Programming 2 (SS 2023)
Saarland University
Faculty MI
Compiler Design Lab

Sample Solution 9
Java

Prof. Dr. Sebastian Hack
Pascal Lauer, M. Sc.
Marcel Ullrich, B. Sc.

The calendar indicates which script chapters you should study in conjuction with each lecture. The exercises are designed to enhance your understanding of the lecture material and prepare you for the mini-tests and the final exam. Additional exercises can be found at the end of each chapter in the script.
The difficulty of an exercise on the sheet is determined by the number of annotated 'X' and 'O' marks in the tic-tac-toe field, with four levels (1-4) increasing by one mark per level.

Exercise 9.1:

1. Which access modifiers exist in Java? Which properties do they have? Which visibility scope does a variable have that is declared in a class without any access modifiers? How can you check your hypothesis?
2. What are the fundamental data types in Java? Which types are automatically converted into others, which are not?
3. What is the `equals` method used for in Java? Why is the usage of == insufficient in many cases?

Solution

1. The access modifiers are `public`, `private` and `protected`. Their properties are depicted in the table below.

	Class	**Package**	**Derived Class**	**Rest**
`public`	✓	✓	✓	✓
`protected`	✓	✓	✓	
no modifier	✓	✓		
`private`	✓			

As can be seen in the table, variables without an access modifier are a special case that allows access to the variable within the package, but prevents access otherwise. One can figure this out by creating a minimal Java project consisting of a class with one member, another class inside the same package and two classes in another package, where one of these two classes inherits from the original class. By trying to access the member within each class, one can easily see that this is only possible from within the class in the same package.

2. The fundamental datatypes and possible conversions are depicted below.

$$\begin{tikzcd}
\texttt{byte} \arrow[r] & \texttt{short} \arrow[r] & \texttt{int} \arrow[r] & \texttt{long} \arrow[r] & \texttt{float>} \arrow[r] & \texttt{double} \\
\texttt{boolean} & \texttt{char} \arrow[ur] & & & &
\end{tikzcd}$$

3. The method `equals` is used to check for semantic equality and to specify when two objects should be semantically equal.

```java
    Fraction A = new Fraction(1, 2);
    Fraction B = new Fraction(2, 4);

    boolean eq = (A == B); \\ evaluates to false
```

In the example above, both fractions represent the same value, but the == comparison evaluates to `false` since two different objects are involved. A self-written `equals` method can declare that both objects are equal and return the value `true`.

Exercise 9.2:

1. Implement a simple calculator in Java. Your program shall receive the following three parameters:

   - `int op` describes the operator used for the calculation.
   - `int l` and `int r` are the operands. The operands must be between 0 and 10000 (both inclusive).

   The following operators exist:

Operator	Result
1	$l + r$
2	$l - r$
3	$l * r$
4	$l \, / \, r$

   Use the code skeleton provided below and implement the given method `compute`:

```java
public static int compute(int op, int l, int r)
        throws IllegalArgumentException {
    res = 0;
    ...
    return res;
}
```

   If your program receives invalid arguments, it should throw an `IllegalArgumentException`. Make sure to consider all reasonable edge cases (e.g. an invalid number being passed as an operand).

2. Write a JUnit test suite for the program described in a) which achieves maximal code coverage.

Solution

```java
package calculator;

public class Calculator {
    public static int compute(int op, int l, int r)
            throws IllegalArgumentException {
        int res = 0;

        if (l < 0 || r < 0 || l > 10000 || r > 10000) {
            throw new IllegalArgumentException("Value out of range!");
        }
        else if (op == 4 && r == 0) {
            throw new IllegalArgumentException("Division by zero!");
        }

        switch (op) {
            case 1:
                res = l + r;
                break;
            case 2:
                res = l - r;
                break;
            case 3:
                res = l * r;
                break;
            case 4:
                res = l / r;
                break;
            default:
                throw new IllegalArgumentException("Invalid operand!");
        }

        return res;
    }
}
```

```java
package calculator;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertThrows;

import org.junit.Test;

public class CalculatorTests {
    @Test
    public void test_basic_op() {
        assertEquals(42, Calculator.compute(1,32,10));
        assertEquals(7, Calculator.compute(2, 15, 8));
        assertEquals(120, Calculator.compute(3, 12, 10));
        assertEquals(10, Calculator.compute(4, 9001, 900));
    }

    @Test
    public void test_div_invalid() {
        assertThrows(IllegalArgumentException.class,
                () -> Calculator.compute(4, 20, 0));
    }

    @Test
    public void test_invalid_l_range() {
        assertThrows(IllegalArgumentException.class,
                () -> Calculator.compute(3, -10, 2));
        assertThrows(IllegalArgumentException.class,
                () -> Calculator.compute(2, 1, 11111));
    }

    @Test
    public void test_invalid_op_range() {
        assertThrows(IllegalArgumentException.class,
                () -> Calculator.compute(-8, 10, 4));
        assertThrows(IllegalArgumentException.class,
                () -> Calculator.compute(42, 8, 13));
    }
}
```

