Java advanced examples and exercises

Enums

```
public class Main {
  public static void main(String[] args) {
    Season[] values = Season.values();
    for (Season s: values) {
      System.out.println(s);
    }
    Scanner scanner = new Scanner(System.in);
    System.out.print("Which season do you like? ");
    String myseason = scanner.nextLine();
    Season favorit = Season.valueOf(myseason.toUpperCase());
    System.out.printf("So you like %s?%n", favorit);
    System.out.printf("It is number %d in the list%n", favorit.ordinal());
  }
}
enum Season {SUMMER, AUTUMN, WINTER, SPRING};
```

Generics

Generics01

We have to add @SuppressWarnings because we are using a collection class without generics (normally we do not do that anymore). Because we are not using generics the arraylist is not typesafe (can contain strings and integers)

```
public class Main {
	@SuppressWarnings("unchecked")
	public static void main(String[] args) {
		@SuppressWarnings("rawtypes")
		ArrayList list1 = new ArrayList();
		list1.add("Elvis"); // add a String
		list1.add(42);          // add a number
		list1.add(new Person("Elvis", 42)); // add a person
		for (Object o: list1) {
			System.out.println(o);
		}
	}
}

class Person{
	private String name;
	private int age;
	public Person(String name, int age) {
		this.name = name;
		this.age = age;
	}
	public String getName() {
		return name;
	}
	public int getAge() {
		return age;
	}
}
```

Generics02

When we declare the arraylist as an arraylist of Strings (ArrayList<String>) it is impossible to add integers:

```
public class Main {
	public static void main(String[] args) {
		ArrayList<String> list1 = new ArrayList<String>();
		list1.add("Elvis");
		//list1.add(42); //not possible
		//list1.add(new Person("Elvis", 42)); //not possible
		for(String s:list1) {
			System.out.println(s);
		}
	}
}
```

Generics03

We deciare our own generic Trap-class:

```
public class Trap<E> {
	private E prisoner;

	public void catchPrisoner(E prisoner) {
		this.prisoner = prisoner;
	}

	public E releasePrisoner() {
		E tmpPrisoner = this.prisoner;
		this.prisoner = null;
		return tmpPrisoner;
	}

	/*public String getPrisonerName() {
	 * //not possible, not every class has a getName() method
	 *    return prisoner.getName();
	}*/
}
```

And now we can declare a Trap for certain types of animals:

```
public class Main {

  public static void main(String[] args) {
    Trap<Bear> bearTrap = new Trap<Bear>();
    Bear baloo = new Bear("Baloo");
    Duck donald = new Duck("Donald");
    bearTrap.catchPrisoner(baloo);
    //bearTrap.catchPrisoner(donald); //error: bearTrap cannot catch ducks
    Animal a = bearTrap.releasePrisoner();
    System.out.printf("%s is released", a.getName());
  }
}

class Animal {
	private String name;
	public Animal(String name) {
```

```
            this.name = name;
      }
      public String getName() {
            return name;
      }
}

class Bear extends Animal{
      public Bear(String name) {
            super(name);
      }
}
class Duck extends Animal{
      public Duck(String name) {
            super(name);
  }
}
```

Generics04

When can also declare a Trap that can only be used for Animal (or derived from Animal) objects:

```
public class AnimalTrap <E extends Animal> {
      private E prisoner;

      public void catchPrisoner(E prisoner) {
            this.prisoner = prisoner;
      }

      public E releasePrisoner() {
            E tmpPrisoner = this.prisoner;
            this.prisoner = null;
            return tmpPrisoner;
      }

      public String getPrisonerName() {
            //Every prisoner is an Animal with a getName() method
            return prisoner.getName();
      }
}
```

Arrays and Covariance(cocontravariance)

Covariance: reading also works for subclasses, writing covariant does not work in Java.

```
public class CoContraVariance {
    public static void main(String[] args) {
        //Covariant arrays reading
        String[] K3_old = {"Karen", "Kristel", "Kathleen"};
        String[] K3_less_old = {"Karen", "Kristel", "Josje"};
        if (equalArrays(K3_old, K3_less_old)){
            System.out.println("Arrays are equal");
        }else {
            System.out.println("Arrays are not equal");
        }
        //ArrayStoreException at runtime while writing
        doSomethingWithArray(K3_old);

    }
```

```
    //works CoVariant: also OK for child classes
    public static boolean equalArrays(Object[] o1, Object[] o2) {
        //skip null checking
        if (o1.length != o2.length) return false;
        for(int i=0;i<o1.length;i++){
            if (! o1[i].equals(o2[i])) return false;
        }
        return true;
    }
    public static void doSomethingWithArray(Object[] objects) {
        //reads are OK
        Object o = objects[0];
        System.out.println("First object is "+o);
        //writes give error at runtime
        //objects[0] = LocalDate.now();
    }
}
```

Generics: Covariant (extends) reading and ContraVariant (super) writing(cocontravariancegenerics)

The mnemonic here is PECS (Producer Extends, Consumer Super). When you “get” something from the generic class it is a producer (hence “extends” in printAnimalFromShelter). When you “set” something in a generic class it is a consumer (hence “super” in saveAnimalInShelter).

```
public class CoContraVarianceGenerics {
    public static void main(String[] args) {
        Shelter<Bird> birdshelter = new Shelter<>();
        saveAnimalInShelter(birdshelter);
        printAnimalFromShelter(birdshelter);
    }
    //writing
    public static void saveAnimalInShelter(Shelter<? super Chicken>
shelter) {
        shelter.setAnimal(new Chicken("Tweety"));
    }
    //reading
    public static void printAnimalFromShelter(Shelter<? extends
Animal> shelter) {
        Animal a = shelter.getAnimal();
        System.out.println(a);

    }
}
class Animal {
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    private String name;
```

```
    public Animal(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return "Animal{" +
                "name='" + name + '\'' +
                '}';
    }
}
class Bird extends Animal{
    public Bird(String name) {
        super(name);
    }

    @Override
    public String toString() {
        return "Bird{" +
                "name='" + getName() + '\'' +
                '}';
    }
}
class Chicken extends Bird{
    public Chicken(String name) {
        super(name);
    }
    @Override
    public String toString() {
        return "Chicken{" +
                "name='" + getName() + '\'' +
                '}';
    }
}
class Shelter <T> {
    private T animal;

    public T getAnimal() {
        return animai;
    }

    public void setAnimal(T animai) {
        this.animai = animai;
    }
}
```

Collections

JavaCollection01

A class (like ArrayList) that implements Iterable can be iterated over. When it implements Collection it can manage a group of Objects (add, remove, …). We use generics to limit the content of the collection to Strings.

When we declare a Collection variable of generic type String it can only be implemented by a class (ArrayList) of type String. From Java 7 on, we can use the *diamond* operator and leave out the 'String'.

```
	public static void main(String[] args) {
		//Collection<String> building = new ArrayList<String>();
		// From Java 7 onwards
		Collection<String> building = new ArrayList<>();
		building.add("Elvis");
		if(building.contains("Elvis")) {
			System.out.println("Elvis is in the building");
		} else {
			System.out.println("Elvis has left the building");
		}
		if (building.isEmpty()) {
			System.out.println("The building is empty");
		} else {
			System.out.println("There is someone in the building");
		}
		building.add("Priscilla");
		Iterator<String> it = building.iterator();
		while (it.hasNext()) {
			String name = it.next();
			System.out.println(name);
		}
		//When a class has an iterator, we can use a for each
		for (String s: building) {
			System.out.println(s);
		}
		building.clear();
	}
```

JavaCollection02

Collections can be converted into streams (Java 8). We can filter streams, sort them, use a foreach operation on them,…

```
	public static void main(String[] args) {
		Collection<Employee> company = new ArrayList<Employee>();
		company.add(new Employee("Elvis", "Entertainment"));
		company.add(new Employee("Priscilla", "Accounting"));
		company.add(new Employee("Colonel Parker", "Management"));
		company.add(new Employee("Lisa Marie", "Accounting"));
		//Java 8
		company.stream()
		.filter(e -> e.getDepartment().equals("Accounting"))
		.forEach(e -> System.out.println(e.getName()));
	}
public class Employee {
	private String name, department;

	public Employee(String name, String department) {
		this.name = name;
		this.department = department;
	}

	public String getName() {
		return name;
	}

	public String getDepartment() {
```

```
            return department;
      }

}
```

Javacollection03

A set contains unique items(the same string cannot be added twice)

```
public static void main(String[] args) {
      Set<String> items = new HashSet<String>();
      items.add("Elvis");
      boolean succeeded = items.add("Elvis"); // false
      if (!succeeded) {
            System.out.println("There is only one Elvis");
      }
      System.out.println("The set contains " + items.size() + " elements");
}
```

