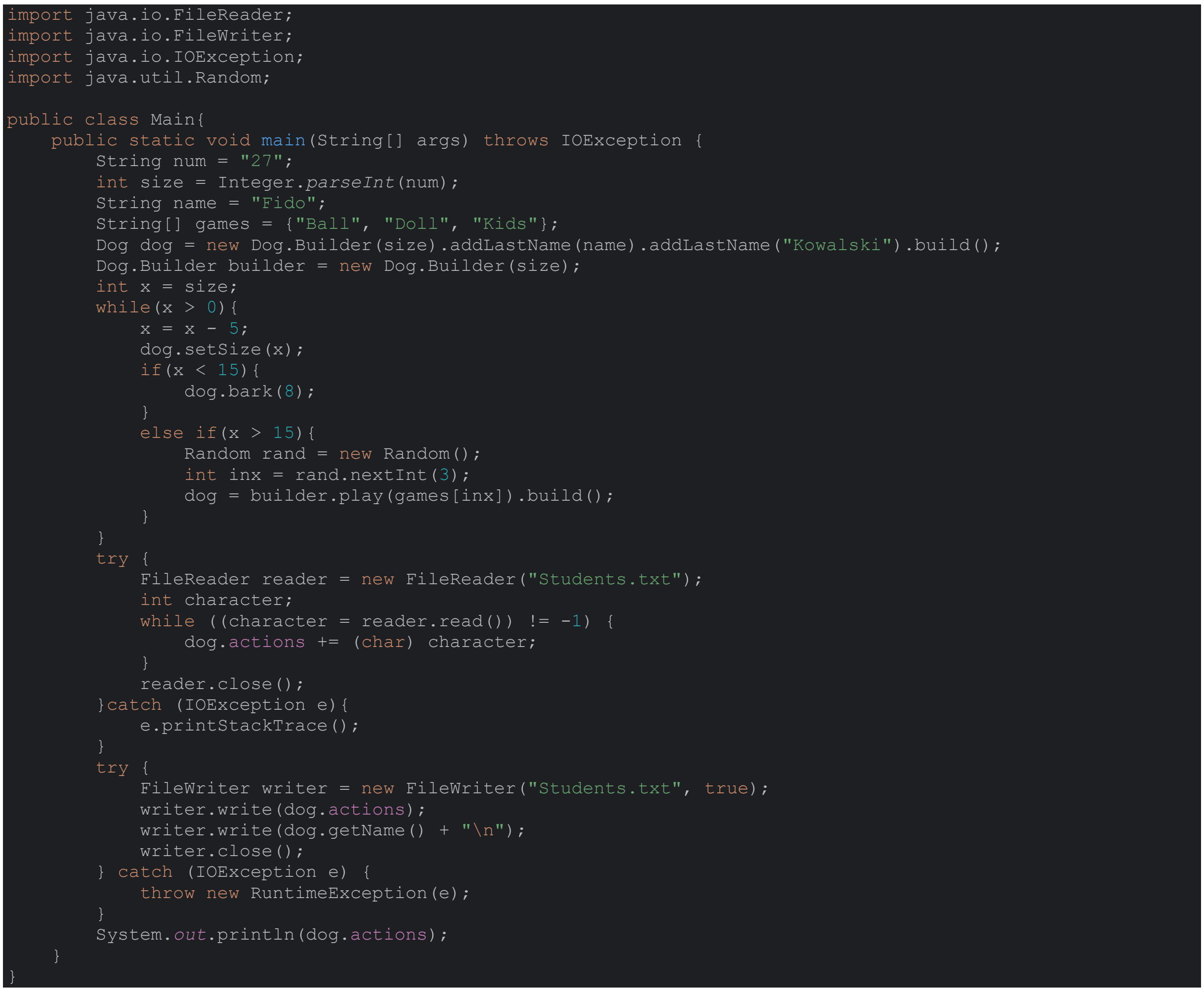
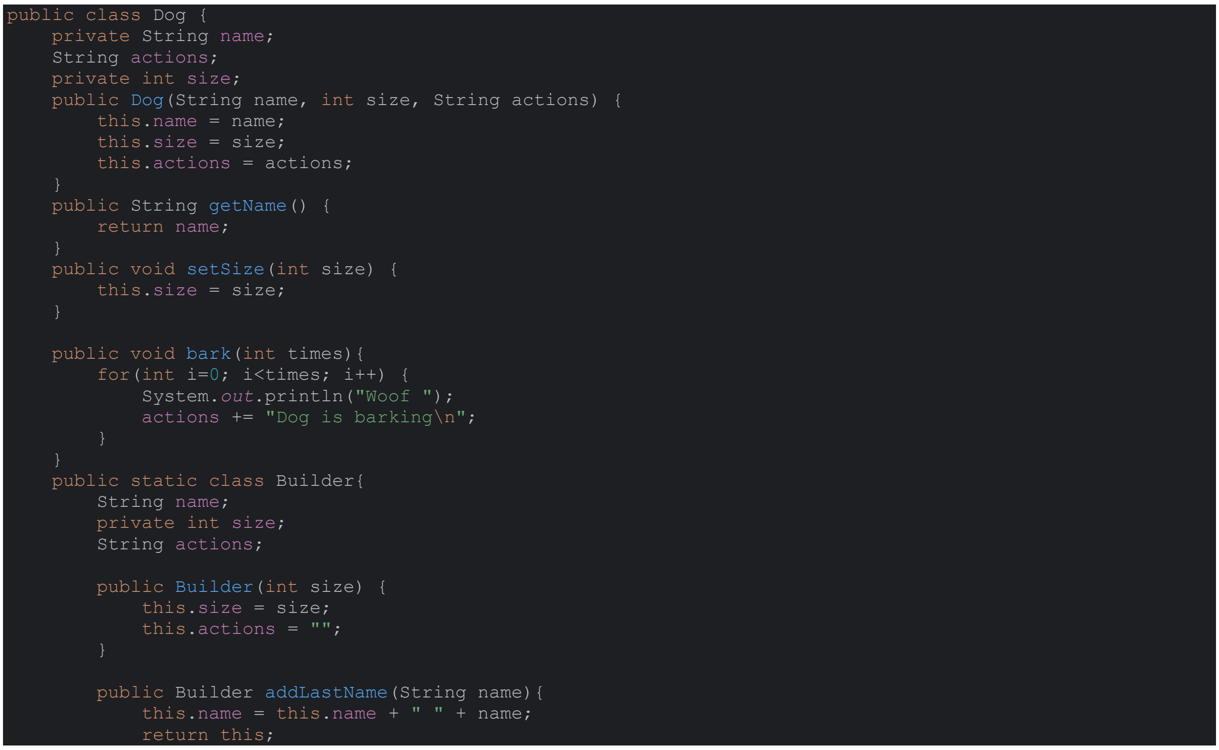
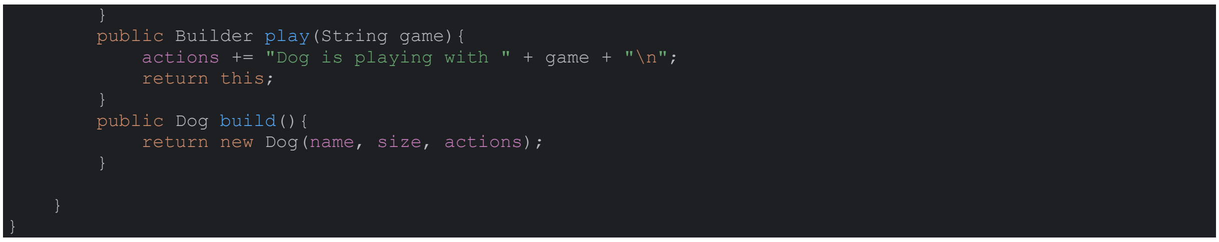
Basic Read/Write to files, random, String to Int, Builder



```java
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Random;

public class Main{
    public static void main(String[] args) throws IOException {
        String num = "27";
        int size = Integer.parseInt(num);
        String name = "Fido";
        String[] games = {"Ball", "Doll", "Kids"};
        Dog dog = new Dog.Builder(size).addLastName(name).addLastName("Kowalski").build();
        Dog.Builder builder = new Dog.Builder(size);
        int x = size;
        while(x > 0){
            x = x - 5;
            dog.setSize(x);
            if(x < 15){
                dog.bark(8);
            }
            else if(x > 15){
                Random rand = new Random();
                int inx = rand.nextInt(3);
                dog = builder.play(games[inx]).build();
            }
        }
        try {
            FileReader reader = new FileReader("Students.txt");
            int character;
            while ((character = reader.read()) != -1) {
                dog.actions += (char) character;
            }
            reader.close();
        }catch (IOException e){
            e.printStackTrace();
        }
        try {
            FileWriter writer = new FileWriter("Students.txt", true);
            writer.write(dog.actions);
            writer.write(dog.getName() + "\n");
            writer.close();
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
        System.out.println(dog.actions);
    }
}
```

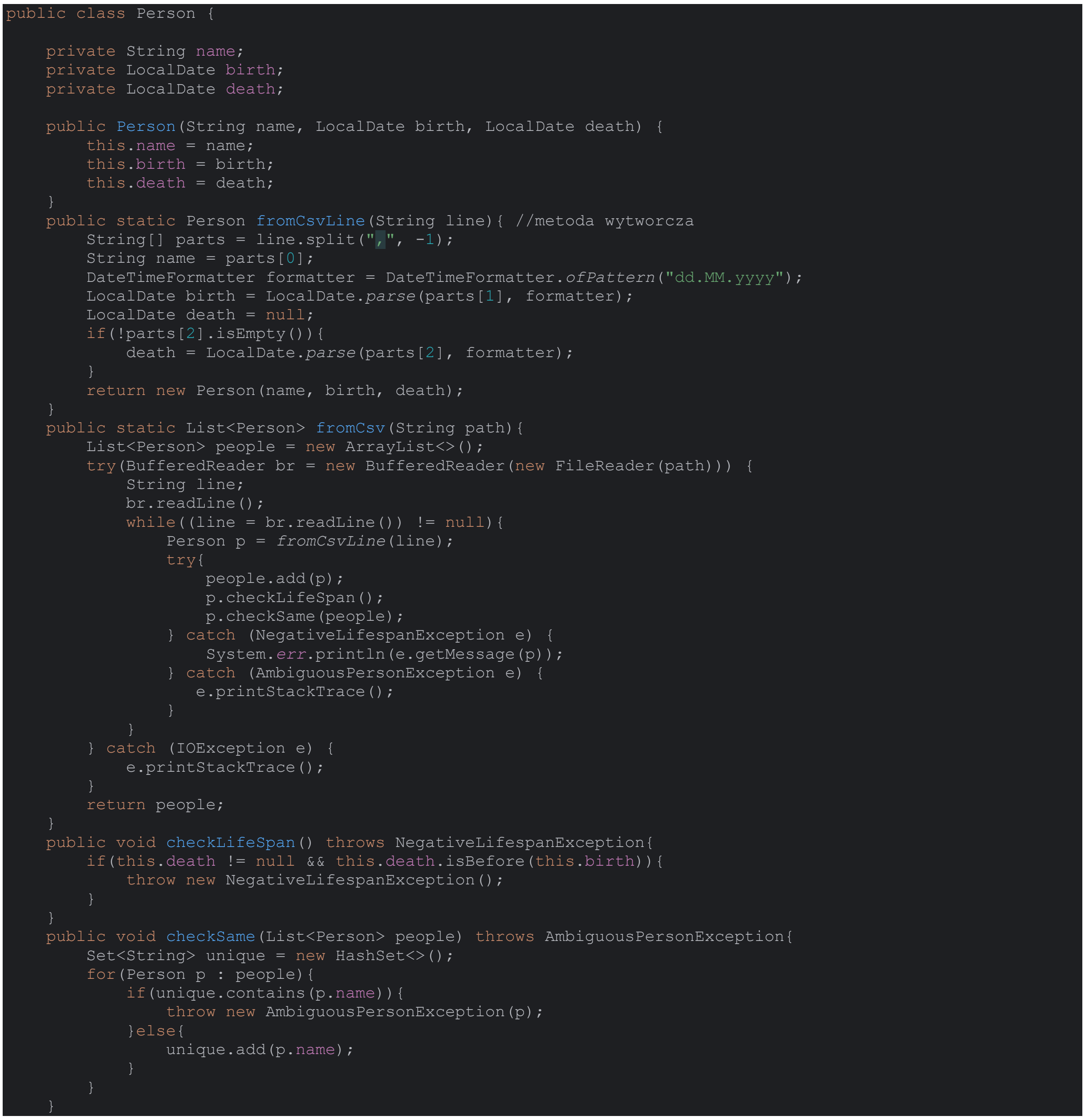


```java
public class Dog {
    private String name;
    String actions;
    private int size;
    public Dog(String name, int size, String actions) {
        this.name = name;
        this.size = size;
        this.actions = actions;
    }
    public String getName() {
        return name;
    }
    public void setSize(int size) {
        this.size = size;
    }

    public void bark(int times){
        for(int i=0; i<times; i++) {
            System.out.println("Woof ");
            actions += "Dog is barking\n";
        }
    }
    public static class Builder{
        String name;
        private int size;
        String actions;

        public Builder(int size) {
            this.size = size;
            this.actions = "";
        }

        public Builder addLastName(String name){
            this.name = this.name + " " + name;
            return this;
```

