Programming Languages and Concepts

Siegfried Benkner

Research Group Scientific Computing

Universität Wien

Contents

- **Classes and Objects**
- **Inheritance**
- **Polymorphism**
- **Dynamic Binding**
- **Encapsulation**
- **Reuse**
- Appendix

Literature

Java Tutorial, Lesson: Object-Oriented Programming Concepts

http://docs.oracle.com/javase/tutorial/java/concepts/

Appendix

- Serialization
- Shadowing
- Arrays
- Strings
- Classes – Scopes
- Local Variable Type Inference
- Immutable Objects
- Records

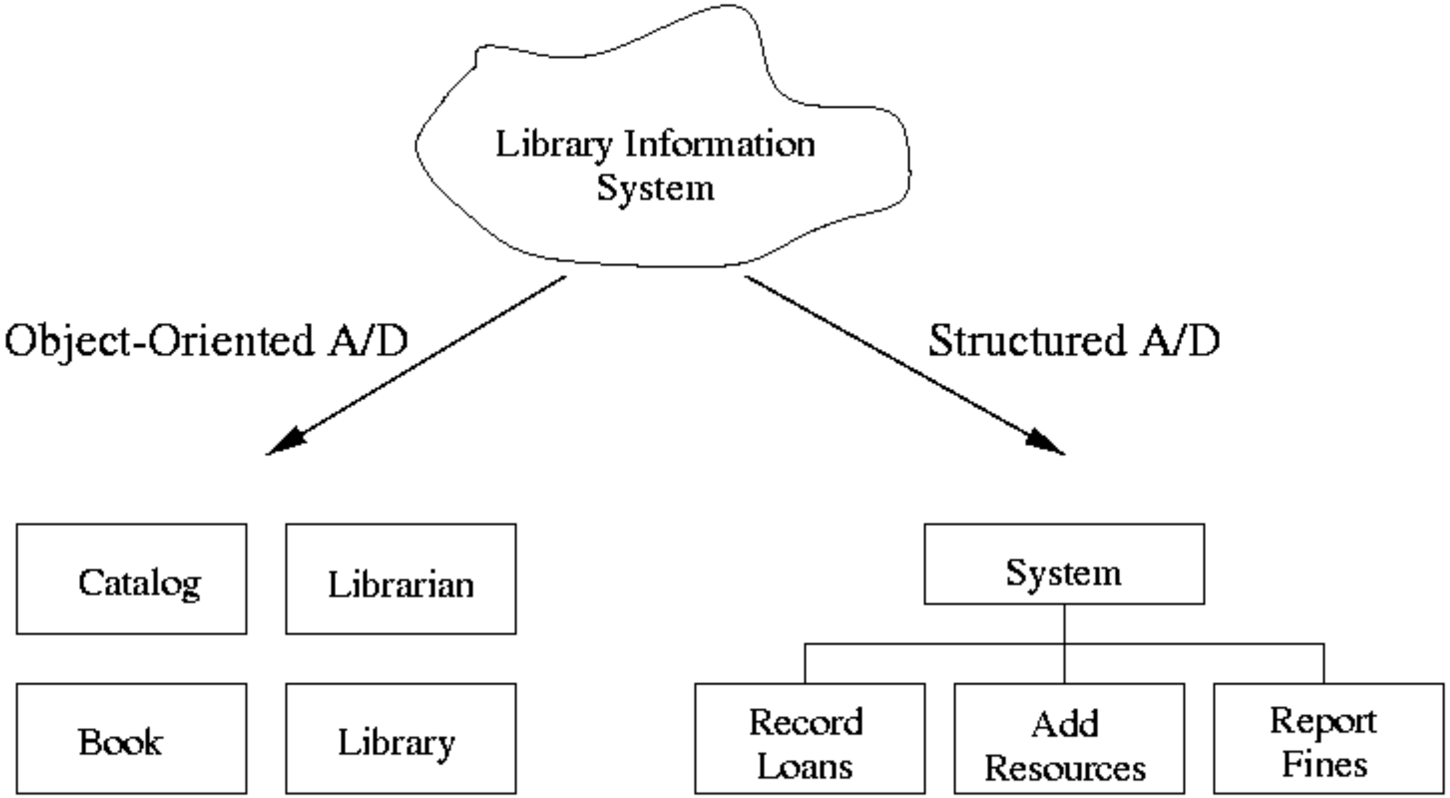
Object-Oriented Languages

- OO languages provide the following main concepts:
  - **Classes and objects**
  - **Inheritance**
  - **Dynamic binding**
  - **Polymorphism**
  - **Encapsulation**
- Software systems based on OO languages often are characterized by lower complexity, better reusability, and easier maintainability.
- The roots of OO languages go back to the late 1960ies - Simula 67 (Nygaard, Dahl), Smalltalk (Xerox, 1970).
- Some OO languages: C++, Objective C, Smalltalk, Simula, Eiffel, Object Pascal, CLOS (Common LISP Object System), Ada 95, Java, Scala, Kotlin.

OO vs. Procedural Programming Paradigms

- **Object-Oriented Paradigm**
  - Data and operations (methods) form a unit → **Class**
  - Primary question: **WHAT?**
  - Smalltalk, C++, Java, ...
- **Procedural Programming Paradigm**
  - Separation of data and operations (procedures)
  - Primary question: **HOW?**
  - Pascal, C, Fortran, ...

OO vs Procedural Software Development



OO

- Decomposition based on concepts i.e., classes.

Procedural

- Decomposition based on processes i.e., procedures, functions.

Classes and Objects

- A class describes the properties of objects.
- A class declaration defines the variables (fields) and methods of its objects.

```
class Person {                         // class declaration
   String name;                        // variable (field)
   private String svnr;                //    - " -
   ...
   public Person(String name, String svnr) { // constructor
      this.name = name; this.svnr = svnr;
   }
   public int getAge() { ... }         // method
}
```

- A class declaration defines a new type (reference type), e.g. the type **`Person`**.