Javacollection04

Two employees are the same when their equals()-method says they are the same or when they refer to the same object in memory

```
public static void main(String[] args) {
      Set<Employee> items = new HashSet<Employee>();
      items.add(new Employee("Elvis"));
      boolean succeeded = items.add(new Employee("Elvis")); //true!!!
      if (succeeded) {
            System.out.println("There is an Elvis impersonator here");
      }
      Iterator<Employee> it = items.iterator();
      while(it.hasNext()) {
            Employee e = it.next();
            if (e.getName().equals("Elvis")) it.remove();
      }
      System.out.println("The set contains " + items.size() + " elements");
}
public class Employee {
      private String name;

      public Employee(String name) {
            this.name = name;
      }

      public String getName() {
            return name;
      }
      /*@Override
      public boolean equals(Object o) {
            boolean equal = false;
            if (o instanceof Employee) {
                  Employee e = (Employee)o;
                  equal = this.getName().equals(e.getName());
            }
            return equal;
      }
      @Override
      public int hashCode() {
            return name.hashCode();
```

```
	}*/

}
```

Javacollection05

A class that implements the list-interface has the concept of 'position' (cfr. Array).

```
public static void main(String[] args) {
	List<String> items = new ArrayList<String>();
	items.add("Elvis"); // position 0
	items.add("Colonel Parker"); // position 1
	items.set(1, "Dries Van Kuijk"); // replace element 1
	items.add("Vernon");
	int index = items.indexOf("Elvis");
	System.out.println("Elvis is on position " + index);
	items.remove(1);
	String name = items.get(1); //Vernon is shifted to position 1
	System.out.println("On position 1 we have " + name);
	ListIterator<String> it = items.listIterator(items.size());
	while (it.hasPrevious()) {
		System.out.println(it.previous());
	}
	/*for(ListIterator<String> it = items.listIterator(items.size());
it.hasPrevious();) {
		System.out.println(it.previous());
	}*/
}
```

Javacollection06

A map has key-value pairs

```
public static void main(String[] args) {
	String[] words = {"one", "two", "one", "three", "three", "one"};
	Map<String, Integer> frequency = new HashMap<String, Integer>();
	for(String word: words) {
		Integer freq = frequency.get(word);
		if (freq == null) {
			freq = 1;
		} else {
			freq = freq + 1;
		}
		frequency.put(word, freq);
	}
	System.out.println(frequency);
}
```

Enhanced for-loop (forloop)

A class that has to be used in an enhanced forloop must implement the Iterable interface. Normally we will use some kind of Collection class. But in order to illustrate the workings of the Iterator interface we use an anonymous class that lets the K3 class return three names. (without using a collection). This is also an example of the usage of strings in a switch statement. (string comparison based on hashcode).

```
import java.util.Iterator;
import java.util.NoSuchElementException;
public class ForLoop {
    public static void main(String[] args) {
        K3 k3 = new K3();
```

```
        for (Iterator<String> it = k3.iterator();it.hasNext();){
            System.out.println(it.next());
        }
        for (String name :k3) {
            System.out.println(name);
        }
    }
}
class K3 implements Iterable<String> {
    @Override
    public Iterator<String> iterator() {
        return new Iterator<String>(){
            private String current = "";
            @Override
            public boolean hasNext() {
                return ! "Kathleen".equals(current);
            }
            @Override
            public String next() {
                switch(current) {
                    case "":
                        current="Karen";
                        break;
                    case "Karen":
                        current="Kristel";
                        break;
                    case "Kristel":
                        current="Kathleen";
                        break;
                    case "Kathleen":
                        throw new NoSuchElementException();
                }
                return current;
            }
        };
    }
}
```

Exercises

1) Write a program that asks how many names the user wants to input, asks for those names and shows the names afterwards. But it should only show names that are longer than 5 characters:

```
How many names: 4
Give name 1:Elvis
Give name 2:Priscilla
Give name 3:Vernon
Give name 4:Lisa Marie
Lisa Marie
Vernon
Priscilla
```

2) Write the classes Student and Classroom to make the following main program work

```
public static void main(String[] args) {
      Classroom classroom = new Classroom();
      Scanner scanner = new Scanner(System.in);
```

```
        Random rand = new Random();
        System.out.print("How many students? ");
        int n = Integer.parseInt(scanner.nextLine());
        for (int i= 0; i< n; i++) {
              System.out.printf("Give name of student %d: ", i + 1);
              String name = scanner.nextLine();
              Student s = new Student();
              s.setId(i + 1);
              s.setName(name);
              s.setPoints(rand.nextInt(20) + 1);
              classroom.addStudent(s);
        }
        for (Student s: classroom) {
              System.out.println(s);
        }
        System.out.printf("Average: %.2f", classroom.getAverage());
  }
```

Solutions

Javacollectionex01

```
	public static void main(String[] args) {
		List<String> names = new ArrayList<String>();
		Scanner scanner = new Scanner(System.in);
		System.out.print("How many names: ");
		int n = Integer.parseInt(scanner.nextLine());
		for (int i = 0; i < n; i++) {
			System.out.printf("Give name %d:", i + 1);
			String s = scanner.nextLine();
			if (s.length() > 5) {
				names.add(s);
			}
		}
		for (String s : names) {
			System.out.println(s);
		}

	}
```

Javacollectionex02

```
public class Student {
	private int id;
	private String name;
	private int points;

	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}
	public int getPoints() {
		return points;
	}
	public void setPoints(int points) {
		this.points = points;
	}
	public int getId() {
		return id;
	}
	public void setId(int id) {
		this.id = id;
	}
	@Override
	public boolean equals(Object o) {
		boolean equal = false;
		if (o instanceof Student) {
			equal = this.getId() == ((Student)o).getId();
		}
		return equal;
	}
	@Override
	public int hashCode() {
		return this.id;
	}
```

```
	@Override
	public String toString(){
		return String.format("%2d: %s (%d points)", id, name, points);
	}
}
public class Classroom implements Iterable<Student> {
	private Set<Student> students = new LinkedHashSet<Student>();
	public boolean addStudent(Student s) {
		return students.add(s);
	}

	public float getAverage() {
		int sum= 0;
		for (Student s: students) {
			sum += s.getPoints();
		}

		return (float)sum/students.size();
	}
	@Override
	public Iterator<Student> iterator() {
		return students.iterator();
	}
}
```

Anonymous classes

Anonymous01

```
public class Main {
	public static void main(String[] args) {
		MenuItem item1 = new MenuItem();
		item1.execute();
		Command item2 = new Command() {
			@Override
			public void execute() {
				System.out.println("By all");

			}
		};
		item2.execute();
	}
}
class MenuItem implements Command{
	@Override
	public void execute() {
		System.out.println("Hi all");

	}
}
interface Command{
	void execute();
}
```

Anonymous02

We can sort a collection using Collections.sort(coll). The second argument can be an object of type Comparator.

```
public class Main {
```

```
	public static void main(String[] args) {
		List<String> people = new ArrayList<>();
		people.add("Elvis");
		people.add("Priscilla");
		people.add("Vernon");
		people.add("Tom");
		Collections.sort(people);
		for(String s: people){
			System.out.println(s);
		}
		Collections.sort(people, new Comparator<String>() {

			@Override
			public int compare(String o1, String o2) {
				return o1.substring(1).compareTo(o2.substring(1));
			}
		});
		System.out.println("Sorted by second character");
		for(String s: people){
			System.out.println(s);
		}

	}

}
```

Exercises

Create a class Person. A person has a first name and a last name. Write a main-program that creates an ArrayList with a number of people (e.g. Priscilla Wagner, Tom Parker, Elvis Presley). Sort the arraylist based on the first name and show the result:

```
Elvis Presley
Priscilla Wagner
Tom Parker
```

Extra: when a class implements the Comparable interface it can be sorted without a comparator: Collections.sort(people). Can you change the Person class so that it will be sorted according to last name?

