed was " + pagesPrinted +" new total print count for printer = " +printer.getPagesPrinted());

## Polymorphism

A polimorfizmus többalakúságot jelet. A szülőosztálynak olyan gyerekosztályai vannak, amelyeknek pontosan ugyanolyan változói vannak, mint a neki. Pl. Film és a konkrét filmek. Van extends.

class Movie {  
 private String name;  
  
 public Movie(String name) {  
 this.name = name;  
 }  
  
 public String plot() {  
 return "No plot here";  
 }  
  
 public String getName() {  
 return name;  
 }  
}

class Jaws extends Movie {  
 public Jaws() {  
 super("Jaws");  
 }  
  
 public String plot() {  
 return "A shark eats lots of people";  
 }  
}

public static Movie randomMovie() {  
 int randomNumber = (int) (Math.*random*() \* 5) +1;  
 System.*out*.println("Random number generated was: " + randomNumber);  
 switch (randomNumber) {  
 case 1:  
 return new Jaws();  
 case 2:  
 return new IndependenceDay();  
 case 3:  
 return new MazeRunner();  
 case 4:  
 return new StarWars();  
 case 5:  
 return new Forgetable();  
 }  
  
 return null;  
}

Movie movie = *randomMovie*();  
System.*out*.println("Movie #" + i +  
 " : " + movie.getName() + "\n" +  
 "Plot: " + movie.plot() + "\n");

# Tömbök

int[] myIntArray = new int[10];  
myIntArray[0] = 45;

A deklaráció után a tömb elemszáma nem változhat meg.

int[] firstTen = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

int[] newArray;  
newArray = new int[] {5, 4, 3, 2, 1};

int arrayLength = firstTen.length;

for (int i = 0; i < newArray.length; i++) {  
 System.out.print(newArray[i] + " ");  
}

Enchanded for loop:

for (int element : newArray) {  
 System.out.print(element + " ");

System.out.println(Arrays.toString(newArray));

Object objectVariable = newArray;  
if (objectVariable instanceof int[]) { //JDK 17  
 System.out.println("objectVariable is really an int array");  
}

Object[] objectArray = new Object[3];  
objectArray[0] = "Hello"; //JDK 17  
objectArray[1] = new StringBuilder("World");  
objectArray[2] = newArray; //JDK 17

Utility a tömbhöz és a véletlenekhez

import java.util.Arrays;  
import java.util.Random;

Véletlenszám

Random random = new Random();

newInt[i] = random.nextInt(100);

Rendezés

Arrays.*sort*(firstArray);

Feltöltés értékkel

Arrays.*fill*(secondArray, 5);

Átmásolás

int[] smallerArray = Arrays.*copyOf*(thirdArray, 5);

A bináris keresésnél az elemek rendezettek kellenek, hogy legyenek. Ha két ugyanolyan elem van, nem lehet tudni melyiket találja meg. Ha megvan az elem, akkor az indexével tér vissza, ellenkező esetben -1-gyel.

Arrays.*sort*(sArray);

if (Arrays.*binarySearch*(sArray, "Mark") >= 0) {  
 System.*out*.println("Found Mark in the list");  
}

Két tömb összehasonlítása

if (Arrays.*equals*(s1, s2)) {  
 System.*out*.println("Arrays are equal");  
} else {  
 System.*out*.println("Arrays are not equal");  
}

Ha két másnéven szereplő tömböt egyenlővé teszünk, a másodikat nem új tömbként hozzuk létre, akkor az reference, együtt változnak.

A tömb indexét csökkenthetjük egyszerűbb módon egy ciklusban.

reversedArray[maxIndex--] = el;

## Szöveges tömbök

Tömb deklarálása, egy mondatot szedünk szét szavakra, amelyek a tömb elemei lesznek. Egymás alá íratjuk ki.

String[] splitStrings = "Hello World again".split(" ");

*printText*(splitStrings);

A tömb elemeit elválasztva íratjuk ki.

System.*out*.println(String.*join*(",", sArray));

Szöveg átalakítása egésszé, szóköz levágással.

values[i] = Integer.*parseInt*(splits[i].trim());

## Kétdimenziós tömbök

A kétdimenziós tömbökben különböző hosszúságú tömbök is lehetnek.

Deklarálása:

int[][] array2 = new int[4][4];

Ha így deklaráljuk, üres tömbök lesznek benne.

int[][] array2 = new int[4][];

Kétdimenziós tömb formázott kiíratása.

for (var outer : array2) {  
 for (var element : outer) {  
 System.out.print(element + " ");  
 }  
 System.out.println();  
 }

Egyszerűbb, de más kiíratás. Változóként viselkedik a tömb benne.

System.*out*.println(Arrays.*deepToString*(array2));

# Tömblisták (ArrayList)

A lista egy speciális típus a Javaban, amelyet Interface-nek neveznek.

import java.util.ArrayList;  
import java.util.Arrays;

A rekord egy adatleíró osztály, amelynek paraméterei vannak. Benne egyszerűsíthetjük a használatát. A rekordok egyszerű adatszerkezetek, amelyek csak adattagokat tartalmaznak, és nem tartalmaznak metódusokat. A rekordok automatikusan generálnak egy equals(), hashCode() és toString() metódust, így a programozónak nem kell kézzel implementálnia ezeket. A rekordok könnyen olvashatóak, és jelentősen csökkentik a kód redundanciáját.

record GroceryItem(String name, String type, int count) {  
  
 public GroceryItem(String name) {  
 this(name, "DAIRY", 1);  
 }  
}

A rekordban felülírhatom a toString() metódust, a szebb kiíratásért.

@Override  
public String toString() {  
 return String.*format*("%d %s in %s", count, name.toUpperCase(), type);  
}

Ez a tömblista bővülhet, nem fix a nagysága. <> Diamond operator, engedi az új megvalósulás (instance) deklarációját.

ArrayList<GroceryItem> groceryList = new ArrayList<>();  
groceryList.add(new GroceryItem("Butter"));

int elementCount = groceryList.size();

Egy elemet az általam kiválasztott helyre rakhatok, a többi arrébb tolódik.

groceryList.add(0,  
 new GroceryItem("apples", "PRODUCE", 6));

Kicserélhetek egy elemet.

groceryList.set(0,  
 new GroceryItem("apples", "PRODUCE", 6));

Törölhetek egy elemet.

groceryList.remove(1);

## Lista készítése a String beépített típusból

String[] items = {"apples", "bananas", "milk", "eggs"};  
List<String> list = List.*of*(items);

ArrayList<String> groceries = new ArrayList<>(list);

ArrayList<String> nextList = new ArrayList<>(  
 List.*of*("pickles", "mustard", "cheese"));

A listához hozzáfűzhetünk egy másikat.

groceries.addAll(nextList);

Index használata.

System.*out*.println("Third item = " + groceries.get(2));

Tartalmazza-e a listánk az értéket? Ha többet, akkor containsAll.

if (groceries.contains("mustard")) {  
 System.*out*.println("List contains mustard");  
}

Az elem első és utolsó előfordulása. Ha nincs ilyen, -1-gyel tér vissza.

System.*out*.println("first = " + groceries.indexOf("yogurt"));  
System.*out*.println("last = " + groceries.lastIndexOf("yogurt"));

Elem törlése.

groceries.remove(1);

groceries.remove("yogurt");

Törlés -tól -tólig.

groceries.removeAll(List.*of*("apples", "eggs"));

Törlés a felsoroltak kivételével.

groceries.retainAll(List.*of*("apples", "milk", "mustard",  
 "cheese"));

Minden törlése.

groceries.clear();

Üresség vizsgálata.

System.*out*.println("isEmpty = " + groceries.isEmpty());

Több elem hozzáadása.

groceries.addAll(List.*of*("apples", "milk", "mustard",  
 "cheese"));  
groceries.addAll(Arrays.*asList*("eggs", "pickles", "mustard", "ham"));

Növekvő és csökkenő sorrendbe rendezés.

groceries.sort(Comparator.*naturalOrder*());

groceries.sort(Comparator.*reverseOrder*());

Tömblista visszaalakítása tömbbé.

var groceryArray = groceries.toArray(new String[groceries.size()]);

Tömb alakítása tömblistává.

String[] originalArray = new String[] {"First", "Second", "Third"};  
var originalList = Arrays.*asList*(originalArray);

A képen asztal látható

Automatikusan generált leírás

Az ArrayList választása az alábbi esetekben jó: adattárolás és olvasás, ha előre tudjuk a maximális nagyságát, ami az Intereger.MAX\_VALUE lehet.

# LinkedList

Akkor érdemes használni, ha nagy mennyiségű adatot adok hozzá, feldolgozom, vagy manipulálom. A maximális elemszám nem ismert, de hatalmas, esetleg túllépi az Intereger.MAX\_VALUE-t. Akkor lehet hatékony, ha az adatok a fejtől a lábig vannak feldolgozva.

Használható sornak: FIFO (First In First Out) vagy veremnek: LIFO (Last In First Out). A dupla végű sornál az elejére és a végére is lehet rakni elemet. Tartalmazza az ArrayList műveleteit is.

Deklarálás.

var placesToVisit = new LinkedList<String>();

Elem hozzáadása.

placesToVisit.add("Sydney");  
placesToVisit.add(0, "Canberra");

list.addFirst("Darwin");  
list.addLast("Hobart");

Hozzáadás sorhoz. A sima hozzáadás az utolsó helyre történik.

list.offer("Melbourne");  
list.offerFirst("Brisbane");  
list.offerLast("Toowoomba");

Hozzáadás veremhez. Az első helyre történik.

list.push("Alice Springs");

Elem törlése. Az első elemet törli alapértelmezetten.

list.remove(4);  
list.remove("Brisbane");

list.remove();

list.removeFirst();

list.removeLast();

Elem törlése sorból. Alapértelmezetten az első elemet törli.

list.poll();

list.pollFirst();

list.pollLast();

Törlés veremből. Alapértelmezetten az első elemet törli.

list.pop();

Elemek megkapása az index alapján. Ez nem túl hatékony.

list.get(4);

list.getFirst();

list.getLast();

Index megkapása az elem alapján.

list.indexOf("Darwin"))

Sorral kapcsolatos műveletek. Ez az első elemet adja vissza.

list.element();

Veremmel kapcsolatos műveletek. Alapértelmezetten az első elemet adja vissza.

list.peek();

list.peekFirst();

list.peekLast();

Iterálás. Nem hatékony.

for (int i = 1; i < list.size(); i++) {  
 System.*out*.println("--> From: " + list.get(i - 1) + " to " + list.get(i));  
}

Iterálás másik módszerrel, nem túl hatékony.

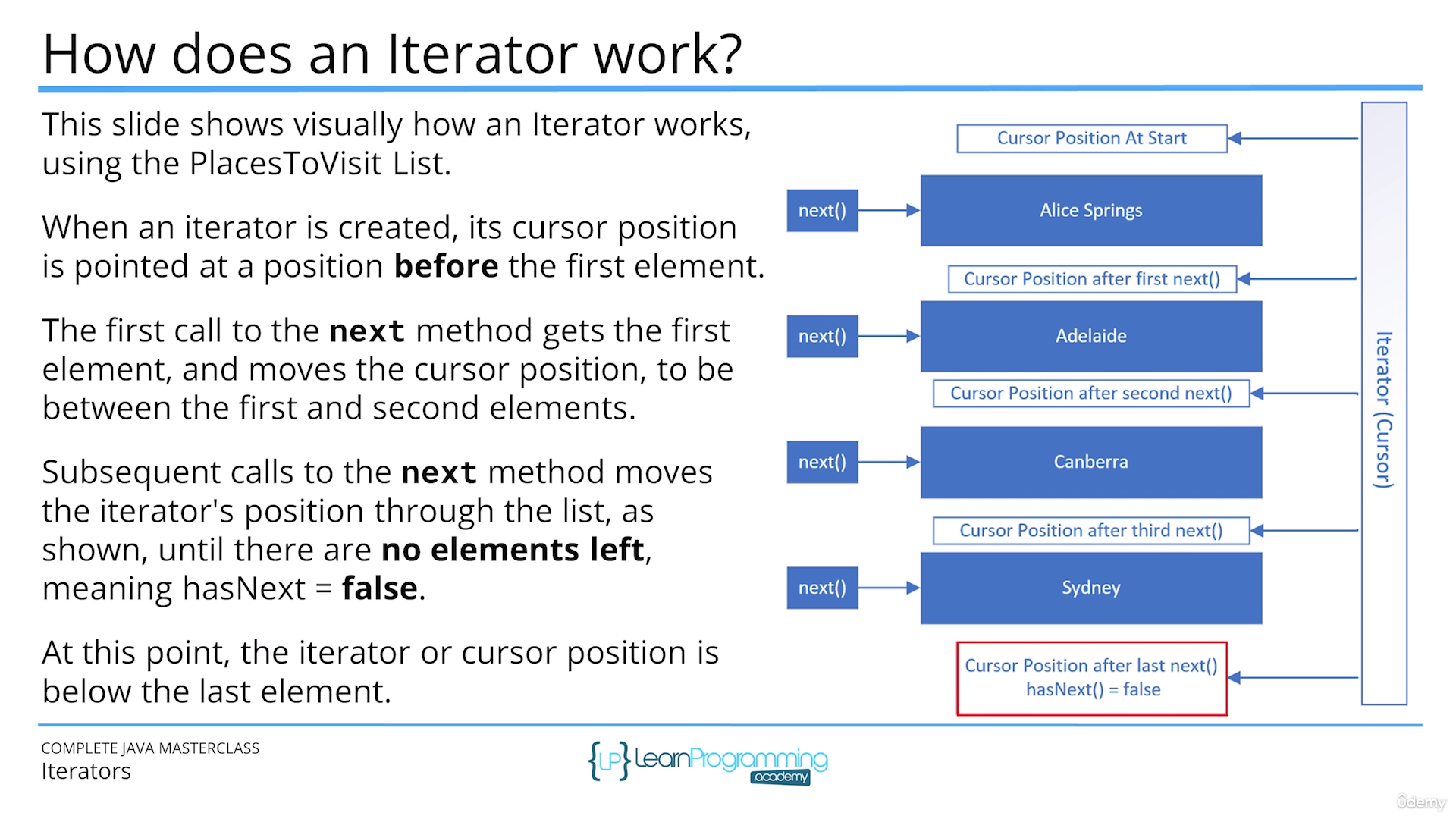
String previousTown = list.getFirst();  
for (String town : list) {  
 System.*out*.println("--> From: " + previousTown + " to " + town);  
 previousTown = town;  
}