Classes and Objects

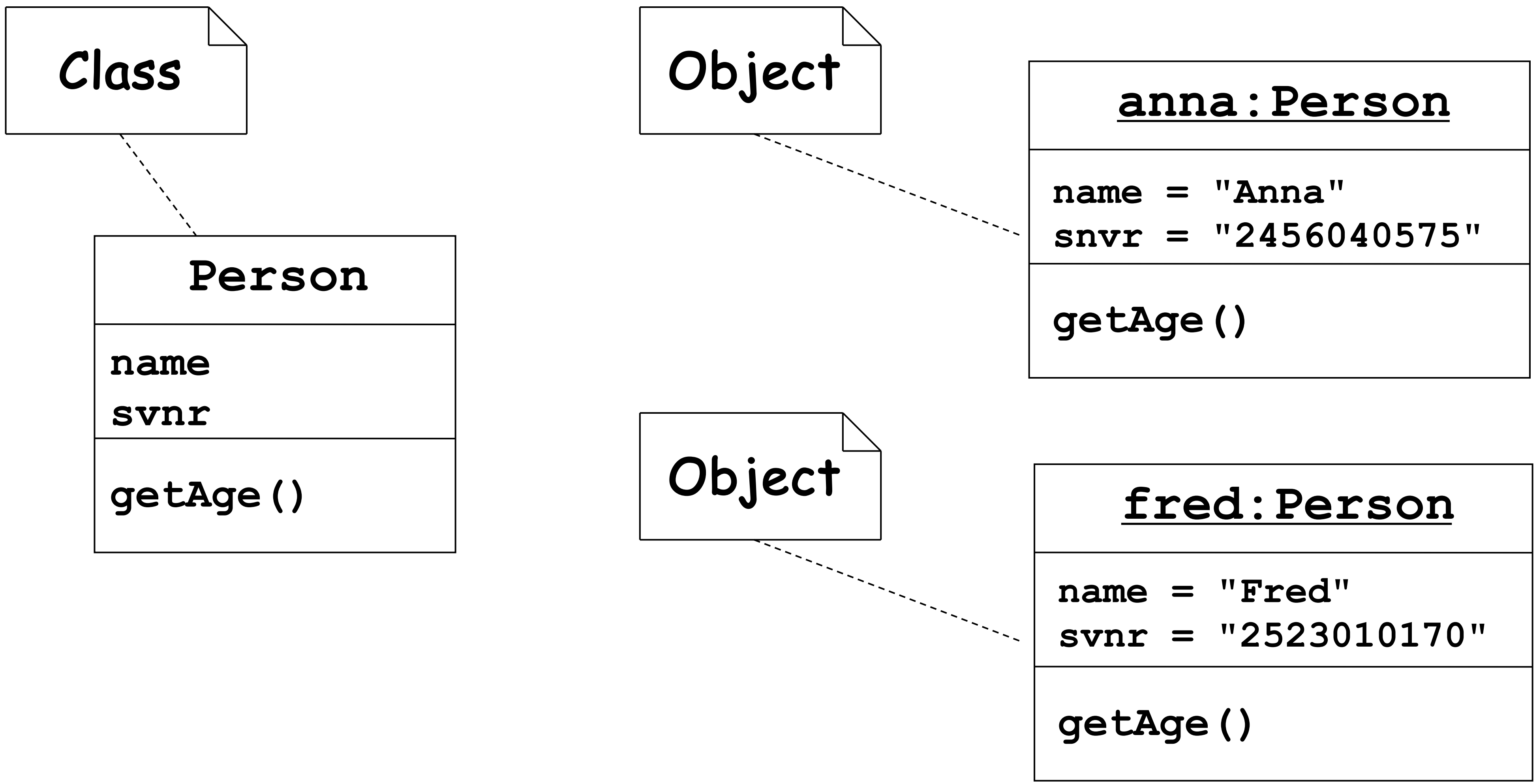
- Objects are instances of a class and are instantiated upon invocation of a constructor (new operator) of the class.

```
// create instance (object) of class Person
Person hugo = new Person("Hugo", "1121030333");
```

- Accessing variables or calling methods of an object is done via the "." operator.

```
String name = hugo.name;          // access variable
int a = hugo.getAlter();          // call method
```

Classes and Objects: UML Diagrams

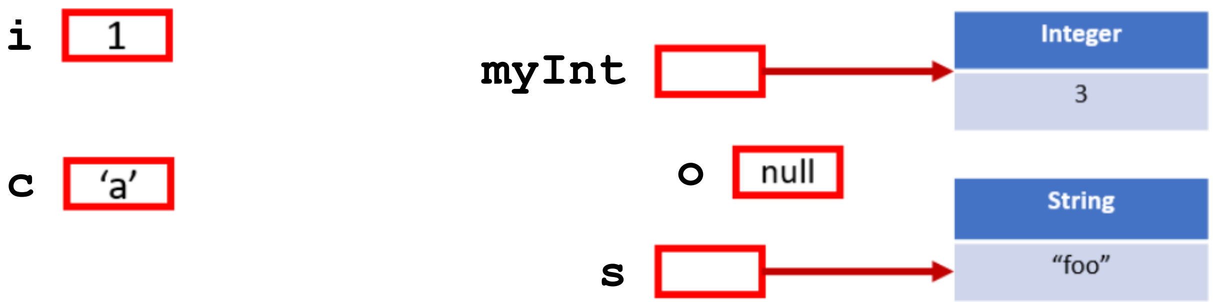


Primitive Types vs. Reference Types

- A primitive type is a built-in predefined data type with a standardized representation.

  `byte, short, int, long, float, double, char, boolean`

- A reference type is usually defined by a class, interface or array declaration.




```
int i = 1;
char c= 'a';
Integer myInt = 3;
Object o;
String s = "foo";
```

Java - Primitive Data Types

Keyword	Description	Size/Format
	integers	
`byte`	Byte-length integer	8-bit two's complement
`short`	Short integer	16-bit two's complement
`int`	Integer	32-bit two's complement
`long`	Long integer	64-bit two's complement
	real numbers	
`float`	Single-precision floating point	32-bit IEEE 754
`double`	Double-precision floating point	64-bit IEEE 754
	other types	
`char`	A single character	16-bit Unicode character
`boolean`	A boolean value	true or false

Primitive Types vs. Reference Types

- **Primitives Types**
  - Better performance, because they are typically *inlined*—stored directly (without headers or pointers) on the stack and in registers.
  - Memory access is more efficient (no additional indirections).
  - Primitive arrays are stored contiguously in memory – better locality.
  - Primitive values do not require garbage collection.