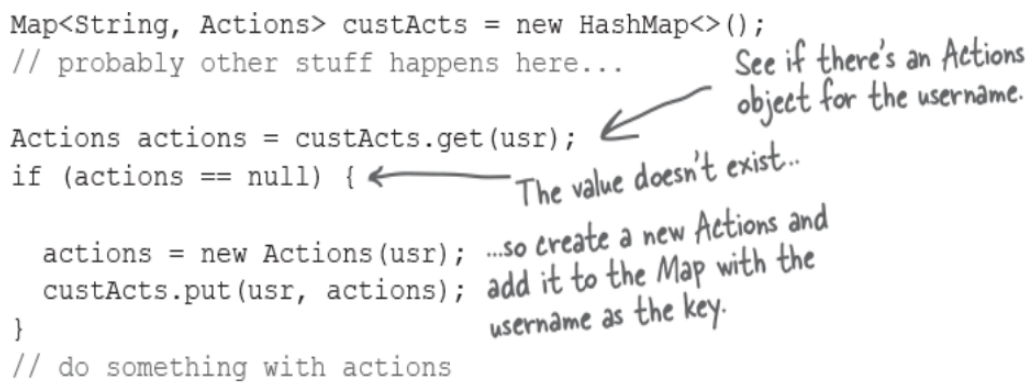


```
        }
        public Builder play(String game){
            actions += "Dog is playing with " + game + "\n";
            return this;
        }
        public Dog build(){
            return new Dog(name, size, actions);
        }

    }
}
```

Exceptions, Sets(only unique values, values not in order), File Reader, metoda wytwórcza



```
public class Person {

    private String name;
    private LocalDate birth;
    private LocalDate death;

    public Person(String name, LocalDate birth, LocalDate death) {
        this.name = name;
        this.birth = birth;
        this.death = death;
    }
    public static Person fromCsvLine(String line){ //metoda wytworcza
        String[] parts = line.split(",", -1);
        String name = parts[0];
        DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd.MM.yyyy");
        LocalDate birth = LocalDate.parse(parts[1], formatter);
        LocalDate death = null;
        if(!parts[2].isEmpty()){
            death = LocalDate.parse(parts[2], formatter);
        }
        return new Person(name, birth, death);
    }
    public static List<Person> fromCsv(String path){
        List<Person> people = new ArrayList<>();
        try(BufferedReader br = new BufferedReader(new FileReader(path))) {
            String line;
            br.readLine();
            while((line = br.readLine()) != null){
                Person p = fromCsvLine(line);
                try{
                    people.add(p);
                    p.checkLifeSpan();
                    p.checkSame(people);
                } catch (NegativeLifespanException e) {
                    System.err.println(e.getMessage(p));
                } catch (AmbiguousPersonException e) {
                   e.printStackTrace();
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
        return people;
    }
    public void checkLifeSpan() throws NegativeLifespanException{
        if(this.death != null && this.death.isBefore(this.birth)){
            throw new NegativeLifespanException();
        }
    }
    public void checkSame(List<Person> people) throws AmbiguousPersonException{
        Set<String> unique = new HashSet<>();
        for(Person p : people){
            if(unique.contains(p.name)){
                throw new AmbiguousPersonException(p);
            }else{
                unique.add(p.name);
            }
        }
    }
```



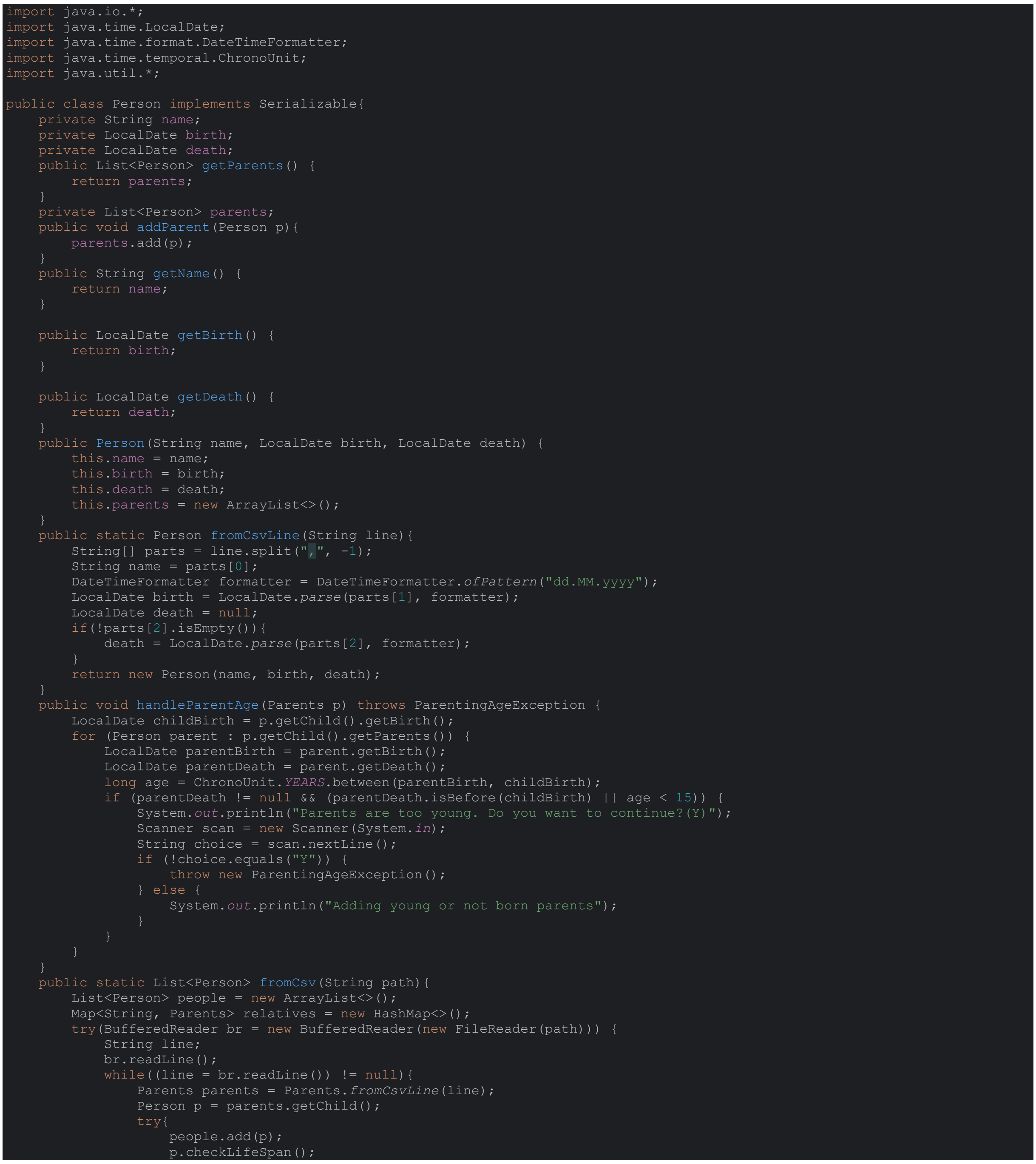
```
Map<String, Actions> custActs = new HashMap<>();
// probably other stuff happens here...

Actions actions = custActs.get(usr);
if (actions == null) {

  actions = new Actions(usr);
  custActs.put(usr, actions);
}
// do something with actions
```

See if there's an Actions object for the username.