Exercise 9.3:

Dieter Schlau found a crate full of phones and would like to sort them in a catalogue to find their owners. To this end, he has created a class `Mobilephone` and a class `Smartphone`. Help him by completing the implementations:

1. Add a reasonable constructor for both classes. The battery life of a mobile phone exhausts after 1500 minutes, the one of a smart phone after 800 minutes.

2. Create two methods for adding and removing contacts for the class `Mobilephone`. Both procedures use 2 minutes of the battery life.

3. Create two methods for adding and removing applications for the class `Smartphone`. An installation costs 5 minutes of battery life, a deinstallation only 2 minutes.

```java
public class Mobilephone {
  int contacts;
  int number;
  int battery;
  //...
}
```

```java
public class Smartphone extends Mobilephone {
  int apps;
  //...
}
```

Solution

```java
public class Mobilephone {
  int contacts;
  int number;
  int battery;

  public Mobilephone(int contacts, int number){
    this.contacts = contacts;
    this.number = number;
    this.battery = 1500;
  }

  public void addContact(){
    contacts++;
    battery -= 2;
  }

  public void removeContact(){
    contacts--;
    battery -= 2;
  }
}
```

```java
public class Smartphone extends Mobilephone {
  int apps;

  public Smartphone (int contacts, int number, int apps){
    super(contacts, number);
    this.apps = apps;
    this.battery = 800;
  }

  public void installApp(){
    apps++;
    battery -= 5;
  }

  public void deinstallApp(){
    apps--;
    battery -= 2;
  }
}
```

Exercise 9.4:

In this exercise, you will get to know an anomaly of Java regarding the modulo operator. Consider the natural numbers modulo n, i.e. the numbers in range $[0, n-1]$. For $n = 10$, we expect that $3 - 5 \equiv 8 \mod 10$, since $-2 - \left\lfloor \frac{-2}{10} \right\rfloor \cdot 10 = 8$. What does `(3-5) % 10` evaluate to in Java? State an expression which gives the correct, positive modulus with respect to a number n for a negative number a. The expression should evaluate to 8 for the numbers $n = 10$ and $a = -2$.

Solution

`(3-5) % 10` evaluates to -2 in Java.

```java
/**
 * Returns a mod n.
 *
 * Calculates the standard modulus if a is positive,
 * otherwise casts the negative modulus to a positive one.
 *
 * @param a - Dividend
 * @param n - Divisor
 * @return
 */
public static int modMinus(int a, int n) {
  while (a < 0) {
    a = a + n;
  }

  return a % n;
}
```

Exercise 9.5:

Konrad Klug has discovered Java for himself. He wants to find out whether arguments in Java are passed by *value* or by *reference*. In this exercise, you will help Konrad to evaluate his experiments:

1. What is the output of the program below?

2. How does a program behave when you pass fundamental data types versus reference data types? Will these be passed by reference or by value?

```java
public class Main {
    public static void main(String[] args) {
        Color color = new Color(0,0,0);

        color.incRed(color);
        color.incValue(color.green);

        System.out.println(color.red);
        System.out.println(color.green);
    }
}
```

```java
public class Color {
    public int red, green, blue;

    public void incRed(Color color) {
        color.red++;
    }

    public void incValue(int value) {
        value++;
    }

    public Color(int red, int green, int blue) {
        this.red = red;
        this.green = green;
        this.blue = blue;
    }
}
```

Solution

Attention: While this implementation is a good example, it is otherwise unreasonable. Normally, one would expect that `Color::incRed` increments the red value of the object it is called on and that it does not have parameters. The given methods generally do not really make sense for a class representing a color.

