Languages of AI

1.1) Object-Oriented Programming, Java, and Evolutionary Algorithms

Prof. Dr. Petra Hofstedt,
Dr. Sven Löffler

Brandenburg University of Technology Cottbus-Senftenberg

Chair Programming Languages and Compiler Construction

Brandenburg
University of Technology
Cottbus - Senftenberg

Overview

Object-Oriented Programming (OOP)

1971, Alan Kay: inventor of the programming language Smalltalk and the term “object oriented“, defined it in the context of Smalltalk as follows:

1. Everything is an object
2. Objects communicate by sending and receiving messages (in terms of objects)
3. Objects have their own memory (in terms of objects)
4. Every object is an instance of a class (which must be an object)
5. The class holds the shared behavior for its instances (in the form of objects in a program list)
6. To evaluate a program list, control is passed to the first object and the remainder is treated as its message

Object-Oriented Programming (OOP)

1999, The ISO/IEC 2382-15 standard defines the term object-oriented as follows:

- Pertaining to a technique or a programming language that supports objects, classes, and inheritance.

History

- 1967: Simula 67, O.-J. Dahl (University of Oslo)
  - Language for implementing event-driven simulations
  - The terms “object“ and “class“ first appeared
- 1971, Smalltalk, Alan Kay, Adele Goldberg
  - First purely object-oriented programming language
  - Aim: to anticipate human-computer communication for the year 2000 (concepts for mouse and window interfaces considered)
  - In practice, Smalltalk could not prevail
  - The idea was too revolutionary
  - Performance issues, significant memory requirements, and lacked static typing
- Since around 1985, Object Pascal
  - Various approaches to extend the Pascal language into an object-oriented language

Object-Oriented Programming (OOP)

History

- 1985, Eiffel, Bertrand Meyer (French)
  - A purely object-oriented language
  - Statically typed, allows multiple inheritance
  - An excellent OOP language, but not gained adoption
- Other:
  - Flavors, Loops, XLisp, CLOS are object-oriented extensions of functional languages (e.g., Lisp)
  - Application: primarily in specialized areas of artificial intelligence
- 1983, C++, Bjarne Stroustrup
  - Breakthrough of object-oriented programming
  - Found first applications outside of research labs
- 1991, Python, Guido van Rossum
  - A purely object-oriented language
  - Dynamic typing
- 1995 Java, James Gosling, Sun Microsystems
  - No multiple inheritance
  - No explicit pointer arithmetic

Overview

Abstraction

Separation between concept and implementation

Examples:

- Recipe vs. food
- Technical handbook vs. machine
- Table of contents vs. book

In OOP: classes vs. objects

Object (implementation)

- Is a real existing thing from the application world of a program

Class (concept)

- Is a description of similar things

Abstraction

In OOP: classes vs. objects

Object

- Is a real existing thing from the application world of a program

Class

- Is a description of similar things
- Describes at least 3 aspects
  - How to create an object?
  - What properties does the object have?
  - How to use the object?

Abstraction

Class Definition

```
<Visibility> class <Class name> {
    <Attribute declarations>
    <Method declarations>
}
```

Example Class

```
class Point {
    double x, y;
    public Point (double x, double y) { // Constructor
        this.x = x;
        this.y = y;
    }
    double dist () { // distance from the origin
        return Math.sqrt(x*x+y*y);
    }
}
```

Example Use of a Class

```
public class AProgram {
  public static void main (String[] args) {
     Point p1 = new Point (3,4);
     Point p2 = new Point (5,9);

     int dist = p1.dist();
     System.out.println(dist);

     ...
```

Encapsulation

Each class is defined by a set of data (attributes or member variables) and methods operating on it

- Member variables represent the state of an object
- Methods represent the behaivor of objects
- Apart from intended exceptions methods are the only way ...
  - ... to communicate with an object
  - ... to earn informations about an object
  - ... to manipulate the object

The combination of methods and variables into classes is called encapsulation

- Reduces the complexity of operating and implementing an object

Encapsulation

Class Declaration

```
class Point {
  double x; // Member variables
  double y;
  public Point (double x, double y) {
    // Constructor
    this.x = x;
    this.y = y;
  }
  double dist () { // distance from the origin
    double d = Math.sqrt(x*x+y*y);
    return d;
  }
  void shift (double dx, double dy) { // manipulate
    // manipulate the object
    this.x = this.x + dx;
    this.y = this.y + dy;
  }
  public double getX() { // earn information
    return x;
  }
  public void setX(double x) { // communicate
    this.x = x;
  }
  ...
```

Reusability

Abstraction and encapsulation support another property of OOP: the reusability of components

Example: Collections (Objects which contain other objects and manage them)

- Are often implemented in a complicated way (to increase speed and reduce storage space) ...
  - Sets (tree sets, hash sets)
  - Lists (array lists, linked lists)
- ... but have a simple interface
- Complex details can be abstracted away and allow an easy reuse
  - e.g. implementing a queue by the use of a list

Reusability

Queue Class I

```
public class Queue<T> {
   List<T> queue;

   public Queue() {
      this (new LinkedList<T>());
   }
   public Queue(List<T> list) {
      this.queue = list;
   }
   public void add(T element) {
      this.queue.add(element);
   }
   public T peek() {
      if(isNullOrEmpty()) {
         return null;
      }
      return queue.get(0);
   }
```

Queue Class II

```
   public T poll() {
      if(isNullOrEmpty()) {
         return null;
      }
      return this.queue.remove(0);
   }
   public boolean isNullOrEmpty() {
      return this.queue == null
            || this.queue.size() == 0;
   }
}
```

Relations

There are at least 3 basic relations between classes and objects

- “is-a“ relation
- “part-of“ relation
- “Use“ or “call“ relation

Relations

there are at least 3 basic relations between classes and objects

- “is-a“ relation generalisation, specialisation

  the relation: class B “is-a“ class A means:

  - B contains all characteristics of A (all not private variables and methods) of A
  - B can have additional characteristics
  - B is a specialisation of A
  - A is a generalisation of B

  Realized in Java with inheritance

  Other languagues allow multiple inheritance

- “part-of“ relation
- Use or call relation

“is-a“ relation

Class A

```
public class Human {
   public String name;
   public int age;
   public Human(String name) {
      this(name, 0);
   }
   public Human(String name, int age) {
      this.name = name;
      this.age = age;
   }
   public void grow() {
      this.age++;
   }
}
```