The Comparable interface has only one method: compareTo(other). It is defined as follows:

```
int compareTo(T o)

Compares this object with the specified object for order. Returns a
negative integer, zero, or a positive integer as this object is less than,
equal to, or greater than the specified object.
```

Solution

```
public static void main(String[] args) {
  List<Person> people = new ArrayList<>();
  people.add(new Person("Priscilla", "Wagner"));
  people.add(new Person("Tom", "Parker"));
  people.add(new Person("Elvis", "Presley"));
  Collections.sort(people, new Comparator<Person>() {
    @Override
    public int compare(Person o1, Person o2) {
      return o1.getFirstName().compareTo(o2.getFirstName());
    }
  });
  for(Person p: people){
    System.out.println(p);
  }
}
class Person implements Comparable<Person>{
      private String firstName;
      private String lastName;

      public Person(String firstName, String lastName) {
            super();
            this.firstName = firstName;
            this.lastName = lastName;
      }
      public String getFirstName() {
            return firstName;
      }
      public void setFirstName(String firstName) {
            this.firstName = firstName;
      }
      public String getLastName() {
            return lastName;
      }
      public void setLastName(String lastName) {
            this.lastName = lastName;
      }
      @Override
      public String toString() {
            return String.format("%s %s",firstName, lastName);
      }
      @Override
      public int compareTo(Person arg0) {
            // TODO Auto-generated method stub
            return 0;
      }
}
```

Varargs(varargs)

Varargs are mainly interesting because they make calling the function easier. In this example we have to add an extra string as first parameter in the varargs-constructor. Otherwise both constructors would be the same for the compiler.

```
public class VarArgs {
    public static void main(String[] args) {
        Group K3_old = new Group(new String[]{"Karen", "Kristel",
"Kathleen"});
```

```
        Group K3_new = new Group("Karen", "Kristel", "Kathleen");
        System.out.println("Using older syntax:");
        for (String s: K3_old){
            System.out.println(s);
        }
        System.out.println("Using newer syntax");
        for(String s: K3_new){
            System.out.println(s);
        }

    }
}
class Group implements Iterable<String> {
    private final String[] participants;

    public Group(String[] participants) {
        if (participants.length == 0) throw new
IllegalArgumentException("At least one participant is needed.");
        this.participants = participants.clone();
    }
    public Group(String p1, String... participants) {
        this.participants = new String[participants.length + 1];
        this.participants[0] = p1;
        System.arraycopy(participants, 0, this.participants, 1,
participants.length);
    }

    @Override
    public Iterator<String> iterator() {
        return Arrays.asList(this.participants).iterator();
    }
}
```

Static imports(staticimport)

Static imports let you use static methods and values without a namespace.

```
import java.text.NumberFormat;
import java.text.ParseException;
import java.util.Locale;
import java.util.Scanner;

//Maybe a good idea?
import static java.lang.Math.sqrt;
//100% certain this is not a good idea...
import static java.text.NumberFormat.getInstance;

public class StaticImport {
    public static void main(String[] args) throws ParseException {
        Locale locale = Locale.getDefault();
        //Which getInstance()? What getInstance()?
        NumberFormat nf = getInstance(locale);
        Scanner scanner = new Scanner(System.in);
        System.out.print("Give a number: ");
        String sNumber = scanner.nextLine();
        double number = (Double)nf.parse(sNumber);
        //I can imagine what sqrt does.
```

```
        double root = sqrt(number);
        System.out.printf("The square root of %.2f is %f.%n",
number, root);
    }
}
```

Annotations(annotationexample)

This is just an example of how we could create our own annotations and use them. This is especially useful for frameworks. Unless you create your own frameworks, you will have little use for this example. Just creating annotations is useful in a dependency injection container.

We start with the definition of an annotation:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.PARAMETER)
public @interface NonNegative {
}
```

In this case it is an annotation that will be accessible at runtime. We can use it for parameters. An example of the use of this annotation is:

```
public class TestClass {
    public void method(@NonNegative Integer number, Integer
number2){
        System.out.println("method: "+number + ", " + number);
    }
}
```

We can define that the @NonNegatieve annotation checks whether the parameter is negative. That could be done in a function like this:

```
public static void saveCall(Object o, Integer ... args) throws
NoSuchMethodException, InvocationTargetException,
IllegalAccessException {
  Class c = o.getClass();
  Method m = c.getMethod("method", Integer.class, Integer.class);
  for (int i=0; i< m.getParameterCount(); i++){
    if (m.getParameters()[i].getAnnotation(NonNegative.class) !=
null) {
     if (args[i] < 0) throw new IllegalArgumentException("Parameter
cannot be negative");
    }
  }
  m.invoke(o, args);
}
```

In this example we check every argument in the “method”-function to see whether it has a NonNegative annotation. If that is the case we check the value of the parameter. If that parameter is negative, we throw an exception.

Lambda functions and streams

Is it really that special? (lambda00)

In this example we give an object to Arrays.sort, but in fact we are offering a compare()-method because the class only has one method defined.

```
public class Main {
```

```
    public static void main(String[] args) {
        String[] names = {"Karen", "Kristel", "Kathleen"};
        Arrays.sort(names, new Comparator<String>(){
           public int compare(String o1, String o2) {
              return Integer.compare(o1.length(), o2.length());
           }
        });
        for (String n: names){
            System.out.println(n);
        }
    }
}
//
// class that can be used when we don't want an anonymous class
//
class LengthComparator implements Comparator<String> {

    @Override
    public int compare(String o1, String o2) {
        return Integer.compare(o1.length(), o2.length());
    }
}
```

And now with a lambda function

```
public class Main {
    public static void main(String[] args) {
        String[] names = {"Karen", "Kristel", "Kathleen"};
        Arrays.sort(names, (o1, o2) -> -Integer.compare(o1.length(),
o2.length()));
        for (String n: names){
            System.out.println(n);
        }
    }
}
```

Functional interfaces(predicateexample)

There are a number of predefined functional interfaces in java.util.function. In the following class we declare a filter method that takes a Predicate<T> as an argument. A Predicate<T> object has a test() method that takes an argument of type T and returns a boolean. So the function can be used as rule for filters:

```
class K3{
    private String[] k3= {"Karen", "Kristel", "Kathleen", "Josje",
"Hanne", "Marthe", "Klaasje"};
    public String[] filter(Predicate<String> p) {
        List<String> filtered = new ArrayList<>();
        for (String n: k3) {
            if (p.test(n)){
                filtered.add(n);
            }
        }
        return filtered.toArray(new String[filtered.size()]);
    }

}
```

When we call the filter function we do not have the create a new class anonymous or not), create an object and give the object as a parameter. We can use a lambda function:

```
public class javaadvanced {
    public static void main(String[] args) {
        K3 k3 = new K3();
        String[] names = k3.filter(s -> s.charAt(1) == 'a');
        for(String n: names) {
            System.out.println(n);
        }
    }
}
```

Lambdaex

```
public class Main {

	public static void main(String[] args) {
		List<Student> students = new ArrayList<>();
		students.add(new Student("john", 12));
		students.add(new Student("paul", 15));
		students.add(new Student("george", 13));
		students.add(new Student("richard", 12));
		students.add(new Student("pete", 16));
		students.add(new Student("stuart", 17));
		students.stream()
			.filter(s -> s.getAge() >= 15 )
			.sorted((s1, s2)->s1.getName().compareTo(s2.getName()))
			.limit(2)
			.forEach(s -> System.out.println(s));
		//alternative: .foreach(System.out::println)
	}
}
class Student{
	private int age;
	private String name;
	public Student(String name, int age) {
		super();
		this.age = age;
		this.name = name;
	}
	public int getAge() {
		return age;
	}
	public void setAge(int age) {
		this.age = age;
	}
	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}
	@Override
	public String toString() {

		return getName();
	}
```

```
}
```

Comparator.comparing (comparatorcomparing)

Comparator.comparing is a static method that returns a comparator. The argument (String::length) is a *keyextractor*. It is a function that returns the property of the objects that is used for the comparison.

The method *comparing* is an example of a static method in an interface. Because we can add static methods to interfaces in Java 8 we could do without *utility classes* like Collections. It would be possible to add all methods from Collections to the interface Collection (or to List, Map, …). But for backward compatibility reasons they are kept in Collections.

```
public class Main {
    public static void main(String[] args) {
        String[] names = {"Karen", "Kristel", "Kathleen"};
        Arrays.sort(names, Comparator.comparing(String::length));
        for(String n: names) {
            System.out.println(n);
        }
    }
}
```

Streams(streams01)

The difference betaween “classical Java” and streams

```
public class javaadvanced {
  public static void main(String[] args) {
    String[] names = {"Karen", "Kristel", "Kathleen", "Josje",
"Hanne", "Marthe", "Klaasje"};
    List<String> lnames = Arrays.asList(names);
    lnames.forEach((s) -> System.out.println(s));
    System.out.println("Names ending with 'n'.");
    lnames.stream().filter( (s) ->
s.endsWith("n")).forEach(System.out::println);
  }
}
```

Streams works as a pipeline (filtering early is good), limit uses short-circuiting(streams02)

```
public class Main {
  public static void main(String[] args) {
    List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8);
    List<Integer> twoEvenSquares =
          numbers.stream()
                .filter(n -> {
                    System.out.println("filtering " + n);
                    return n % 2 == 0;
                })
                .map(n -> {
                    System.out.println("mapping " + n);
                    return n * n;
                })
                .limit(2)
                .collect(toList());
  }
}
```

Output:

```
filtering 1
filtering 2
mapping 2
filtering 3
filtering 4
mapping 4
```

Method references

Sometimes we can shorten the function body in the lambda expressions. One example of this is String.length in the previous example. Instead of using n->n.length, we can also use String::length. These are called *method references*. There are four types of method references:

Kind	Example
Reference to a static method	Class::staticMethodName
Reference to an instance method of an object	Object::instanceMethodName
Reference to an instance method of an object of a particular type	Type::methodName
Reference to a constructor	ClassName::new

Some predefined functional interfaces (java.util.function)

A functional interface is an interface with only one method (except for default implementations). We can use the attribute @FunctionalInterface to denote a functional interface.

BiConsumer<T,U>	Operation with two input arguments, no results
BiFunction<T,U,R>	Function with two input arguments and result R
BiPredicate<T,U>	Function with two input arguments, Boolean return
Consumer<T>	Operation with one input argument, no results
DoubleFunction<R>	Function with double argument and result R
DoubleToIntFunction	Function with double argument and int result
Function<T,R>	Function with one argument and result R
Predicate<T>	Function with one argument and Boolean result
ToIntBiFunction<T,U>	Function with two arguments and int result

Some examples of Lambda functions and streams

The stream()-function turns a List into a stream.