Iterálás harmadik módszerrel. Hatékony.

ListIterator<String> iterator = list.listIterator(1);  
while (iterator.hasNext()) {  
 var town = iterator.next();  
 System.*out*.println("--> From: " + previousTown + " to " + town);  
 previousTown = town;  
}

## Iteratorok

Az iterator egy adatbázis kurzor.



Igaz vagy hamis értéket ad arra, hogy van-e előző vagy következő elem.

iterator.hasNext()

iterator.hasPrevious()

Ugrás az előző vagy a következő elemre.

iterator.next();

iterator.previous();

Az iterator csak előre megy és a remove metódust támogatja. A ListIterator előre és hátra is megy és a remove, add, set metódusokat támogatja.

Egy példa:

var iterator = list.listIterator();  
 while (iterator.hasNext()) {  
// System.out.println(iterator.next());  
 if (iterator.next().equals("Brisbane")) {  
 iterator.add("Lake Wivenhoe");  
 }  
 }

Substring: a megjelölt résszel csinál valamit, jelen esetben nagybetűt.

String menuItem = scanner.nextLine().toUpperCase().substring(0, 1);

# Autoboxing and Unboxing

Az ArrayList és a LinkedList nem használhat primitív típusokat. A Java minden primitív típus mellé ad egy becsomagolt osztályt (wrapped class).

A becsomagolás és kicsomagolás kézi esetei:

Integer boxedInt = Integer.*valueOf*(15); // preferred but unnecessary  
Integer deprecatedBoxing = new Integer(15); // deprecated since JDK 9  
int unboxedInt = boxedInt.intValue(); // unnecessary

A Javaban van lehetőség automatikus be- és kicsomagolásra.

Integer autoBoxed = 15;  
int autoUnboxed = autoBoxed;

Az alábbi esetben a metódus primitív típussal tér vissza, de a Java automatikusan becsomagolja, mert a java.lang.Double osztály egy változójához van csatolva.

Double resultBoxed = *getLiteralDoublePrimitive*();

private static double getLiteralDoublePrimitive() {  
  
 return 100.0;  
}

A fordítottja is igaz, ez a metódus egy objektum megvalósulásával tér vissza, amit egy primitív változóhoz csatoltunk, az Unboxing automatikusan megvalósul.

double resultUnboxed = *getDoubleObject*();

private static Double getDoubleObject() {  
  
 return Double.*valueOf*(100.00);  
}

A becsomagolás egyéb lehetőségei:

Integer[] wrapperArray = new Integer[5];

Character[] characterArray = {'a', 'b', 'c', 'd'};

var ourList = List.*of*(1, 2, 3, 4, 5);  
System.*out*.println(ourList);

A varargs használata leegyszerűsíti a kódot, mivel több argumentum átadásának szükségességét egyetlen varargs paraméterrel lehet megoldani, és nem kell külön kódot írni minden argumentum számára. A varargs csak utolsó paraméterként lehet megadni, és csak egy varargs paraméter lehet egy metódusban.

var ourList = getList(…varags: 1, 2, 3, 4, 5);  
System.*out*.println(ourList);

private static ArrayList<Integer> getList(Integer... varargs) {  
  
 ArrayList<Integer> aList = new ArrayList<>();  
 for (int i : varargs) {  
 aList.add(i);  
 }  
 return aList;  
}

# Enumeration

Az enum típus olyan, mint egy tömb, de az elemei nem változtathatóak meg, és osztályként hozhatjuk létre. Rendezett. Korlátozott konstans számmal rendelkezik. Csupa nagybetűvel kell írni.

public enum DayOfTheWeek {  
  
 *SUN*, *MON*, *TUES*, *WED*, *THURS*, *FRI*, *SAT*}

DayOfTheWeek weekDay = DayOfTheWeek.*TUES*;

System.out.printf("Name is %s, Ordinal Value = %d%n",  
 weekDay.name(), weekDay.ordinal());

public static void switchDayOfWeek(DayOfTheWeek weekDay) {  
  
 int weekDayInteger = weekDay.ordinal() + 1;  
 switch (weekDay) {  
 case *WED* -> System.*out*.println("Wednesday is Day " + weekDayInteger);  
 case *SAT* -> System.*out*.println("Saturday is Day " + weekDayInteger);  
 default -> System.*out*.println(weekDay.name().charAt(0) +  
 weekDay.name().substring(1).toLowerCase() +  
 "day is Day " + weekDayInteger);  
 }  
}

for (Topping topping : Topping.*values*()) {  
 System.*out*.println(topping.name() + " : " + topping.getPrice());  
}

public double getPrice() {  
  
 return switch (this) {  
 case *BACON* -> 1.5;  
 case *CHEDDAR* -> 1.0;  
 default -> 0.0;  
 };  
}

# Absztrakció a Javaban

Private: csak azon osztályok számára láthatóak, amelyek ugyanabban az osztályban vannak deklarálva.

Protected: a leszármazottak is láthatják.

Public: bárhonnan elérhető.

Published: olyan, mint a public, de minden taghoz generálódik egy RTTI infó, azért például, hogy bizonyos metódusokat bizonyos property-khez rendeljük.

Automated: olyan, mint a public, kompatibilitási okok miatt van WinAPI részére.

Abstract nem párosítható össze a private-tal. Az abstract osztály nem teljes, nem lehet létrehozni egy megvalósulását, nem konkrét. Lehet egy konkrét osztály kiterjesztése, lehet egy abstract osztályt kiterjeszteni egy konkréttal. Az abstract metódusok csak a deklarácót tartalmazzák, a konkrét osztályban felül lesz írva.

public abstract class Animal {  
  
 protected String type;  
 private String size;  
 private double weight;  
  
 public Animal(String type, String size, double weight) {  
 this.type = type;  
 this.size = size;  
 this.weight = weight;  
 }  
  
 public abstract void move(String speed);  
 public abstract void makeNoise();  
  
 public final String getExplicitType() {  
 return getClass().getSimpleName() + " (" + type + ")";  
 }  
}

public class Dog extends Mammal {  
  
 public Dog(String type, String size, double weight) {  
 super(type, size, weight);  
 }  
  
 @Override  
 public void move(String speed) {

}

}

public static void main(String[] args) {  
  
// Animal animal = new Animal("animal", "big", 100);  
  
 Dog dog = new Dog("Wolf", "big", 100 );  
 dog.makeNoise();  
 *doAnimalStuff*(dog);

}

private static void doAnimalStuff(Animal animal) {  
  
 animal.makeNoise();  
 animal.move("slow");  
}

ArrayList<Animal> animals = new ArrayList<>();  
animals.add(dog);  
animals.add(new Dog("German Shepard", "big", 150));  
animals.add(new Fish("Goldfish", "small", 1));

Az alosztályokat írja ki:

getClass().getSimpleName();

Final metódus: nem írható felül.

Ha egy abstract osztálynak egy másik abstract kiterjesztése van, akkor

* megvalósíthatja a szülő osztály összes vagy néhány abstract metódusait,
* vagy egyet sem tartalmaz,
* egyéb abstract metódusokat tartalmazhat.

for (Animal animal : animals) {  
 *doAnimalStuff*(animal);  
 if (animal instanceof Mammal currentMammal) {  
 currentMammal.shedHair();  
 }  
}

Az interface hasonlít az abstract osztályhoz, mégsem osztály, hanem speciális típus. A deklarációban a class szó helyett interface-t használunk. Egy osztály társulhat egy interface-hez:

interface FlightEnabled {  
  
 double *MILES\_TO\_KM* = 1.60934;  
 double *KM\_TO\_MILES* = 0.621371;  
  
 void takeOff();  
 void land();  
 void fly();  
  
}

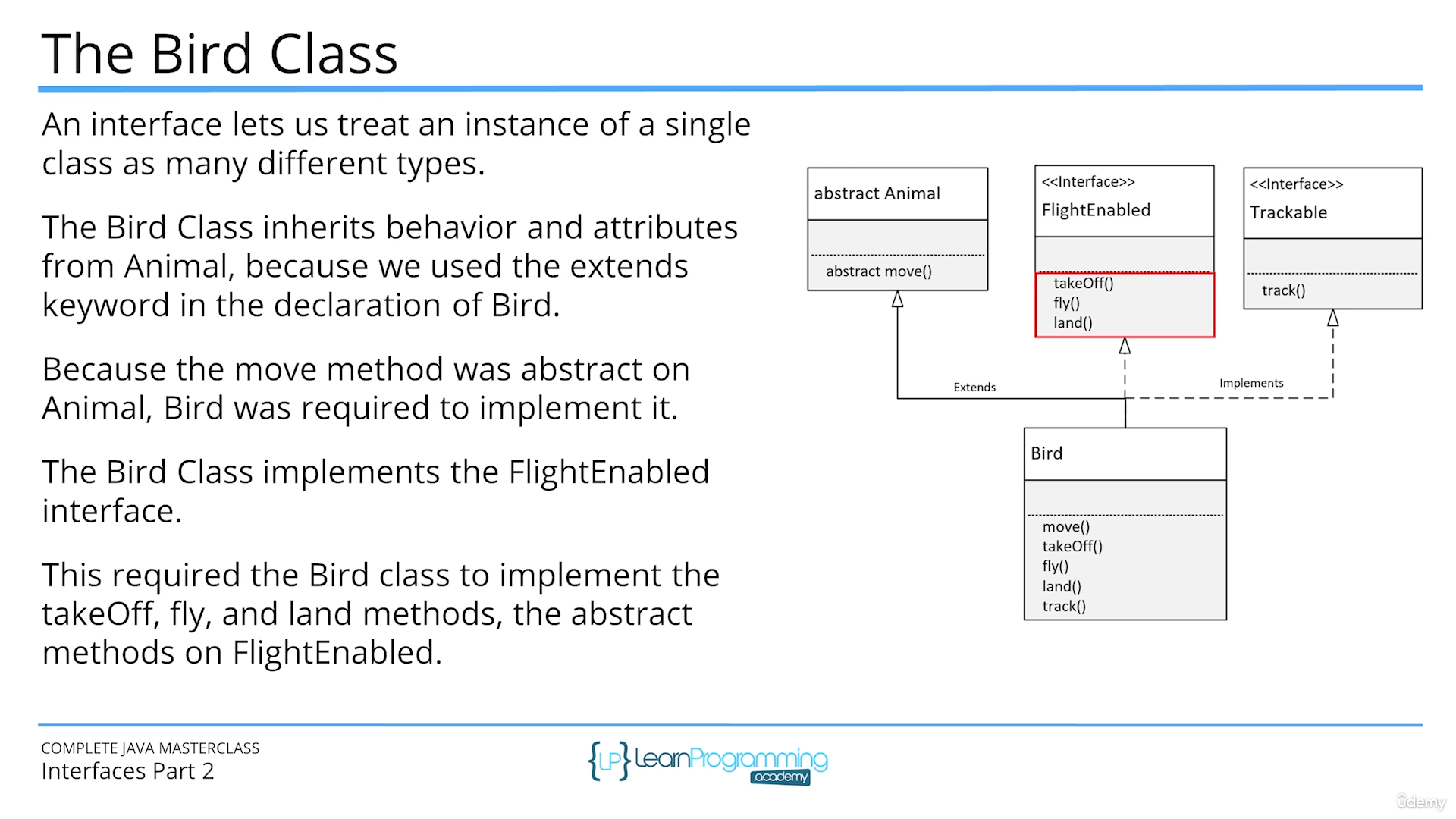
public class Bird implements FlightEnabled {