1. The output of the program is

```
1
0
```

2. If you pass a fundamental type like `int` or `bool` the data is copied into the function. That is called *call by value*. If you change this value in the function, nothing happens outside of the function. If you pass other datatypes like complex objects it is called *call by reverence*. If you change the fields of the object, they are changed in the object so the change is noticeable outside of the function. Hence, in `incRed` a reference to the object `color` is passed and the member `red` of this object is incremented. For the second part, since fundamental data types are passed by value, `value` is only locally incremented inside the method `incValue`.

Exercise 9.6:

In this exercise, you will learn how function calls are resolved in Java. The code parts are to be interpreted as connected, the divisions are supposed to highlight the program parts relevant for each subtask.

1. Determine the static and dynamic type of each variable.

```java
A a = new A();
A b = new B();
A c = new C();

B bB = new B();
C cC = new C();
```

2. Is the static or dynamic type used when a method is overloaded? What about overwritten methods?

3. For all classes, determine those methods which are overloaded and those which are overwritten.

```java
class A {
  void f (boolean b) { System.out.println("A.f(boolean)"); }

  void f (double d) { System.out.println("A.f(double)"); }
}

class B extends A {
  void f (boolean b) { System.out.println("B.f(boolean)"); }

  void f (int i) { System.out.println("B.f(int)"); }
}

class C extends B {
  void f (boolean b) { System.out.println("C.f(boolean)"); }

  void f (float f) { System.out.println("C.f(float)"); }

  void f (int i) { System.out.println("C.f(int)"); }

  void f (short s) { System.out.println("C.f(short)"); }
}
```

4. Now determine the output of the following calls by searching for the most specific signature for each call (with respect to the static and dynamic type).

```java
cC.f(1);
cC.f(1.0f);

a.f(true);
b.f(true);
c.f(true);

bB.f(1);
b.f(1);
cC.f(1.0);
c.f((short) 1);
```

Solution

1.

Variable	Static Type	Dynamic Type
a	A	A
b	A	B
c	A	C
bB	B	B
cC	C	C

2. Overloading: Static, Overwriting: Dynamic

3.

Class	Methods				
A	A.f(double)	A.f(boolean)			
B		B.f(boolean)	B.f(int)		
C		C.f(boolean)	C.f(int)	C.f(short)	C.f(float)

If methods are vertically adjacent in the table, the lower method overwrites the upper method (e.g. `C.f(int)` overwrites `B.f(int)`). Methods in a row with the same name but differing argument types are overloaded (e.g. `B.f(int)` overloads `A.f(double)`).

4.

`cC.f(1)`	$\Longrightarrow$	`C.f(int)`
`cC.f(1.0f)`	$\Longrightarrow$	`C.f(float)`
`a.f(true)`	$\Longrightarrow$	`A.f(boolean)`
`b.f(true)`	$\Longrightarrow$	`B.f(boolean)`
`c.f(true)`	$\Longrightarrow$	`C.f(boolean)`
`bB.f(1)`	$\Longrightarrow$	`B.f(int)`
`b.f(1)`	$\Longrightarrow$	`A.f(double)`
`cC.f(1.0)`	$\Longrightarrow$	`A.f(double)`
`c.f((short) 1)`	$\Longrightarrow$	`A.f(double)`

The most specific signature appears in the calls `b.f(1)` and `c.f((short) 1)`, since here the signature `f(double)` is chosen according to the static type `A`. Since this method is neither defined in `B` nor `C`, the method of `A` is taken which is transitively inherited by `B` and `C`.

Exercise 9.7: Cutpoint

Dieter Schlau has not yet given up on his dream of becoming a video game developer. He is working to exhaustion on his new game, "Collierycraft." In this game, there are supposed to be different kinds of animals with different characteristics. The animals "Sheep," "Cow," and "Goat" shall be able to be milked. The animals "Donkey", "Horse" and "Unicorn" shall be able to be ridden. To make the game more difficult, the animals "Cow", "Goat", "Horse" and "Unicorn" are able to attack other animals.
To represent the animals Dieter decides to create a central parent class, `Animal` from which all animals inherit. Additionally, he wants to create a separate class for each ability, such as the class `Milkable` with the method `milk()` from which the respective animals inherit.

1. Dieter quickly notices a problem with this approach. Describe why he cannot create the class structure in this way.

2. His friend, No Hao, proposes to use an interface. Dieter is confused, but she explains that an interface simply works like an abstract class, except that inside interfaces, methods can only be declared and not implemented. Noticing that Dieter still has a confused look on his face, she points him to Subsection 8.7.1: Interfaces in the lecture notes.