Class B

```
public class Student extends Human {
   public int studentIdNumber;
   public static int numberOfStudens;
   public static int semester;
   public Student(String name) {
      super(name);
      numberOfStudens++;
      this.studentIdNumber = numberOfStudens;
   }
   public Student(String name, int age) {
      super(name, age);
      numberOfStudens++;
      this.studentIdNumber = numberOfStudens;
   }
   @Override
   public void grow() {
      super.grow();
      semester = semester + 2;
   }
}
```

Student “is-a“ Human

Relations

there are at least 3 basic relations between classes and objects

- “is-a“ relation — generalisation, specialisation
- “part-of“ relation — aggregation and composition
  - An object is a composition of other objects (composition)
  - An object contains other objects (aggregation)

  While object-oriented modeling distinguishes between the two cases, object-oriented programming languages implement them in the same way (member variables)
- Use or call relation

“part-of“ relation

Point is Part of Quadrilateral

```
public class Quadrilateral {
   private Point a;
   private Point b;
   private Point c;
   private Point d;

   public Quadrilateral(Point a, Point b, Point c, Point d) {
      this.a = a;
      this.b = b;
      this.c = c;
      this.d = d;
   }
}
```

Point is “part-of“ Quadrilateral

Relations

there are at least 3 basic relations between classes and objects

- “is-a“ relation — generalisation, specialisation
- “part-of“ relation — aggregation and composition
- “Use“ or “call“ relation
  - Is the most general relationship between objects
  - An object A uses an object B to do special operations

    In OOP realiced with local variables

“Use“ or “call“ relation

Shifter Use Quadrilateraly

```
public class Shifter {
   private double shiftX;
   private double shiftY;
   public Shifter(double shiftX, double shiftY) {
      this.shiftX = shiftX;
      this.shiftY = shiftY;
   }
   public void shiftQuadliteral(Quadrilateral quadrilateral) {
      for (Point p: quadrilateral.getPoints()) {
         p.setX(p.getX() + shiftX);
         p.setY(p.getY() + shiftY);
      }
   }
}
```

Polymorphism

Is the possibility that object variables can get objects of different classes

- This happens not arbitrarily
- An object variable of type A can only get objects of type A or objects inheriting from A
- Distinction between early binding and late binding
  - Early binding: at compile time it is determined which version of a method is called
  - Late binding: overloading of methods, an object variable A can contain objects which inherit from A. At compile time it is not clear which version of a function must be used.

Polymorphism: object of another class and late binding

Class A

```
public class Human {
   public String name;
   public int age;
   public Human(String name) {
      this(name, 0);
   }
   public Human(String name, int age) {
      this.name = name;
      this.age = age;
   }
   public void grow() {
      this.age++;
   }
}
```

Object of Another Class & Late Binding

```
public class Main {
   public static void main(String[] args) {
      Human human = new Student("Sven");
      human.grow();
   }
}
```

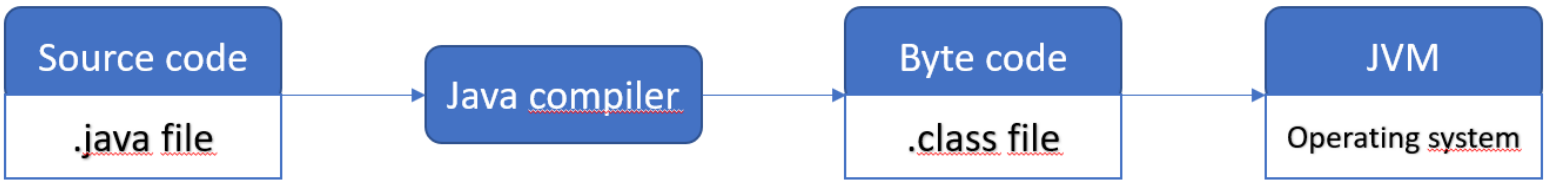
Class B

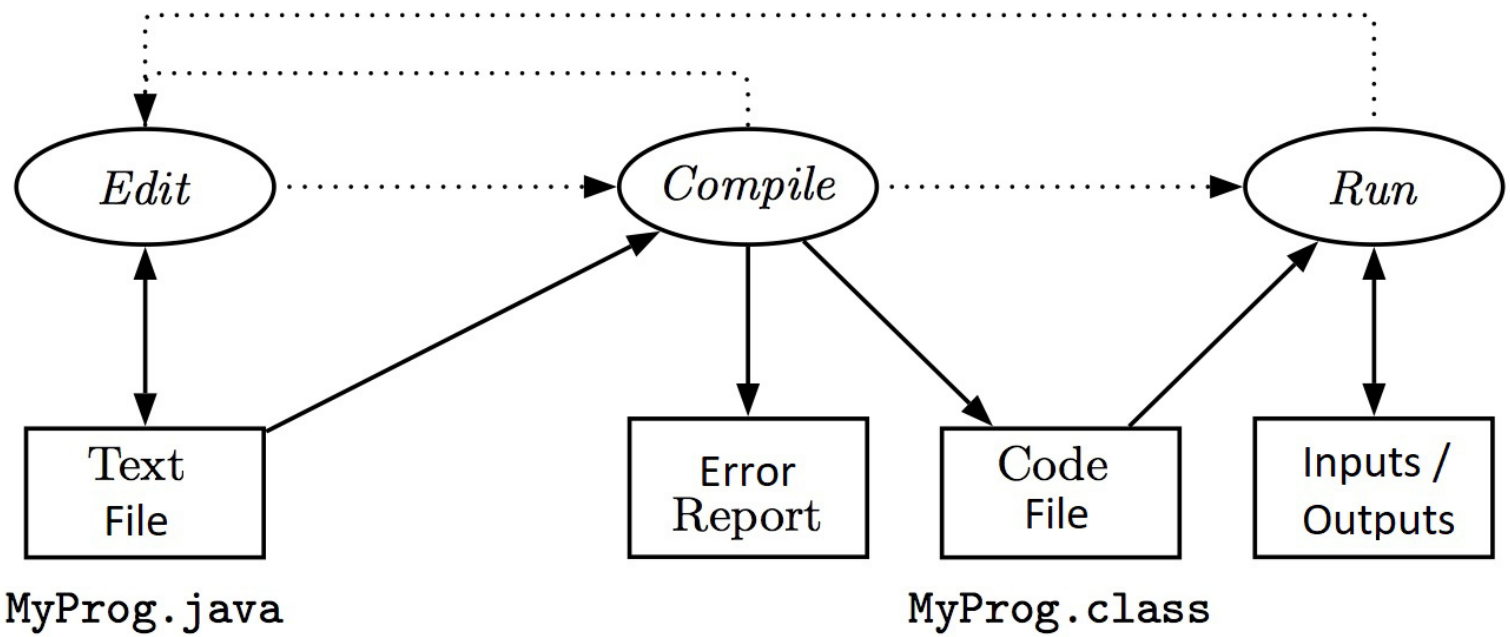
```
public class Student extends Human {
   public int studentIdNumber;
   public static int numberOfStudens;
   public static int semester;
   public Student(String name) {
      super(name);
      numberOfStudens++;
      this.studentIdNumber = numberOfStudens;
   }
   public Student(String name, int age) {
      super(name, age);
      numberOfStudens++;
      this.studentIdNumber = numberOfStudens;
   }
   @Override
   public void grow() {
      super.grow();
      semester = semester + 2;
   }
}
```