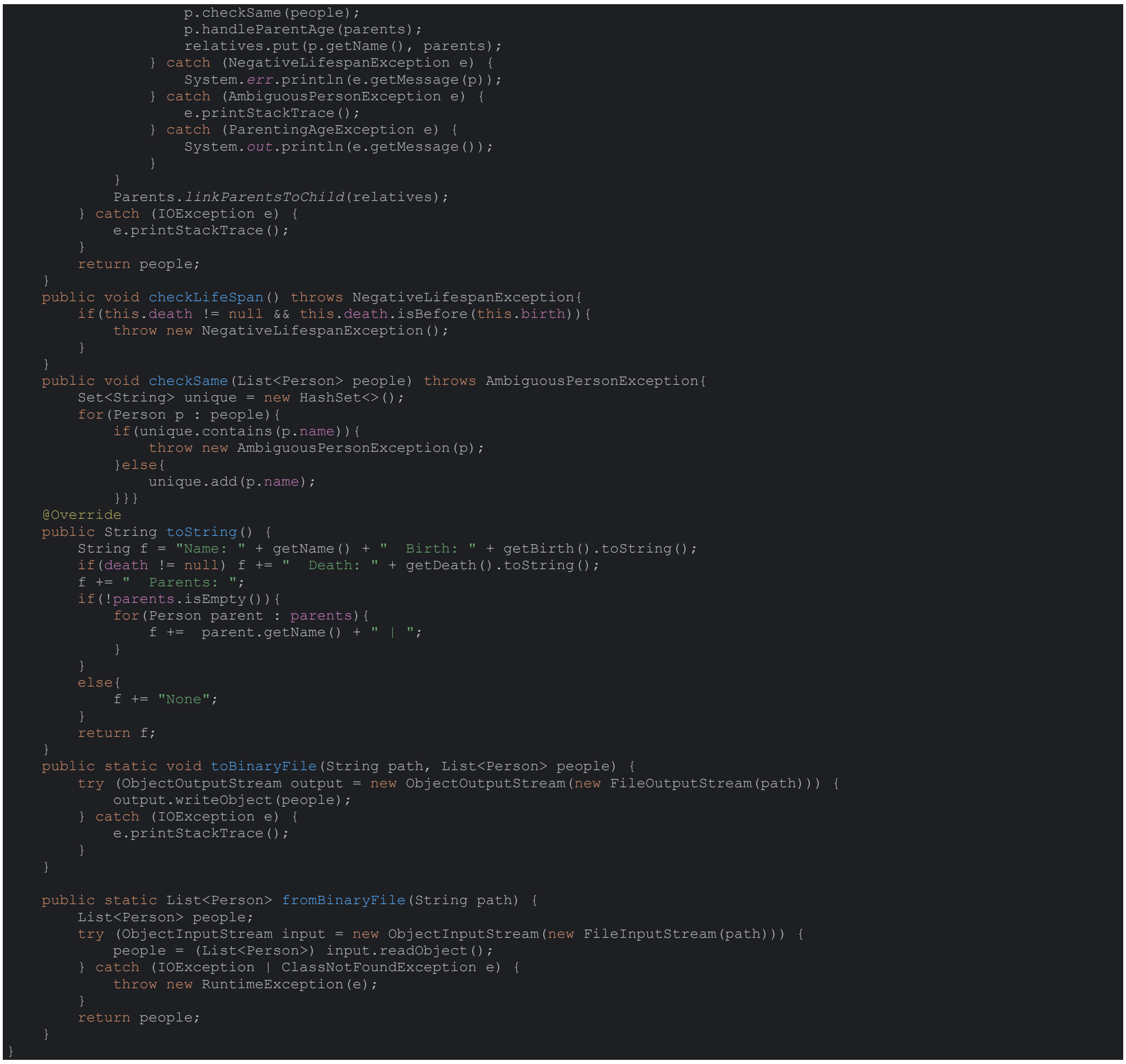
The value doesn't exist...

...so create a new Actions and add it to the Map with the username as the key.

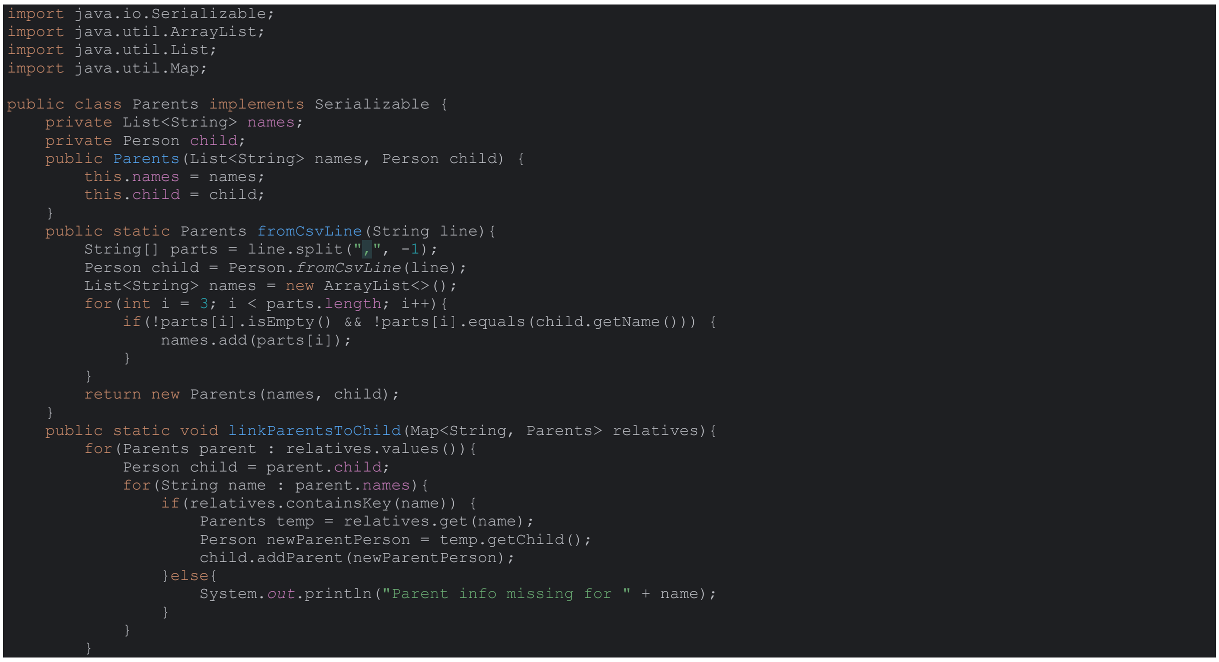
MAPY, PLIKI BINARNE WYJATKI



```
import java.io.*;
import java.time.LocalDate;
import java.time.format.DateTimeFormatter;
import java.time.temporal.ChronoUnit;
import java.util.*;

public class Person implements Serializable{
    private String name;
    private LocalDate birth;
    private LocalDate death;
    public List<Person> getParents() {
        return parents;
    }
    private List<Person> parents;
    public void addParent(Person p){
        parents.add(p);
    }
    public String getName() {
        return name;
    }

    public LocalDate getBirth() {
        return birth;
    }

    public LocalDate getDeath() {
        return death;
    }
    public Person(String name, LocalDate birth, LocalDate death) {
        this.name = name;
        this.birth = birth;
        this.death = death;
        this.parents = new ArrayList<>();
    }
    public static Person fromCsvLine(String line){
        String[] parts = line.split(",", -1);
        String name = parts[0];
        DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd.MM.yyyy");
        LocalDate birth = LocalDate.parse(parts[1], formatter);
        LocalDate death = null;
        if(!parts[2].isEmpty()){
            death = LocalDate.parse(parts[2], formatter);
        }
        return new Person(name, birth, death);
    }
    public void handleParentAge(Parents p) throws ParentingAgeException {
        LocalDate childBirth = p.getChild().getBirth();
        for (Person parent : p.getChild().getParents()) {
            LocalDate parentBirth = parent.getBirth();
            LocalDate parentDeath = parent.getDeath();
            long age = ChronoUnit.YEARS.between(parentBirth, childBirth);
            if (parentDeath != null && (parentDeath.isBefore(childBirth) || age < 15)) {
                System.out.println("Parents are too young. Do you want to continue?(Y)");
                Scanner scan = new Scanner(System.in);
                String choice = scan.nextLine();
                if (!choice.equals("Y")) {
                    throw new ParentingAgeException();
                } else {
                    System.out.println("Adding young or not born parents");
                }
            }
        }
    }
    public static List<Person> fromCsv(String path){
        List<Person> people = new ArrayList<>();
        Map<String, Parents> relatives = new HashMap<>();
        try(BufferedReader br = new BufferedReader(new FileReader(path))) {
            String line;
            br.readLine();
            while((line = br.readLine()) != null){
                Parents parents = Parents.fromCsvLine(line);
                Person p = parents.getChild();
                try{
                    people.add(p);
                    p.checkLifeSpan();
```



```java
                    p.checkSame(people);
                    p.handleParentAge(parents);
                    relatives.put(p.getName(), parents);
                } catch (NegativeLifespanException e) {
                    System.err.println(e.getMessage(p));
                } catch (AmbiguousPersonException e) {
                    e.printStackTrace();
                } catch (ParentingAgeException e) {
                    System.out.println(e.getMessage());
                }
            }
            Parents.linkParentsToChild(relatives);
        } catch (IOException e) {
            e.printStackTrace();
        }
        return people;
    }
    public void checkLifeSpan() throws NegativeLifespanException{
        if(this.death != null && this.death.isBefore(this.birth)){
            throw new NegativeLifespanException();
        }
    }
    public void checkSame(List<Person> people) throws AmbiguousPersonException{
        Set<String> unique = new HashSet<>();
        for(Person p : people){
            if(unique.contains(p.name)){
                throw new AmbiguousPersonException(p);
            }else{
                unique.add(p.name);
            }}}
    @Override
    public String toString() {
        String f = "Name: " + getName() + "  Birth: " + getBirth().toString();
        if(death != null) f += "  Death: " + getDeath().toString();
        f += "  Parents: ";
        if(!parents.isEmpty()){
            for(Person parent : parents){
                f +=  parent.getName() + " | ";
            }
        }
        else{
            f += "None";
        }
        return f;
    }
    public static void toBinaryFile(String path, List<Person> people) {
        try (ObjectOutputStream output = new ObjectOutputStream(new FileOutputStream(path))) {
            output.writeObject(people);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public static List<Person> fromBinaryFile(String path) {
        List<Person> people;
        try (ObjectInputStream input = new ObjectInputStream(new FileInputStream(path))) {
            people = (List<Person>) input.readObject();
        } catch (IOException | ClassNotFoundException e) {
            throw new RuntimeException(e);
        }
        return people;
    }
}
```



```java
import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;
import java.util.Map;

public class Parents implements Serializable {
    private List<String> names;
    private Person child;
    public Parents(List<String> names, Person child) {
        this.names = names;
        this.child = child;
    }
    public static Parents fromCsvLine(String line){
        String[] parts = line.split(",", -1);
        Person child = Person.fromCsvLine(line);
        List<String> names = new ArrayList<>();
        for(int i = 3; i < parts.length; i++){
            if(!parts[i].isEmpty() && !parts[i].equals(child.getName())) {
                names.add(parts[i]);
            }
        }
        return new Parents(names, child);
    }
    public static void linkParentsToChild(Map<String, Parents> relatives){
        for(Parents parent : relatives.values()){
            Person child = parent.child;
            for(String name : parent.names){
                if(relatives.containsKey(name)) {
                    Parents temp = relatives.get(name);
                    Person newParentPerson = temp.getChild();
                    child.addParent(newParentPerson);
                }else{
                    System.out.println("Parent info missing for " + name);
                }
            }
        }
    }
    public Person getChild() {
        return child;
    }

}
```

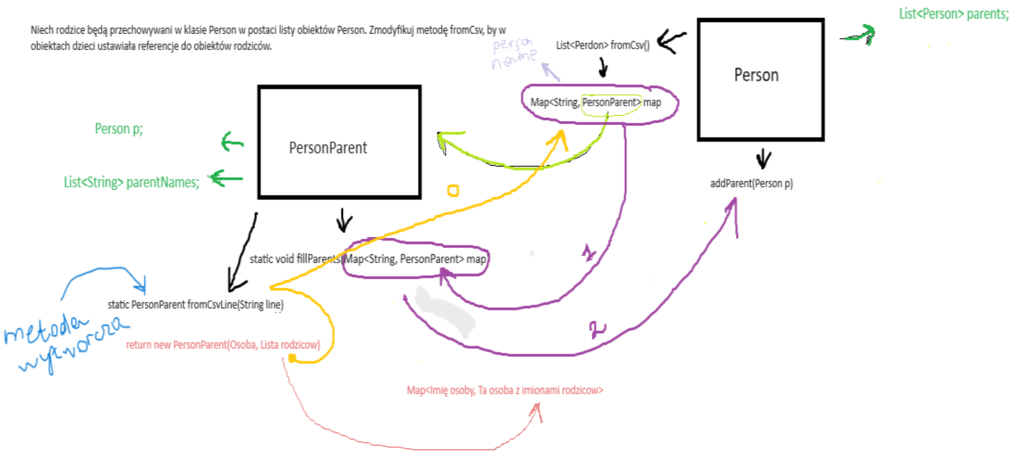
Code analize

Binary Files(use need to implement Serializable interface)

```
public static void toBinaryFile(String path, List<Person> people) {
    try (ObjectOutputStream output = new ObjectOutputStream(new FileOutputStream(path))) {
        output.writeObject(people);
    } catch (IOException e) {
        e.printStackTrace();
    }
}

public static List<Person> fromBinaryFile(String path) {
    List<Person> people;
    try (ObjectInputStream input = new ObjectInputStream(new FileInputStream(path))) {
        people = (List<Person>) input.readObject();
    } catch (IOException | ClassNotFoundException e) {
        throw new RuntimeException(e);
    }
    return people;
}
```

```
public class Person implements Serializable{
```

Maps Diagram



Niech rodzice będą przechowywani w klasie Person w postaci listy obiektów Person. Zmodyfikuj metodę fromCsv, by w obiektach dzieci ustawiała referencje do obiektów rodziców.