   Declare suitable interfaces, classes, and methods for Dieter's animals. Milkable animals should be able to be milked. Riding animals should be able to be saddled and ridden by other animals. Aggressive animals can attack other animals. Make it clear which methods are only declared and which actually have an implementation.

   How do interfaces solve the problem from task 1?

3. Dieter now wants to be able to specify more precisely whether the animals attack with their hooves or their horns. For this, he wants to create two new interfaces `Hooved` and `Horned` that inherit from the interface for aggressive animals. Hooved animals (i.e., unicorns and horses) can trample other animals, and animals with horns (unicorns and goats) can sting other animals. Also change the already declared *classes* accordingly and adapt the inheritance hierarchy as a whole.

4. In the meantime, Dieter has learned that you can actually specify default implementations for methods in interfaces by using the `default` keyword. Help him by specifying a default implementation for the attack method in the interfaces declared in Task 3 that each call the more specific method. Does Dieter still need to implement the `attack()` method in the `Unicorn` class?

Solution

1. All animals inherit from `Animal`, but they can not also inherit from `Milkable` and `Attacking`, because in Java every class can have at most one parent class.

2. This restriction does not hold for interfaces. Every class can implement any number of interfaces. The interfaces, classes and methods are declared/implemented as follows:

```java
abstract class Animal {
  // implementation
}
interface Milkable {
  void milk();
}
interface Ridable {
  void saddle();
  void ride(Animal rider);
}
interface Attacking {
  void attack(Animal target);
}
class Cow extends Animal implements Attacking, Milkable {
  public void milk() {...}
  public void attack(Animal other) {...}
}
class Goat extends Animal implements Attacking, Milkable {
  public void milk() {...}
  public void attack(Animal other) {...}
}
class Sheep extends Animal implements Milkable {
  public void milk() {...}
}
class Donkey extends Animal implements Ridable {
  public void saddle() {...}
  public void ride(Animal rider) {...}
}
class Horse extends Animal implements Ridable, Attacking {
  public void saddle() {...}
  public void ride(Animal rider) {...}
  public void attack(Animal other) {...}
}
class Unicorn extends Animal implements Ridable, Attacking {
  public void saddle() {...}
  public void ride(Animal rider) {...}
  public void attack(Animal other) {...}
}
```

   Methods such as `void saddle();` are declarations. However, if the method is followed by curly braces (such as `void saddle() {...}`), there would be an implementation, which is omitted here to save space. It is important that methods in interfaces are `public` by default, and thus the overriding methods must also be `public`.

3. The interfaces and implemented methods in the respective classes look as follows:

```java
interface Hooved extends Attacking {
  void trample(Animal target);
}
interface Horned extends Attacking {
  void sting(Animal target);
}
class Unicorn extends Animal implements Ridable, Horned, Hooved {
  public void trample(Animal target) {}
  public void sting(Animal target) {}
}
class Horse extends Animal implements Ridable, Hooved {
  public void trample(Animal target) {}
}
```

```java
class Goat extends Animal implements Ridable, Horned {
  public void sting(Animal target) {}
}
```

The modified classes also implement `Attacking` by transitivity. So we do not have to specify this explicitly (but it would also not be wrong).

```java
interface Hooved extends Attacking {
  void trample(Animal target);
  default void attack(Animal target) {
    trample(target);
  }
}
interface Horned extends Attacking {
  void sting(Animal target);
  default void attack(Animal target) {
    sting(target);
  }
}
class Unicorn extends Animal implements Ridable, Horned, Hooved {
  public void attack(Animal target) {
    //Code
  }
}
```

Note that we can remove the implementation of `attack()` from the `Horse` and `Goat` class, leading to the default implementation being used. For the `Unicorn` class we must implement the `attack()` method, since it has default implementations in both interfaces and it would not be clear which one should be used.

Exercise 9.8:

We are a huge car company and want to structure our manufacturing process a bit better to increase the throughput. Therefore, we will use the factory pattern.

1. Create interfaces `Tire`, `Chassis`, `Engine` and a class `Car`. Your `Tire` should have a method `getPressure`, your `Chassis` should have a method `getType` and your `Engine` should have methods `getPower` and `getConsumption`.

2. Now create classes implementing the respective interfaces: `CheapTire` should have a pressure of 2 bar. A `PremiumTire` should have a pressure of 3 bar. For chassis, there is the option Sedane and SUV. They should return their type as string. For the engine, there are `SportEngine` and `CheapEngine`. Sport engines have 300PS and consume 9l/100km, cheap engines have 100PS and consume 5l/100km.