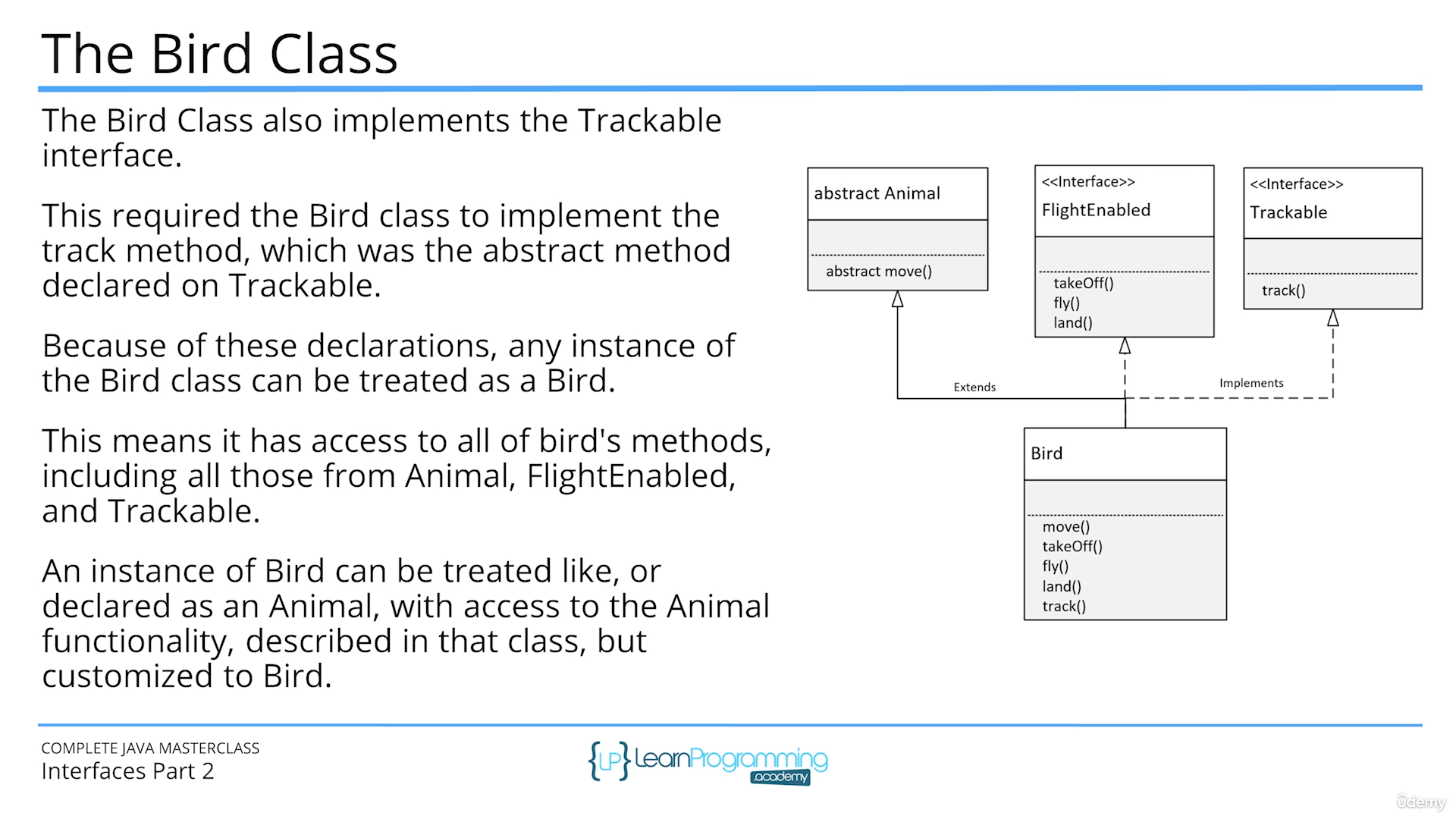
}

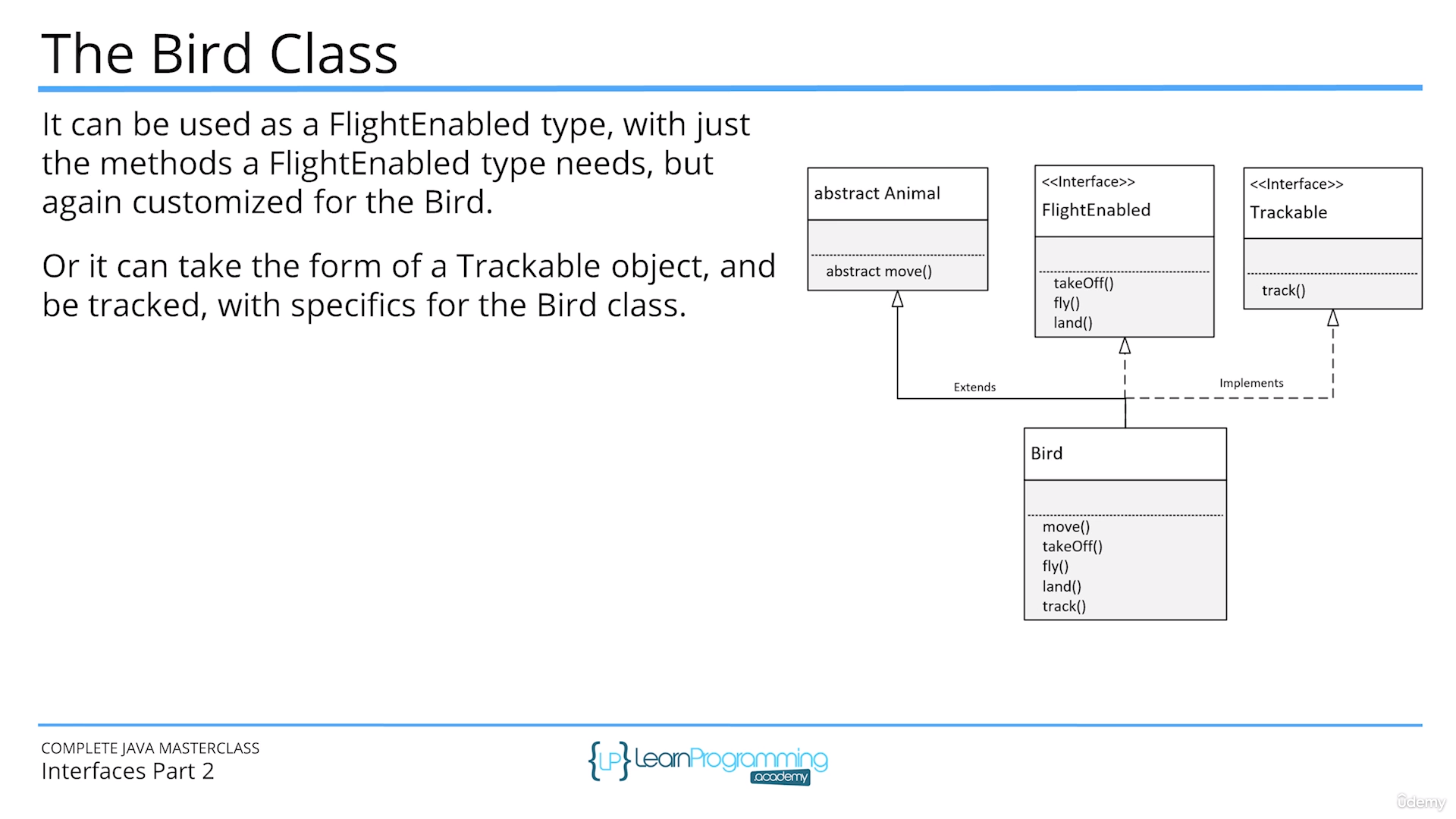
FlightEnabled flier = new Bird();

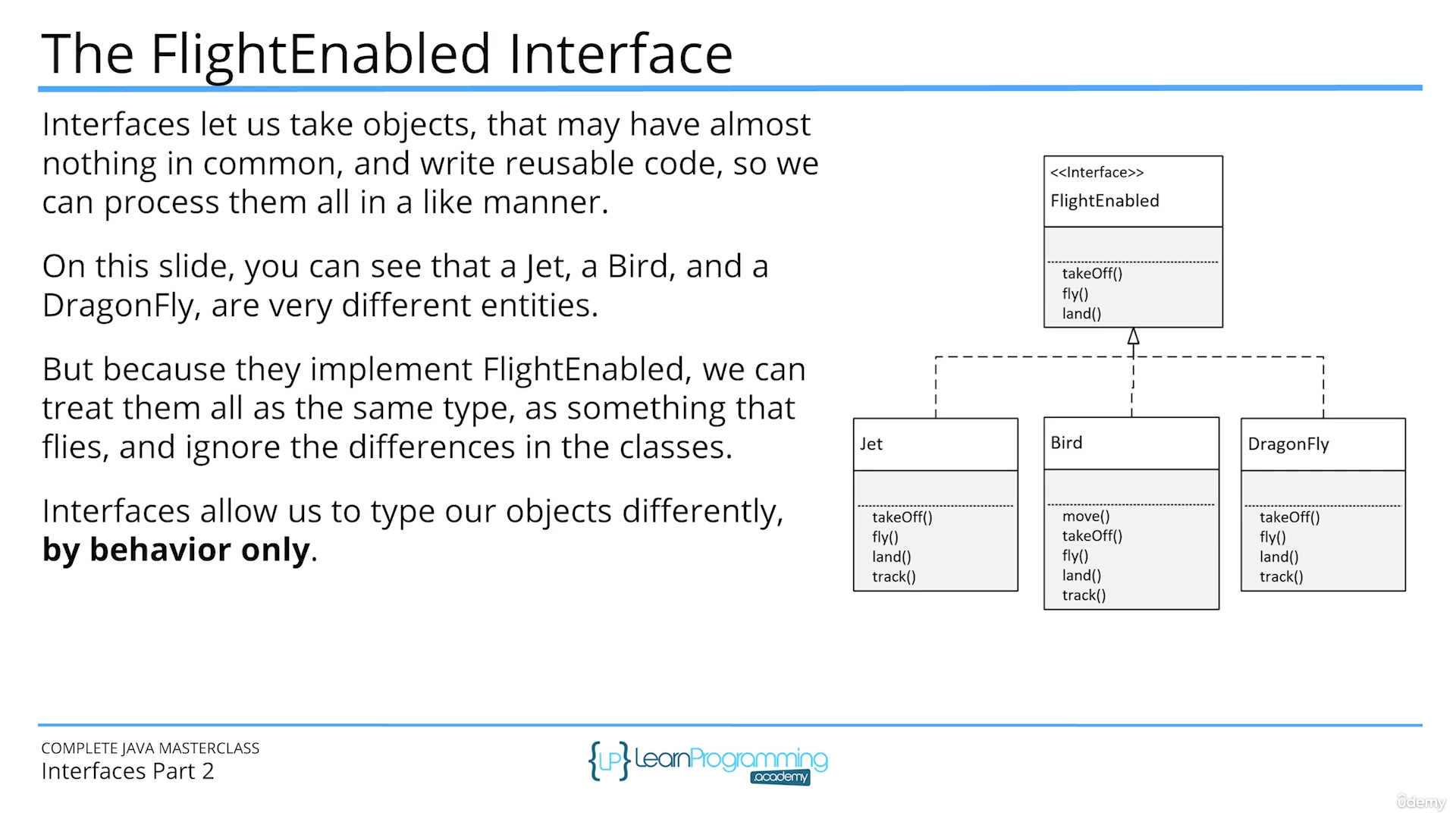
Bird bird = new Bird();  
Animal animal = bird;  
FlightEnabled flier = bird;  
Trackable tracked = bird;

Az interface deklarációjánál szükségtelen abstract-ot vagy public-ot használni, eleve olyan, a protected-re hibaüzenetet kapunk. Az osztály alapértelmezetten private.