List<Perdon> fromCsv()
person mozne
Map<String, PersonParent> map
Person
List<Person> parents;
addParent(Person p)
PersonParent
Person p;
List<String> parentNames;
static void fillParents(Map<String, PersonParent> map)
static PersonParent fromCsvLine(String line)
return new PersonParent(Osoba, Lista rodzicow)
metoda wytworcza
1
2
Map<Imię osoby, Ta osoba z imionami rodzicow>

Maps Steps

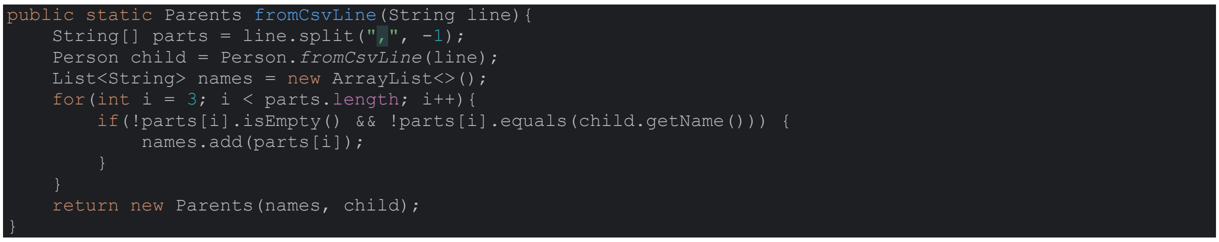
Step 1

```
public List<Person> getParents() {
    return parents;
}
private List<Person> parents;
```

Step 2

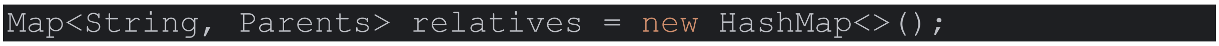
```
public class Parents implements Serializable {
    private List<String> names;
    private Person child;
    public Parents(List<String> names, Person child) {
        this.names = names;
        this.child = child;
    }
```

Step 3 Returns person and parents name from line(like in Person class)



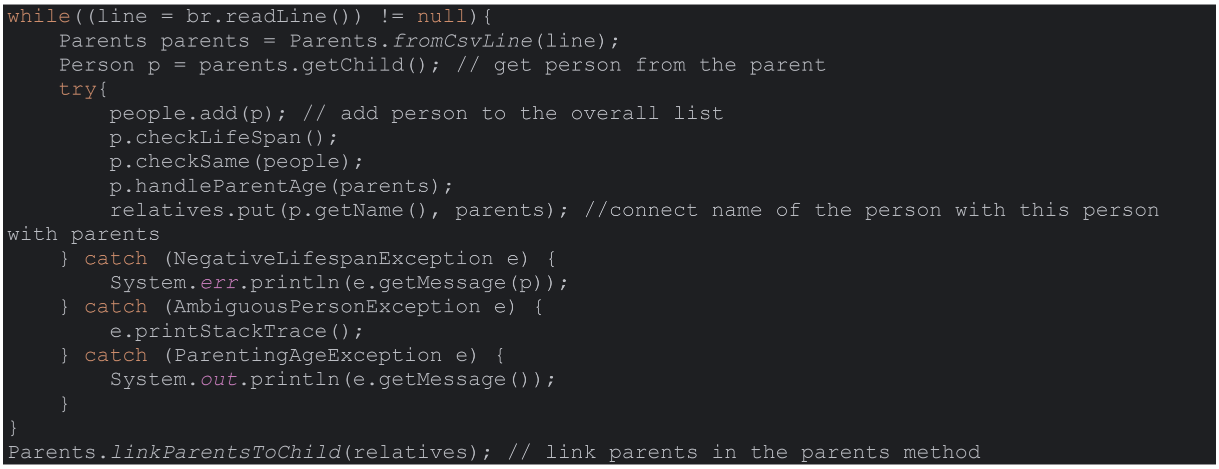
```java
public static Parents fromCsvLine(String line){
    String[] parts = line.split(",", -1);
    Person child = Person.fromCsvLine(line);
    List<String> names = new ArrayList<>();
    for(int i = 3; i < parts.length; i++){
        if(!parts[i].isEmpty() && !parts[i].equals(child.getName())) {
            names.add(parts[i]);
        }
    }
    return new Parents(names, child);
}
```

Step 4 (Create map in person from File method)



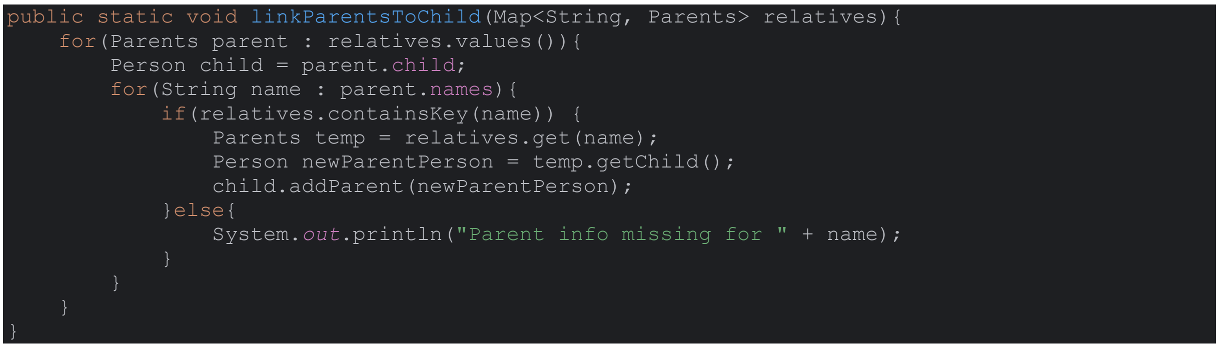
```java
Map<String, Parents> relatives = new HashMap<>();
```

Add person



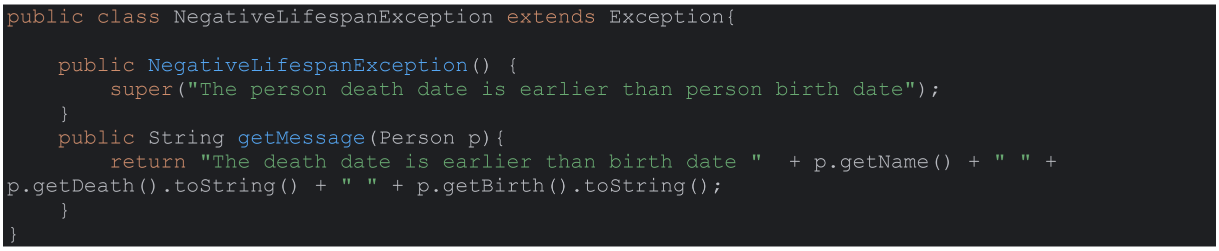
```java
while((line = br.readLine()) != null){
    Parents parents = Parents.fromCsvLine(line);
    Person p = parents.getChild(); // get person from the parent
    try{
        people.add(p); // add person to the overall list
        p.checkLifeSpan();
        p.checkSame(people);
        p.handleParentAge(parents);
        relatives.put(p.getName(), parents); //connect name of the person with this person
with parents
    } catch (NegativeLifespanException e) {
        System.err.println(e.getMessage(p));
    } catch (AmbiguousPersonException e) {
        e.printStackTrace();
    } catch (ParentingAgeException e) {
        System.out.println(e.getMessage());
    }
}
Parents.linkParentsToChild(relatives); // link parents in the parents method
```

Step 5 (Link person with parents and add Parent to this person with static method addParent)



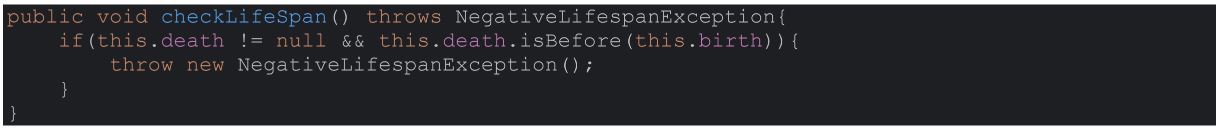
```java
public static void linkParentsToChild(Map<String, Parents> relatives){
    for(Parents parent : relatives.values()){
        Person child = parent.child;
        for(String name : parent.names){
            if(relatives.containsKey(name)) {
                Parents temp = relatives.get(name);
                Person newParentPerson = temp.getChild();
                child.addParent(newParentPerson);
            }else{
                System.out.println("Parent info missing for " + name);
            }
        }
    }
}
```

Exception example

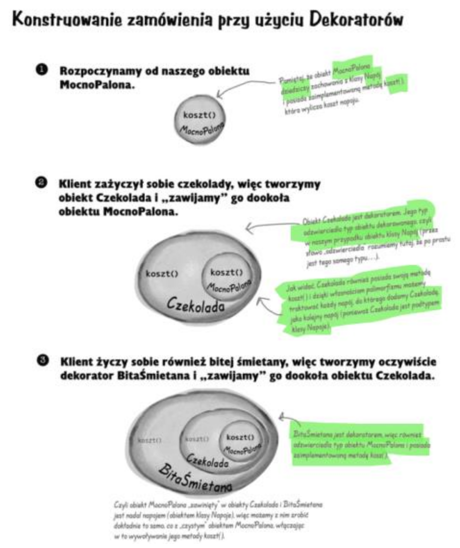


```java
public class NegativeLifespanException extends Exception{

    public NegativeLifespanException() {
        super("The person death date is earlier than person birth date");
    }
    public String getMessage(Person p){
        return "The death date is earlier than birth date "  + p.getName() + " " +
p.getDeath().toString() + " " + p.getBirth().toString();
    }
}
```

Implementation of this exception



```java
public void checkLifeSpan() throws NegativeLifespanException{
    if(this.death != null && this.death.isBefore(this.birth)){
        throw new NegativeLifespanException();
    }
}
```



Konstruowanie zamówienia przy użyciu Dekoratorów

❶ **Rozpoczynamy od naszego obiektu MocnoPalona.**

Pamiętaj, że obiekt MocnoPalona dziedziczy zachowania z klasy Napój i posiada zaimplementowaną metodę koszt(), która wylicza koszt napoju.

koszt()
MocnoPalona

❷ **Klient zażyczył sobie czekolady, więc tworzymy obiekt Czekolada i „zawijamy" go dookoła obiektu MocnoPalona.**

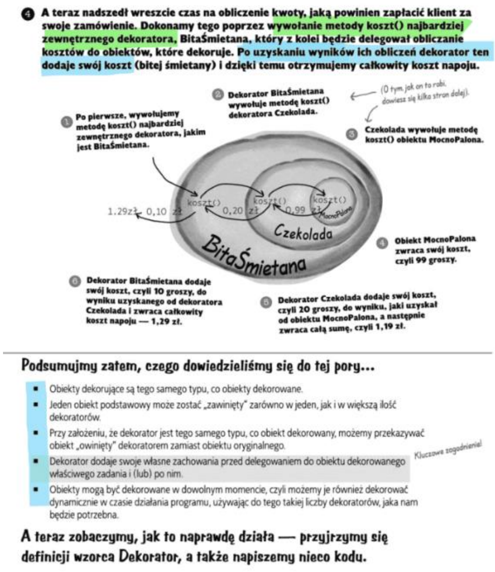
Obiekt Czekolada jest dekoratorem. Jego typ odzwierciedla typ obiektu dekorowanego, czyli w naszym przypadku obiektu klasy Napój (przez słowo „odzwierciedla" rozumiemy tutaj, że po prostu jest tego samego typu…).