3. Next, create a `Car` class that contains tires, chassis, engine and a color. You should have methods that provide access to the methods of the different parts of the car.

4. Now, we can implement the actual factory class. Create a new class `StandardFactory`. Your factory should be able to produce a Sedane (using only the cheap parts). This should be possible by calling the method `createSedane`. Furthermore, implement `create...` methods for the individual components (Tire, Engine, ...). The class should also have a method `createSUV`. Unfortunately, our car company has no cheap SUVs, so it should always return `null`. The methods for creating cars take the color as an argument.

5. Finally we want to build a new factory. It should now be able to produce premium cars. To keep a nice structure, first create an abstract class `CarFactory` containing the before mentioned methods. It might be possible to already implement some methods in this abstract class, because they are the same for all subclasses. Then, make `StandardFactory` extend your new `CarFactory` and create a new class `PremiumFactory` also extending CarFactory. This `PremiumFactory` should be able to create both car types, but uses the premium component version.

Solution

```java
package com.example.carproject;

public interface Tire {
    public int getPressure();
}
```

```java
package com.example.carproject;

public interface Chassis {
    public String getType();
}
```

```java
package com.example.carproject;

public interface Engine {
    public int getPower();
    public int getConsumption();
}
```

```java
package com.example.carproject;

public class CheapTire implements Tire {
    @Override
    public int getPressure() {
        return 2;
    }
}
```

```java
package com.example.carproject;

public class ExpensiveTire implements Tire {
    @Override
    public int getPressure() {
        return 3;
    }
}
```

```java
package com.example.carproject;

public class SedaneChassis implements Chassis{
    @Override
    public String getType() {
        return "Sedane";
    }
}
```

```java
package com.example.carproject;

public class SUVChassis implements Chassis{
    @Override
    public String getType() {
        return "SUV";
    }
}
```

```java
package com.example.carproject;

public class CheapEngine implements Engine {
    @Override
    public int getPower() {
        return 100;
    }

    @Override
    public int getConsumption() {
        return 5;
    }
}
```

```java
package com.example.carproject;

public class SportsEngine implements Engine {
    @Override
    public int getPower() {
        return 300;
    }

    @Override
    public int getConsumption() {
        return 8;
    }
}
```

```java
package com.example.carproject;

public class Car {
    Chassis chassis;
    Tire tires;
    Engine engine;
    String color;

    public Car(Chassis chassis, Engine engine, Tire tires, String color) {
        this.engine = engine;
        this.tires = tires;
        this.color = color;
        this.chassis = chassis;
    }

    public String getChassisType() {
        return chassis.getType();
    }
    public int getPressure() {
        return tires.getPressure();
    }
    public String getColor() {
        return color;
    }
    public int getPower() {
        return engine.getPower();
    }
    public int getConsumption() {
        return engine.getConsumption();
    }
}
```

```java
package com.example.carproject;

public abstract class CarFactory {
    public abstract Tire createTire();
    public abstract Engine createEngine();
    public abstract Car createSedane(String color);
    public abstract Car createSUV(String color);

    public Chassis createChassis(String type) {
            if (type.equalsIgnoreCase("SEDANE")) {
                return new SedaneChassis();
            }
            if (type.equalsIgnoreCase("SUV")) {
                return new SUVChassis();
            }
            return null;
    }
}
```

```java
package com.example.carproject;

public class StandardFactory extends CarFactory {
    public Engine createEngine() {
        return new CheapEngine();
    }

    @Override
    public Car createSUV(String color) {
        return null;
    }

    @Override
    public Tire createTire() {
        return new CheapTire();
    }

    @Override
    public Car createSedane(String color) {
        return new Car(createChassis("SEDANE"), createEngine(),
            createTire(), color);
    }
}
```

```java
package com.example.carproject;

public class PremiumFactory extends CarFactory{
    @Override
    public Tire createTire() {
        return new ExpensiveTire();
    }
    @Override
    public Engine createEngine() {
        return new SportsEngine();
    }

    @Override
    public Car createSedane(String color) {
        return new Car(createChassis("SEDANE"), createEngine(),
            createTire(), color);
    }

    @Override
    public Car createSUV(String color) {
        return new Car(createChassis("SUV"), createEngine(),
            createTire(), color);
    }
}
```