A konstans public, static, final.

Az interface-k az egyszerű osztályokhoz hasonlóan lehetnek extends szóval hivatkozva. Az interface használhat kibővített kifejezést és többszörös interface-eket. Egy interface nem implementálhat másik interface-t.

JDK8, metódus felülírása egy interface-en:

* Nem írjuk felül.
* Ha felülírjuk, az interface metódus nem fut.
* Megírom a saját kódomat, felidézem az interface metódusát, mint a megvalósítás részét.

Mai dátum:

var today = new java.util.Date();

Az abstract osztályoknál nem használunk static-ot és final-t. Használhatok 4 belépési módosítót, a konkrét metódusokért. Használhatunk privátot az abstract metódusokért. Az absract class csak egy szülő osztályt tud kiegészíteni, de többszörös interface-eket tud megvalósítani. Amikor alosztályai vannak, az alosztály általában az összes abstract metódus megvalósítását nyújtja a szülőosztályában. Az alosztály lehet abstract-nak deklarálva.

Az abstract osztály akkor használom, ha

* kódot akarok megosztani néhány hasonló megvalósítású osztály között,
* kiegészítek vele osztályokat metódusokért, vagy más belépési módosító kell a publicon kívül,
* nem statikus és nem final fieldeket akarok deklarálni, lehet metódusokat definiálni, amelyek be tudnak lépni és objektumok állapotát módosítani.
* elvárod az alaposztálytól a fő metódusok alapértelmezett megvalósításának megosztását, de más metódusoknak felülírhatóaknak kell lenniük a gyerekosztály által.

Az abstract osztály egy általános definíciót nyújt, mint egy alap osztály, amit a többszörös, örökölt osztályok megoszthatnak.

Az interface csak osztályok deklarációja, amit szeretnék, ha meglenne bizonyos osztályoknak, de nem megvalósítás. Definiáljuk, hogy az objektumnak milyen műveletei vannak, ezeket azok az osztályok definiálják, amelyek megvalósítják. Ez egy egyezség két osztály között és a külső világgal, ami a fordítás során valósul meg. Nem tudod bemutatni az interface-t, de megvalósítással vagy anélküli metódusokat tartalmaz. A metódusok törzs nélküliek, automatikusan public és static állapotúak. Ki lehetnek egészítve más interface-szel. Flexibilisek, kevesebb a stressz velük, mert nem részei az osztály hierarchiának. Tartalmaznak default metódusokat megvalósítással a visszamenő kompatibilitás miatt. Public static metódus a Java 8-ban volt bevezetve. Java 9-ig csak private metódust tartalmaztak, az alapértelmezett metódusok gyakran használt kódokat osztottak meg.

Az interface-t akkor használom, ha:

* Sok, nem összefüggő osztály által tudom megvalósítani.
* Amikor specifikálom egy adattípus viselkedését, de mindegy mi valósítja meg.
* El akarok különíteni különféle viselkedéseket.
* Collections Framework: List és Quene, a megvalósításuk, ArrayList és LinkedList.
* A lambda expressions alapja, pl.
* JBDC

A képen asztal látható

Automatikusan generált leírás

# Generics

A Java generics olyan típusos adatstruktúrák, amelyek tetszőleges típusú elemeket tartalmazhatnak. A generics használatával lehetővé válik a kód egységesítése, mert ugyanaz a kódblokk tetszőleges típusú adatokat kezelhet. A generics használata lehetővé teszi a kód futási időben való ellenőrzését, hogy biztosítsuk, hogy csak a megfelelő típusú adatok kerülnek az adatstruktúrába.

public class Team<T extends Player> implements Comparable<Team<T>>

Ez egy Java osztály, amely Team néven van meghatározva. Az osztály használja a generikus típusokat (T extends Player), amely azt jelenti, hogy a T típus csak olyan osztályokat lehet, amelyek Player osztályból származnak, vagy Player osztályból származó osztályokat tartalmaznak.

A Comparable az egyik alapértelmezett Java interface, amely segítséget nyújt a típusok összehasonlításához. Az osztály, amely implementálja ezt az interfészt, megvalósítja a compareTo metódust, amely lehetővé teszi, hogy az osztályok példányait összehasonlítsák egymással. Az interfész használható generikus típusként, amely lehetővé teszi, hogy csak olyan típusokat hasonlítson össze, amelyek megvalósítják az interfészt.

# Naming Conventions

* Packages: kisbetűs, com.supplier.\_switch
* Class: ArrayList
* Interface: CamelCase
* Metódus: mixedCase
* Constant: MAX\_INT
* Változók: mixedCase
* Paraméter típus: karakter, nagybetű, K -Key, T – Type, V – Value

# Packages

A package (csomag) egy Java programozási nyelvben használt mechanizmus, amely lehetővé teszi, hogy az osztályokat logikai egységekbe rendezzük. A csomagok az osztályok csoportosítására és a névtér konfliktusok elkerülésére szolgálnak.

package com.example.mypackage;

public class MyClass {

// osztály kódja

}

A csomagok nevei az általános Java konvenciók szerint fordított internetes domain nevek formájában íródnak. Ezzel biztosítható, hogy a csomagok nevei egyediek legyenek, és ne ütközzenek más csomagok neveivel.

* Settings/Auto Import/Show, Optimize, Add pipa.
* Settings/Auto Import/Insert imports on past: Ask
* Három, main nélküli java mentése jar-ként: File/Project Structure/Artifacts/+Java/From modules with dependencies, Build/Make Project/Build Artifact/Build, File/Project Structure/Libraries, az új mainnél automatikusan megtalálja.

# Scope

A "scope" szó a programozásban a változók vagy metódusok érvényességi körét jelenti. Az érvényességi kör megadja, hogy melyik része a programnak férhet hozzá a változókhoz vagy metódusokhoz. A scope lehet lokális vagy globális. A lokális scope a metódusok vagy blokkok belsejében van, míg a globális scope a teljes programon belül érvényes.

# Access Modifiers

* public: Mindenhol látható a csomagban és ahova be van importálva.
* package-private: Csak a saját csomagjában elérhető.
* private: Ott elérhető, ahol deklarálták.
* protected: A saját csomagjában mindenhol elérhető, de olyan alosztályban is, ami más csomagban található.

Az interface alapértelmezett láthatósága package-private. Az összes interface-ban lévő változó alapértelmezetten public static final. A metódusok is publicok.

## The static statement

A statikus adattagok és metódusok azzal a tulajdonsággal bírnak, hogy nem az objektumok példányaiban léteznek, hanem az osztályban, vagyis egyszerre csak egy példánya van belőlük a program futása során. A statikus adattagokat és metódusokat az osztály nevével lehet elérni, nem pedig az objektum példányaival.

A statikus metódusoknál nem szükséges példányosítani az osztályt, hanem közvetlenül az osztály nevével hívhatóak.

A statikus adattagok és metódusok nagyon hasznosak lehetnek például osztályszintű változók tárolásához, vagy olyan metódusokhoz, amelyeket az osztály egésze számára használnak.

## The final statement

A final változók egyszer módosíthatóak, a constructorban.

Alapértelmezett láthatósági szintek:

* Interface: public static final
* Abstract class: package-private
* Class: package-private
* Változók: package-private
* Listák: public
* Tömbök: Azonos a deklaráló osztályéval.

# Java Collections

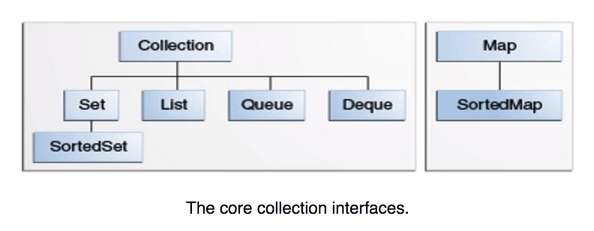
Karakter és szám változó keverése:

public Theatre(String theatreName, int numRows, int seatsPerRow) {  
 this.theatreName = theatreName;  
  
 int lastRow = 'A' + (numRows -1);  
 for (char row = 'A'; row <= lastRow; row++) {  
 for(int seatNum = 1; seatNum <= seatsPerRow; seatNum++) {  
 Seat seat = new Seat(row + String.*format*("%02d", seatNum));  
 seats.add(seat);  
 }  
 }  
}

LinkedList deklarálása:

private List<Seat> seats = new ArrayList<>();

A Collection az egyetemes Java keretrendszerben található interfész, amely lehetővé teszi, hogy egy csoport objektumot tároljunk és manipuláljunk egyetlen egységes felületen keresztül. A Collection interfész a gyűjtemények általános tulajdonságait definiálja, például az elemek hozzáadását, eltávolítását és lekérdezését. Az ArrayList az egyik olyan osztály, amely implementálja a Collection interfészt, és lehetővé teszi a dinamikusan változó méretű tömbök használatát.



A HashSet egy olyan konkrét implementáció a Collection interfészből, amely egy halmazt (set) reprezentál, vagyis egy olyan gyűjteményt, amely csak egyedi elemeket tartalmaz. A HashSet használatakor nincs garantált sorrend, és az elemek hozzáadásának és eltávolításának ideje állandó marad az összes elem esetében, függetlenül attól, hogy a halmaz mérete milyen nagy.

private Collection<Seat> seats = new HashSet<>();

private Collection<Seat> seats = new LinkedHashSet<>();

private class Seat implements Comparable<Seat>

@Override  
public int compareTo(Seat seat) {  
 return this.seatNumber.compareToIgnoreCase(seat.getSeatNumber());  
}

seatCopy.get(1).reserve();

Collections.*shuffle*(seatCopy);

Theatre.Seat minSeat = Collections.*min*(seatCopy);

theatre.seats.size()

Collections.*swap*(list, i, j);

## Bináris keresés

public boolean reserveSeat(String seatNumber) {  
 int low = 0;  
 int high = seats.size()-1;  
  
 while (low <= high) {  
 System.*out*.print(".");  
 int mid = (low + high) / 2;  
 Seat midVal = seats.get(mid);  
 int cmp = midVal.getSeatNumber().compareTo(seatNumber);  
  
 if (cmp < 0) {  
 low = mid + 1;  
 } else if (cmp > 0) {  
 high = mid - 1;  
 } else {  
 return seats.get(mid).reserve();  
 }  
 }  
 System.*out*.println("There is no seat " + seatNumber);  
 return false;  
}

## Sort rendezés

public static void sortList(List<? extends Theatre.Seat> list) {  
 for(int i=0; i<list.size() -1; i++) {  
 for(int j=i+1; j<list.size(); j++) {  
 if(list.get(i).compareTo(list.get(j)) >0) {  
 Collections.*swap*(list, i, j);  
 }  
 }  
 }  
}

## Comparable and Comparator

A Comparable és a Comparator mindkettő interfészek, amelyek segítenek az objektumok összehasonlításában. A különbség az, hogy a Comparable egy alapértelmezett rendezési szabályt ad az objektumokhoz, amely alapján összehasonlíthatók, a compareTo egy abstract metódus, míg a Comparator segítségével saját egyedi rendezési szabályokat definiálhatunk az objektumokhoz, amelyek eltérhetnek az alapértelmezetttől.

public class Seat implements Comparable<Seat> {

@Override  
 public int compareTo(Seat seat) {  
 return this.seatNumber.compareToIgnoreCase(seat.getSeatNumber());  
 }

}

static final Comparator<Seat> *PRICE\_ORDER*;

static {  
 *PRICE\_ORDER* = new Comparator<Seat>() {  
 @Override  
 public int compare(Seat seat1, Seat seat2) {  
 if (seat1.getPrice() < seat2.getPrice()) {  
 return -1;  
 } else if (seat1.getPrice() > seat2.getPrice()) {  
 return 1;  
 } else {  
 return 0;  
 }  
 }  
 };  
}

## Az inner és nested, anonim class

Az inner és a nested class kifejezések gyakran felcserélhetőek, és ugyanarra az osztály típusra utalnak, amely egy másik osztályon belül van deklarálva. Azonban van egy kis különbség a kettő között:

Inner class: Az inner class egy olyan osztály, amely belső osztályként van deklarálva egy másik osztályban. Az inner class eléri az anyaosztály privát adattagjait és metódusait, míg az anyaosztály nem férhet hozzá az inner class privát tagjaihoz. Az inner class példányosítható csak az anyaosztály egy példánya birtokában.

Nested class: A nested class egy olyan osztály, amelyet egy másik osztályba ágyaztak, és lehet statikus vagy nem statikus. A statikus nested class eléri az anyaosztály statikus tagjait, míg a nem statikus nested class eléri az anyaosztály statikus és nem statikus tagjait is. A nested class példányosítható az anyaosztály példánya nélkül is.