Java in a Nutshell

- Development of Java in 1991/92 by *James Gosling* and other developers on behalf of Sun Microsystems (originally for embedded systems)
- First public demonstration and release in 1995
- Acquisition by Oracle in 2010
- Imperative and object-oriented language

Program Creation & Compilation





(from [Pepper2007], Chap. 4)

The *entry point* of every Java application is the **main** method.

The **main** Method

```
public class MyProgram {
    public static void main(String[] args) {
        System.out.println("Today we start with Java.");
    }
}
```

Execution of a program:

- Starting the JVM
- Loading the class, Linking, Initializing, ...
- Executing the actions/commands in the body of **main**
- Termination of the program by the Java system

Introduction of variables through **declarations**
Changing variable values by **assignments**

Declaration of and Assignments to Variables

```
int counter;
counter = 0;
double radius = 5.5;
Point p = new Point(2.2f, 1.7f);
Circle c = new Circle(p1, radius);
int x = 5;
int y = 6;
x = y + 1;
int z = y % 3;
```

Definition (Casting)

The **conversion** of a type t_1 to another type t_2 – also known as **casting** – is written by enclosing the new type t_2 in parentheses before the value or variable of type t_1, for example, `(float)1.7` or `(float)x`.

Implicit type conversion from a subtype to a supertype is performed automatically by the compiler.

From type ...		... to type
`byte`	$\rightarrowtail$	`short, char, int, long, float, double`
`char, short`	$\rightarrowtail$	`int, long, float, double`
`int`	$\rightarrowtail$	`long, float, double`
`long`	$\rightarrowtail$	`float, double`
`float`	$\rightarrowtail$	`double`
<all>	$\rightarrowtail$	`String`

automatic, harmless casting by the compiler (from [Pepper2007], Chap. 2)

Explicit type conversion can be enforced.

Arithmetic and Casting I

```
1 + 2
1.0 + 2
double d = 4;
int i = 4.0;
```

Arithmetic and Casting II

```
2.5 * 3
(int) 2.5*3
(int)(2.5*3)
```

Arithmetic and Casting III

```
float tax = 0.19;
double d = 3.14;
float f = d + 1;
```

Arithmetic and Casting I

```
1 + 2
1.0 + 2
double d = 4;
int i = 4.0;
```

Arithmetic and Casting II

```
2.5 * 3
(int) 2.5*3
(int)(2.5*3)
```

Arithmetic and Casting III

```
float tax = 0.19F;
double d = 3.14;
float f = (float)d + 1;
```

Arithmetic and Casting IV

```
float pi = 3.14159f;
String text = "Pi is " + pi + ".";
```

Libraries/Packages: Collection of algorithms and data structures, such as mathematics, input and output, useful data structures, ...

We consider the package `java.lang` with the class `Math`, which contains mathematical functions.

static double	**PI** The double value that is closer than any other to *pi*, the ratio of the circumference of a circle to its diameter.
static double	**ceil**(double a) Returns the smallest (closest to negative infinity) double value that is greater than or equal to the argument and is equal to a mathematical integer.
static double	**floor**(double a) Returns the largest (closest to positive infinity) double value that is less than or equal to the argument and is equal to a mathematical integer.
static double	**cos**(double a) Returns the trigonometric cosine of an angle.
static double	**sin**(double a) Returns the trigonometric sine of an angle.
static double	**pow**(double a, double b) Returns the value of the first argument raised to the power of the second argument.

```
Math.sin(1.7)
Math.cos(2*Math.PI)
Math.pow(2,3)
Math.pow(3,Math.pow(3,3))
Math.ceil(3.4)
Math.floor(3.4)
```

The **if** statement for conditional branching. Dynamic dependency of the program flow on conditions.

If Statements

```
if (Condition) { statements }
if (Condition) { statements_1 } else { statements_2 }
```

If Statements: Example I

```
int maximum (int x, int y) {
  if (x > y) { return x; }
  else       { return y; }
}
```

If Statements: Example II

```
int sign = 0;
if (a > 0) { sign = 1; }
else if (a == 0) { sign = 0; }
     else        { sign = -1; }
```

If Statements: Example III

```
if (temp >= 100) { System.out.println("Alarm"); }
```

The **switch** statement for selection based on simple values.
The expression and values must be of one of the types **byte**, **char**, **short**, **int**, or **long**. The default case may be missing.
Note: The statements in a case will be executed until a break or the end of the switch statement is reached.

Switch Statements

```
switch (expresion) {
  case Value_1: statements_1; break;
  case Value_2 : statements_2; break;
  ...
  case Value_n : statements_n; break;
  default : statements_n+1; break;
}
```

Switch Statements: Example I

```
switch (month) {
  case  1: days = 31; break;
  case  2: days = 28; break;
  case  3: days = 31; break;
  case  4: days = 30; break;
  ...
  case  9: days = 30; break;
  case 10: days = 31; break;
  case 11: days = 30; break;
  case 12: days = 31; break;
}
```

The **switch** statement for selection based on simple values.
The expression and values must be of one of the types **byte**, **char**, **short**, **int**, or **long**. The default case may be missing.
Note: The statements in a case will be executed until a break or the end of the switch statement is reached.

Switch Statements

```
switch (expresion) {
  case Value_1: statements_1; break;
  case Value_2 : statements_2; break;
  ...
  case Value_n : statements_n; break;
  default : statements_n+1; break;
}
```

Switch Statements: Example II

```
switch (month) {
  case 2: days = 28; break;
  case 4:
  case 6:
  case 9:
  case 11: days = 30; break;
  default: days = 31; break;
}
```

Loops for repeating statements.

While Loops

```
while (Condition) {Statements} // while-do-loop

do {Statements} while (Condition); // do-while-loop
```

Calculation of the Greatest Common Divisor

```
int gcd (int a, int b) {
  while (a != b) {
     if (a > b) a = a - b;
     else b = b - a;
  }
  return a;
}
```

Loops for repeating statements.