- **Reference types (objects)**
  - Better abstraction, including access control, and subtyping.
  - But objects traditionally perform poorly in comparison to primitives, because they are primarily stored in heap-allocated memory and accessed by reference.

Primitive Types vs. Reference Types

- **Primitives Types**
  - Have a corresponding wrapper type, e.g. `Integer`, which is a reference type, and there are boxing and unboxing conversion between primitive types and their corresponding wrappers;
  - The value set of primitive types never includes null;

- **Reference types (objects)**
  - The value set of reference types consists not of objects, but of references to objects, and always includes null;
  - Objects have object identity.

Java - Reference Data Types

- Class types are defined via a class declaration, which describes the structure and implementation of objects.

```
class Circle { ... }
```

- Interface types usually only describe the signature of abstract methods, but not their implementation.

```
interface Sender { ... }
```

- Array types define arrays.

```
int table[];
Circle[] circles;
```

Classes and Objects - Instantiation

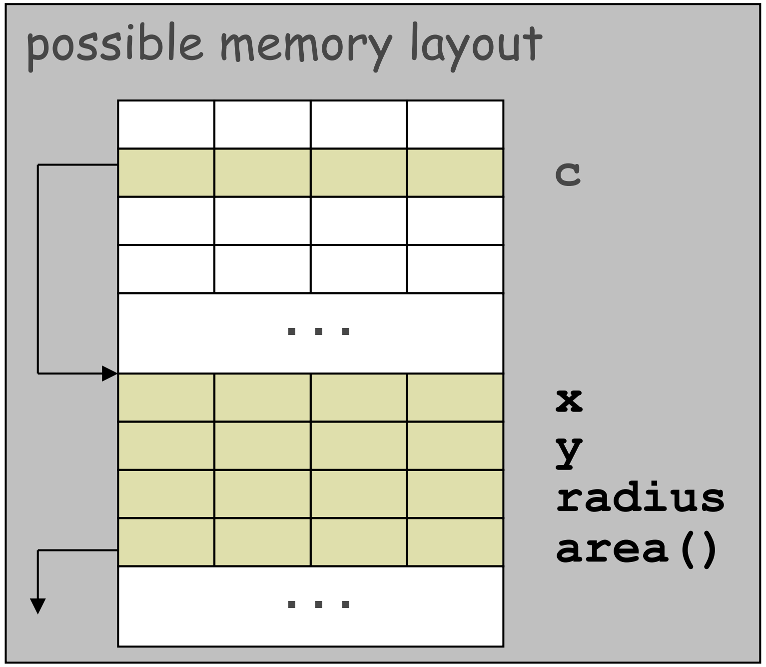
- A class is instantiated with the **`new`** operator
  - Memory allocation.
  - Initialization of all variables with default values and invocation of a (default) constructor.
  - Return a reference to the newly generated object.

```
class Circle {
   public double x,y;
   public double radius;
   public double area() {...}
}
...

Circle c = new Circle();

c.radius = 3.7;

a = c.area();
```



Classes and Objects

Constructors example:

```
class Circle {
   public double x,y, radius;

   public Circle(double x, double y, double radius) {
      this.x = x; this.y = y; this.radius = radius;
   }

   public Circle(Point p, double radius) {
      this.x = p.x; this.y = p.y; this.radius = radius;
   }
   ...
}
...
Point  o = new Point(0.0, 0.0);
Circle c = new Circle(1.1, 2.2, 3.0);
Circle d = new Circle(o, 5.0);
```

Classes and Objects

- Reference operator `this`
  - `this` can be used to refer to the actual object.
  - `this(…)` can be used to refer to a constructor of the actual class.

```
class Circle {
   public double x,y, radius;

   public Circle(double x, double y, double radius) {
      this.x = x; this.y = y; this.radius = radius;
   }

   public Circle(Point p, double radius) {
      this(p.x, p.y, radius);
   }
   ...
}
```

Classes and Objects

- **Deleting Objects**
  - The Java runtime system automatically frees memory for objects, which are not longer used (automatic garbage collection).
  - A garbage collector (thread) runs in the background with each program.

Classes and Objects

- **Object finalization (deprecated)**
  - The user may provide a finalize method, e.g., to free certain resources used by an object, which will be called before an object is garbage collected.

```
class FileIO { ...
   protected void finalize() { ... }
}
```

  - Since Java 18, object finalization is deprecated and should not be used anymore (see: https://openjdk.org/jeps/421).
  - Use other resource management techniques such as the *try-with-resources* statement and *cleaners*.

Classes and Objects – State, Identity, Behavior

- **Objects are stateful.**
  - The state of an object is determined by the content of its variables.
  - The state of an object may change during runtime.

- **Objects have an identity.**
  - The identity of an object is its being distinct from any other object.
  - As opposed to objects, primitive types have no identity.

- **Behavior.**
  - The behavior of an object is determined by its methods.
  - Objects interact with each other by exchanging messages (i.e., calling methods).

Classes and Objects

Identity vs. Equality

- The **==** operator checks if two variables refer to the same object.

```
String s1 = "text"; String s2 = s1;
String s3 = new String("text");
String s4 = "text";
System.out.println(s1 == s2); // true, s1,s2 refer to same object
System.out.println(s1 == s3); // false
System.out.println(s1 == s4); // true, compiler generates
                              // single object “text”
```

Identity vs. Equality

- The `equals()` method (inherited from class Object) may be overridden to implement equality checks, i.e., if two objects have the same content.

```
...(s1.equals(s2) && s1.equals(s3) && s1.equals(s4))... // true
```

Immutable Objects

```
final class Point {
    public final int x;
    public final int y;

    public Point(int x, int y) { this.x = x; this.y = y; }
}
```

Assume we have an array of **`Point`** objects (and a 64-bit JVM)

- Two integers **`x,y`** require 8 bytes
- Object header requires 16 bytes
- Reference requires 8 bytes

➢ 8 bytes of data take 32 bytes of heap space (4 X)

➢ Iterating over the array requires a pointer dereference every Point iteration, destroying the inherent locality of arrays and limiting program performance.