Jak widać, Czekolada również posiada swoją metodę koszt() i dzięki właściwościom polimorfizmu możemy traktować każdy napój, do którego dodamy Czekoladę, jako kolejny napój (ponieważ Czekolada jest podtypem klasy Napoje).

❸ **Klient życzy sobie również bitej śmietany, więc tworzymy oczywiście dekorator BitaŚmietana i „zawijamy" go dookoła obiektu Czekolada.**

BitaŚmietana jest dekoratorem, więc również odzwierciedla typ obiektu MocnoPalona i posiada zaimplementowaną metodę koszt().

Czyli obiekt MocnoPalona „zawinięty" w obiekty Czekolada i BitaŚmietana jest nadal napojem (obiektem klasy Napoje), więc możemy z nim zrobić dokładnie to samo, co z „czystym" obiektem MocnoPalona, włączając w to wywoływanie jego metody koszt().



❹ **A teraz nadszedł wreszcie czas na obliczenie kwoty, jaką powinien zapłacić klient za swoje zamówienie. Dokonamy tego poprzez wywołanie metody koszt() najbardziej zewnętrznego dekoratora, BitaŚmietana, który z kolei będzie delegował obliczanie kosztów do obiektów, które dekoruje. Po uzyskaniu wyników ich obliczeń dekorator ten dodaje swój koszt (bitej śmietany) i dzięki temu otrzymujemy całkowity koszt napoju.**

① **Po pierwsze, wywołujemy metodę koszt() najbardziej zewnętrznego dekoratora, jakim jest BitaŚmietana.**

② **Dekorator BitaŚmietana wywołuje metodę koszt() dekoratora Czekolada.**

(O tym, jak on to robi, dowiesz się kilka stron dalej).

③ **Czekolada wywołuje metodę koszt() obiektu MocnoPalona.**

④ **Obiekt MocnoPalona zwraca swój koszt, czyli 99 groszy.**

⑤ **Dekorator Czekolada dodaje swój koszt, czyli 20 groszy, do wyniku, jaki uzyskał od obiektu MocnoPalona, a następnie zwraca całą sumę, czyli 1,19 zł.**

⑥ **Dekorator BitaŚmietana dodaje swój koszt, czyli 10 groszy, do wyniku uzyskanego od dekoratora Czekolada i zwraca całkowity koszt napoju — 1,29 zł.**

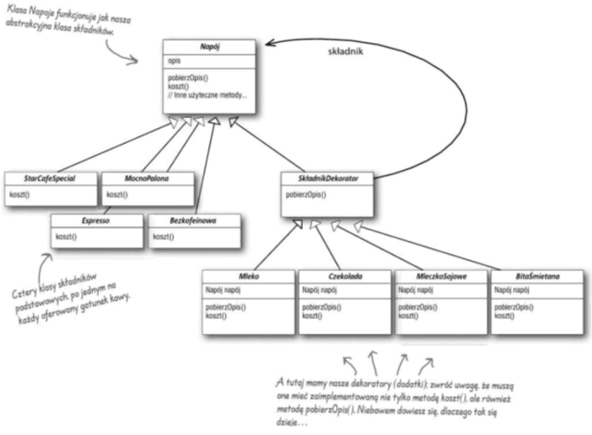
Podsumujmy zatem, czego dowiedzieliśmy się do tej pory…

- Obiekty dekorujące są tego samego typu, co obiekty dekorowane.
- Jeden obiekt podstawowy może zostać „zawinięty" zarówno w jeden, jak i w większą ilość dekoratorów.
- Przy założeniu, że dekorator jest tego samego typu, co obiekt dekorowany, możemy przekazywać obiekt „owinięty" dekoratorem zamiast obiektu oryginalnego.
- Dekorator dodaje swoje własne zachowania przed delegowaniem do obiektu dekorowanego właściwego zadania i (lub) po nim.
- Obiekty mogą być dekorowane w dowolnym momencie, czyli możemy je również dekorować dynamicznie w czasie działania programu, używając do tego takiej liczby dekoratorów, jaka nam będzie potrzebna.

Kluczowe zagadnienie!

A teraz zobaczymy, jak to naprawdę działa — przyjrzymy się definicji wzorca Dekorator, a także napiszemy nieco kodu.

Dekoratory



Klasa Napoje funkcjonuje jak nasza abstrakcyjna klasa składników.

Cztery klasy składników podstawowych, po jednym na każdy oferowany gatunek kawy.

A tutaj mamy nasze dekoratory (dodatki); zwróć uwagę, że muszą one mieć zaimplementowaną nie tylko metodę koszt(), ale również metodę pobierzOpis(). Niebawem dowiesz się, dlaczego tak się dzieje…

1. Create an interface.
2. Create concrete classes implementing the same interface.
3. Create an abstract decorator class implementing the above same interface.
4. Create a concrete decorator class extending the above abstract decorator class.

```
public interface Book {
    public String description();
    public double calculateCost();
}
```

```
public abstract class BookDecorator implements Book{
    protected Book book;

    public BookDecorator(Book book) {
        this.book = book;
    }
}
```

```
public class HardCoverDecorator extends BookDecorator{

    public HardCoverDecorator(Book book) {
        super(book);
    }
    @Override
    public String description() {
        return book.description() + "," + "hard cover,";
    }
    @Override
    public double calculateCost() {
        return book.calculateCost() + 15;
    }
}
```

```
public class BasicBook implements Book{
    protected String title;
    protected LocalDate date;
    protected int pages;
    protected List<BasicBook> sameAuthor;

    public BasicBook(String title, LocalDate date, int pages) {
        this.title = title;
        this.date = date;
        this.pages = pages;
        this.sameAuthor = new ArrayList<>();
    }
    @Override
    public String description() {
        return this.title + "," + this.date.toString() + "," + this.pages + "," +
this.sameAuthor.toString();
    }
    @Override
    public double calculateCost() {
        return 20;
    }
}
  public class Horror extends BasicBook{
                              public Horror(String title, LocalDate date, int pages)
{
                              super(title, date, pages);
            }
```

Main

```
      public static void main(String[] args) {
    Book book1 = new Horror("Dark tower", LocalDate.of(1999, 5, 12), 635);
    book1 = new HardCoverDecorator(book1);
    System.out.println(book1.description() + " " + book1.calculateCost());
```

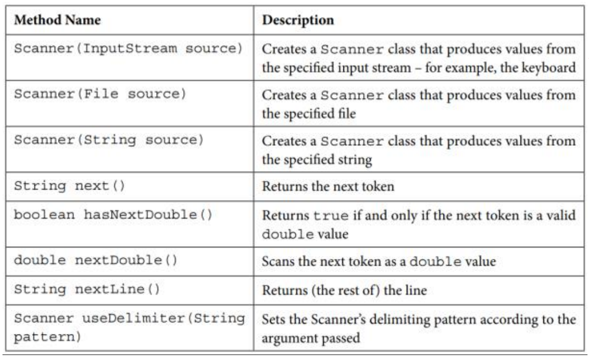
Int to String

```
int intValue = 123;
String strValue = String.valueOf(intValue);
```

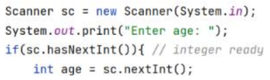
String to int

```
String str = "123";
int intValue = Integer.parseInt(str);
```

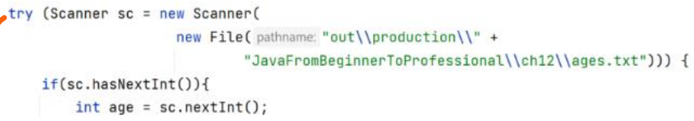
Scanner class (don’t close a scanner only when System.in)



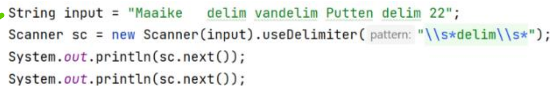
Method Name	Description
Scanner(InputStream source)	Creates a Scanner class that produces values from the specified input stream – for example, the keyboard
Scanner(File source)	Creates a Scanner class that produces values from the specified file
Scanner(String source)	Creates a Scanner class that produces values from the specified string
String next()	Returns the next token
boolean hasNextDouble()	Returns true if and only if the next token is a valid double value
double nextDouble()	Scans the next token as a double value
String nextLine()	Returns (the rest of) the line
Scanner useDelimiter(String pattern)	Sets the Scanner’s delimiting pattern according to the argument passed



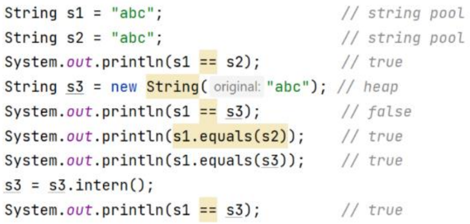
```
Scanner sc = new Scanner(System.in);
System.out.print("Enter age: ");
if(sc.hasNextInt()){ // integer ready
    int age = sc.nextInt();
```



```
try (Scanner sc = new Scanner(
                        new File( pathname: "out\\production\\" +
                                "JavaFromBeginnerToProfessional\\ch12\\ages.txt"))) {
    if(sc.hasNextInt()){
        int age = sc.nextInt();
```