While Loops

```
while (Condition) {Statements} // while-do-loop

do {Statements} while (Condition); // do-while-loop
```

Calculation of the Product

```
int prod (int a, int b) {
  int s = 1;
  do {
    s = s * a;
    a = a + 1;
  } while (a <= b);
  return s;
}
```

The for Loop (Counting Loop): Perform a Specific Number of Repetitions

```
for (Initialization; Termination Test; Step) {
    Instructions
}
```

Calculation of the Multiplication Table

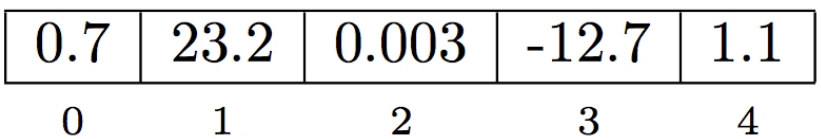
```
for (a=1; a<11; a++) {
  for (b=1; b<11; b++) {
    System.out.print(a*b + " ");
  }
  System.out.println();
}
```

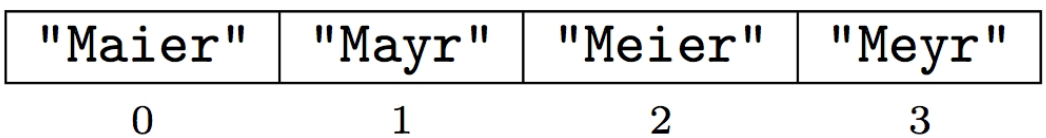
```
1 2 3 4 5 6 7 8 9 10
2 4 6 8 10 12 14 16 18 20
3 6 9 12 15 18 21 24 27 30
4 8 12 16 20 24 28 32 36 40
5 10 15 20 25 30 35 40 45 50
6 12 18 24 30 36 42 48 54 60
7 14 21 28 35 42 49 56 63 70
8 16 24 32 40 48 56 64 72 80
9 18 27 36 45 54 63 72 81 90
10 20 30 40 50 60 70 80 90 100
```

Array ([Pepper2007], Section 1.5.2)

Arrays (in Java) are characterized by the following properties:

- An array is an ordered collection of elements.
- All elements in the array must have the same type, referred to as the "base type" of the array.
- The number of elements in the array, denoted by n, is referred to as its "length".
- The elements in the array are indexed from 0 to $n - 1$.





Declaration of Arrays:

Declaration of Arrays

```
int[] myIntArray = new int[6]; // an empty Array with size 6 for ints
String[] b = new String[50]; // an empty Array with size 50 for Strings
```

Declaration and Assignment:

Declaration and Assignment of Arrays

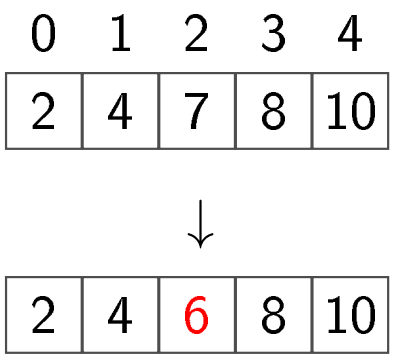
```
double[] nums = {0.7, 23.2, 0.003, -12.7, 1.1};
String[] names = {"Maier", "Mayr", "Meier", "Meyr"};
int[] primes = {2,3,5,7,11};
```

Selection and definition of elements:

Selection and Definition of Elements

```
String[] names = {"Maier", "Mayr", "Meier", "Meyr"};
...
String a = names[3]; // set a to "Meyr"
String b = names[4]; // Error!

int[] inums = {2,4,7,8,10};
...
int c = inums[0]; // set c to 2
...
inums[2] = 6; // change inums
```



Selection and definition of elements:

Selection and Definition of Elements

```
String[] names = {"Maier", "Mayr", "Meier", "Meyr"};
...
String a = names[3]; // set a to "Meyr"
String b = names[4]; // Error!

int[] inums = {2,4,7,8,10};
...
int c = inums[0]; // set c to 2
...
inums[2] = 6; // change inums
```

Length of an Array:

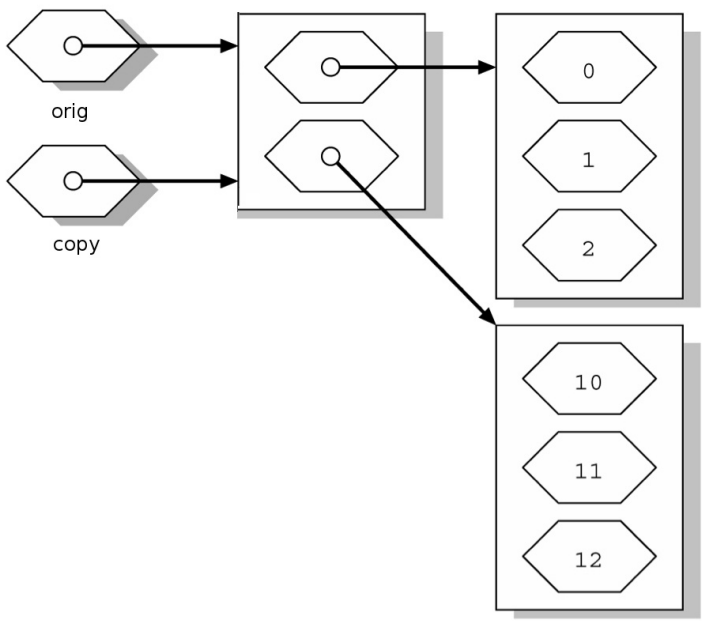
Length of an Array

```
int inumsLen = inums.length;
int n = names.length;
```

An assignment copies only the reference to an array.

References and Arrays

```
int[][] orig = new int[][] {{0,1,2},{10,11,12}};
int[][] copy = orig;
```



Consequence: Local changes are visible in both arrays.

References and Arrays

```
copy[0][2] = 15;
System.out.println(orig[0][2]); // has value 15
```

Programming principle:

A program part that solves a self-contained task can be encapsulated as a subroutine/function/procedure/method.

The subroutine can be called from different parts of the program.

$\rightsquigarrow$ Improved structuring, readability, reusability

Java: A *method* provides a named, parameterized sequence of instructions that is executed through a *method call*.