See: http://cr.openjdk.java.net/~jrose/values/values-0.html

Records

- Records are classes that act as transparent carriers for immutable data.
- The header of a record describes its state (the types and names of its fields).
- As objects, instances of records also have an identity.
- As opposed to classes, records do not support decoupling the internal implementation from its API.
- As opposed to classes, records cannot be extended.

```
record Rectangle(double length, double width) { }
...
record r = new Rectangle(4,5);
```

See: https://openjdk.org/jeps/384

Records

The previous record declaration is equivalent to the following class.

```
public final class Rectangle {
    private final double length;
    private final double width;

    public Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }

    double length() { return this.length; }
    double width()  { return this.width; }

    // Implementation of equals() and hashCode(), which specify
    // that two record objects are equal if they
    // are of the same type and contain equal field values.
    public boolean equals...
    public int hashCode...

    // An implementation of toString() that returns a string
    // representation of all the record class's fields,
    // including their names.
    public String toString() {...}
}
```

Records, Value Objects, Primitive Classes

- **Records** (JEP 384, https://openjdk.org/jeps/384)
  - Classes that act as transparent carriers for immutable data. Can be thought of nominal tuples.
- **Value Objects** (JEP Draft, https://openjdk.org/jeps/8277163)
  - Instances of *value classes* that have only final instance fields and no object identity.
- **Primitive Classes** (JEP 401, https://openjdk.org/jeps/401)
  - Special case of value classes.
  - Instances of primitive classes have no identity and can be converted between value objects and simpler primitive values, e.g., a sequence of field values, without any headers or extra pointers.

➢ These extensions try to combine advantages of primitive and reference types.

Serialization

- Serialization is the process of automatically converting an object that exists in memory during runtime into a byte stream that allows, for example,
  - to persistently store the object to a file, or,
  - to send it over the network to a remote Java program (cf. Java RMI).

- An object that has been serialized to a file can be deserialized again into the internal format used by the JVM.

- Serialization/Deserialization preserves the state of an object.

See also appendix of this slide stack.

Serialization

Example: Serialization of an object of type **`List<Article>`** to a file.

```
void serializeArticles(List<Article> articles) {
      File file = new File(filename);
      if (file.exists()) file.delete();

      try {
        ObjectOutputStream writer = new ObjectOutputStream(new
                                      FileOutputStream(filename, true));
        writer.writeObject(articles);
        writer.close();
      }
      catch (Exception e) {
        System.err.println("Error during serialization: " +
                                                e.getMessage());
        System.exit(1);
      }
}
```

Instance Variables vs. Class Variables

- **Instance Variables**
  - Every object has its own copy of an instance variable.
  - Access: **`objectName.variableName`**
  - Example: variables **`x,y`** and **`radius`** of class **`Circle`**

- **Class variables** (static variables)
  - Exist once per class
  - Must be declared with the modifier **`static`**
  - Access: **`className.variableName`** or
    `objectName.variableName` (not recommended)

Instance Variables vs. Class Variables

- Class variables exist once per class, regardless of how many objects of a class have been generated.

```
class Circle {
   public double x,y, radius;    // instance variables
   static int nCircles;          // class variable

   public Circle(double x, double y, double radius) {
      this.x = x; this.y = y; this.radius = radius;
      nCircles++;
   }
}

class MainCircle {
   public static void main (String[] args) {
      System.out.println("Nr = " + Circle.nCircles); // Nr = 0
      Circle c = new Circle(1.0,1.0,1.0);
      System.out.println("Nr = " + c.nCircles);      // Nr = 1
   }
}
```

Methods

- **Instance methods**
  - Always relate to an object.
  - Call: `objectName.methodName(...)`
- **Class methods**
  - Relate to the class, not to an object.
  - Must be declared with the modifier `static`; access to class variables only.
  - Call: `className.methodName(...)` or

    `objectName.methodName(...)` (not recommended)

```
class Circle { ...
   // instance method
   public double area() { return 3.14159*radius*radius; }

   // class method
   static int total() { return nCircles; }
}
```

Singleton Classes

- The Singleton pattern guarantees that a class can be instantiated at most once.

```
class Singleton {
   private static Singleton instance;

   private Singleton() { }

   public static Singleton getInstance() {

   if (instance == null) {
        instance = new Singleton();
   }
   return instance;
}
```

```
Singleton s = Singleton.getInstance();
...
Singleton s2 = Singleton.getInstance();
```

Method Overloading

- A class can have multiple variants of a method, all with the same name.
- If there are multiple methods with the same name, the compiler decides based on the method signature which one to call.
- **Signature**: method name & number and type of formal parameters
- Methods with the same signature but different return type are not allowed

```
class Graphics { ...
   public void moveTo(double x, double y) { ... }
   public void moveTo(Point p) { ... }
}
```

Methods – Parameter Passing

- The parameter passing mechanism in Java is "**call by value**".
- Parameters of primitive types (int, boolean, ...)
  - Value of the parameter will be passed.
- Parameters of reference types (referring to objects or arrays)
  - Value of the parameter (=reference) will be passed.

Methods – Parameter Passing

Example

```
public class ParameterTransfer {
   static void setRadius(double r, Circle c) {
      c.radius = r; r = 0; c = null;
   }
   public static void main(String[] args) {
      double maxr = 100.0;
      Circle c    = new Circle(1.0,1.0,1.0);
      setRadius(maxr, c);
      System.out.println(”maxr = ” + maxr);
      System.out.println(”c.radius = ” + c.radius);
   }
}
```

```
> java ParameterTransfer
maxr = 100.0
c.radius = 100.0
```

Inheritance

- Inheritance (specialization, generalization) allows to derive a subclass from an existing superclass.