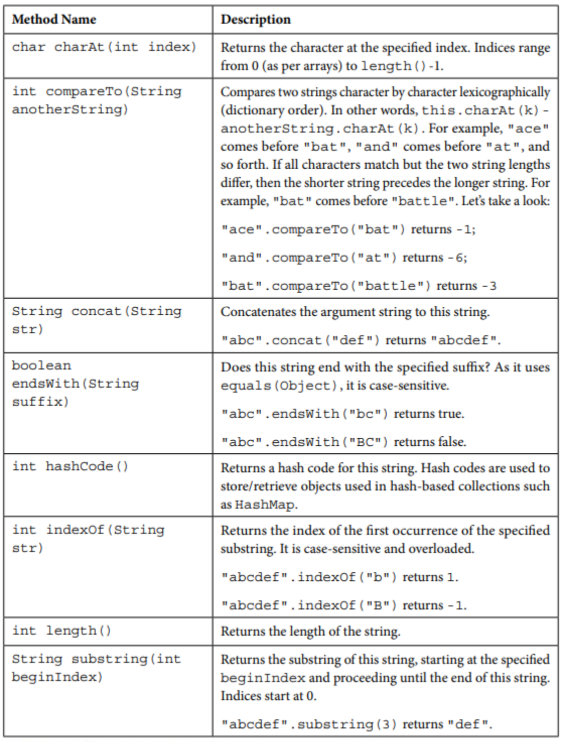


```
String input = "Maaike   delim vandelim Putten delim 22";
Scanner sc = new Scanner(input).useDelimiter( pattern: "\\s*delim\\s*");
System.out.println(sc.next());
System.out.println(sc.next());
```



```
String s1 = "abc";                              // string pool
String s2 = "abc";                              // string pool
System.out.println(s1 == s2);                   // true
String s3 = new String( original: "abc"); // heap
System.out.println(s1 == s3);                   // false
System.out.println(s1.equals(s2));              // true
System.out.println(s1.equals(s3));              // true
s3 = s3.intern();
System.out.println(s1 == s3);                   // true
```

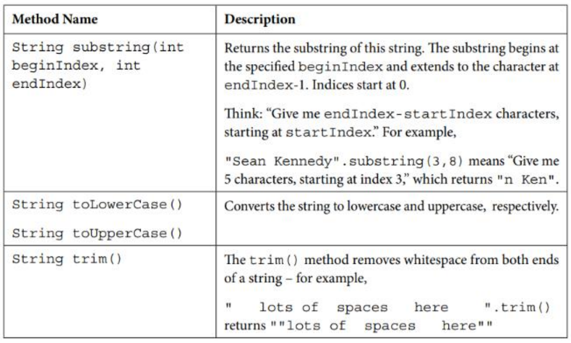
The \s* regular expression translates into 0 or more whitespace characters. * represents 0 or more and \s represents a single whitespace character. The delim string is hardcoded. This means that the input tokens are delimited by 0 or more spaces, followed by the delim token, followed by 0 or more spaces.



Method Name	Description
`char charAt(int index)`	Returns the character at the specified index. Indices range from 0 (as per arrays) to `length()`-1.

`int compareTo(String anotherString)`	Compares two strings character by character lexicographically (dictionary order). In other words, `this.charAt(k)` - `anotherString.charAt(k)`. For example, `"ace"` comes before `"bat"`, `"and"` comes before `"at"`, and so forth. If all characters match but the two string lengths differ, then the shorter string precedes the longer string. For example, `"bat"` comes before `"battle"`. Let's take a look:
	`"ace".compareTo("bat")` returns -1;
	`"and".compareTo("at")` returns -6;
	`"bat".compareTo("battle")` returns -3

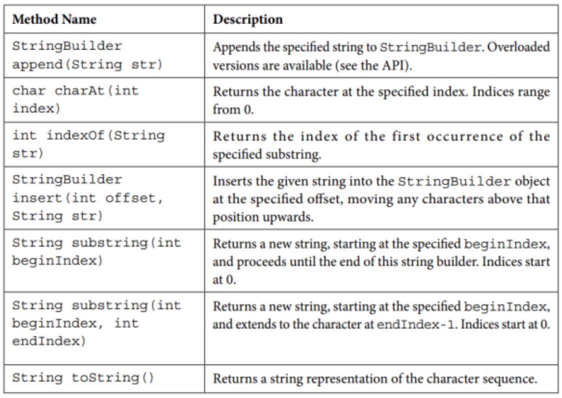
`String concat(String str)`	Concatenates the argument string to this string. `"abc".concat("def")` returns `"abcdef"`.
`boolean endsWith(String suffix)`	Does this string end with the specified suffix? As it uses `equals(Object)`, it is case-sensitive. `"abc".endsWith("bc")` returns true. `"abc".endsWith("BC")` returns false.
`int hashCode()`	Returns a hash code for this string. Hash codes are used to store/retrieve objects used in hash-based collections such as `HashMap`.
`int indexOf(String str)`	Returns the index of the first occurrence of the specified substring. It is case-sensitive and overloaded. `"abcdef".indexOf("b")` returns 1. `"abcdef".indexOf("B")` returns -1.
`int length()`	Returns the length of the string.
`String substring(int beginIndex)`	Returns the substring of this string, starting at the specified `beginIndex` and proceeding until the end of this string. Indices start at 0. `"abcdef".substring(3)` returns `"def"`.



Method Name	Description
`String substring(int beginIndex, int endIndex)`	Returns the substring of this string. The substring begins at the specified `beginIndex` and extends to the character at `endIndex`-1. Indices start at 0. Think: "Give me `endIndex-startIndex` characters, starting at `startIndex`." For example, `"Sean Kennedy".substring(3,8)` means "Give me 5 characters, starting at index 3," which returns `"n Ken"`.

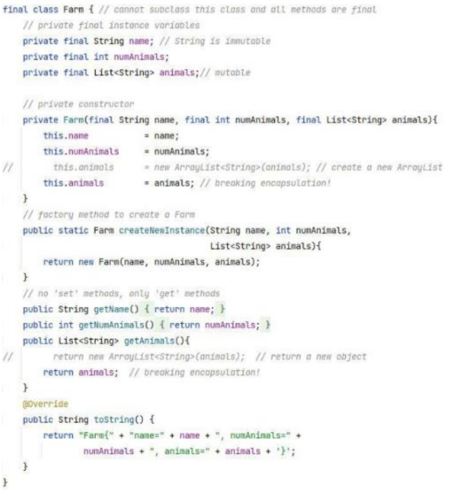
`String toLowerCase()`
`String toUpperCase()`
	Converts the string to lowercase and uppercase, respectively.

`String trim()`	The `trim()` method removes whitespace from both ends of a string – for example, `"   lots of   spaces   here    ".trim()` returns `""lots of   spaces   here""`

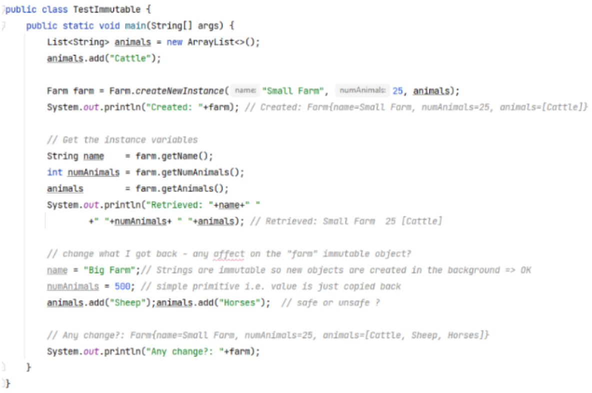


Method Name	Description
`StringBuilder append(String str)`	Appends the specified string to `StringBuilder`. Overloaded versions are available (see the API).
`char charAt(int index)`	Returns the character at the specified index. Indices range from 0.
`int indexOf(String str)`	Returns the index of the first occurrence of the specified substring.
`StringBuilder insert(int offset, String str)`	Inserts the given string into the `StringBuilder` object at the specified offset, moving any characters above that position upwards.
`String substring(int beginIndex)`	Returns a new string, starting at the specified `beginIndex`, and proceeds until the end of this string builder. Indices start at 0.
`String substring(int beginIndex, int endIndex)`	Returns a new string, starting at the specified `beginIndex`, and extends to the character at `endIndex-1`. Indices start at 0.
`String toString()`	Returns a string representation of the character sequence.

Immutable class(week)



```java
final class Farm { // cannot subclass this class and all methods are final
    // private final instance variables
    private final String name; // String is immutable
    private final int numAnimals;
    private final List<String> animals;// mutable

    // private constructor
    private Farm(final String name, final int numAnimals, final List<String> animals){
        this.name          = name;
        this.numAnimals    = numAnimals;
//        this.animals       = new ArrayList<String>(animals); // create a new ArrayList
        this.animals       = animals; // breaking encapsulation!
    }
    // factory method to create a Farm
    public static Farm createNewInstance(String name, int numAnimals,
                                         List<String> animals){
        return new Farm(name, numAnimals, animals);
    }
    // no 'set' methods, only 'get' methods
    public String getName() { return name; }
    public int getNumAnimals() { return numAnimals; }
    public List<String> getAnimals(){
//        return new ArrayList<String>(animals);  // return a new object
        return animals;  // breaking encapsulation!
    }
    @Override
    public String toString() {
        return "Farm{" + "name=" + name + ", numAnimals=" +
                numAnimals + ", animals=" + animals + '}';
    }
}
```



```java
public class TestImmutable {
    public static void main(String[] args) {
        List<String> animals = new ArrayList<>();
        animals.add("Cattle");

        Farm farm = Farm.createNewInstance( name: "Small Farm",  numAnimals: 25, animals);
        System.out.println("Created: "+farm); // Created: Farm{name=Small Farm, numAnimals=25, animals=[Cattle]}

        // Get the instance variables
        String name    = farm.getName();
        int numAnimals = farm.getNumAnimals();
        animals        = farm.getAnimals();
        System.out.println("Retrieved: "+name+" "
                +" "+numAnimals+ " "+animals); // Retrieved: Small Farm  25 [Cattle]

        // change what I got back - any affect on the "farm" immutable object?
        name = "Big Farm";// Strings are immutable so new objects are created in the background => OK
        numAnimals = 500; // simple primitive i.e. value is just copied back
        animals.add("Sheep");animals.add("Horses");  // safe or unsafe ?

        // Any change?: Farm{name=Small Farm, numAnimals=25, animals=[Cattle, Sheep, Horses]}
        System.out.println("Any change?: "+farm);
    }
}
```

The `static` block in Java is a block of code that is executed when the class is loaded into memory. It is typically used for performing initialization tasks that need to be done once, such as populating static fields or initializing static maps.

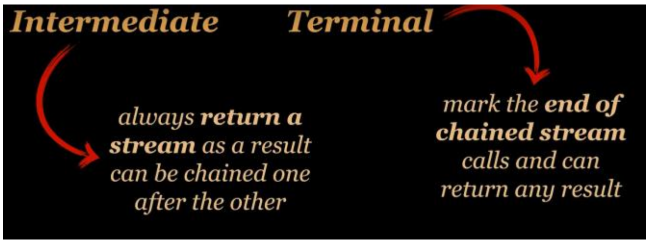
POTOKI

every country returns deaths

```java
@Override
public int getDeaths(LocalDate date) {
    return provinces.stream().mapToInt(p -> p.getDeaths(date)).sum();
}
```

```java
List<String> provinces = new ArrayList<>(reader.lines()
        .map(s -> s.split(";")[0]).toList());//reader.lines() creates a stream of lines from
the BufferedReader
```

```java
Product.addProducts(FoodProduct::fromCsv, //to use functional object(Function<takes, returns> )
Path.of("C:\\Users\\Yulia\\OneDrive\\Obrazy\\javaUMCS\\kolokwium1_2022\\kolokwium2022\\data\\food"));
```



Intermediate
Terminal
always return a stream as a result can be chained one after the other
mark the end of chained stream calls and can return any result

To create a stream:

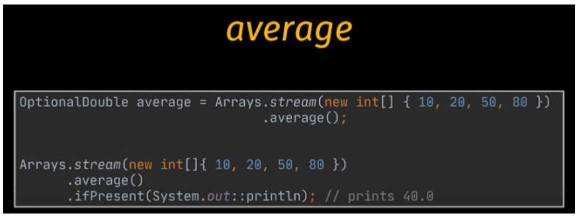
Stream<Type> stream = Stream.of(data)

Type array = stream.toArray(Type[]::new) – you need casting because by default it returns Object

Array.max(Comparator.comparing(b -> b.getPrice()).