Az anonim osztályokat általában akkor használjuk, amikor csak egyszeri alkalommal szeretnénk definiálni egy olyan osztályt, amelyet nem tervezünk újrahasznosítani. Az ilyen osztályokat általában egy interfész implementálására vagy egy szuperosztályból való öröklésre használjuk. Az anonim osztály nem rendelkezik saját névvel, mert az a célja, hogy egyedi és egyszeri implementációt adjon az adott interfésznek vagy osztálynak, amelyhez kapcsolódik. Az anonim osztály létrehozása során az osztály definíciója és példányosítása egy lépésben történik, és az osztály neve nem szükséges, mert csak az adott kódblokkban létezik.

## Map

A Map egy kulcs-érték párok gyűjteménye, ahol a kulcsok egyedi azonosítóként szolgálnak az értékekhez. A kulcsok és az értékek lehetnek bármilyen típusúak, és a Map azonos kulcsokat nem enged meg. A Map lehetővé teszi az adatok hatékony keresését és hozzáférését a kulcsok alapján, és számos implementációja létezik a Java-ban, például a HashMap, TreeMap, ConcurrentHashMap stb. A Map használata széles körben elterjedt a programozásban, például adatbázis-kapcsolatok, webalkalmazások és játékok területén.

Ha két ugyanolyan kulccsal rögzítünk két értéket, akkor a második felülírja az elsőt. A kulcsok olyan sorrendben jelennek meg, ahogy felvettük, nem ABC sorrendben. A kulcs és az érték is lehet objektum. A kulcsot nem lehet megváltoztatni.

Map<String, String> languages = new HashMap<>();  
if(languages.containsKey("Java")) {  
 System.*out*.println("Java already exists");  
} else {  
 languages.put("Java", "a compiled high level, object-oriented, platform independent language");  
 System.*out*.println("Java added successfully");  
}

languages.put("Python", "an interpreted, object-oriented, high-level programming language with dynamic semantics");

for(String key: languages.keySet()) {  
 System.*out*.println(key + " : " + languages.get(key));  
}

languages.remove("Algol", "an algorithmic language");

languages.replace("Lisp", "Therein lies madness", "a functional programming language with imperative features");

Van-e már ez az egység?

StockItem inStock = list.getOrDefault(item.getName(), item);

Nem módosítható map-pal való visszatérés:

return Collections.*unmodifiableMap*(list);

## Immutable classes

1. Nem engedélyezi a settereket.
2. Fields: final, private
3. Nem lehet alosztályba sorolni.
4. Ha a példánymezők hivatkozásokat tartalmaznak módosítható objektumokra, ne lehessen ezeket az objektumokat a módosítani:
   1. Ne legyen olyan metódus, amely módosítja a módosítható objektumokat.
   2. Ne legye megosztva hivatkozás a változtatható objektumokra.

## Set és HashSet

Halmaz típus.

private final Set<HeavenlyBody> satellites;

### Equals() és hashCode()

Az == operátor összehasonlítja két objektum referenciáját, és igazat ad, ha a két referencia ugyanarra az objektumra mutat. Ezzel szemben az equals() metódus az objektum tartalmát hasonlítja össze, nem pedig a referenciákat. Az equals() metódus az Object osztályban van definiálva, de az általunk létrehozott saját osztályok esetén általában érdemes felülírni, hogy saját igényeink szerinti összehasonlítást végezhessünk.

A hashCode() hasításhoz alkalmazható, feladata megadni az adott példányhoz rendelhető hasítókódot, amely lényegében egy 32 bites egész szám. Ezt egyes adatszerkezetek, például halmazok vagy hasítótáblák (avagy kulcstranszformációs táblázatok, ld. később) esetén használják fel. Emiatt a metódus megvalósítása során elvárás, hogy:

* ha az objektum nem változik, ugyanazt az értéket adja;
* ha két objektum az equals() szerint megegyezik, a hashCode() is egyezzen meg;
* két különböző objektumra nem kell különböző értéket adni (noha ez a hasítókód ütközéséhez vezet, amelynek vannak bizonyos következményei).

@Override  
public boolean equals(Object obj) {  
 if(this == obj) {  
 return true;  
 }  
  
 System.*out*.println("obj.getClass() is " + obj.getClass());  
 System.*out*.println("this.getClass() is " + this.getClass());  
 if ((obj == null) || (obj.getClass() != this.getClass())) {  
 return false;  
 }  
  
 String objName = ((HeavenlyBody) obj).getName();  
 return this.name.equals(objName);  
}

@Override  
public int hashCode() {  
 System.*out*.println("hashcode called");  
 return this.name.hashCode() + 57;  
}

### Unió

Set<Integer> union = new HashSet<>(squares);

union.addAll(cubes);

Az egyformák csak egyszer vannak benne.

### Metszet

Set<Integer> intersection = new HashSet<>(squares);  
intersection.retainAll(cubes);

### Különbség

System.*out*.println("nature - divine:");  
Set<String> diff1 = new HashSet<>(nature);  
diff1.removeAll(divine);  
*printSet*(diff1);

Tömb hozzáadása halmazhoz

Set<String> nature = new HashSet<>();  
Set<String> divine = new HashSet<>();  
String[] natureWords = {"all", "nature", "is", "but", "art", "unknown", "to", "thee"};  
nature.addAll(Arrays.*asList*(natureWords));  
  
String[] divineWords = {"to", "err", "is", "human", "to", "forgive", "divine"};  
divine.addAll(Arrays.*asList*(divineWords));

## LinkedHashMap

A HashMap és a LinkedHashMap is a Map interfész implementációi Java-ban. A két implementáció közötti különbség a belső adatszerkezetekben van.

A HashMap a bejegyzéseket egy hash táblában tárolja, amely gyors keresést tesz lehetővé a kulcsok alapján. Az elemek sorrendje véletlenszerű lehet, mert a hash tábla az elemeket a hash-kódjuk alapján rendezi el.

A LinkedHashMap hasonlóan működik, mint a HashMap, de egy kiegészítő adatszerkezetet is tartalmaz, amely egy kétirányú láncolt lista. Ennek eredményeként az elemek hozzáadása és eltávolítása lassabb lehet, mint a HashMap esetében, de az elemek sorrendje megegyezik a beszúrási sorrenddel.

Egy másik fontos különbség a LinkedHashMap és a HashMap között az, hogy a LinkedHashMap lehetővé teszi a belső adatszerkezet iterátorainak módosítását, míg a HashMap nem engedi meg.

Az iterátor egy olyan objektum, amely lehetővé teszi a kollekciók (pl. listák, halmazok, térképek stb.) elemeinek végigiterálását, vagyis sorban lehívását. Az iterátorral lehetőségünk van az adatszerkezet elemein egyesével végigmenni, és az aktuális elemre hivatkozni. Az iterátorok a java.util csomagban találhatók.

A kétirányú láncolt lista egy adatszerkezet, amely lehetővé teszi az elemek hozzáadását és eltávolítását mindkét irányból. Az elemek egymásra mutató mutatók segítségével vannak tárolva, és minden elemnek van egy előző és egy következő mutatója. A kétirányú láncolt lista hasznos lehet, ha olyan adatokat kell tárolni, amelyeket gyakran kell hozzáadni vagy eltávolítani a lista elejéről vagy végéről.

## TreeMap

A TreeMap egy olyan implementációja a Map interfésznek, amely a kulcsokat rendezetten tárolja. A TreeMap az elemeket a kulcsok alapján rendezi, így amikor iterálunk rajta, az elemek rendezett sorrendben fognak megjelenni. A TreeMap az összes Map implementáció közül a leglassabb, mert a kulcsok rendezése folyamatosan történik, de ha szükségünk van rendezett adattárolásra, akkor érdemes lehet használni.

this.list = new TreeMap<>();

## UnmodifiableMap

Az unmodifiableMap egy olyan metódus a java.util.Collections osztályban, amely egy adott Map-ot visszaad egy nem módosítható (immutable) Map-ként. Ez azt jelenti, hogy az eredeti Map-ot nem lehet módosítani az unmodifiableMap segítségével visszaadott Map-on keresztül, mivel az csak olvasható. Az unmodifiableMap segítségével biztosíthatjuk, hogy más kód ne változtathassa meg az adatokat a Map-ban. Ha mégis szeretnénk módosítani a Map-ot, akkor az eredeti Map-ot kell használnunk. Az unmodifiableMap hasznos lehet, ha azt szeretnénk, hogy az adatok csak olvashatóak legyenek bizonyos részein a kódunknak, például ha egy kliensnek szeretnénk biztosítani egy adatstruktúrát, de nem akarjuk, hogy módosítsa azt.

# Streams

A helyes megnevezés "Stream API", és az a Java nyelvbe beépített funkcionális programozási lehetőséget jelent, amely segítségével hatékonyan dolgozhatunk az adatokkal. A Stream API lehetővé teszi a kollekciók, tömbök vagy adatfolyamok elemzését és manipulálását. A Stream objektumok olyan egyszerű funkciókat kínálnak, mint a szűrés, a rendezés, a mappázás, a csoportosítás és az aggregálás. Az egyik legnagyobb előnye az, hogy a műveletek végrehajtása lazán történik, így csak azok az elemek dolgoznak fel, amelyekre szükség van.

List<String> someBingoNumbers = Arrays.*asList*(  
 "N40", "N36",  
 "B12", "B6",  
 "G53", "G49", "G60", "G50", "g64",  
 "I26", "I17", "I29",  
 "O71");  
  
List<String> gNumbers = new ArrayList<>();

someBingoNumbers  
 .stream()  
 .map(String::toUpperCase)  
 .filter(s->s.startsWith("G"))  
 .sorted()  
 .forEach(System.*out*::println);

Stream<String> ioNumberStream = Stream.*of*("I26", "I17", "I29", "O71");  
Stream<String> inNumberStream = Stream.*of*("N40", "N36", "I26", "I17", "I29", "O71");  
Stream<String> concatStream = Stream.*concat*(ioNumberStream, inNumberStream);  
System.*out*.println("-----------------------");  
System.*out*.println(concatStream  
 .distinct()  
 .peek(System.*out*::println)  
 .count());

A flatMap() metódus a Java 8-as Stream osztály egy olyan operátora, amely lehetővé teszi a Stream elemeinek összevonását és újabb Stream létrehozását. A flatMap() metódus a map() metódus egy változata, amely egy Stream helyett több Stream-et állít elő, amelyeket össze lehet vonni. A flatMap() metódus egy függvényt kap, amely a Stream elemeiből újabb Stream-eket állít elő, majd ezeket összevonja egyetlen Streammé. A flatMap() metódus hasznos, ha egy Stream elemeiből egy másik, az eredetivel össze nem vonható típusú Stream-et kell előállítani.

Map<Integer, List<Employee>> groupedByAge = departments.stream()  
 .flatMap(department -> department.getEmployees().stream())  
 .collect(Collectors.*groupingBy*(employee -> employee.getAge()));  
  
departments.stream()  
 .flatMap(department -> department.getEmployees().stream())  
 .reduce((e1, e2) -> e1.getAge() < e2.getAge() ? e1 : e2)  
 .ifPresent(System.*out*::println);  
  
Stream.*of*("ABC", "AC", "BAA", "CCCC", "XY", "ST")  
 .filter(s -> {  
 System.*out*.println(s);  
 return s.length() == 3;  
 })  
 .count();

# JavaFX

## Elemei

### .iml

Az .iml állomány a JavaFX projekt IntelliJ IDEA-ben történő konfigurációjához szükséges. Az IntelliJ IDEA azonosítja a projektet a fájlban tárolt információk alapján, és beállítja a projektet a megfelelő JDK-val és a szükséges könyvtárakkal. Az .iml állomány az IntelliJ IDEA projekt konfigurációs fájlja, és nem tartalmaz semmilyen forráskódot. Az állományokat az IntelliJ IDEA automatikusan létrehozza, amikor új projektet hozunk létre vagy meglévő projektet importálunk.

<?xml version="1.0" encoding="UTF-8"?>  
<module type="JAVA\_MODULE" version="4">  
 <component name="NewModuleRootManager" LANGUAGE\_LEVEL="JDK\_11" inherit-compiler-output="true">  
 <exclude-output />  
 <content url="file://$MODULE\_DIR$">  
 <sourceFolder url="file://$MODULE\_DIR$/src" isTestSource="false" />  
 </content>  
 <orderEntry type="inheritedJdk" />  
 <orderEntry type="sourceFolder" forTests="false" />  
 <orderEntry type="library" name="javafx-11" level="application" />  
 </component>  
</module>

### FXML

Az FXML (JavaFX Scene Builder Extensible Markup Language) egy XML-alapú nyelv, amelyet a JavaFX Scene Builderrel használnak felhasználói felületek létrehozásához. Az FXML leírja a felhasználói felület összetevőit, például a gombokat, a szövegmezőket, az elrendezési módokat stb., és definiálja, hogyan kell rendezni ezeket a komponenseket. Az FXML fájlok JavaFX alkalmazásokban használhatók, és lehetővé teszik az alkalmazás logikájának és a felhasználói felület megjelenítésének elkülönítését.