```
List<String> myList =
    Arrays.asList("a1", "a2", "b1", "c2", "c1");

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

// C1
// C2
```

But we do not need a list on beforehand. We can create a list on-the-fly:

```
Stream.of("a1", "a2", "a3")
    .findFirst()
    .ifPresent(System.out::println);  // a1
```

Or (working with a stream of integers)

```
IntStream.range(1, 4)
    .forEach(System.out::println);

// 1
// 2
// 3
```

We can “debug” a stream by using the peek() function. The peek() function creates a second stream consisting of the element of this stream. Because we work on a copy, we do not disturb the original stream. Using peek() we can print the intermediate results:

```
List<String> myList
                = Arrays.asList("a1", "a2", "b1", "c2", "c1");

        myList
                .stream()
                .peek(s->{System.out.println("after stream:"+s);})
                .filter(s -> s.startsWith("c"))
                .peek(s->{System.out.println("after filter:"+s);})
                .map(String::toUpperCase)
                .peek(s->{System.out.println("after map:"+s);})
                .sorted()
               .forEach(System.out::println);
```

We can turn a stream of objects into primitive types:

```
Stream.of("a1", "a2", "a3")
    .map(s -> s.substring(1))
    .mapToInt(Integer::parseInt)
    .max()
    .ifPresent(System.out::println);  // 3
```

Using collectors we turn the output of a stream into a collection:

```
List<Person> filtered =
    persons
        .stream()
        .filter(p -> p.name.startsWith("P"))
        .collect(Collectors.toList());

System.out.println(filtered);    // [Peter, Pamela]
```

We can for instance group objects together in a map of ages and a list of people:

```
Map<Integer, List<Person>> personsByAge = persons
    .stream()
    .collect(Collectors.groupingBy(p -> p.age));

personsByAge
    .forEach((age, p) -> System.out.format("age %s: %s\n", age, p));

// age 18: [Max]
// age 23: [Peter, Pamela]
// age 12: [David]
```

But instead of turning the result of a stream in a collection we can also aggregate the elements into one value: the average age:

```
Double averageAge = persons
    .stream()
    .collect(Collectors.averagingInt(p -> p.age));

System.out.println(averageAge);     // 19.0
```

In this next example we turn the list into a string by joining the elements (and adding a prefix and a suffix:

```
String phrase = persons
    .stream()
    .filter(p -> p.age >= 18)
    .map(p -> p.name)
    .collect(Collectors.joining(" and ", "In Belgium ", " are of legal
age."));

System.out.println(phrase);
// In Belgium Max and Peter and Pamela are of legal age.
```

A collector consists of a supplier, an accumulator a combiner and a finisher:

```
Collector<Person, StringJoiner, String> personNameCollector =
    Collector.of(
        () -> new StringJoiner(" | "),          // supplier
        (j, p) -> j.add(p.name.toUpperCase()),  // accumulator
        (j1, j2) -> j1.merge(j2),               // combiner
        StringJoiner::toString);                // finisher

String names = persons
    .stream()
    .collect(personNameCollector);

System.out.println(names);  // MAX | PETER | PAMELA | DAVID
```

A Reduce operation combines all elements of a stream into a single result. In this example the person with the highest age is printed

```
persons
    .stream()
    .reduce((p1, p2) -> p1.age > p2.age ? p1 : p2)
    .ifPresent(System.out::println);
```

Another example is where we calculate the sum of the ages of all persons. The first argument of the Reduce function assigns 0 to sum:

```
Integer ageSum = persons
    .stream()
    .reduce(0, (sum, p) -> sum += p.age, (sum1, sum2) -> sum1 + sum2);

System.out.println(ageSum);
```

Exercise streams and collectors: exstreamcollect

Given a Person class with name and age and a number of persons, e.g.

```
Person[] persons=  {new Person("Jan", 14), new Person("Marieke",
14), new Person("Jan", 12), new Person("Piet", 13),
new Person("Joris", 15), new Person("Corneel", 13)
};
```

Use streams and lambda functions to filter the list (age > 13), order the list(ordered by age ascending) and show the result:

```
Jan (14)
Marieke (14)
Joris (15)
```

Extra: Create a map of <Integer, List<String>> that groups people per age:

```
{12=[Jan], 13=[Piet, Corneel], 14=[Jan, Marieke], 15=[Joris]}
```

Solution

```
public static void main(String[] args) {
   Person[] persons=  {new Person("Jan", 14),
     new Person("Marieke", 14), new Person("Jan", 12),
     new Person("Piet", 13), new Person("Joris", 15),
     new Person("Corneel", 13)
   };
   Arrays.stream(persons)
      .filter(p -> p.getAge()>13)
      .sorted((p1, p2)->Integer.compare(p1.getAge(),  p2.getAge()))
      .forEach(System.out::println);
   System.out.println(Arrays.stream(persons)
        .collect(Collectors.groupingBy(Person::getAge,
                Collectors.mapping(Person::getName,
                Collectors.toList()))));

     }
class Person{
    private String name;
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
    private int age;
    @Override
    public String toString() {
         return "Person [name=" + name + ", age=" + age + "]";
    }
}
```

Exercise streams/lambda

Given the following Java classes:

```
public class Main {
    public static void main(String[] args) {
        Singer[] singers= {new Singer("Karen", "Daemen"), new
Singer("Kristel", "Verbeke"), new Singer("Kathleen", "Aerts")};
         //String result = …;
        System.out.println(result);

    }
}
class Singer{
```

```
    private String firstName;
    private String lastName;
    public Singer(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }
    public String getFirstName() {
        return firstName;
    }
    public String getLastName() {
        return lastName;
    }
}
```

Write the right expression for “result = …;” so that the output will be:
Karen, Kristel and Kathleen.

Solution

Probably the best solution is taking length-1 elements and joining them with a comma. The last name is added “manually”.

```
public class Main {
    public static void main(String[] args) {
        Singer[] singers= {new Singer("Karen", "Daemen"), new
Singer("Kristel", "Verbeke"), new Singer("Kathleen", "Aerts")};
        String result = Arrays.stream(singers).limit(singers.length-
1).map(Singer::getFirstName)
                .collect(Collectors.joining(", ")).concat(" and
").concat(singers[singers.length-1].getFirstName());
        System.out.println(result);
    }
}
```

JDBC (databases)

Jdbc01

```
//private static final String driver = "org.apache.derby.jdbc.EmbeddedDriver";
public static void main(String[] args) throws SQLException, InstantiationException,
IllegalAccessException, ClassNotFoundException {
  //Class.forName(driver).newInstance();
  Connection conn = null;
  try{
    conn = DriverManager.getConnection("jdbc:derby:memory:sample;create=true");
    System.out.println("Connected to database");
  }finally{
    if (conn != null) {
      conn.close();
    }
    try{
      DriverManager.getConnection("jdbc:derby:memory:sample;drop=true");
      //dropping an inmemory derby database will also shutdown the database server
      //DriverManager.getConnection("jdbc:derby:memory:;shutdown=true");
    }catch(SQLException ex) {
      //Successful shutdown will throw an exception with state XJ015
      if ("XJ015".equals(ex.getSQLState())) {
        System.out.println("Embedded Derby stopped successfully");
        //succesful drop will throw an exception with state 08006
      }else if ("08006".equals(ex.getSQLState())) {
        System.out.println("Database dropped successfully");
      }else{
        throw ex;
    }
  }
}
```

The alternative is a try-with-resources statement:

```
try(Connection conn =
DriverManager.getConnection("jdbc:derby:memory:sample;create=true")){
  System.out.println("Connected to database");
}catch(SQLException ex) {
  //Successful shutdown will throw an exception with state XJ015
  if ("XJ015".equals(ex.getSQLState())) {
      System.out.println("Embedded Derby stopped successfully");
      //succesful drop will throw an exception with state 08006
  }else if ("08006".equals(ex.getSQLState())) {
      System.out.println("Database dropped successfully");
```