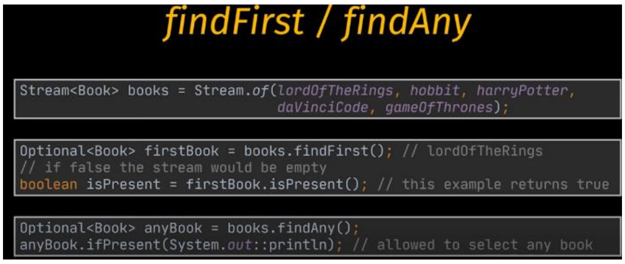
Returns max/min Object type(not value)

```
Optional<Author> a1 = usaAuthors.stream().max(Comparator.comparing(a -> a.getBirthYear()));

Author a2 = usaAuthors.stream().min(Comparator.comparing(a ->
((Author)a).getBirthYear()).reversed()).orElse(null);
```



average

```
OptionalDouble average = Arrays.stream(new int[] { 10, 20, 50, 80 })
                                .average();


Arrays.stream(new int[]{ 10, 20, 50, 80 })
      .average()
      .ifPresent(System.out::println); // prints 40.0
```

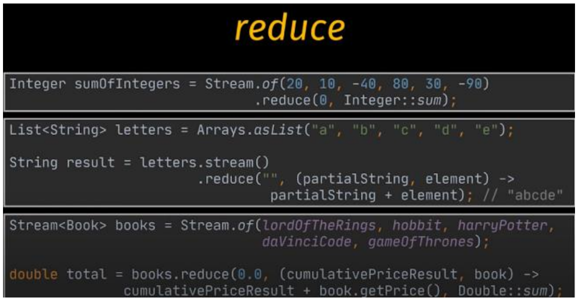


findFirst / findAny

```
Stream<Book> books = Stream.of(lordOfTheRings, hobbit, harryPotter,
                               daVinciCode, gameOfThrones);
```

```
Optional<Book> firstBook = books.findFirst(); // lordOfTheRings
// if false the stream would be empty
boolean isPresent = firstBook.isPresent(); // this example returns true
```

```
Optional<Book> anyBook = books.findAny();
anyBook.ifPresent(System.out::println); // allowed to select any book
```

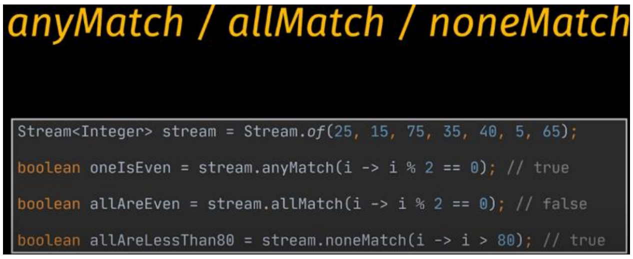


reduce

```
Integer sumOfIntegers = Stream.of(20, 10, -40, 80, 30, -90)
                              .reduce(0, Integer::sum);
```

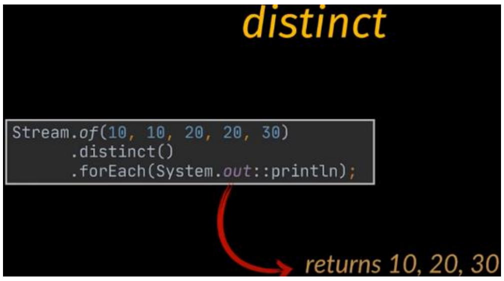
```
List<String> letters = Arrays.asList("a", "b", "c", "d", "e");

String result = letters.stream()
                       .reduce("", (partialString, element) ->
                               partialString + element); // "abcde"
```

```
Stream<Book> books = Stream.of(lordOfTheRings, hobbit, harryPotter,
                               daVinciCode, gameOfThrones);

double total = books.reduce(0.0, (cumulativePriceResult, book) ->
               cumulativePriceResult + book.getPrice(), Double::sum);
```



anyMatch / allMatch / noneMatch

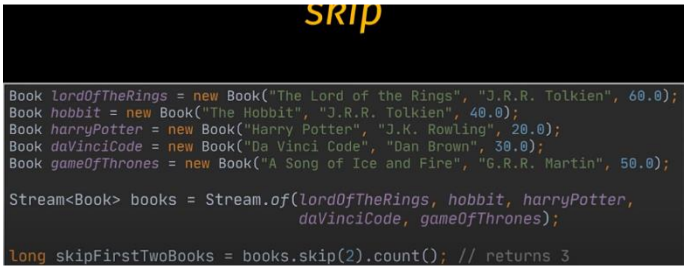
```
Stream<Integer> stream = Stream.of(25, 15, 75, 35, 40, 5, 65);

boolean oneIsEven = stream.anyMatch(i -> i % 2 == 0); // true

boolean allAreEven = stream.allMatch(i -> i % 2 == 0); // false

boolean allAreLessThan80 = stream.noneMatch(i -> i > 80); // true
```



distinct

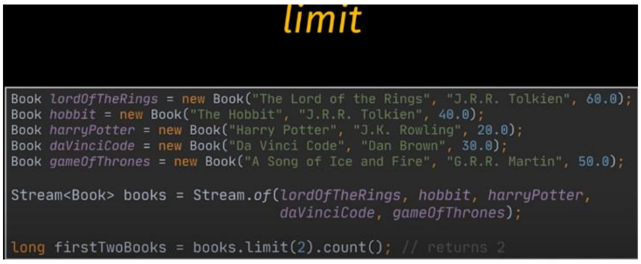
```
Stream.of(10, 10, 20, 20, 30)
      .distinct()
      .forEach(System.out::println);
```

returns 10, 20, 30



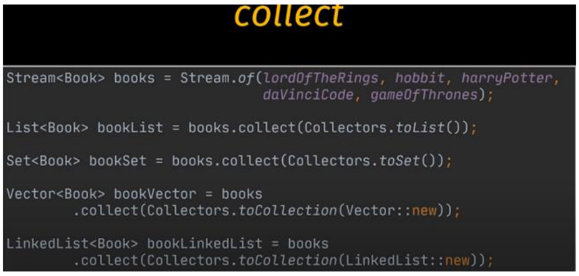
skip

```
Book lordOfTheRings = new Book("The Lord of the Rings", "J.R.R. Tolkien", 60.0);
Book hobbit = new Book("The Hobbit", "J.R.R. Tolkien", 40.0);
Book harryPotter = new Book("Harry Potter", "J.K. Rowling", 20.0);
Book daVinciCode = new Book("Da Vinci Code", "Dan Brown", 30.0);
Book gameOfThrones = new Book("A Song of Ice and Fire", "G.R.R. Martin", 50.0);

Stream<Book> books = Stream.of(lordOfTheRings, hobbit, harryPotter,
                               daVinciCode, gameOfThrones);

long skipFirstTwoBooks = books.skip(2).count(); // returns 3
```



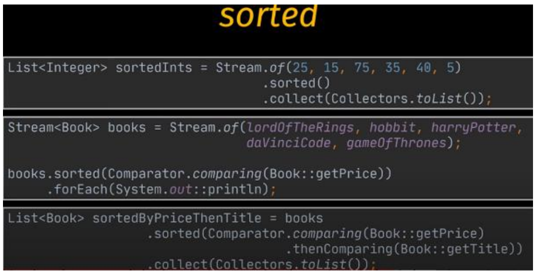
limit

```
Book lordOfTheRings = new Book("The Lord of the Rings", "J.R.R. Tolkien", 60.0);
Book hobbit = new Book("The Hobbit", "J.R.R. Tolkien", 40.0);
Book harryPotter = new Book("Harry Potter", "J.K. Rowling", 20.0);
Book daVinciCode = new Book("Da Vinci Code", "Dan Brown", 30.0);
Book gameOfThrones = new Book("A Song of Ice and Fire", "G.R.R. Martin", 50.0);

Stream<Book> books = Stream.of(lordOfTheRings, hobbit, harryPotter,
                               daVinciCode, gameOfThrones);

long firstTwoBooks = books.limit(2).count(); // returns 2
```



collect

```
Stream<Book> books = Stream.of(lordOfTheRings, hobbit, harryPotter,
                               daVinciCode, gameOfThrones);

List<Book> bookList = books.collect(Collectors.toList());

Set<Book> bookSet = books.collect(Collectors.toSet());

Vector<Book> bookVector = books
        .collect(Collectors.toCollection(Vector::new));

LinkedList<Book> bookLinkedList = books
        .collect(Collectors.toCollection(LinkedList::new));
```

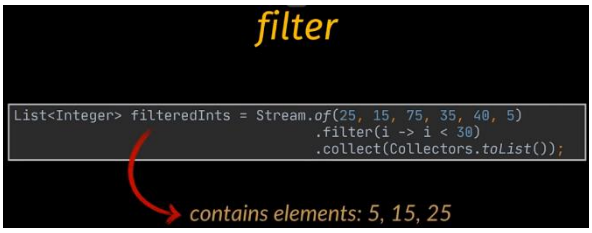


sorted

```java
List<Integer> sortedInts = Stream.of(25, 15, 75, 35, 40, 5)
                                  .sorted()
                                  .collect(Collectors.toList());
```

```java
Stream<Book> books = Stream.of(lordOfTheRings, hobbit, harryPotter,
                              daVinciCode, gameOfThrones);

books.sorted(Comparator.comparing(Book::getPrice))
     .forEach(System.out::println);
```

```java
List<Book> sortedByPriceThenTitle = books
                .sorted(Comparator.comparing(Book::getPrice)
                                  .thenComparing(Book::getTitle))
                .collect(Collectors.toList());
```

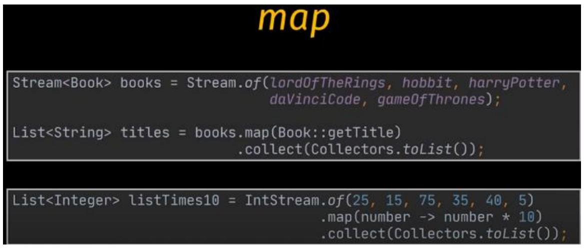


filter

```java
List<Integer> filteredInts = Stream.of(25, 15, 75, 35, 40, 5)
                                    .filter(i -> i < 30)
                                    .collect(Collectors.toList());
```

contains elements: 5, 15, 25

Map – returns a new stream after applying a function on the existing stream



map

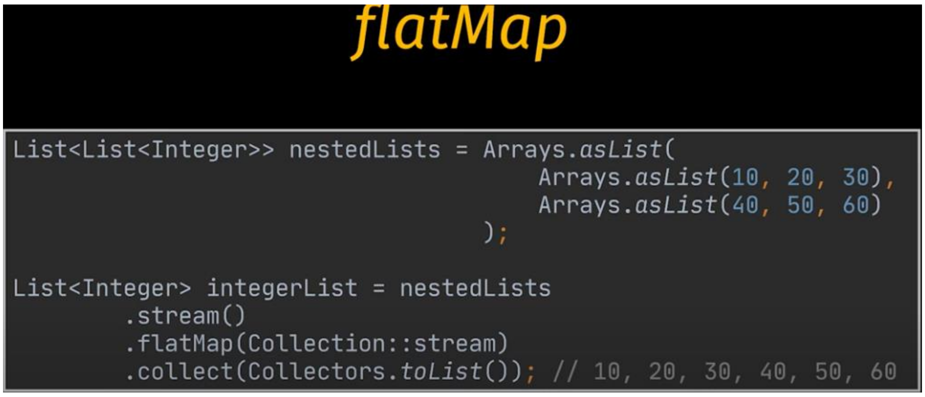
```java
Stream<Book> books = Stream.of(lordOfTheRings, hobbit, harryPotter,
                              daVinciCode, gameOfThrones);

List<String> titles = books.map(Book::getTitle)
                          .collect(Collectors.toList());
```

```java
List<Integer> listTimes10 = IntStream.of(25, 15, 75, 35, 40, 5)
                                      .map(number -> number * 10)
                                      .collect(Collectors.toList());
```