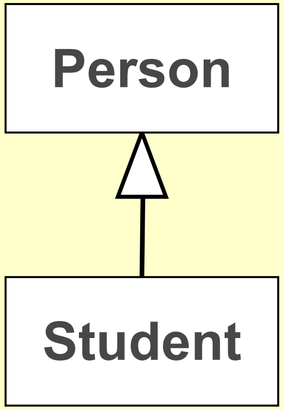
```
class Subclass1 extends Superclass {...}
```

- A subclass
  - inherits the methods and fields from its superclass(es).
  - may defined new methods and fields.
  - may override methods inherited from a superclass.

Inheritance

```
// superclass
class Person {
      private String name;
      private String svnr;
      public int getAge() {...}
}


// subclass
class Student extends Person {
      String matrNr;
}
```

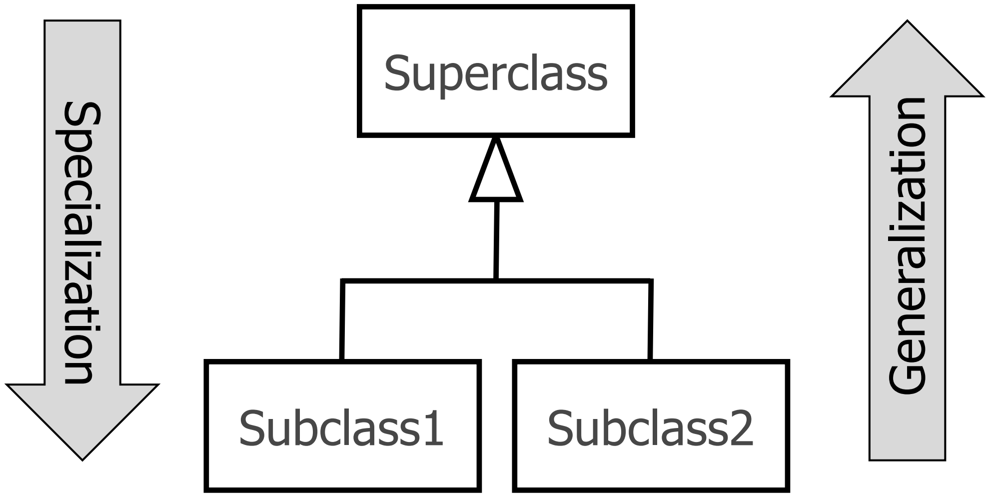


- Student objects have
  - fields: **name, svnr, matrNr**;
  - methods: **getAge()**

- An inheritance relation is also referred to as an "**is-a**" relation.

Inheritance

- **Class Hierarchy**
  - A superclass is an **abstraction** of a subclass with less details.
- **Specialization**
  - A subclass represents a specialized concept of a superclass
- **Generalization**
  - A superclass represents a more general concept than a subclass



- Other abstraction mechanisms in Java are abstract classes and interfaces.

Inheritance – Constructor Chaining

- A constructor of a subclass must call a constructor of its superclass, using the keyword **`super`**.

- If a constructor of the superclass is not explicitly called, a call to the default constructor of the superclass (**`super()`**) is generated by the compiler. This will lead to a compile time error if the superclass does not have a default constructor.

- A call to a constructor of a superclass must always be the first statement of a constructor.

Inheritance – Constructor Chaining

```
class Person {
      String name;
      private int age;

      public Person(String name, int alter) {
         this.name = name; this.alter = alter;
      }
      ...
}

class Student extends Person {
      String matrNr;

      public Student(String name, int age, String matrNr) {
         super(name, age);      // call constructor of Person
         this.matrNr = matrNr;
      }
}
```

Inheritance - Polymorphism

- Polymorphism designates the ability of a variable to refer to objects of different types (instances of different classes).

```
class Student extends Person { ... }
Person p = new Person();
p = new Student();
```

- An object of a subclass **`B`** has also the types of all superclasses of **`B`**. Thus, a variable of of type **`B`** has also the type **`A`**, if **`A`** is a superclass of **`B`**.

- Example:

  `s` is of type **`Student`** and of type **`Person`**, since every student is-a person.

```
Student s = new Student();
boolean test1 = s instanceof Student;    // true
boolean test2 = s instanceof Person;     // true
```

Inheritance – `instanceof` Operator

- The **`instanceof`** operator may be used to check if a variable refers to an instance of a particular class.

```
Person p;
...
if (p instanceof Student) {
   Student s = (Student) p;
   System.out.println(s.name + ": " + s.matrNr);
}
```

- The **`instanceof`** operator supports **pattern matching**, e.g., to allow the above *instanceof-and-cast idiom* to be expressed more concisely.

```
if (p instanceof Student s) {
   System.out.println(s.name + ": " + s.matrNr);
}
```

Inheritance - Assignment Compatibility

- A variable of type T can be assigned to a variable of type T or any supertype of T.

```
class Person { ... }
class Student extends Person { ... }
class Course { void register(Person p) { ... } }
...

Person p = new Person();
Student s = new Student();
s = p;                              // error
p = s;                              // compatible
Course c = new Course();
c.register(s);                      // compatible
```

Inheritance – Cast Operator

- If a variable **`a`** of type **`A`** refers to an object of type **`B`** and **`B`** is a subclass of **`A`**, then **`a`** can be assigned to a variable **`b`** of type **`B`** only when an explicit type cast is performed.

```
class A { ... }
class B extends A { ... }

A a;
B b = new B();

a = b;
b = (B) a;        // cast type of a to B
```

- If the variable **`a`** did not refer to an object of type B in the above example, a runtime exception would be raised (**`ClassCastException`**).

Inheritance – Cast Operator

```
class Person {
        private String name;
        private String svnr;
        public int getAge() {...}
}

class Student extends Person {
        private String matrNr;
}


...
Person p;
Student s = new Student();
...
p = s;              // compatible (every student is a person)
s = (Student) p;    // cast p to Student
```

Inheritance – Method Overriding

- A subclass can override methods inherited from a superclass, i.e., newly implement or extend the implementation from the superclass.
- Method overriding requires that the method in the subclass has the same signature as the method of the superclass that is overridden.
- The annotation **`@Override`** may be used to indicate method overriding.
- The keyword super may be used to extend methods inherited from a superclass, e.g., **`super.m()`** calls method **`m()`** of a superclass.
- Overridden methods are also referred to as polymorphic methods.