```
  }else{
      throw ex;
  }
}
```

Jdbc02

```
public class Main {
      private static final String CREATEPERSON = "CREATE TABLE person (id
int PRIMARY KEY, name VARCHAR(50))";
      private static final String INSERTPEOPLE = "INSERT INTO person VALUES
(1, 'Peter'), (2, 'Paul'), (3, 'Mary')";
      public static void main(String[] args) throws SQLException {
      try(Connection
conn=DriverManager.getConnection("jdbc:derby://localhost:1527/sample;create
=true");
        Statement stmt = conn.createStatement()){
        //will issue a warning if database already exists
        SQLWarning warnings = conn.getWarnings();
        if (warnings !=null) {
            printWarnings(warnings);
        }
        stmt.executeUpdate(CREATEPERSON);
        stmt.executeUpdate(INSERTPEOPLE);
      }
    }
    private static void printWarnings(SQLWarning warnings) {
      SQLWarning w = null;
      while ((w= warnings.getNextWarning()) != null) {
        System.out.println(w.getMessage());
      }
    }
}
```

Jdbc03

```
private static final String SELECT = "SELECT * FROM PERSON";
private static final String COUNT = "SELECT COUNT(*) FROM PERSON";
public static void main(String[] args) throws SQLException {
  try(Connection conn=
DriverManager.getConnection("jdbc:derby://localhost:1527/sample");
      Statement stmt = conn.createStatement()){
    try(ResultSet rs= stmt.executeQuery(COUNT)){
       rs.next(); // go to the first (and only) row
       int number = rs.getInt(1); //row contains one column with an integer
       System.out.printf("TABLE contains %d people\n", number);
    }
    try(ResultSet rs = stmt.executeQuery(SELECT)){
       while(rs.next()) {
         int id = rs.getInt(1);
         String name = rs.getString(2);
         System.out.printf("%d: %s\n", id, name);
       }
    }
  }
}
```

jdbcex01

Start derby as a network database.

Write a Java program that can execute CREATE TABLE/DROP TABLE statements in the sample database. Test the program with the following SQL statements:

CREATE TABLE testtbl (id int primary key)

And

DROP TABLE testtbl

Check whether the table is created in the database using the ij tool (connect 'jdbc:derby://localhost:1527/sample';)

Jdbcex02

Write your own ij-tool. It should connect to the sample database automatically. It should work with create table, drop table, insert, update, delete and select statements. When the user enters a select statement, only the rows must be shown (not the column names). Maybe the method getMetaData() from ResultSet can be of help here?

You can read lines from the console using the Scanner class:

Scanner scanner = new Scanner(System.in);

String line = scanner.nextLine();

while(! "exit".equals(line)) {

}

The program should show an error when the user makes a syntax error. After showing the error it must continue.

A typical session might look like this:

```
Enter command: create table test(id int primary key, name varchar(50))
Enter command: insert into test values (1, 'John')
Enter command: select * from test
                  ID                 NAME
                   1                John
Enter command: drop table test
Enter command: exit
```

Solution jdbcex01

```
public static void main(String[] args) {
  try(Connection conn=
DriverManager.getConnection("jdbc:derby://localhost:1527/sample");
      Statement stmt = conn.createStatement()){
    Scanner scanner = new Scanner(System.in);
    System.out.print("Enter CREATE TABLE/DROP TABLE statement:");
    String statement = scanner.nextLine();
    stmt.executeUpdate(statement);
    System.out.println("Statement executed");
  }catch(SQLException ex){
    System.out.println(ex.getMessage());
  }
}
```

Solution jdbcex02

```
private static final String CONN= "jdbc:derby://localhost:1527/sample";
public static void main(String[] args) {
  try(Connection conn = DriverManager.getConnection(CONN);
      Statement stm = conn.createStatement()){
    Scanner scanner = new Scanner(System.in);
    System.out.print("Enter command: ");
    String command = scanner.nextLine();
    while (!"exit".equals(command)){
      if (command.substring(0, 6).equalsIgnoreCase("select")){
        executeSelect(stm, command);
      }else{
        executeNonQuery(stm, command);
      }
      System.out.print("Enter command: ");
      command = scanner.nextLine();
    }
  }catch(SQLException ex){
      System.out.println(ex.getMessage());
  }
}
private static void executeSelect(Statement stm, String command){
  try(ResultSet rs = stm.executeQuery(command)){
    ResultSetMetaData rsmd = rs.getMetaData();
    int numCols = rsmd.getColumnCount();
    for (int i=0;i<numCols;i++){
      System.out.printf("%20s", rsmd.getColumnName(i + 1));
    }
    System.out.println();
    while(rs.next()){
      for (int i=0;i<numCols;i++){
        System.out.printf("%20s", rs.getObject(i+1));
      }
      System.out.println();
    }
  }catch (SQLException ex) {
    System.out.println(ex.getMessage());
  }
}
private static void executeNonQuery(Statement stm, String command){
  try{
    stm.executeUpdate(command);
```

```
  }catch(SQLException ex){
    System.out.println(ex.getMessage());
  }
}
```

Jdbc04

```
private static final String CONN ="jdbc:derby://localhost:1527/sample";
private static final String INSERT = "INSERT INTO PERSON (id, name) VALUES
(?,?)";
private static final String DELETE = "DELETE FROM PERSON WHERE id = ?";
private static final String SELECT = "SELECT * FROM PERSON";
public static void main(String[] args) throws SQLException {
  Person[] persons = {new Person(4,"Elvis"), new Person(5, "Lisa Marie")};
  try(Connection conn = DriverManager.getConnection(CONN)){
    try(PreparedStatement prepstmt = conn.prepareStatement(INSERT)){
      for(Person p:persons) {
        prepstmt.setInt(1, p.getId()); // first ?
        prepstmt.setString(2, p.getName()); // second ?
        prepstmt.executeUpdate();
      }
    }
    try(Statement stmt = conn.createStatement();
       ResultSet rs = stmt.executeQuery(SELECT)){
       while(rs.next()){
          int id = rs.getInt(1);
          String name = rs.getString(2);
          System.out.printf("%d: %s\n", id, name);
       }
    }
    try(PreparedStatement prepstmt = conn.prepareStatement(DELETE)){
      for (Person p:persons) {
        prepstmt.setInt(1, p.getId());
        prepstmt.executeUpdate();
      }
    }
  }
}
```

jdbcex03

Write a Java program that lets the user insert rows in the person table. It will produce the following output and will ask the following input (*italics*):

It starts with a list of the rows in the Person table:

```
1: Peter

2: Paul

3: Mary
```

Then it will ask for an ID and a name:

```
Give ID: 4

Give name: Elvis
```

The program will insert the row and show the new table

```
1: Peter
```

```
2: Paul
3: Mary
4: Elvis
```

It will then ask if the user wants to add another person:

```
Do you want to insert another? (y/n) y
```

If the user answers ‘y’, the program will ask for another ID and name. If the answer is no, the program will stop.

Jdbcex03 solution

```
private static final String CONN="jdbc:derby://localhost:1527/sample";
private static final String SELECT="SELECT ID, NAME FROM person";
private static final String INSERT="INSERT INTO person (ID, NAME) VALUES
(?, ?)";
public static void main(String[] args) {
  Scanner scanner = new Scanner(System.in);
  try(Connection conn = DriverManager.getConnection(CONN);
    Statement stm = conn.createStatement();
    PreparedStatement pstm = conn.prepareStatement(INSERT)){
    showTable(stm);
    String answer;
    do{
      System.out.print("ID?: ");
      int id = Integer.parseInt(scanner.nextLine());
      System.out.print("Name?: ");
      String name = scanner.nextLine();
      pstm.setInt(1, id);
      pstm.setString(2, name);
      pstm.executeUpdate();
      showTable(stm);
      System.out.print("Do you want to insert another? ");
      answer = scanner.nextLine();
    }while(answer.substring(0,1).equalsIgnoreCase("y"));
  }catch(SQLException ex){
    System.out.println(ex.getMessage());
  }
}
private static void showTable(Statement stm) throws SQLException {
  try(ResultSet rs = stm.executeQuery(SELECT)){
    while(rs.next()){
      int id = rs.getInt(1);
      String name = rs.getString(2);
      System.out.printf("%5d %20s%n", id, name);
    }
  }
}
```

jdbcex04: refactoring the previous exercise

When working with JDBC it is often a good a idea to separate the database code from the rest of the program. We can do that using a Data Access Layer (DAL). When using a separate *layer* we need a way to communicate between the two layers. For this we will us the Person class;

```
public class Person {
      private int ID;
      public int getID() {
            return ID;
      }
      public String getName() {
            return name;
      }
      private String name;
      public Person(int iD, String name) {
            ID = iD;
            this.name = name;
      }
}
```

A Data Access Layer should have all the methods that are needed to communicate with the database: get a list of Persons, get one person, insert one person, update one person, delete one person. In this case we will only implement the methods we need for this program: a list of persons and insert one person.

The DAL object will need a connection String. That is why the constructor accepts a connection string:

```
public class PersonDal {
  private String connectionString;
  private static final String INSERT = "INSERT INTO person (ID, NAME)
VALUES (?,?)";
  private static final String SELECT = "SELECT ID, NAME FROM person";
  public PersonDal(String connectionString){
    this.connectionString = connectionString;
  }
  public void insertPerson(Person p) throws SQLException {
    try(Connection conn = DriverManager.getConnection(connectionString);
        PreparedStatement pstm = conn.prepareStatement(INSERT)){
            pstm.setInt(1, p.getID());
            pstm.setString(2, p.getName());
            pstm.executeUpdate();
      }
  }
  public List<Person> getAllPersons() throws SQLException{
    List<Person> persons = new ArrayList<>();
    try(Connection conn = DriverManager.getConnection(connectionString);
        Statement stm = conn.createStatement();
        ResultSet rs = stm.executeQuery(SELECT)){
          while(rs.next()){
            int id = rs.getInt(1);
            String name = rs.getString(2);
            Person p = new Person(id, name);
            persons.add(p);
          }
    }
    return persons;
  }
}
```

And this is the main method that is refactored to use the DAL object:

```
public class Main {
     public static void main(String[] args) throws SQLException {
          PersonDAL dal = new PersonDAL(
                    "jdbc:derby://localhost:1527/sample");
          Scanner scanner = new Scanner(System.in);
          List<Person> persons = dal.getAllPersons();
          showTable(persons);
          String answer;
          do {
               Person p = askPerson();
               dal.insertPerson(p);
               persons = dal.getAllPersons();
               showTable(persons);
               System.out.print("Do you want to insert another?
```

```
(y/n)");
                  answer = scanner.nextLine();
            } while (answer.substring(0, 1).equalsIgnoreCase("y"));
      }
      private static Person askPerson() {
            Scanner scanner = new Scanner(System.in);
            System.out.print("Give ID: ");
            int id = Integer.parseInt(scanner.nextLine());
            System.out.print("Give name: ");
            String name = scanner.nextLine();
            Person p = new Person(id, name);
            return p;
      }
      private static void showTable(List<Person> persons){
            for (Person p : persons) {
                  System.out.printf("%d: %s\n", p.getID(),
p.getName());
            }
      }
}
```

jdbc05: autogenerated values

```
private static final String CREATE = "CREATE TABLE products (id int
generated always as identity primary key, name varchar(50), price
decimal)";
private static final String INSERT = "INSERT INTO products (name, price)
VALUES (?, ?)";
private static final String SELECT = "SELECT id, name, price FROM products
WHERE id = ?";
private static final String CONN = "jdbc:derby://localhost:1527/sample";
public static void main(String[] args) throws SQLException {
  try (Connection conn = DriverManager.getConnection(CONN);
       Statement stmt = conn.createStatement()) {
      if (!tableExist(conn, "PRODUCTS")) {
        stmt.executeUpdate(CREATE);
        System.out.println("TABLE products created");
      }
      try (PreparedStatement pstmt = conn.prepareStatement(INSERT,
PreparedStatement.RETURN_GENERATED_KEYS)) {
        pstmt.setString(1, "Pencil");
        pstmt.setBigDecimal(2, new BigDecimal(10));
        pstmt.executeUpdate();
        try (ResultSet rsgen = pstmt.getGeneratedKeys()) { // get generated
                                                           // id
            if (rsgen.next()) {
              int id = rsgen.getInt(1);
              System.out.printf("Pencil added with key %d\n", id);
              showProduct(conn, id);
            }
        }
      }
   }
}
private static void showProduct(Connection conn, int id) throws
SQLException {
```

```
  try (PreparedStatement pstmtselect = conn.prepareStatement(SELECT)) {
      pstmtselect.setInt(1, id);
      try (ResultSet rs = pstmtselect.executeQuery()) {
        while (rs.next()) {
            id = rs.getInt(1);
            String name = rs.getString(2);
            BigDecimal price = rs.getBigDecimal(3);
            System.out.printf("%d: %s (%.2f)\n", id, name, price);
        }
      }
  }
}
private static boolean tableExist(Connection conn, String tablename) throws
SQLException {
  DatabaseMetaData meta = conn.getMetaData();
  try (ResultSet rs = meta.getTables(null, null, tablename.toUpperCase(),
null)) {
      if (rs.next()) {
        return true;
      } else {
        return false;
    }
  }
```

Jdbc06

```
private static final String SELECT= "SELECT * FROM person";
private static final String CONN = "jdbc:derby://localhost:1527/sample";
public static void main(String[] args) throws SQLException {
  try(Connection conn = DriverManager.getConnection(CONN);
      JdbcRowSet jdbcRs = new JdbcRowSetImpl(conn)){
    jdbcRs.setCommand(SELECT);
    jdbcRs.execute();
    jdbcRs.last(); // goto last row in rowset
    while (!jdbcRs.isBeforeFirst()) {
      int id = jdbcRs.getInt(1);
      String name = jdbcRs.getString(2);
      System.out.printf("%d: %s\n", id, name);
      jdbcRs.previous();
    }
  }
}
```

Jdbc07

```
private static final String SELECT = "SELECT * FROM person";
private static final String CONN = "jdbc:derby://localhost:1527/sample";
public static void main(String[] args) throws SQLException {
  try(JdbcRowSet jdbcRs = new JdbcRowSetImpl()){
    jdbcRs.setUrl(CONN);
    jdbcRs.setCommand(SELECT);
    jdbcRs.execute();
    jdbcRs.moveToInsertRow();
    jdbcRs.updateInt(1, 4);
    jdbcRs.updateString(2, "Elvis");
    jdbcRs.insertRow();
    jdbcRs.moveToCurrentRow(); // move back to the row we were on
    System.out.println("Row added");
  }
}
```

Jdbc08

```
private static final String CONN = "jdbc:derby://localhost:1527/sample";
public static void main(String[] args) throws SQLException {
  try(CachedRowSet crs = new CachedRowSetImpl()) {
    crs.setCommand("SELECT * FROM person");
    crs.setKeyColumns(new int[] { 1 }); // Define primary keys
    readData(crs);
    displayData(crs);
    doubleName(crs);
    Scanner scanner = new Scanner(System.in);
    System.out.print("Hit <return> key");
    scanner.nextLine();
    writeData(crs);
    readData(crs);
    displayData(crs);
    divideName(crs);
    writeData(crs);
  }
}
private static void divideName(CachedRowSet crs) throws SQLException {
  for (crs.first(); !crs.isAfterLast(); crs.next()) {
      String name = crs.getString(2);
      name = name.substring(0, name.length() / 2);
      crs.updateString(2, name);
      crs.updateRow();
  }
}
private static void doubleName(CachedRowSet crs) throws SQLException {
  for (crs.first(); !crs.isAfterLast(); crs.next()) {
      String name = crs.getString(2);
      name = name + name;
      crs.updateString(2, name);
      crs.updateRow(); //does not write to the database
  }
}
private static void writeData(CachedRowSet crs) throws SQLException,
     SyncProviderException {
  try(Connection conn = DriverManager.getConnection(CONN) ) {
      crs.acceptChanges(conn);
  }
}
private static void readData(CachedRowSet crs) throws SQLException{
  try(Connection conn = DriverManager.getConnection(CONN) ) {
      crs.execute(conn);
  }
}
private static void displayData(CachedRowSet crs) throws SQLException {
  while (crs.next()) {
      int id = crs.getInt(1);
      String name = crs.getString(2);
      System.out.printf("%d: %s\n", id, name);
  }
}
```

Jdbc09

```
public static void main(String[] args) throws SQLException {
  Scanner scanner = new Scanner(System.in);
```

```
  CachedRowSet crs = null;
  try {
    crs = new CachedRowSetImpl();
    crs.setUrl("jdbc:derby://localhost:1527/sample");
    crs.setCommand("SELECT * FROM person");
    crs.setKeyColumns(new int[] { 1 });
    crs.execute();
    System.out.print("Hit any key");
    scanner.nextLine(); // change data in database (e.g. using ij-tool)
    doubleName(crs);
    crs.acceptChanges();
  } catch (SyncProviderException ex) {
    resolveConflict(crs, ex);
  } finally {
    if (crs != null)
      crs.close();