<?import javafx.geometry.Insets?>  
<?import javafx.scene.layout.GridPane?>  
  
<?import javafx.scene.control.Button?>  
<?import javafx.scene.control.Label?>  
<?import javafx.scene.text.Font?>  
<GridPane fx:controller="sample.Controller"  
 xmlns:fx="http://javafx.com/fxml" alignment="center" hgap="10" vgap="10">  
 <Label text="Welcome to JavaFX!" textFill="green">  
 <font>  
 <Font name="Times New Roman bold" size="70"/>  
 </font>  
  
 </Label>  
</GridPane>

### Controller.java

A Controller.java egy JavaFX alkalmazásban az a Java osztály, amely összeköti a felhasználói felületet leíró fxml fájlt a Java kódunkkal. Az fxml fájl tartalmazza a felhasználói felületen látható elemeket és azok elrendezését, míg a Controller.java osztály tartalmazza a kódokat, amelyek meghatározzák az egyes elemek működését és viselkedését. A JavaFX framework segítségével az fxml fájl automatikusan betöltődik, és a megfelelő Controller osztályhoz kapcsolódik, így lehetőség van arra, hogy a felhasználói felület és a Java kód szorosan együtt működjön.

### Start metódus fxml nélkül

public void start(Stage primaryStage) throws Exception{  
 Parent root = FXMLLoader.*load*(getClass().getResource("sample.fxml"));  
  
 GridPane root = new GridPane();  
 root.setAlignment(Pos.*CENTER*);  
 root.setVgap(10);  
 root.setHgap(10);  
  
 Label greeting = new Label("Welcome to JavaFX!");  
 greeting.setTextFill(Color.*GREEN*);  
 greeting.setFont(Font.*font*("Times New Roman", FontWeight.*BOLD*, 70));  
  
 root.getChildren().add(greeting);  
  
 primaryStage.setTitle("Hello JavaFX!");  
 primaryStage.setScene(new Scene(root, 700, 275));  
 primaryStage.show();  
}

## Új JavaFX készítése

1. File/Project Structure beállítani mindent.
2. JavaFX 11 hozzáadása a modulhoz
3. A modul-info.java hozzáadása a sample-höz.

module GridPaneLayout {  
 requires javafx.fxml;  
 requires javafx.controls;  
  
 opens sample;  
}

## Layouts

### GridPane Layout

A GridPane osztályt akkor használjuk, ha olyan elrendezést szeretnénk létrehozni a felhasználói felületen, ahol a tartalmakat egy rácsos rendszerben helyezzük el. A GridPane az oszlopok és sorok elrendezését biztosítja, a tartalmakat pedig az egyes cellákba helyezhetjük el. A GridPane különösen hasznos, ha a tartalmakat egyenletesen kell elhelyezni, vagy ha több elemet szeretnénk elhelyezni ugyanabban a sorban vagy oszlopban.

<?import javafx.geometry.Insets?>  
<?import javafx.scene.layout.GridPane?>  
  
<?import javafx.scene.control.Button?>  
<?import javafx.scene.control.Label?>  
<?import javafx.scene.layout.ColumnConstraints?>  
<GridPane fx:controller="sample.Controller"  
 xmlns:fx="http://javafx.com/fxml" alignment="center" hgap="10" vgap="10"  
 gridLinesVisible="true">  
 <!--<columnConstraints>-->  
 <!--<ColumnConstraints percentWidth="50.0"/>-->  
 <!--<ColumnConstraints percentWidth="50.0"/>-->  
 <!--</columnConstraints>-->  
  
 <Button text="Button One" GridPane.rowIndex="0" GridPane.columnIndex="0"/>  
 <Button text="Button Two" GridPane.rowIndex="0" GridPane.columnIndex="1"/>  
 <Button text="Button Three" GridPane.rowIndex="1" GridPane.columnIndex="0"/>  
 <Button text="Really Long Button Four more text more text more text more text more text more text " GridPane.rowIndex="1" GridPane.columnIndex="1"/>  
 <Button text="Button Five" GridPane.rowIndex="2" GridPane.columnIndex="0"/>  
</GridPane>

### HBox Layout

A HBox az egyik JavaFX layout konténer, amelyben az elemeket vízszintesen rendezhetjük el egymás mellett. A HBox használata különösen hasznos, ha például széles, alacsony méretű felhasználói felületeket szeretnénk tervezni, ahol az elemek egymás mellett jelennek meg. Egy példa a HBox használatára lehet egy egyszerű navigációs sáv vagy gombok elhelyezése egymás mellett az alkalmazásunkban. A HBox-on belül elhelyezhetünk más layout konténereket és bármilyen JavaFX vezérlőt, és a HBox automatikusan elrendezésre kerül az előre meghatározott módon.

<HBox fx:controller="sample.Controller"  
 xmlns:fx="http://javafx.com/fxml" alignment="bottom\_right"  
 style="-fx-border-color: red; -fx-border-width: 3; -fx-border-style: dashed"  
 spacing="10">  
 <padding>  
 <Insets bottom="10" right="10"/>  
 </padding>  
  
 <Button text="Okay" prefWidth="90"/>  
 <Button text="Cancel" prefWidth="90"/>  
 <Button text="Help" prefWidth="90"/>  
</HBox>

### BorderPane Layout

A BorderPane az elrendezés (layout) egyik típusa a JavaFX-ben, amelyet általában a fő ablakok, szélesebb vagy magasabb területek megjelenítésére használnak. A BorderPane-nak 5 része van: top, bottom, left, right, center. Az egyes részekhez adhatunk elemeket, amelyek a megfelelő részen jelennek meg, így lehetőséget nyújt arra, hogy az alkalmazásunkban egyszerűen elrendezzük az elemeket. A BorderPane használata különösen alkalmas olyan felhasználói felületek megjelenítésére, ahol a központi terület körül több kisebb terület található, például a menüsor, a szerszámsor, a navigációs sáv stb. A BorderPane alkalmazása egy kényelmes és rugalmas módja az alkalmazások elrendezésének.

<BorderPane fx:controller="sample.Controller"  
 xmlns:fx="http://javafx.com/fxml" >  
 <top>  
 <Label text="This label is in the top position" alignment="center"  
 BorderPane.alignment="center"  
 style="-fx-border-color: blue; -fx-border-width: 3; -fx-border-style: dashed"/>  
 </top>  
 <bottom>  
 <HBox spacing="10" alignment="bottom\_right">  
 <padding>  
 <Insets bottom="10" right="10"/>  
 </padding>  
 <Button text="Okay" prefWidth="90"/>  
 <Button text="Cancel" prefWidth="90"/>  
 <Button text="Help" prefWidth="90"/>  
 </HBox>  
 </bottom>  
</BorderPane>

### AnchorPane

Az AnchorPane JavaFX-ben elhelyezési elrendezés (layout) típus, amely lehetővé teszi, hogy az egyes gombok, címkék és más vezérlőelemek a szülőkonténerhez viszonyítva megadott négy pont egyikén legyenek pozicionálva. Ezáltal a tartalom mérete, elhelyezése és viselkedése viszonylag stabil marad a felület nagyítása vagy kicsinyítése során is.

Az AnchorPane akkor használható, ha a felhasználói felületen lévő elemeknek pontosan meghatározott helyük van a szülőkonténeren belül, például egy beállítások panelen vagy a bejelentkezési felületen.

### FlowPane

A FlowPane a JavaFX-ben egy elrendezési (layout) osztály, amelynek segítségével a felhasználói felület elemeit egymás mellé lehet rendezni, és ha elfogy a rendelkezésre álló hely, automatikusan átugrik az új sorba. Ez a vezérlő nagyon hasznos lehet olyan felhasználói felületekhez, ahol dinamikus elemek listája van, és ahol a dinamikus elemek különböző méretűek. A FlowPane-t általában olyan felhasználói felületekhez használják, ahol a térhatás fontos, és ahol a felhasználói felület elemeinek egymáshoz viszonyított sorrendje fontos szerepet játszik. A FlowPane lehetővé teszi a felhasználói felületelemek számára a folytonos áramlást, így a felület könnyen áttekinthetővé és navigálhatóvá válik.

### TilPane

A TilePane osztályt általában olyan felhasználói felületek kialakításához használjuk, amelyek szabályos méretű téglalapokból állnak. Például egy képgaléria vagy egy csempézett elrendezésű felület kialakítására lehet alkalmas. A TilePane egymás mellé helyezi az elemeket, és igyekszik a sorokat és az oszlopokat egységes méretűre tenni, így lehetővé téve az összehangolt megjelenést. A téglalapok közötti térköz és a gyerekelemek elrendezése számos módon testre szabható.

### StackPane

A StackPane a JavaFX-ben egy elrendezési osztály, amely az összes gyermekét azonos pozícióban, egymásra helyezi, hasonlóan egy veremhez. Ez azt jelenti, hogy a StackPane minden gyermekét a legfelső rétegre helyezi, így a legfelső gyermek lesz az elsődleges, míg a háttérben lévő gyerekek átlátszóak lesznek. Ez a pane hasznos lehet, ha olyan felhasználói felületet szeretnénk létrehozni, ahol az egyes elemek átfedhetik egymást.

## Controls

<GridPane fx:controller="sample.Controller"  
 xmlns:fx="http://javafx.com/fxml" alignment="center" hgap="10" vgap="10">  
 <Button GridPane.rowIndex="0" GridPane.columnIndex="0" text="Click me">  
 <graphic>  
 <ImageView>  
 <Image url="@/toolbarButtonGraphics/general/TipOfTheDay24.gif"/>  
 </ImageView>  
 </graphic>  
 </Button>  
 <Label GridPane.rowIndex="0" GridPane.columnIndex="1" text="This is a label">  
 <graphic>  
 <ImageView>  
 <Image url="@/toolbarButtonGraphics/general/Information24.gif"/>  
 </ImageView>  
 </graphic>  
 </Label>  
</GridPane>

Ikongyűjtemény: <https://www.oracle.com/java/technologies/java-look-and-feel-graphics-repository.html>, bele kell másolni a projektmappába.

### Imports

<?import javafx.geometry.Insets?>  
<?import javafx.scene.layout.GridPane?>  
  
<?import javafx.scene.control.Button?>  
<?import javafx.scene.control.Label?>  
<?import javafx.scene.image.ImageView?>  
<?import javafx.scene.image.Image?>  
<?import javafx.scene.text.Font?>  
<?import javafx.scene.control.RadioButton?>  
<?import javafx.scene.control.ToggleGroup?>  
<?import javafx.scene.control.CheckBox?>  
<?import javafx.scene.layout.VBox?>  
<?import javafx.scene.control.ToggleButton?>  
<?import javafx.scene.layout.HBox?>  
<?import javafx.scene.control.TextField?>  
<?import javafx.scene.control.PasswordField?>  
<?import javafx.scene.control.ComboBox?>  
<?import javafx.collections.\*?>  
<?import java.lang.String?>  
  
<?import javafx.scene.control.ChoiceBox?>  
<?import javafx.scene.control.Slider?>  
<?import javafx.scene.control.Spinner?>  
<?import javafx.scene.control.ColorPicker?>  
<?import javafx.scene.control.DatePicker?>  
<?import javafx.scene.control.TitledPane?>  
<?import javafx.scene.control.Accordion?>

### RadioButton and CheckBox

<fx:define>  
 <ToggleGroup fx:id="colorToggleGroup"/>  
</fx:define>  
<RadioButton GridPane.rowIndex="0" GridPane.columnIndex="2" text="Red" toggleGroup="$colorToggleGroup"/>  
<RadioButton GridPane.rowIndex="0" GridPane.columnIndex="3" text="Blue" toggleGroup="$colorToggleGroup" selected="true"/>  
<RadioButton GridPane.rowIndex="0" GridPane.columnIndex="4" text="Green" toggleGroup="$colorToggleGroup"/>  
  