Method Declaration

```
<Visability> <Type|void> <MName> (<Type> <ParamName>, ...) {
   <Instructions>
}
```

Example

```
public static void main(String[] args) {
    int m = 7;
    int f = foo(m);
    System.out.println("foo(" + m + ")" + " = " + f);
}
```

The *method call* is executed in the following steps (*call sequence*) (see [Schiedemeier2010], p. 121):

1. The calling program (the caller) is interrupted.
2. The method body is executed.
3. The caller is resumed.

The Caller

```
public class MyProg {
  public static void main(String[] args) {
    int m = 7;
    int f = foo(m);
    System.out.println("foo(" + m + ")" + " = " + f);
  }
...
```

The Caller and the Called Method

```
public static int foo (int n){
  int m = max (n,2*n);
  return m;
}
```

The Called Method

```
public static int max (int a, int b) {
  if (a>b) { return a; }
  else { return b; }
}
```

The Method Call

The *method call* is executed in the following steps (*call sequence*) (see [Schiedemeier2010], p. 121):

1. The calling program (the caller) is interrupted.
2. The method body is executed.
3. The caller is resumed.

Methods with result type `void`:

$\rightsquigarrow$ Method calls (resp. *prozedure* calls) are instructions.

Methods with other result types:

$\rightsquigarrow$ Method calls (resp. *function* calls) return a value, they are expressions.

Recursion ([Pepper2007], Section 6.1)

A method `f` is called (directly) **recursive** if it contains calls to itself within its body.
The method `f` is called *indirectly recursive* if it calls another method `g` within its body, and that method `g` directly or indirectly leads to calls to `f`.

Example. The factorial function:

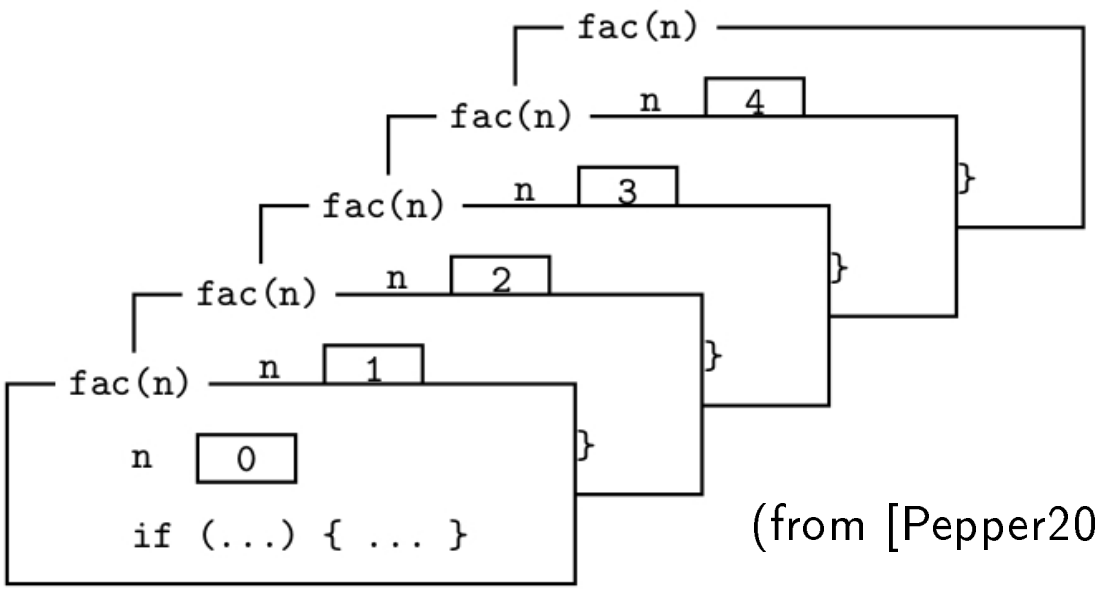
$$\begin{aligned} 0! &= 1 \\ (n+1)! &= (n+1) * n! \end{aligned}$$

Recursion

```
int fac(int n){
  if (n>0) {
    return n * fac(n-1); // recursive call
  }
  else {
    return 1;
  }
}
```

The Factorial Function

```
int fac(int n){
  if (n>0) {
    return n * fac(n-1); // recursive call
  }
  else {
    return 1;
  }
}
```



(from [Pepper2007], S.106)

Object-oriented programming:
Summary: Object-oriented programming involves the encapsulation of data and functions into objects, enabling the description of complex systems through the interaction of cooperating objects.

Object

An **object** is a unit/entity characterized by three aspects: independent identity, state, and behavior.

Object-oriented programming (OOP) supports:

- Abstraction
- Encapsulation
- Reusability
- Relations
- Polymorphism

Classes and Objects

A Java program consists of a collection of classes. The "main class" contains a start method called `main`, which serves as the entry point of the Java application.

- A *class* is a blueprint for creating similar objects. It describes the attributes and methods that the objects of that class will have.
- An *object* is an instance or manifestation of a class. It represents a specific occurrence or realization of the attributes and behaviors defined by the class.

- A *class* is a blueprint for creating similar objects. It describes the attributes and methods that the objects of that class will have.
- An *object* is an instance or manifestation of a class. It represents a specific occurrence or realization of the attributes and behaviors defined by the class.

Class Declaration

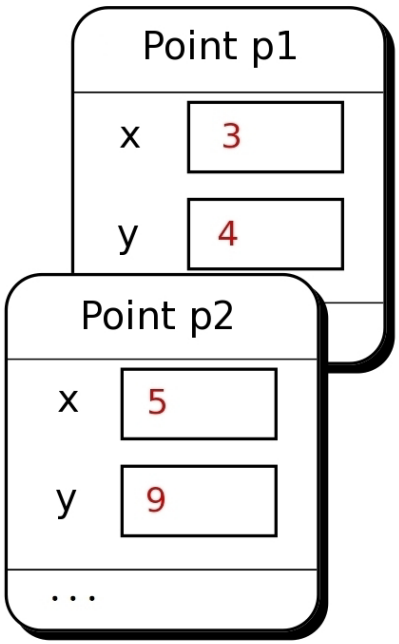
```
<Visibility> class <Class name> {
    <Attribute declarations>
    <Method declarations>
}
```

Example (see slide 9)

```
class Point {
    double x, y;
    public Point (double x, double y) { // Constructor
        this.x = x;
        this.y = y;
    }
    double dist () { // distance from the origin
        return Math.sqrt(x*x+y*y);
    }
}
```

Creation of an Object: Constructors

An *object* is an instance or occurrence of a class. It is a concrete representation or manifestation of the attributes and behaviors defined by the class.