  }
}
private static void resolveConflict(CachedRowSet crs,
      SyncProviderException ex) throws SQLException {
  SyncResolver resolver = ex.getSyncResolver();
  while (resolver.nextConflict()) {
    int row = resolver.getRow(); //get rownumber
    crs.absolute(row); // goto to conflicting row in CachedRowSet
    switch (resolver.getStatus()) {
      case SyncResolver.UPDATE_ROW_CONFLICT:
            String nameInRowSet = crs.getString(2);
            Object nameInDataBase = resolver.getConflictValue(2);
            if (nameInDataBase != null) {
                  System.out.printf("No update: %s(rowset) vs %s(db)\n",
                                          nameInRowSet, nameInDataBase);
            } else {
                  System.out.printf("%s not updated because of previous
errors",nameInRowSet);
            }
            break;
      case SyncResolver.INSERT_ROW_CONFLICT:
      case SyncResolver.DELETE_ROW_CONFLICT:
            System.out.println("Other conflict");
            break;
    }
  }
}

private static void doubleName(CachedRowSet crs) throws SQLException {
      for (crs.first(); !crs.isAfterLast(); crs.next()) {
            String name = crs.getString(2);
            name = name + name;
            crs.updateString(2, name);
            crs.updateRow(); // does not write to the database
      }
}
```

Internationalisation/localisaton

International01

```
public static void main(String[] args) {
  //Locale currentLocale = Locale.FRENCH;
  //Locale currentLocale = new Locale("nl");
  Locale currentLocale = new Locale("de");
  // getBundle(basename, locale) loads the right locale file
  // if the file is not available (German), it will use the default locale
  // of the system and the corresponding properties file
  // if the corresponding properties file for the default locale of the
system
  // is not available, the base file is used (MessageBundle.properties)
  ResourceBundle messages =
ResourceBundle.getBundle("MessageBundle",currentLocale);
  System.out.println(messages.getString("greetings"));
  System.out.println(messages.getString("inquiry"));
  System.out.println(messages.getString("farewell"));
}
```

MessageBuncle.properties

```
greetings = Hello.
farewell = Goodbye.
inquiry = How are you?
```

MessageBundle_fr.properties

```
greetings = Bonjour.
farewell = Au revoir.
inquiry = Comment allez-vous?
```

International02

```
public static void main(String[] args) {
  //Locale currentLocale = new Locale("fr", "BE");
  Locale currentLocale = new Locale("nl", "BE");
  ResourceBundle messages =
ResourceBundle.getBundle("org.betavzw.MyBundle",currentLocale);
  System.out.println(messages.getString("greetings"));
  System.out.println(messages.getString("inquiry"));
  System.out.println(messages.getString("farewell"));
}
public class MyBundle_fr_BE extends ListResourceBundle {
      private Object[][] contents = {
                  {"greetings", "Bonjour"},
                  {"farewell", "Au revoir"},
                  {"inquiry", "Comment allez-vous?"}
      };
      @Override
      protected Object[][] getContents() {
            return contents;
      }
}
```

International03

```
public static void main(String[] args) {
  Locale currentLocale = Locale.US;
  //Locale currentLocale = new Locale("fr", "BE");
  double pi = Math.PI;
  int answer = 4200;
  NumberFormat formatter = NumberFormat.getNumberInstance(currentLocale);
```

```
  String doubleOutput = formatter.format(pi);
  String intOutput = formatter.format(answer);
  ResourceBundle bundle = ResourceBundle.getBundle("org.betavzw.Bundle",
currentLocale);
  System.out.printf(bundle.getString("PI_Format"),doubleOutput);
  System.out.printf(bundle.getString("ANSWER_Format"), intOutput);
  DateTimeFormatter dateFormatter = DateTimeFormatter
         .ofLocalizedDate(FormatStyle.SHORT).withLocale(currentLocale);
  LocalDate date = LocalDate.of(1935, Month.JANUARY, 8);
  String dateString = dateFormatter.format(date);
  System.out.printf(bundle.getString("ELVIS_Format"), dateString);
}
public class Bundle_en_US extends ListResourceBundle {
  private Object[][] contents = {
      {"PI_Format", "PI is %s\n"},
      {"ANSWER_Format", "%s is the answer to the ultimate question of Life,
the Universe and Everything\n"},
      {"ELVIS_Format", "Elvis was born on %s\n"},
  };
  @Override
  protected Object[][] getContents() {
            return contents;
  }
}
```

International04

```
private static Collator collator = Collator.getInstance(Locale.FRANCE);
//private static Collator collator = Collator.getInstance(Locale.US);
public static void main(String[] args) {
      List<String> strings = new ArrayList<String>();
      strings.add("péché");
      strings.add("pêche");
      strings.add("sin");
      strings.add("peach");
      Collections.sort(strings, new Comparator<String>() {
            @Override
            public int compare(String o1, String o2) {
                  return collator.compare(o1, o2);
            }
      });
      for(String s: strings) {
            System.out.println(s);
      }
}
```

File streams

fileIO01

```
public static void main(String[] args) throws IOException {
  Scanner scanner = new Scanner(System.in);
  try(PrintWriter writer = new PrintWriter(new FileWriter("out.txt"))) {
  String line;
  System.out.println("Give line (end with empty line)");
  while(!"".equals(line = scanner.nextLine())) {
      writer.println(line);
  }
  System.out.println("<TERMINATED PROGRAM>");
```

```
  }
}
```

FileIO02

```
public static void main(String[] args) throws IOException {
  try(BufferedReader reader = new BufferedReader(new
FileReader("../fileIO01/out.txt"))){
      String line;
      while ((line = reader.readLine()) != null) {
            System.out.println(line);
      }
  }
}
```

FileIO03

```
public static void main(String[] args) throws IOException {
  ArrayList<Person> persons = new ArrayList<Person>();
  persons.add(new Person("Elvis", 79));
  persons.add(new Person("Priscilla", 69));
  try (ObjectOutputStream out = new ObjectOutputStream(new
FileOutputStream("persons.bin"))){
      out.writeObject(persons);
      System.out.println("<Program terminated>");
  }
}
public class Person implements Serializable {
      private static final long serialVersionUID = 1L;
      private String name;
      private int age;
      public Person(String name, int age) {
            this.name = name;
            this.age = age;
      }
      public String getName() {
            return name;
      }
      public int getAge() {
            return age;
      }
}
```

FileIO04

```
public static void main(String[] args) throws IOException,
ClassNotFoundException {
  List<Person> persons = null;
  try (ObjectInputStream in = new ObjectInputStream(new
FileInputStream("../fileIO03/persons.bin"))){
      persons = (List<Person>)in.readObject();
      for (Person p: persons){
            System.out.printf("%s: %s\n", p.getName(), p.getAge());
      }
  }
}
```

FIleIOex01

Write a Java program that shows the contents of a file and gives the user the opportunity to add one line. When the file does not exist yet (the first time the program is run) the program should output "File does not exist yet".

Input/Output could be the following:

File does not exist yet
Give new line: *Hi there*
Line is added

The next time the program is run it should produce the following output:

Hi there
Give new line: *What?*
Line is added

FIleIOex01 solution

```
private static final String FILENAME = "myfile.txt";
public static void main(String[] args) throws IOException {
 try(BufferedReader reader = new BufferedReader(new FileReader(FILENAME))){
      String line;
      while ((line = reader.readLine()) != null) {
            System.out.println(line);
      }
 }catch(FileNotFoundException fe) {
      System.out.println("File does not exist yet");
 }
 Scanner scanner = new Scanner(System.in);
 System.out.print("Give new line: ");
 String newLine = scanner.nextLine();
 //Open for append
 try(PrintWriter writer = new PrintWriter(new FileWriter(FILENAME, true))){
      writer.println(newLine);
      System.out.println("Line is added");
 }
}
```

A slightly more extensive exercise

Given the following textfile:

```
I:1;table;40.25
I:2;chair;25.30
U:1;table;50.25
D:2;chair;25.30
```

Write a program that can read this file and execute the commands in a database:

- I means insert a product (id, name, price)
- U means update a product
- D means delete a product

I have written this program with a Data Access Layer based on the following interface:

```
public interface ProductDAL {
      void createTable() throws SQLException;
      void insertProduct(Product product) throws SQLException;
      void deleteProduct(Product product) throws SQLException;
      void updateProduct(Product product) throws SQLException;
      List<Product> getAllProducts() throws SQLException;
}
```

The CREATE TABLE statement for the database looks like this:

```
CREATE TABLE product (id int primary key, name varchar(50),
price Decimal(7,2))
```

After executing the commands the program should show the content of the *product* table:

```
1: table costs €   50,25
```

Solution

I have started with a Data Transfer Object: Product:

```
public class Product {
	private int id;
	private String name;
	private BigDecimal price;
	public Product(int id, String name, BigDecimal price) {
		super();
		this.id = id;
		this.name = name;
		this.price = price;
	}
	public int getId() {
		return id;
	}
	public String getName() {
		return name;
	}
	public BigDecimal getPrice() {
		return price;
	}
}
```

I also thought it would be a good idea to define an enumeration for the different commands:

```
public enum DbUpdater { INSERT, UPDATE, DELETE}
```

The last 'simple' class I created was a combination of the enum and the DTO:

```
public class ProductUpdater {
	private DbUpdater updateType;
	private Product product;
	public ProductUpdater(DbUpdater updateType, Product product) {
		super();
		this.updateType = updateType;
		this.product = product;
	}
	public DbUpdater getUpdateType() {
		return updateType;
	}
	public Product getProduct() {
		return product;
	}
}
```

The class that reads the file is initialised with the filename. It reads the file and returns a list of ProductUpdaters. In order to ease the conversion from the characters I, U and D to a DbUpdater enum I used a map. This map is initialized with a static initializer.