  
<VBox GridPane.rowIndex="0" GridPane.columnIndex="5">  
 <CheckBox text="Dog"/>  
 <CheckBox text="Cat"/>  
 <CheckBox text="Bird"/>  
</VBox>  
<HBox GridPane.rowIndex="0" GridPane.columnIndex="6">  
 <ToggleButton text="Toggle Me" />  
 <ToggleButton text="Hello" />  
 <ToggleButton text="Good-Bye" />  
</HBox>

### TextField, PasswordField, ComboBox and ChoiceBox

<TextField GridPane.rowIndex="1" GridPane.columnIndex="0"/>  
<PasswordField GridPane.rowIndex="1" GridPane.columnIndex="1"/>  
<ComboBox GridPane.rowIndex="1" GridPane.columnIndex="2" GridPane.columnSpan="2">  
 <items>  
 <FXCollections fx:factory="observableArrayList">  
 <String fx:value="This is Option 1"/>  
 <String fx:value="This is Option 2"/>  
 <String fx:value="This is Option 3"/>  
 <String fx:value="This is Option 4"/>  
 <String fx:value="This is Option 5"/>  
 </FXCollections>  
 </items>  
 <value>  
 <String fx:value="This is Option 4"/>  
 </value>  
</ComboBox>  
<ChoiceBox GridPane.rowIndex="1" GridPane.columnIndex="4">  
 <items>  
 <FXCollections fx:factory="observableArrayList">  
 <String fx:value="cb 1"/>  
 <String fx:value="cb 2"/>  
 <String fx:value="cb 3"/>  
 <String fx:value="cb 4"/>  
 <String fx:value="cb 5"/>  
 </FXCollections>  
 </items>  
</ChoiceBox>

### Slider, Spiner, ColorPicker and DatePicker Controls

<Slider GridPane.rowIndex="2" GridPane.columnIndex="0" GridPane.columnSpan="4" min="0" max="100"  
 showTickLabels="true" showTickMarks="true" minorTickCount="4" snapToTicks="true"/>  
<Spinner GridPane.rowIndex="2" GridPane.columnIndex="4" min="0" max="100"  
 editable="true" initialValue="50"/>  
  
<ColorPicker GridPane.rowIndex="3" GridPane.columnIndex="0"/>  
<DatePicker GridPane.rowIndex="3" GridPane.columnIndex="1"/>

### Titled Pane

<Accordion GridPane.rowIndex="3" GridPane.columnIndex="2" GridPane.columnSpan="2"  
 expandedPane="$tp3">  
 <panes>  
 <TitledPane text="Titled Pane">  
 <Label text="Label in titled pane"/>  
 </TitledPane>  
 <TitledPane text="Titled Pane 2">  
 <Label text="Label 2 in titled pane"/>  
 </TitledPane>  
 <TitledPane fx:id="tp3" text="Titled Pane 3">  
 <Label text="Label 3 in titled pane"/>  
 </TitledPane>  
  
 </panes>  
</Accordion>

## Events and Event Handlers

### Sample.fxml

<GridPane fx:controller="sample.Controller"  
 xmlns:fx="http://javafx.com/fxml" alignment="center" hgap="10" vgap="10">  
 <TextField fx:id="nameField" GridPane.rowIndex="0" GridPane.columnIndex="0"  
 onKeyReleased="#handleKeyReleased"/>  
 <Button fx:id="helloButton" text="Say Hello" GridPane.rowIndex="1" GridPane.columnIndex="0"  
 onAction="#onButtonClicked"/>  
 <Button fx:id="byeButton" text="Say Bye" GridPane.rowIndex="1" GridPane.columnIndex="1"  
 onAction="#onButtonClicked"/>  
</GridPane>

### Controller.java

package sample;  
  
import javafx.event.ActionEvent;  
import javafx.fxml.FXML;  
import javafx.scene.control.Button;  
import javafx.scene.control.TextField;  
  
public class Controller {  
 @FXML  
 private TextField nameField;  
 @FXML  
 private Button helloButton;  
 @FXML  
 private Button byeButton;  
  
 @FXML  
 public void initialize() {  
 helloButton.setDisable(true);  
 byeButton.setDisable(true);  
  
 }  
 @FXML  
 public void onButtonClicked(ActionEvent e) {  
 if(e.getSource().equals(helloButton)) {  
 System.*out*.println("Hello, " + nameField.getText());  
 } else if(e.getSource().equals(byeButton)) {  
 System.*out*.println("Bye, " + nameField.getText());  
 }  
 }  
  
 @FXML  
 public void handleKeyReleased() {  
 String text = nameField.getText();  
 boolean disableButtons = text.isEmpty() || text.trim().isEmpty();  
 helloButton.setDisable(disableButtons);  
 byeButton.setDisable(disableButtons);  
 }  
}

## UI Thread, Runnable

Az UI Thread a JavaFX-ben a grafikus felhasználói felülettel kapcsolatos események kezeléséért felelős szál. Ez a szál felelős az UI komponensek megjelenítéséért és az eseménykezelő függvények végrehajtásáért. Az UI Thread fontos szerepet játszik az alkalmazás általános teljesítményének szempontjából, mert ha túl sokáig dolgozik, az alkalmazás válaszideje lassúvá válhat. A hosszú ideig futó folyamatokat a UI Threadtől külön szálakon kell elvégezni, hogy az alkalmazás reagáló maradjon.

A Runnable egy interfész a Java nyelvben, amely egyetlen metódust tartalmaz, a run() metódust. A Runnable-t általában akkor használjuk, amikor szálakat szeretnénk létrehozni. A run() metódusban lévő utasításokat a szálban fogjuk végrehajtani.

Ebben az esetben a MyRunnable osztály implementálja a Runnable interfészt, és felülírja a run() metódust, amely egy egyszerű szöveges üzenetet jelenít meg a konzolra. A main() metódusban létrehozunk egy új szálat a Thread osztály segítségével, amelynek paraméterként átadjuk az MyRunnable objektumot. Az start() metódussal indítjuk el a szálat, majd kiírunk egy másik szöveges üzenetet a fő szálon. Amikor a program lefut, a két szöveges üzenetet párhuzamosan jelenítjük meg a konzolon, mivel a MyRunnable szálának futása nem blokkolja a fő szál végrehajtását.

sample.fxml

<CheckBox fx:id="ourCheckBox" text="Clear the field after saying hello or bye" GridPane.rowIndex="2"  
 GridPane.columnIndex="0" GridPane.columnSpan="2"  
 onAction="#handleChange"/>  
<Label fx:id="ourLabel" text="Nothing has happened" GridPane.rowIndex="3" GridPane.columnIndex="0"/>

controller.java

@FXML  
private CheckBox ourCheckBox;  
@FXML  
private Label ourLabel;

public void handleChange() {  
 System.*out*.println("The checkbox is " + (ourCheckBox.isSelected() ? "checked" : "not checked"));  
}

@FXML  
public void onButtonClicked(ActionEvent e) {  
 if(e.getSource().equals(helloButton)) {  
 System.*out*.println("Hello, " + nameField.getText());  
 } else if(e.getSource().equals(byeButton)) {  
 System.*out*.println("Bye, " + nameField.getText());  
 }  
  
 Runnable task = new Runnable() {  
 @Override  
 public void run() {  
 try {  
 String s = Platform.*isFxApplicationThread*() ? "UI Thread" : "Background Thread";  
 System.*out*.println("I'm going to sleep on the: " + s);  
 Thread.*sleep*(10000);  
 Platform.*runLater*(new Runnable() {  
 @Override  
 public void run() {  
 String s = Platform.*isFxApplicationThread*() ? "UI Thread" : "Background Thread";  
 System.*out*.println("I'm updating the label on the: " + s);  
 ourLabel.setText("We did something!");  
 }  
 });  
 } catch(InterruptedException event) {  
 // we don't care about this  
 }  
 }  
 };  
  
 new Thread(task).start();  
  
 if(ourCheckBox.isSelected()) {  
 nameField.clear();  
 helloButton.setDisable(true);  
 byeButton.setDisable(true);  
 }  
}

## Singleton

A Singleton egy olyan tervezési minta, amelynek célja, hogy biztosítsa, hogy egy adott osztályból csak egyetlen példány létezzen, és hogy a példányhoz az alkalmazás bármely részéből egyszerűen hozzáférhessenek. A Singleton objektumok leggyakrabban az alkalmazás konfigurációjának vagy állapotának tárolására szolgálnak. A Singleton osztályokat általában azért használják, mert biztosítják, hogy egy adott osztályból csak egy példány jöjjön létre, így elkerülhetők az olyan problémák, mint a memóriaterhelés és a teljesítményproblémák.

# Kivételkezelés

private static int getIntEAFP() {  
 Scanner s = new Scanner(System.*in*);  
 System.*out*.println("Please enter an integer ");  
 try {  
 return s.nextInt();  
 } catch(InputMismatchException e) {  
 return 0;  
 }  
}

private static int divideEAFP(int x, int y) {  
 try {  
 return x / y;  
 } catch(ArithmeticException e) {  
 return 0;  
 }  
}

## Több kivétel kezelése

public static void main(String[] args) {  
 try {  
 int result = *divide*();  
 System.*out*.println(result);  
 } catch(ArithmeticException | NoSuchElementException e) {  
 System.*out*.println(e.toString());  
 System.*out*.println("Unable to perform division, autopilot shutting down");  
 }  
 }  
  
 private static int divide() {  
 int x, y;  
 x = *getInt*();  
 y = *getInt*();  
 System.*out*.println("x is " + x + ", y is " + y);  
 return x / y;  
 }  
  
 private static int getInt() {  
 Scanner s = new Scanner(System.*in*);  
 System.*out*.println("Please enter an integer ");  
 while(true) {  
 try {  
 return s.nextInt();  
 } catch(InputMismatchException e) {  
 s.nextLine();  
 System.*out*.println("Please enter a number using only the digits 0 to 9 ");  
 }  
 }  
 }

# I/O

## FileWriter class and Finally block

A FileWriter objektumot akkor használjuk, amikor szöveges adatokat szeretnénk fájlba írni. Általában kisebb fájlok esetén használjuk, mivel a nagyobb fájlok esetén hatékonysági problémák lehetnek.

FileWriter locFile = null;  
try {  
 locFile = new FileWriter("locations.txt");  
 for(Location location : *locations*.values()) {  
 locFile.write(location.getLocationID() + "," + location.getDescription() + "\n");  
 }  
} catch(IOException e) {  
 System.*out*.println("In catch block");  
 e.printStackTrace();  
} finally {  
 System.*out*.println("in finally block");  
 try {  
 if(locFile != null) {  
 System.*out*.println("Attempting to close locfile");  
 locFile.close();  
 }  
 } catch(IOException e) {  
 e.printStackTrace();  
 }  
}

## Throw

A throw kulcsszó lehetővé teszi a fejlesztők számára, hogy saját kivételeket dobjanak, amikor bizonyos feltételek teljesülnek. Például, ha egy metódus bemeneti paramétere érvénytelen, a metódus dobhat egy kivételt, amely jelzi az érvénytelen bemenetet. A kivétel dobása után a hívó programnak meg kell próbálnia kezelni a kivételt.

A throw kulcsszó használata előtt meg kell határozni, milyen típusú kivételt akarunk dobni. Ezután a throw kulcsszó után a kivétel példányát kell megadni, amelyet dobni akarunk.

public static void main(String[] args) throws IOException {  
 try(FileWriter locFile = new FileWriter("locations.txt");  
 FileWriter dirFile = new FileWriter("directions.txt")) {  
 for(Location location : *locations*.values()) {  
 locFile.write(location.getLocationID() + "," + location.getDescription() + "\n");  
 for(String direction : location.getExits().keySet()) {  
 dirFile.write(location.getLocationID() + "," + direction + "," + location.getExits().get(direction) + "\n");  
 }  
 }  
 }

}

## Reader

Scanner scanner = null;  
try {  
 scanner = new Scanner(new FileReader("locations.txt"));  
 scanner.useDelimiter(",");  
 while(scanner.hasNextLine()) {  
 int loc = scanner.nextInt();  
 scanner.skip(scanner.delimiter());  
 String description = scanner.nextLine();  
 System.*out*.println("Imported loc: " + loc + ": " + description);  
 Map<String, Integer> tempExit = new HashMap<>();  
 *locations*.put(loc, new Location(loc, description, tempExit));  
 }  
  
} catch(IOException e) {  
 e.printStackTrace();  
} finally {  
 if(scanner != null) {  
 scanner.close();  
 }  
}

## Buffered reader

A BufferedReader osztály egy olyan osztály a Java nyelvben, amely lehetővé teszi a szövegfájlok olvasását karakterek sorozataként, amelyek nagyobb teljesítményűek, mint ha karakterenként olvasnánk be a fájlt. A BufferedReader lehetővé teszi a szövegfájlok olvasását soronként, illetve a readLine() metódus segítségével egy adott sor beolvasását egy String-be.