Peek == forEach but is terminal (returns a stream)

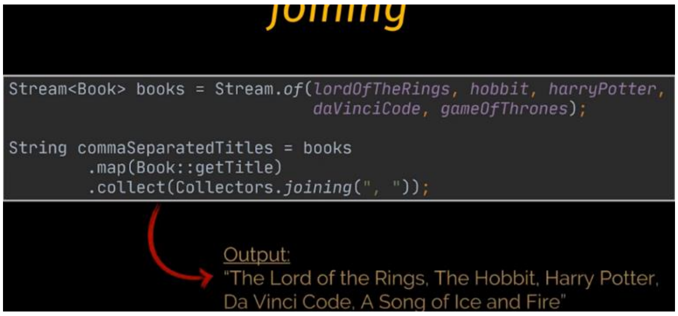
Flat map – method used when you want to collect a values of the map sequentially into the same list

In summary, `map` transforms each element in the stream, while `flatMap` transforms each element into a stream and then flattens these streams into one.



flatMap

```java
List<List<Integer>> nestedLists = Arrays.asList(
                                        Arrays.asList(10, 20, 30),
                                        Arrays.asList(40, 50, 60)
                                  );

List<Integer> integerList = nestedLists
        .stream()
        .flatMap(Collection::stream)
        .collect(Collectors.toList()); // 10, 20, 30, 40, 50, 60
```

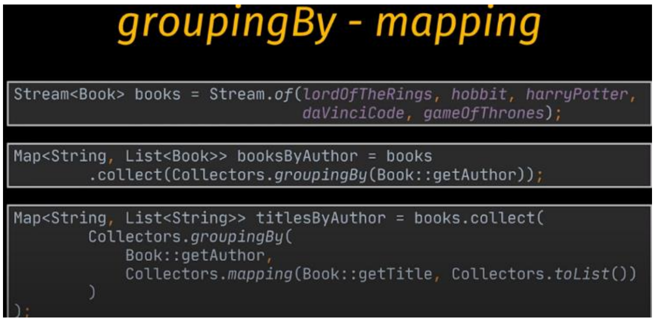


joining

```java
Stream<Book> books = Stream.of(lordOfTheRings, hobbit, harryPotter,
                              daVinciCode, gameOfThrones);

String commaSeparatedTitles = books
        .map(Book::getTitle)
        .collect(Collectors.joining(", "));
```

Output:
"The Lord of the Rings, The Hobbit, Harry Potter,
Da Vinci Code, A Song of Ice and Fire"

Joining joins parts of the stream into the single string using delimeter(",") between them



groupingBy - mapping

```java
Stream<Book> books = Stream.of(lordOfTheRings, hobbit, harryPotter,
                              daVinciCode, gameOfThrones);
```

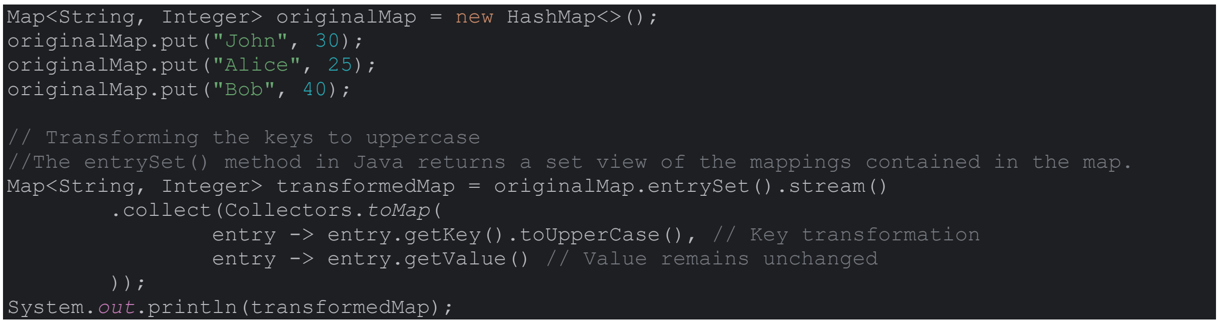
```java
Map<String, List<Book>> booksByAuthor = books
        .collect(Collectors.groupingBy(Book::getAuthor));
```

```java
Map<String, List<String>> titlesByAuthor = books.collect(
        Collectors.groupingBy(
            Book::getAuthor,
            Collectors.mapping(Book::getTitle, Collectors.toList())
        )
);
```

When we use the **mapping** method we link one value of our stream with another value

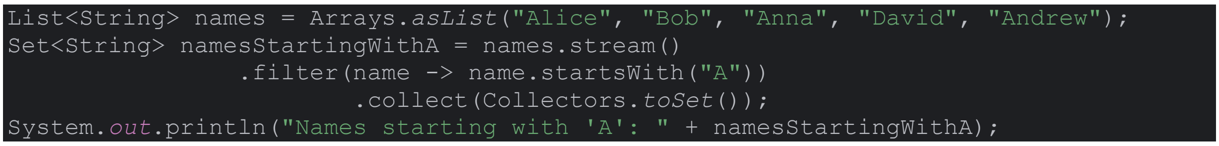
Here we connect to a map author and all his books by **groupingBy**

Using streams with Map

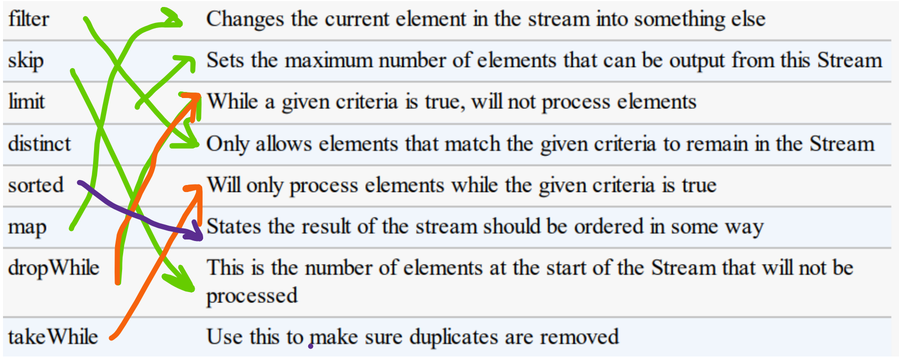


```java
Map<String, Integer> originalMap = new HashMap<>();
originalMap.put("John", 30);
originalMap.put("Alice", 25);
originalMap.put("Bob", 40);

// Transforming the keys to uppercase
//The entrySet() method in Java returns a set view of the mappings contained in the map.
Map<String, Integer> transformedMap = originalMap.entrySet().stream()
        .collect(Collectors.toMap(
                entry -> entry.getKey().toUpperCase(), // Key transformation
                entry -> entry.getValue() // Value remains unchanged
        ));
System.out.println(transformedMap);
```

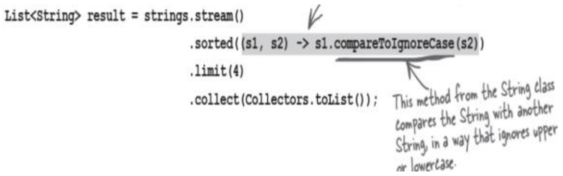
Using streams with Set



```java
List<String> names = Arrays.asList("Alice", "Bob", "Anna", "David", "Andrew");
Set<String> namesStartingWithA = names.stream()
                 .filter(name -> name.startsWith("A"))
                         .collect(Collectors.toSet());
System.out.println("Names starting with 'A': " + namesStartingWithA);
```

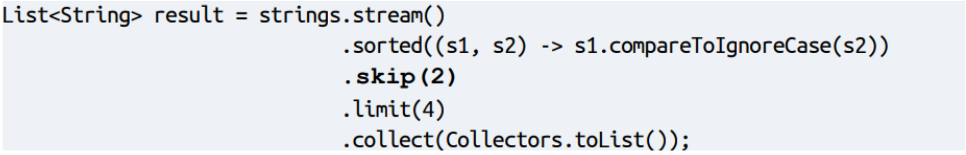


filter	Changes the current element in the stream into something else
skip	Sets the maximum number of elements that can be output from this Stream
limit	While a given criteria is true, will not process elements
distinct	Only allows elements that match the given criteria to remain in the Stream
sorted	Will only process elements while the given criteria is true
map	States the result of the stream should be ordered in some way
dropWhile	This is the number of elements at the start of the Stream that will not be processed
takeWhile	Use this to make sure duplicates are removed

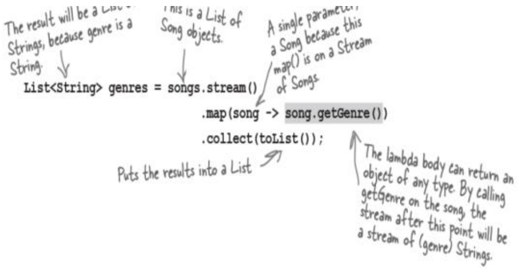


```java
List<String> result = strings.stream()
                             .sorted((s1, s2) -> s1.compareToIgnoreCase(s2))
                             .limit(4)
                             .collect(Collectors.toList());
```

This method from the String class compares the String with another String, in a way that ignores upper or lowercase.

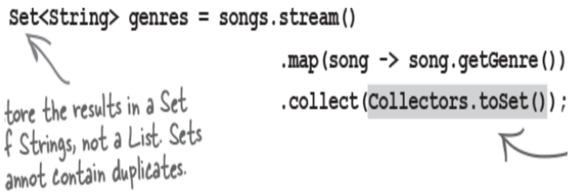


```java
List<String> result = strings.stream()
                             .sorted((s1, s2) -> s1.compareToIgnoreCase(s2))
                             .skip(2)
                             .limit(4)
                             .collect(Collectors.toList());
```



```java
List<String> genres = songs.stream()
                           .map(song -> song.getGenre())
                           .collect(toList());
```

The result will be a List of Strings, because genre is a String.
This is a List of Song objects.
A single parameter, a Song because this map() is on a Stream of Songs
Puts the results into a List
The lambda body can return an object of any type. By calling getGenre on the song, the stream after this point will be a stream of (genre) Strings.



```java
Set<String> genres = songs.stream()
                          .map(song -> song.getGenre())
                          .collect(Collectors.toSet());
```

tore the results in a Set f Strings, not a List. Sets annot contain duplicates.