Exercise 9.9:

- Adapt the implementation of Foo such that it behaves as desired.
- Look up what the annotation @Override does and shortly explain how it could have prevented the mistake.

```java
class Foo {
    private int a;
    public Foo(int a) { this.a = a; }
    public boolean equals(Foo b) { return a == b.a; }
    public static void main(String[] args) {
        Object a = new Foo(1);
        Object b = new Foo(1);
        System.out.println(a.equals(b));
        System.out.println(a.equals(a));
    }
}
```

Solution

- The equals method of Foo is overloaded instead of overwritten, since the equals method expects an object of type Object.

  ```java
  ...
  @Override
  public boolean equals(Object o){
      if(o == null || !(o instanceof Foo)) {
          return false;
      }
      return this.a == ((Foo) o).a;
  }
  ...
  ```

- The annotation @Override indicates to the compiler that you intend to override a method from the superclass (Object in this case). If there is a typo or mistake in the method signature, the compiler will generate an error, preventing accidental method overloading instead of overriding. In this case, using @Override would have alerted you to the fact that you are not correctly overriding the equals method.

Exercise 9.10:

In this exercise, you are required to implement a class on your own, including methods for the class. Your class should represent a matrix, instantiated in the constructor and mutable via the public methods (you can assume that all arguments are valid). Follow the steps below:

1. Create a class Matrix with a constructor that receives a width and height, and allocates a corresponding new integer array.
2. Write a getter and setter with which you can modify and access the matrix values arbitrarily.
3. Write a method which returns the minimum of all entries of the matrix.
4. Write a method which returns the average of all entries of the matrix.

5. Write a method that overrides the `equals` method to compare two matrices. It should return `true` if they have the same elements in each position, otherwise `false`.

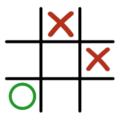
6. Write a method that overrides the `toString` method and outputs the matrix row by row, with each element separated by a tab (`'\t'`) and a new line (`'\n'`) for each row. This is shown below:

```
A 2x2 matrix:

1       4
2       3

A 4X4 matrix:

42      42      42      42
42      42      3       42
42      42      42      42
42      42      42      42
```

Solution

```java
public class Matrix {
  int height;
  int width;
  int[][] matrix;

  public Matrix(int height, int width){
    this.height = height;
    this.width = width;
    this.matrix = new int[height][width];
  }

  public void setValue(int h, int w, int value){
    matrix[h][w] = value;
  }

  public int getValue(int h, int w){
    return matrix[h][w];
  }

  public int getMinimum(){
    int value = matrix[0][0];
    for (int i = 0; i < height; i++){
      for (int j = 0; j < width; j++){
        if(matrix[i][j] < value)
          value = matrix[i][j];
      }
    }
    return value;
  }

  public double average(){
    int value = 0;
    int counter = 0;
    for (int i = 0; i < height; i++){
      for (int j = 0; j < width; j++){
        counter++;
        value += matrix[i][j];
      }
    }
    double res = ((double) value) / counter;
    return res;
  }

  @Override
  public boolean equals (Object obj) {
      if (!(obj instanceof Matrix)) {
          return false;
      }
      Matrix otherMatrix = (Matrix) obj;
      if (width != otherMatrix.width || height != otherMatrix.height) {
          return false;
      }
      for (int i = 0; i < height; i++) {
          for (int j = 0; j < width; j++) {
              if (matrix[i][j] != otherMatrix.matrix[i][j]) {
                  return false;
              }
          }
      }
      return true;
  }

  public String toString() {
    String res = "";
    for (int i=0; i < this.height; i ++) {
        String s = "";
        for (int j=0; j < this.height; j++)
                s += (this.getValue(i, j) + "\t");
        res += (s + "\n");
    }
    return res;
  }
}
```

Exercise 9.11: Tic Tac Toe



In this exercise, your task is to develop and implement different autoplayers for the game Tic-Tac-Toe. To start, use the provided program skeleton which can be found at `https://git.prog2.de/shared/exercise-repos/tictactoe`.

1. Familiarize yourself with the structure of the program. Look at the documentation and find out in particular, how new autoplayers are created and added to the game.
2. First, implement an autoplayer based on randomness, that chooses random (but valid) spaces in its moves.
3. Develop more sensible strategies for the autoplayer. Implement and test these among each other.

Solution

In the `TicTacToe` repository, checkout the branch 'solution'.