Creation of an Object

```
public class AProgram {
  public static void main (String[] args) {
     Point p1 = new Point (3,4);
     Point p2 = new Point (5,9);
     ...
```

Constructors are speciel methods for the initialisation of objects.

A constructor has the same name as the class and is used to create a new object of that class.

Creation of an Object: Constructors

Example

```
class Point {
  double x, y;              // attributes
  Point (double x, double y) { // constructor method
    this.x = x;
    this.y = y;
  }
  ...                       // method
}
```

Observations:

- A constructor method does *not* have a return type.
- When creating an object, the instructions within the constructor are executed.
- If a method refers to the object it belongs to, the keyword `this` can be used to refer to that object (self-reference).
- It is common practice to name the parameters of the constructor the same as the corresponding attributes, using self-references for distinction.

Creation of an Object: Constructors

If no constructor is defined for a class, the compiler automatically generates a default (parameterless) constructor.

Application of the Default constructor

```
...
Point p1 = new Point ();
p1.x = 1;
p1.y = 2;
...
```

A class can have multiple constructors. The condition is that all constructors must have different types or number of parameters (overloading).

Overloading of Methods [Pepper2007] S.13

```
class Point {
  double x, y;              // attributes
  Point () {};              // without coordinates
  Point (double x) {        // equal coordinates
    this.x = x;
    this.y = x;
  Point (double x, double y) { // different coordinates
    this.x = x;
    this.y = y;
  ...                       // other methods
}
```

Chained constructors ([Schiedemeier2010] S.136):

Constructors can have additional tasks besides assigning values to object variables, such as parameter testing.
When there are multiple overloaded constructors, they can be chained together to avoid code duplication.

Chained Constructors ([Schiedemeier2010] S.136)

```
class Rational {
  int n; // numerator
  int d; // denominator
  Rational (int n, int d) {
    ... // Check if d is not equal to 0
    ... // Reduce n and d
    ... // Make the sign of d positive
    ... // etc.
    this.n = n;
    this.d = d;
  }
  Rational () {      // nullary constructor
    this (0,1);
  }
  Rational (int n) { // unary constructor
    this (n,1);
  }
  ...
}
```

The attributes and methods of an object can be accessed using the dot operator ".".

Object Methods

```
class Point {
  double x, y; // attributes: coordinates
  ...
  Point (double x, double y) { // constructor
    this.x = x;
    this.y = y;
  }
  double dist () {          // distance from the origion
    double d = Math.sqrt(x*x+y*y);
    return d;
  }
  void shift (double dx, double dy) { // shifting
    x = x + dx;
    y = y + dy;
  }
}

public class AProgram {
  public static void main (String[] args) {
     Point p1 = new Point (3,4);
     Point p2 = new Point (5,9);
     p1.shift (3,6);
     p2.shift (p1.dist(),p1.y);
     ...
```

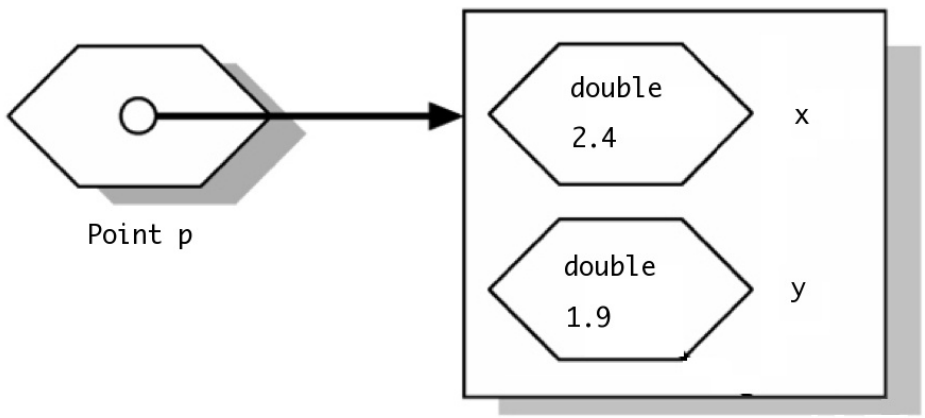
Class definition / *reference types*.
In the example, `p` is a *reference variable* of the reference type `Point`.

Example: Creation of Objects

```
Point p = new Point (2.4,1.9);
```

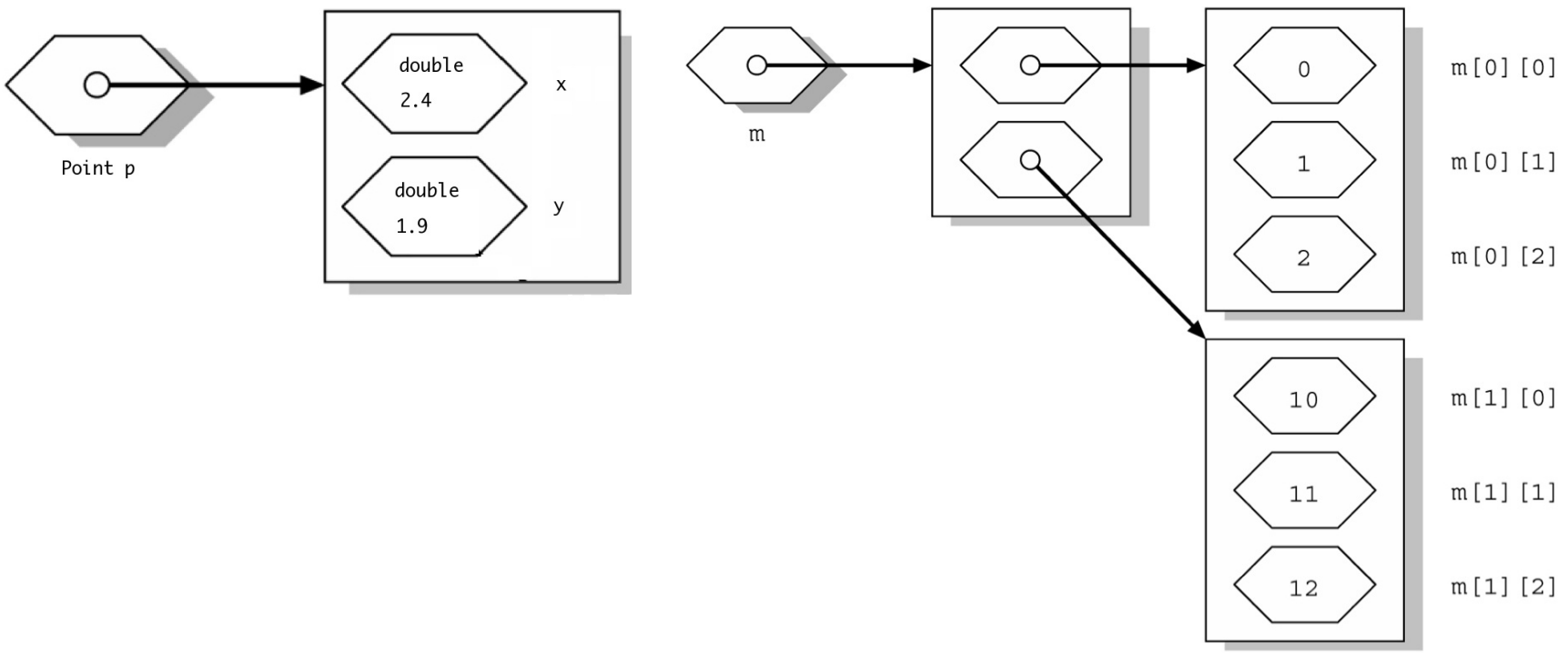
Definition: Reference ([Pepper2007] S.297)