Overriding vs. Overloading

```
public class A {
   ...
   void set(int i) { ... }
}

public class B extends A {
   ...
   void set(int i) { ... }         // overriding
   void set(char c) { ... }        // overloading
}
```

- Same Signature ... Overriding
- Different Signature ... Overloading

Inheritance – Method Overriding

```
public class A {
    ...
    void doSomething() { ... }
}

public class B extends A {
    ...
    @Override
    void doSomething() {          // extend doSomthing() of A
        super.doSomething() ;     // call doSomething() of A
        ...
    }
}
```

Dynamic Binding

- If an overridden method is called using a polymorphic variable, the compiler cannot determine which variant of the overridden method to call.

- Since the actual type of the object a polymorphic variable is referring to cannot be determined statically (i.e. at compile time), the compiler cannot decide which variant of an overridden method to call.

- Thus, which variant of an overridden method will be called is deferred to the runtime, when the concrete type of the object the variable used for the method call refers to is known.

Dynamic Binding

```
public class Point {
    ...
    public void draw() { ... }          // use default color
}

public class Pixel extends Point {
    Color color;
    ...
    @Override
    public void draw() { ... }          // use color color
}
...
Point p = new Point(); Pixel px = new Pixel();
...
if (i == 0) p = px;
p.draw();                               // dynamic binding
```

- If at runtime **`p`** refers to a **`Point`** call **`draw()`** of **`Point`**, else call **`draw()`** of **`Pixel`**.

Dynamic Binding


```
class Animal {
    public void saySomething() { System.out.println("?"); }
}
class Frog extends Animal {
    public void saySomething() { System.out.println("quak!"); }
}
class Dog extends Animal {
    public void saySomething() { System.out.println("wow!"); }
}
...
Animal animals[] = new Animal[10] Random randomGen = new Random();
for (int i = 0; i < 10; i++)
    if (randomGen.nextDouble() < 0.5)
        animals[i] = new Frog();
    else
        animals[i] = new Dog();
animals[1].saySomething();          // Dog or Frog ?
```

Dynamic Binding

- In Java dynamic binding is the default mechanism. It provides more flexibility, since it allows referring to different implementation variants of a method in a uniform way.

- In C++, to enable C++ dynamic binding, a functions must be declared **`virtual`**.

- Dynamic binding is associated with a runtime overhead as compared to static binding.

- Methods declared **`private`**, **`final`** or **`static`** cannot be overridden and thus can be statically bound.

Multiple Dispatch

- While in Java, C++, … the dynamic type of the implicit first parameter (this) is used for method binding, in languages that support multiple dispatch (e.g., Julia, R, Perl) the dynamic types of all parameters are used.
- Multiple dispatch is the ability to define behavior across many combinations of argument types.
- Overloading can be seen as multiple dispatch at compile time.

```
public class TestDynamicDispatch {
   public static void main(String[] args) {
      Float f = new Float(0);
      myMethod(f);              // → Float
      Number n = f;
      myMethod(n);              // → Number (with multiple dispach: Float)
   }
   public static void myMethod(Float x) { System.out.println("Float"); }
   public static void myMethod(Number x) { System.out.println("Number"); }
}
```

Inheritance – Class `Object`

- Every class is an implicit subclass of class **`Object`**.
- The methods of **`Object`** can be used (or overridden) in every class, e.g. **`equals(), clone(), toString(), finalize().`**
- A variable of type **`Object`** can refer to any Java object.

Inheritance – Class `Object`

Note: better solution with Generics (since Java 5)!

```
Person p = null;
LinkedList ll = new LinkedList();    // create empty list
...

while ((p = myReader.readNext()) != null) {
      ll.add(p);                      // insert p into list
}
...

for (Iterator i = ll.iterator(); i.hasNext(); ) {
     p = (Person)i.next();      // get next object from list
                                // cast Object to Person
}
...
```

Methode add der Klasse LinkedList:
`void add(Object o)`

Methode next() der Klasse Iterator:
`Object next()`

Restricting Inheritance

- A class can be declared `sealed` if all its direct subclasses are known when the class is declared and no other direct subclasses are desired or required.
- A class can be declared `final` if its definition is complete and no subclasses are desired or required.
- Every permitted subclass of a sealed class must be declared `final`, `sealed` or `non-sealed`.

```
public abstract sealed class Vehicle permits Car, Truck {
...
}
```

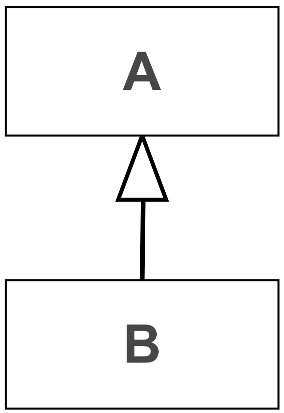
```
public final class Car extends Vehicle {...}
```

```
public non-sealed class Truck extends Vehicle {...}
```

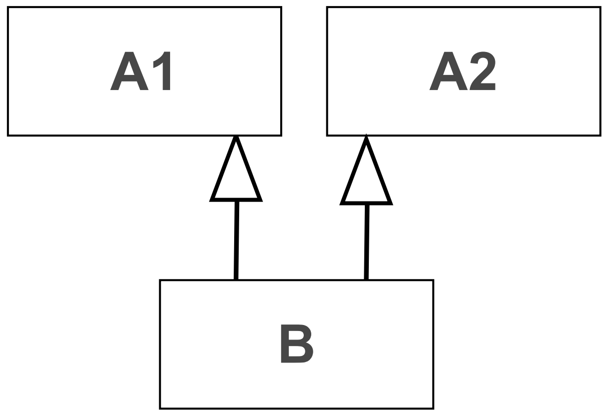
See: https://openjdk.org/jeps/409

Inheritance – Single vs. Multiple Inheritance

- Java supports only single inheritance – a class can extend only one superclass.
- C++ supports multiple inheritance.
- The effect of multiple inheritance can be achieved in Java using interfaces.