A BufferedWriter objektumot akkor használjuk, ha nagyobb mennyiségű szöveget szeretnénk írni a fájlba. Az objektum írási folyamatának sebességét javítja a buffer használata, amely az adatokat ideiglenesen tárolja a memóriában.

try {  
 scanner = new Scanner(new BufferedReader(new FileReader("directions.txt")));  
 scanner.useDelimiter(",");  
 while(scanner.hasNextLine()) {  
// int loc = scanner.nextInt();  
// scanner.skip(scanner.delimiter());  
// String direction = scanner.next();  
// scanner.skip(scanner.delimiter());  
// String dest = scanner.nextLine();  
// int destination = Integer.parseInt(dest);  
 String input = scanner.nextLine();  
 String[] data = input.split(",");  
 int loc = Integer.*parseInt*(data[0]);  
 String direction = data[1];  
 int destination = Integer.*parseInt*(data[2]);  
  
 System.*out*.println(loc + ": " + direction + ": " + destination);  
 Location location = *locations*.get(loc);  
 location.addExit(direction, destination);  
 }  
 } catch (IOException e) {  
 e.printStackTrace();  
 } finally {  
 if(scanner != null) {  
 scanner.close();  
 }  
 }

protected void addExit(String direction, int location) {  
 exits.put(direction, location);  
}

A FileReader a fájlt karakterről karakterre olvassa be, míg a BufferedReader egész sorokat olvas be egy belső pufferről, ami hatékonyabbá teszi az olvasást. A BufferedReader használatával gyakran sok időt spórolhatunk, különösen akkor, ha nagy fájlokat olvasunk be.

## Bufferred Writer

public static void main(String[] args) throws IOException {  
 try(BufferedWriter locFile = new BufferedWriter(new FileWriter("locations.txt"));  
 BufferedWriter dirFile = new BufferedWriter(new FileWriter("directions.txt"))) {  
 for(Location location : *locations*.values()) {  
 locFile.write(location.getLocationID() + "," + location.getDescription() + "\n");  
 for(String direction : location.getExits().keySet()) {  
 if(!direction.equalsIgnoreCase("Q")) {  
 dirFile.write(location.getLocationID() + "," + direction + "," + location.getExits().get(direction) + "\n");  
 }  
 }  
 }  
 }  
}

## Byte Streams

A byte streamok a Java I/O API-jának egy típusa, amelyeket a byte adatok feldolgozására használnak. A byte streamok olyan adatfolyamokat kezelnek, amelyek minden bájtot külön-külön kezelnek, és nincs bennük semmilyen karakterszintű feldolgozás. Az InputStream és az OutputStream osztályok a byte streamok alapját képezik. Az InputStream osztályból származó osztályok olvassák be a byte-okat az adatforrásból, az OutputStream osztályból származó osztályok pedig írják ki a byte-okat az adatfolyamba.

A byte stream (FileOutputStream, BufferedOutputStream) akkor használjuk, ha nem szöveges adatokat szeretnénk fájlba írni, hanem bináris adatokat, például képeket vagy hangfájlokat. Ez azért hasznos, mert a bináris adatok fájlba írása közben az adatok konvertálása a karakterkódolásra nem történik meg, ami hatékonyabbá teszi az írási folyamatot.

Byte stream-mel fájlba írás hasznos lehet, ha olyan adatokat szeretnénk fájlba írni, amelyek nagyok és bináris formátumban vannak, például képek vagy hangfájlok. Ebben az esetben a byte stream lehetővé teszi az adatok hatékonyabb olvasását és írását, mivel a byte-ok egyszerre olvashatók vagy írhatók, és nem szükséges az egész fájlt memóriába beolvasni vagy kiírni.

A byte stream továbbá lehetővé teszi a pontosabb adatkezelést, különösen azok esetében, amelyek bináris formátumban vannak, és az adatok bitjeinek helyes sorrendje fontos a megfelelő olvasásukhoz vagy írásukhoz. A byte stream használata biztosítja, hogy az adatok helyes sorrendben kerülnek írásra vagy olvasásra a fájlban, így megelőzve a hibákat vagy a fájl sérülését.

try (DataOutputStream locFile = new DataOutputStream(new BufferedOutputStream(new FileOutputStream("locations.dat")))) {  
 for (Location location : *locations*.values()) {  
 locFile.writeInt(location.getLocationID());  
 locFile.writeUTF(location.getDescription());  
 System.*out*.println("Writing location " + location.getLocationID() + " : " + location.getDescription());  
 System.*out*.println("Writing " + (location.getExits().size() - 1) + " exits.");  
 locFile.writeInt(location.getExits().size() - 1);  
 for (String direction : location.getExits().keySet()) {  
 if (!direction.equalsIgnoreCase("Q")) {  
 System.*out*.println("\t\t" + direction + "," + location.getExits().get(direction));  
 locFile.writeUTF(direction);  
 locFile.writeInt(location.getExits().get(direction));  
 }  
 }  
 }  
}

try(DataInputStream locFile = new DataInputStream(new BufferedInputStream(new FileInputStream("locations.dat")))) {  
 boolean eof = false;  
 while(!eof) {  
 try {  
 Map<String, Integer> exits = new LinkedHashMap<>();  
 int locID = locFile.readInt();  
 String description = locFile.readUTF();  
 int numExits = locFile.readInt();  
 System.*out*.println("Read location " + locID + " : " + description);  
 System.*out*.println("Found " + numExits + " exits");  
 for(int i=0; i<numExits; i++) {  
 String direction = locFile.readUTF();  
 int destination = locFile.readInt();  
 exits.put(direction, destination);  
 System.*out*.println("\t\t" + direction + "," + destination);  
 }  
 *locations*.put(locID, new Location(locID, description, exits));  
  
 } catch(EOFException e) {  
 eof = true;  
 }  
  
 }  
} catch(IOException io) {  
 System.*out*.println("IO Exception");  
}

## Object I/O including Serialization

A serialization egy folyamat, amely során egy objektum állapotát átalakítjuk egy olyan formátumba, amely könnyen továbbítható, tárolható vagy másolható. Az így elkészített formátumot általában byte stream-ként tároljuk el vagy továbbítjuk hálózaton. A Java-ban az objektum serialization-t az ObjectOutputStream osztály végzi, míg az objektumok visszaállítását az ObjectInputStream osztály végzi. A serialization segítségével lehetőség van objektumok mentésére, adatbázisba írására, hálózaton történő átvitelre vagy objektumok klónozására.

A "serialVersionUID" egy olyan mező, amelyet az objektum szerializálásához használnak. Amikor egy objektumot szerializálnak, az átkerül egy byte tömbbe, amely később visszaállítható az eredeti objektummá. Ha a szerializálás során az objektumot módosítják, és az új verziója inkompatibilis a régi verzióval, akkor a deszerializálás során hibák léphetnek fel.

A "serialVersionUID" segít az ilyen hibák elkerülésében. Ha egy objektumhoz tartozó "serialVersionUID" ugyanaz, mint a deszerializációkor használt programban, akkor az objektum sikeresen visszaállítható. Ha a "serialVersionUID" eltér, akkor a deszerializáció hibát eredményezhet. Az értéke 1L azt jelenti, hogy az első verziót használja az objektum szerializálása és deszerializálása során.

public class Location implements Serializable{

…

private long serialVersionUID = 1L;

try (ObjectOutputStream locFile = new ObjectOutputStream(new BufferedOutputStream(new FileOutputStream("locations.dat")))) {  
 for(Location location : *locations*.values()) {  
 locFile.writeObject(location);  
 }  
}

try(ObjectInputStream locFile = new ObjectInputStream(new BufferedInputStream(new FileInputStream("locations.dat")))) {  
 boolean eof = false;  
 while(!eof) {  
 try {  
 Location location = (Location) locFile.readObject();  
 System.*out*.println("Read location " + location.getLocationID() + " : " + location.getDescription());  
 System.*out*.println("Found " + location.getExits().size() + " exits");  
  
 *locations*.put(location.getLocationID(), location);  
 } catch(EOFException e) {  
 eof = true;  
 }  
 }

} catch(InvalidClassException e) {  
 System.*out*.println("InvalidClassException " + e.getMessage());  
} catch(IOException io) {  
 System.*out*.println("IO Exception" + io.getMessage());  
} catch(ClassNotFoundException e) {  
 System.*out*.println("ClassNotFoundException " + e.getMessage());  
}

## Random Access File

public class IndexRecord {  
 private int startByte;  
 private int length;

…

}

private static Map<Integer, IndexRecord> *index* = new LinkedHashMap<>();

private static RandomAccessFile *ra*;

…

try (RandomAccessFile rao = new RandomAccessFile("locations\_rand.dat", "rwd")) {  
 rao.writeInt(*locations*.size());  
 int indexSize = *locations*.size() \* 3 \* Integer.*BYTES*;  
 int locationStart = (int) (indexSize + rao.getFilePointer() + Integer.*BYTES*);  
 rao.writeInt(locationStart);  
  
 long indexStart = rao.getFilePointer();  
  
 int startPointer = locationStart;  
 rao.seek(startPointer);  
  
 for(Location location : *locations*.values()) {  
 rao.writeInt(location.getLocationID());  
 rao.writeUTF(location.getDescription());  
 StringBuilder builder = new StringBuilder();  
 for(String direction : location.getExits().keySet()) {  
 if(!direction.equalsIgnoreCase("Q")) {  
 builder.append(direction);  
 builder.append(",");  
 builder.append(location.getExits().get(direction));  
 builder.append(",");  
 }  
 }  
 rao.writeUTF(builder.toString());  
  
 IndexRecord record = new IndexRecord(startPointer, (int) (rao.getFilePointer() - startPointer));  
 *index*.put(location.getLocationID(), record);  
  
 startPointer = (int) rao.getFilePointer();  
 }  
  
 rao.seek(indexStart);  
 for(Integer locationID : *index*.keySet()) {  
 rao.writeInt(locationID);  
 rao.writeInt(*index*.get(locationID).getStartByte());  
 rao.writeInt(*index*.get(locationID).getLength());  
 }  
 }  
}

try {  
 *ra* = new RandomAccessFile("locations\_rand.dat", "rwd");  
 int numLocations = *ra*.readInt();  
 long locationStartPoint = *ra*.readInt();  
  
 while(*ra*.getFilePointer() < locationStartPoint) {  
 int locationId = *ra*.readInt();  
 int locationStart = *ra*.readInt();  
 int locationLength = *ra*.readInt();  
  
 IndexRecord record = new IndexRecord(locationStart, locationLength);  
 *index*.put(locationId, record);  
 }  
  
 } catch(IOException e) {  
 System.*out*.println("IOException in static initializer: " + e.getMessage());  
 }  
}

public Location getLocation(int locationId) throws IOException {  
  
 IndexRecord record = *index*.get(locationId);  
 *ra*.seek(record.getStartByte());  
 int id = *ra*.readInt();  
 String description = *ra*.readUTF();  
 String exits = *ra*.readUTF();  
 String[] exitPart = exits.split(",");  
  
 Location location = new Location(locationId, description, null);  
  
 if(locationId != 0) {  
 for(int i=0; i<exitPart.length; i++) {  
 System.*out*.println("exitPart = " + exitPart[i]);  
 System.*out*.println("exitPart[+1] = " + exitPart[i+1]);  
 String direction = exitPart[i];  
 int destination = Integer.*parseInt*(exitPart[++i]);  
 location.addExit(direction, destination);  
 }  
 }  
  
 return location;  
}

public void close() throws IOException {  
 *ra*.close();  
}

Location currentLocation = *locations*.getLocation(1);  
 while(true) {  
 System.*out*.println(currentLocation.getDescription());  
  
 if(currentLocation.getLocationID() == 0) {  
 break;  
 }  
  
 Map<String, Integer> exits = currentLocation.getExits();  
 System.*out*.print("Available exits are ");  
 for(String exit: exits.keySet()) {  
 System.*out*.print(exit + ", ");  
 }  
 System.*out*.println();  
  
 String direction = scanner.nextLine().toUpperCase();  
 if(direction.length() > 1) {  
 String[] words = direction.split(" ");  
 for(String word: words) {  
 if(vocabulary.containsKey(word)) {  
 direction = vocabulary.get(word);  
 break;  
 }  
 }  
 }  
  
 if(exits.containsKey(direction)) {  
 currentLocation = *locations*.getLocation(currentLocation.getExits().get(direction));  
  
 } else {  
 System.*out*.println("You cannot go in that direction");  
 }  
 }  
  
 *locations*.close();  
  
}

## Java NIO