A **reference** is a *pointer* to an object.



Primitive types	reference types
`boolean`, `char`, `byte`, `short`, `int`, `long`, `float`, `double`	all classes, especially `String` and arrays

Types in Java ([Pepper2007] S. 298)

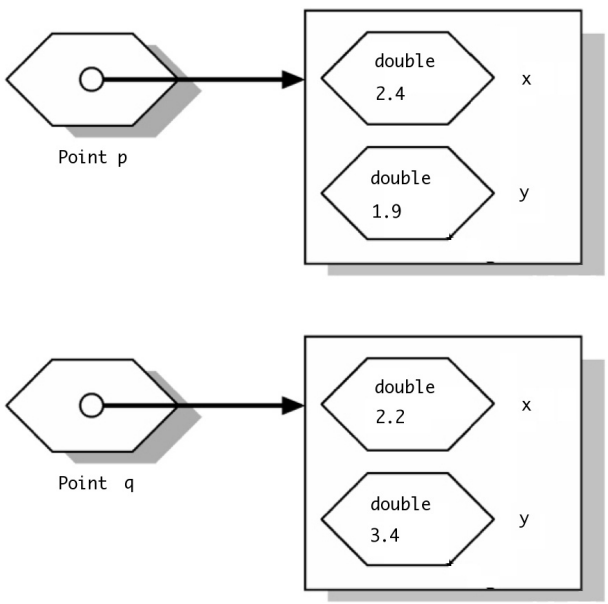


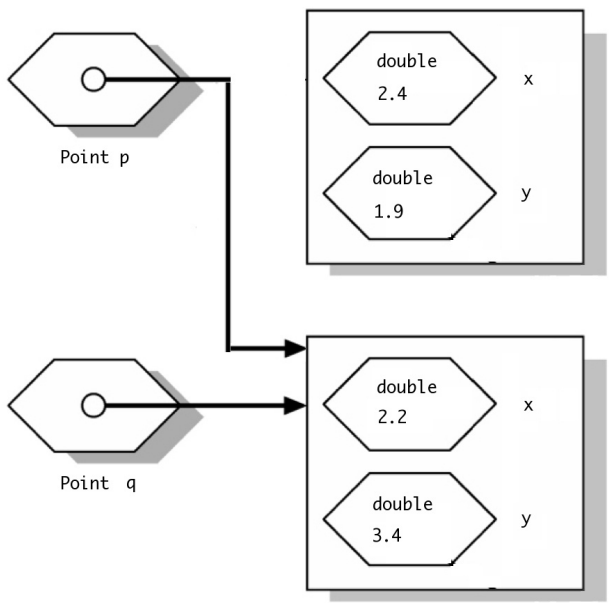
Working with references

We create two objects.

Creating Two Points

```
Point p = new Point (2.4,1.9);
Point q = new Point (2.2,3.4);
p = q;
```





We perform an assignment.
In this process, the object is not copied, but the reference is.

Visibility: Package

A *Package* is a collection of logically related classes and forms a namespace.

Default case (no access modifier):

- All classes in a package can freely use each other's methods and attributes without any restrictions.
- Access from external packages is *shielded* or *protected*.

Using Packages

```
package example;
class JosephusPermutation {
  public static void main(String[] args) {
    int n = StdIn.askInt("n = ");
    ...
```

```
package example;
import java.io.InputStreamReader;
class StdIn {
  static String askString(String prompt) {
    System.out.print(prompt);
    ...
  static int askInt(String prompt) {
    do {
      ...
```

- More fine-grained control of access rights is possible through the keywords: `public`, `private`, and `protected`.

Overview

Modifiers for *visibility* regulate access rights to attributes and methods.

Element is visible in...	Element modifier			
	public	protected	-	private
... the class itself	✓	✓	✓	✓
... classes in the same package	✓	✓	✓	
... subclass of other packages	✓	✓		
... all classes of all packages	✓			

The modifiers can also specify classes.

Class Variables and Static Methods

Class variables and static methods belong to the class (and are not specific to individual objects).

Class variable: There is only one instance per class that is shared by all objects. Object methods can access a class variable.

Static methods: A static method can only access static attributes and not object variables. The *main* method is static because there are no objects existing at the start of the program.

Static Methods

```
package example;
public class JosephusPermutation {
  public static void main(String[] args) {
    int n = StdIn.askInt("n = ");
    ...

package example;
public class StdIn {
  public static String askString(String prompt) {
    System.out.print(prompt);
    ...
  public static int askInt(String prompt) {
    do {
      ...
```

Overview

Brief History

The idea of applying Darwinian principles to automated problem solving dates back to the 1940s, long before the breakthrough of computers [EibenSmith2015]

1948 Turing proposed "genetical or evolutionary search"

1962 Bremermann executed computer experiments on optimization through evolution and recombination

- During 1960s, three different implementations:
  - 1962, USA: Holland called his method a genetic algorithm
  - 1965, Germany: Rechenberg and Schwefel invented evolution strategies
  - 1966, USA: Fogel, Owens, and Walsh introduced evolutionary programming

Now, we call them all **genetic algorithms** or **evolutionary algorithms**

The Inspiration from Biology: Darwinian Evolution [Darwin1859]

- The environment can host only a limited number of individuals
- Basic instinct of individuals to **reproduce**

$\Rightarrow$ **selection** is neccessary (survival of the fittest)

Each individual represents a unique combination of phenotypic traits that is evaluated by the environment (**fitness function**)

If this combination evaluates positive, then the individual has a higher chance for creating offspring

Darwin's insight was that small, random variations **mutations** in phenotypic traits occur during reproduction from generation to generation.