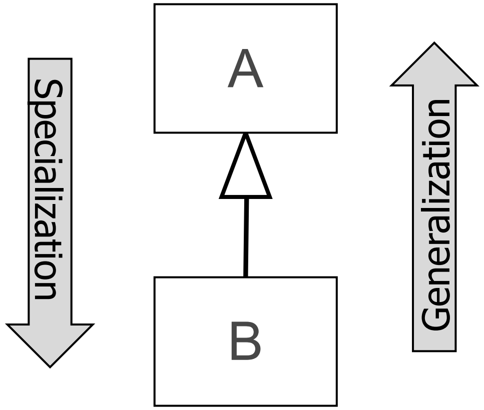
Single inheritance



Multiple inheritance
(not supported in Java)

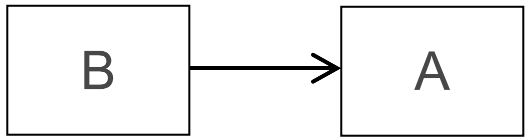
OO Reuse Mechanisms

Inheritance



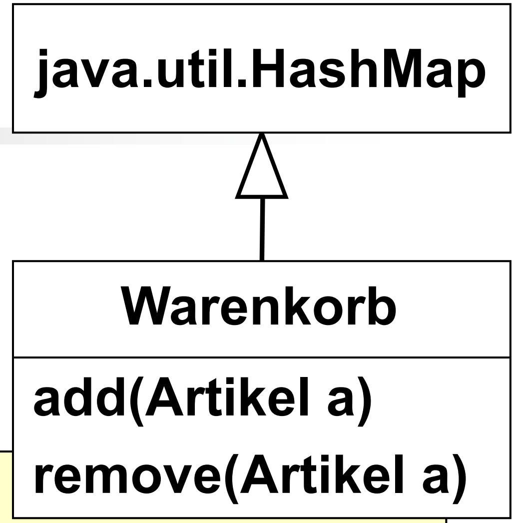
- Reuse of (all) data and methods of superclasses
- ***is-a*** relationship
- Simple reuse mechanism
- Static
- Tight coupling between subclass and superclass (breaks encapsulation)
- Generalization vs. specialization

Delegation



- Reuse other class by using an object of that class
- ***has-a*** relationship
- More complex than inheritance (multiple objects, instantiation …)
- Dynamic
- More flexible; looser coupling between subclass and superclass

Reuse - Inheritance



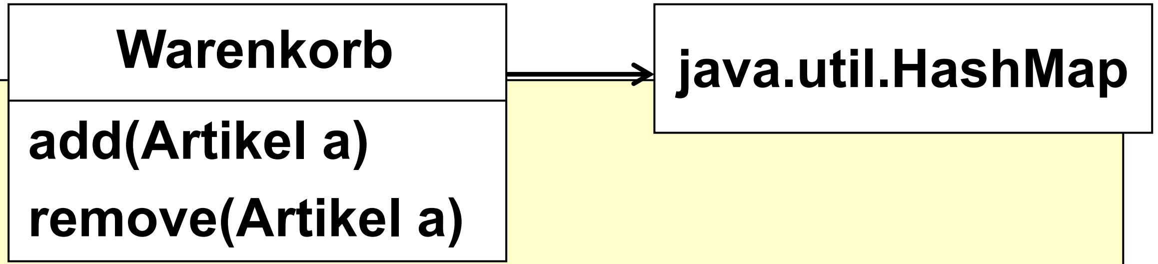
- Inheritance is a suitable mechanism to achieve code reuse.
- A subclass inherits all functionality of its superclass(es).
- A subclass can *specialize* (overwrite) inherited methods.

```
import java.util.HashMap;

public class Warenkorb extends HashMap {
    public void add(Artikel artikel) {
            WarenkorbItem item = new WarenkorbItem(artikel, 1);
            put(artikelId, item);       // put into HashMap
            ...
    }
    public void remove(Artikel artikel) {
            if (containsKey(artikelId)) remove(artikelId);
            ...
    }
```

Reuse - Delegation

- Reuse functionality of class HashMap via a member variable.




```
import java.util.HashMap;

public class Warenkorb {
    private HashMap hm = new HashMap();
    public void add(Artikel artikel) {
            WarenkorbItem item = new WarenkorbItem(artikel, 1);
            hm.put(artikelId, item);        // put into HashMap
            ...
    }
    public void remove(Artikel artikel) {
            if (hm.containsKey(artikelId)) hm.remove(artikelId);
            ...
    }
```

Definition: Polymorphism

[Booch 91]
A concept in type theory, according to which a **name** (such as a variable declaration) **may denote objects of many different classes** that are related by some common superclass; thus, any **object** denoted by this name **is able to respond to some common set of operations in different ways.**

[Meyer 88]
"Polymorphism" means the ability to take several forms. In object-oriented programming, this refers to the **ability of an entity to refer at run-time to instances of various classes**. In a typed environment such as Eiffel, this is constrained by inheritance.

Definition: Polymorphism

[Stroustrup 90]
The use of derived classes and virtual functions is often called "object-oriented programming". Furthermore, the ability to call a variety of functions using exactly the same interface - as is provided by virtual functions - is sometimes called "polymorphism".

[Rumbaugh 91]
"Polymorphism" means that the same operation may behave differently on different classes.

[https://docs.julialang.org/en/v1/manual/types/]
The ability to write code that can operate on different types is called polymorphism.

Liskov Substitution Principle

[Liskov 87]

Let $\phi(x)$ be a property provable about objects x of type T. Then $\phi(y)$ should be true for objects y of type S where S is a subtype of T.

Substitutability states that, if S is a subtype of T, then objects of type T may be *replaced* (substituted) with any object of type S without altering any of the desirable properties of the program (correctness, etc.).

Encapsulation

- The internals of an object should not be accessible (visible) from outside.

- **Visibility attributes**
  - **`public`**: global access
  - **`private`**: no access from outside the class
  - **`protected`**: access from subclasses and the same package
  - default (*package): access from same package*

```
class Person {
   ...
   private String svnr;          // not accessible outside
   public int getSvnr() {...}   // access method
}
```

- Private variables or methods of a class may be changed without requiring any other classes to be changed as well.