```
public class DBManagerReader {
	private static final Map<Character, DbUpdater> converter = new
HashMap<>();
	static{
		converter.put('I', DbUpdater.INSERT);
		converter.put('D', DbUpdater.DELETE);
		converter.put('U', DbUpdater.UPDATE);
	}
```

```
	private String filename;
	public DBManagerReader(String filename){
		this.filename = filename;
	}
	public List<ProductUpdater> readFile() throws IOException{
		List<ProductUpdater> updaters = new ArrayList<>();
		try(BufferedReader reader = new BufferedReader(new
FileReader(filename))){
			String line;
			while ((line = reader.readLine()) != null){
				ProductUpdater productUpdater = processLine(line);
				updaters.add(productUpdater);
			}
		}
		return updaters;
	}
	private ProductUpdater processLine(String line){
		String[] split = line.split(":");
		DbUpdater updater = converter.get(split[0].charAt(0));
		String[] productItems = split[1].split(";");
		int id = Integer.parseInt(productItems[0]);
		String name = productItems[1];
		BigDecimal price = new BigDecimal(productItems[2]);
		ProductUpdater productUpdater = new ProductUpdater(updater, new
Product(id, name, price));
		return productUpdater;
	}
}
```

The implementation class for the ProductDal looks like this:

```
public class ProductDALImpl implements ProductDAL {
  private String connectionString;
  private static final String SELECT = "SELECT id, name, price FROM
product";
  private static final String INSERT = "INSERT INTO product (id, name,
price) values (?,?,?)";
  private static final String DELETE = "DELETE FROM product where id= ?";
  private static final String UPDATE = "UPDATE product SET name=?, price=?
WHERE id=?";
  private static final String CREATE = "CREATE TABLE product (id int
primary key, name varchar(50), price Decimal(7,2))";
  private static final String DROP = "DROP TABLE product";
  public ProductDALImpl(String connectionString) {
    super();
    this.connectionString = connectionString;
  }
  @Override
  public void createTable() throws SQLException{
    try(Connection conn = DriverManager.getConnection(connectionString);
        Statement stm = conn.createStatement()){
      try{
        stm.executeUpdate(CREATE);
      }catch(SQLException ex){
        stm.executeUpdate(DROP);
        stm.executeUpdate(CREATE);
      }
```

```
    }
  }
  @Override
  public void insertProduct(Product product) throws SQLException {
    try(Connection conn = DriverManager.getConnection(connectionString);
            PreparedStatement pstm = conn.prepareStatement(INSERT)){
      pstm.setInt(1, product.getId());
      pstm.setString(2, product.getName());
      pstm.setBigDecimal(3, product.getPrice());
      pstm.executeUpdate();
    }
  }
  @Override
  public void deleteProduct(Product product) throws SQLException {
    try(Connection conn = DriverManager.getConnection(connectionString);
            PreparedStatement pstm = conn.prepareStatement(DELETE)){
      pstm.setInt(1, product.getId());
      pstm.executeUpdate();
    }
  }
  @Override
  public void updateProduct(Product product) throws SQLException {
    try(Connection conn = DriverManager.getConnection(connectionString);
            PreparedStatement pstm = conn.prepareStatement(UPDATE)){
      pstm.setString(1, product.getName());
      pstm.setBigDecimal(2, product.getPrice());
      pstm.setInt(3, product.getId());
      pstm.executeUpdate();
    }
  }
  @Override
  public List<Product> getAllProducts() throws SQLException {
    List<Product> products = new ArrayList<>();
    try(Connection conn = DriverManager.getConnection(connectionString);
            Statement stm = conn.createStatement();
      ResultSet rs = stm.executeQuery(SELECT)){
      while(rs.next()){
        int id = rs.getInt("id");
        String name = rs.getString("name");
        BigDecimal price = rs.getBigDecimal("price");
        Product p = new Product(id, name, price);
        products.add(p);
      }
    }
    return products;
  }
}
```

The main-function combines the two functionalities:

```
private static final String FILE = "src/data.txt";
private static final String CONN = "jdbc:derby:sample;create=true";
public static void main(String[] args) throws SQLException, IOException {
  System.out.println(Paths.get(".").toAbsolutePath());
  DBManagerReader reader = new DBManagerReader(FILE);
  ProductDAL dal = new ProductDALImpl(CONN);
  dal.createTable();
```

```
  List<ProductUpdater> productUpdaters = reader.readFile();
  for(ProductUpdater updater : productUpdaters){
    Product product = updater.getProduct();
    switch(updater.getUpdateType()){
      case INSERT:
            dal.insertProduct(product);
            break;
      case DELETE:
            dal.deleteProduct(product);
            break;
      case UPDATE:
            dal.updateProduct(product);
            break;
      default:
            break;
    }
  }
  List<Product> products = dal.getAllProducts();
  for(Product p: products){
    System.out.printf("%d: %s costs € %7.2f%n", p.getId(), p.getName(),
p.getPrice());
  }
}
```

Multithreading

Multithread01

```
public static void main(String[] args) {
	ThreadRunner tr1 = new ThreadRunner("TR1");
	ThreadRunner tr2 = new ThreadRunner("TR2");
	System.out.println("Both 'threads' run in main thread");
	tr1.run();
	tr2.run();
	System.out.println("Both 'threads' run in different thread");
	Thread t1 = new Thread(new ThreadRunner("TR1"));
	Thread t2 = new Thread(new ThreadRunner("TR2"));
	t1.start(); //invokes run() method from Runnable
	t2.start();
	System.out.println("End of program (will end before TR1 and TR2)");
}
public class ThreadRunner implements Runnable {
	private String name;

	public ThreadRunner(String name) {
		this.name = name;
	}

	@Override
	public void run() {
		try {
			for (int i = 0; i < 5; i++) {
				System.out.println("Running " + name);
				Thread.sleep(1000);
			}
		} catch (InterruptedException e) {
	// InterruptedExceptionwhen an another thread interrupts this thread
			e.printStackTrace();
		}
	}
}
```

Multitreahd02: atomic operations

```
public class Main {

	public static void main(String[] args) throws InterruptedException {
	  Counter counter = new Counter();
	  Thread t1 = new Thread(new Runnable() {
	      @Override
	      public void run() {
	          for (int i=0;i<1000;i++) {
	              counter.increment();
	              counter.incrementAtomic();
	          }
	      }
	  });
	  Thread t2 = new Thread(new Runnable() {
	      @Override
	      public void run() {
	          for (int i=0;i<1000;i++) {
	              counter.increment();
	              counter.incrementAtomic();
	          }
```

```
            }
        });
        t1.start();
        t2.start();
        t1.join();
        t2.join();
        System.out.println("The value of counter is "+counter.getValue());
        System.out.println("The atomic value of counter is " +
                counter.getAtomicValue());
      }

}
class Counter{
    private AtomicInteger atomicCounter = new AtomicInteger();
    private int counter;
    public void incrementAtomic(){
        atomicCounter.incrementAndGet();
    }
    public int getAtomicValue(){
        return atomicCounter.get();
    }
    public void increment(){
        counter++;
    }
    public int getValue(){
        return counter;
    }
}
```

Multithread03

Using an executor service we can setup a threadpool.

```
public class Main {
 private static final int PORT_NUMBER=4242;
 public static void main(String[] args) throws IOException {

     ExecutorService executor = Executors.newFixedThreadPool(2);
     try(ServerSocket socket = new ServerSocket(PORT_NUMBER)){
        while(true) {
           Socket clientSocket = socket.accept();
           executor.execute(new ClientThread(clientSocket));
        }
     }
     //executor.shutdown();
   }
}

class ClientThread implements Runnable{
  private Socket clientSocket;
  public ClientThread(Socket client){
      this.clientSocket = client;
  }
  @Override
  public void run() {
     try(PrintWriter out = new PrintWriter(clientSocket.getOutputStream(),
true);
     BufferedReader in = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()))){
```

```
        String inputLine, outputLine;
        while ((inputLine = in.readLine())!= null){
            System.out.println(inputLine);
            if (inputLine.equals(".exit")) break;
            outputLine = "Echo: " + inputLine;
            out.println(outputLine);
        }
        clientSocket.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
  }
}
```