Evolutionary Computing: Why?

When looking for the most powerful natural problem solver, there are two rather obvious candidates:

- The human brain (that created the wheel, Cottbus, wars and so on)
- The evolutionary process (that created the human brain)

Procedure

There are two main forces that form the basis of evolutionary systems:

- **Variation** operators (**recombination** and **mutation**) create the necessary diversity within the population, and thereby facilitate novelty
  - **Recombination** is applied to two or more selected candidates (parents), producing one or more new candidates (children)
  - **Mutation** is applied to one candidate and results in one new candidate
- **Selection** acts as a force increasing the mean quality of solutions in the population
  - Individuals with a higher fitness value have a greater chance of being selected (**survival of the fittest**)

Procedure

Evolutionary Algorithm

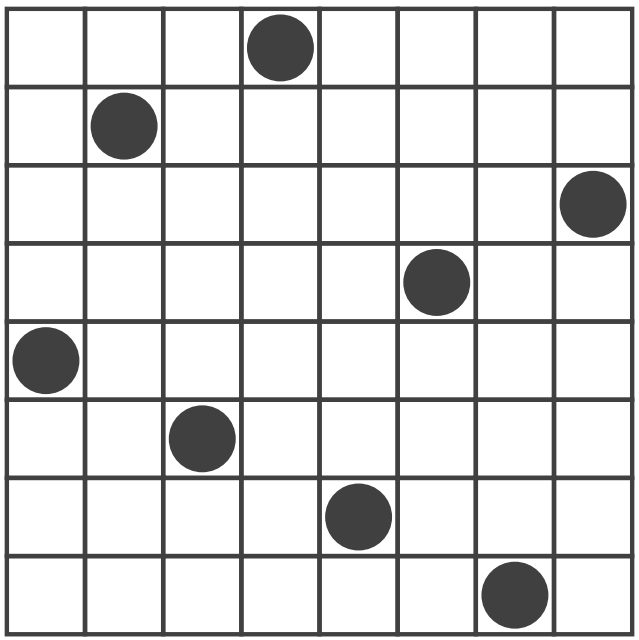
```
public class EvolutionaryAlgorithm {
   public static void main(String[] args) {
      int size = 100;
      Population population = Population.newPopulation(size);

      while (!population.goalFunction()) {
         Population newPopulation = population.copy();
         newPopulation.add(population.recombine());
         newPopulation.add(population.mutate());
         newPopulation.selection();
         population = newPopulation;
      }
   }
}
```

Example: n-Queens

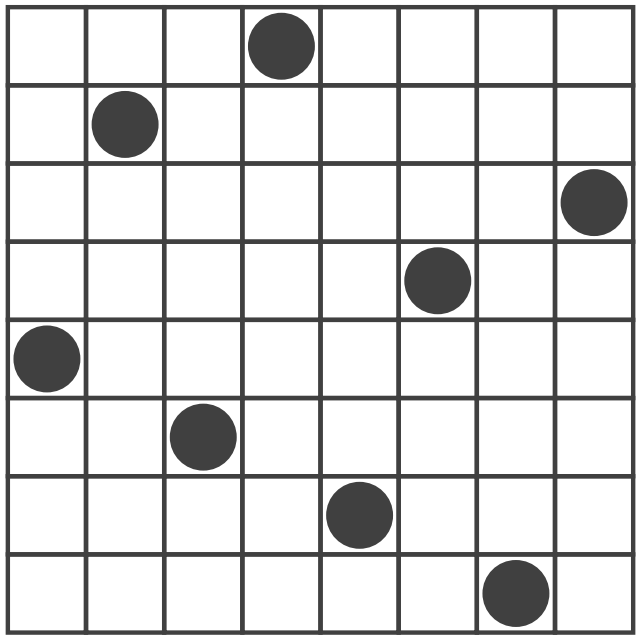
Placing n chess queens on an $n \times n$ chessboard so that no two queens threaten each other: no two queens share the same row, column, or diagonal.

Solution for the 8−Queens problem



Example: n-Queens

Solution for the 8−Queens problem



- A solution can be represented as an array of length n
- Each entry can only hold a value between 0 and $n-1$
- Each entry x_i describes the row in which a queen is positioned in column i

$$[4, 1, 5, 0, 6, 3, 7, 2]$$

Example: n-Queens

The following 6 methods must be implemented

- Population.newPopulation(size)
- population.goalFunction()
- population.copy()
- population.recombine()
- population.mutate()
- newPopulation.selection()

Example: n-Queens

Population.newPopulation(size)

- Generate *size* random instances
  - Every instance contains n random values between 0 and $n-1$
  - Variation: Every instance contains n different random values between 0 and $n-1$

population.goalFunction()

- Creates a value for every instance
- For each pair of queens in the same row or diagonal, the score value (number of conflicts) is increased by 1

Example: n-Queens

population.copy()

- Creates a copy of an instance

population.recombine()

- Choose two random instances $X = [x_1, x_2, ..., x_n]$ and $Y = [y_1, y_2, ..., y_n]$.
- Variation: Select instances with a better goal value with a higher probability.
- Creates a new queens array by selecting one value each from x_i or y_i.
- Variation: Select values from the array with a higher goal value with a higher probability.

Example: n-Queens

population.mutate()

- Choose an instance
- Copy the instance
- Change one or more values of the instance
  - Assign a new value to a random variable.
  - Swap the values of two variables.

newPopulation.selection()

- Select the top n instances based on the goal function.

Summary

Properties and significance of Object-Oriented Programming (OOP): Abstraction, Encapsulation, Reusability, Relations, Polymorphism

For which kind of problems is the paradigm useful?

Java as object-oriented programming language: casting, libraries, cases, iteration, arrays, references, methods, recursion, classes and objects, constructors, object access, visibilities

Evolutionary Algorithms involve selecting the most fit individuals from a population, where these individuals undergo mutation and recombination.

References

- Charles Darwin. On the Origin of Species by Means of Natural Selection. Murray, London, 1859. or the Preservation of Favored Races in the Struggle for Life.
- A. E. Eiben and James E. Smith. Introduction to Evolutionary Computing. Springer Publishing Company, Incorporated, 2nd edition, 2015.
- P. Pepper. Programmieren lernen: Eine grundlegende Einführung mit Java. eXamen.press. Springer Berlin Heidelberg, 2007.
- Reinhard Schiedermeier. Programmieren mit Java . Pearson Deutschland, 2010.