Appendix

- Serialization
- Shadowing
- Arrays
- Strings
- Classes – Scopes
- Local Variable Type Inference
- Immutable Objects
- Records

Serialization

Serialization is the process of automatically converting an object that exists in memory during runtime into a byte stream that allows, for example,

- to persistently store the object to a file, or,
- to send it over the network to a remote Java program (cf. Java RMI).

An object that has been serialized to a file can be deserialized again into the internal format used by the JVM.

Serialization/Deserialization preserves the state of an object.

Serialization

For serialization the class **`ObjectOutputStream`** is provided (package java.io).

```
class ObjectOutputStream {
    ...
    public final void writeObject(Object obj) throws IOException ...
    ...
}
```

During serialization, the following data of an object are written to **`ObjectOutputStream`**, using the method **`writeObject`**:

- the class of the object
- all member variables including all variables inherited from superclasses

Serialization

Deserialisation can be achieved using the class **`ObjectInputStream`**.

```
class ObjectInputStream {
   ...
   public final Object readObject() throws IOException ...
   ...
}
```

In order for an object to be serializable, its class must implement the **Serializable** (marker) interface.

```
class Person implements Serializable {

   ...

}
```

Serialization

Example: Serialization of an object of type **`List`** to a file.

```
void serializeVehicles(List<Vehicle> vehicles) {
      File file = new File(dateiname);
      if (file.exists()) file.delete();

      try {
        ObjectOutputStream writer = new ObjectOutputStream(new
                                    FileOutputStream(dateiname, true));
        writer.writeObject(vehicles);
        writer.close();
      }
      catch (Exception e) {
        System.err.println("Fehler bei Serialisierung: " +
                                                   e.getMessage());
        System.exit(1);
      }
}
```

Serialization

Example: Deserialization of an object of type `List` from a file.

```
@SuppressWarnings("unchecked")
List<Vehicles> deserializeVehicles() {

   File file = new File(dateiname);
   ...

   List<Vehicle> vehicles = null;

   try {
      ObjectInputStream reader;
      reader = new ObjectInputStream(newFileInputStream(dateiname));
      wohungen = (List<Vehicles>) reader.readObject(); // unchecked
      reader.close();
   }
   catch (Exception e) {
      ...
   }
   return vehicles;
}
```

Serialization

Serialization of an object can be a very elaborate and complex process:

- The JVM must determine at runtime all the member variables of the object to be serialized (including all inherited from superclasses). This processes is based on the Java Reflection API.

- Since member variables can be objects themselves, which also need to be serialized, serialization has to correctly handle cyclic references.

Serialization

Deserialization of untrusted data is inherently dangerous and should be avoided. [Secure Coding Guidelines for Java SE, https://www.oracle.com/technetwork/java/seccodeguide-139067.html#8]

Security

Passengers ride free on SF Muni subway after ransomware infects network, demands $73k

Office admin systems derailed by malware

By Chris Williams, Editor in Chief 27 Nov 2016 at 21:39 98 SHARE ▼

`https://www.theregister.co.uk/2016/11/27/san_francisco_muni_ransomware/`

Alternative technologies are, e.g., **JSON** (JavaScript Object Notation) or Protocol Buffers (**protobuf**).

Inheritance – Shadowing/Hiding

- **Static Methods**
  - If a subclass defines a static method with the same signature as a static method of the superclass, then the method in the subclass shadows/hides the method of the superclass.
  - Shadowed static methods are statically bound.

- **Variables/fields**
  - A variable of a subclass shadows a variable with the same name of a superclass shadows.
  - The shadowed variable may be accessed in the subclass via super or via this and an explicit type cast.
  - Shadowed variables are statically bound.

Inheritance – Shadowing/Hiding

```
public class Animal {
    public static void testClassMethod() {
        System.out.println("Static method in Animal"); }
    public void testInstanceMethod() {
        System.out.println("Instance method in Animal"); }
}
```

```
public class Cat extends Animal {
    public static void testClassMethod() {
        System.out.println("Static method in Cat"); }
    public void testInstanceMethod() {
        System.out.println("Instance method in Cat"); }

    public static void main(String[] args) {
        Cat myCat = new Cat();
        Animal myAnimal = myCat;
        myAnimal.testClassMethod();    // Static method in Animal
        myAnimal.testInstanceMethod(); // Instance method in Cat
    }
}
```

See: https://docs.oracle.com/javase/tutorial/java/IandI/override.html

Inheritance – Shadowing/Hiding

```
class A {
    int i = 1;
}

class B extends A {
    int i = 2;

    public void print() {
        System.out.println(i);              // → 2 (i of B)
        System.out.println(this.i);         // → 2 (i of B)
        System.out.println(super.i)         // → 1 (i of A)
        System.out.println(((A)this).i)     // → 1 (i of A)
    }
}
```

Inheritance – Shadowing/Hiding

```
class S             { int x = 0; }
class T extends S { int x = 1; }  // shadowing
class StaticBindingFieldsTest {
    public static void main(String[] args) {
        T t = new T();
        System.out.println("t.x=" + t.x + when("t", t));
        S s = new S();
        System.out.println("s.x=" + s.x + when("s", s));
        s = t;
        System.out.println("s.x=" + s.x + when("s", s));
    }
    static String when(String name, Object t) {
        return " when " + name + " holds a "
                        + t.getClass() + " at run time.";
    }
}
```

```
t.x=1 when t holds a class T at run time.
s.x=0 when s holds a class S at run time.
s.x=0 when s holds a class T at run time.
```

!! Static binding for fields!

Overriding, Overloading, Shadowing

```
class A {
   int s;
   void set(int s) { ... }
}
class B extends A {
   int s;                          // shadowing
   void set(int s) { ... }     // overriding set() of A
   void set(char c){ ... }     // overloading
}
...
A a = new A(); B b = new B();
a = b;
a.set(5);                          // invoke set() of B!
System.out.println(a.s);       // access s of A
```

Arrays

- Arrays are objects of a reference type.
- Array elements may be of primitive type or of reference type (objects)
- All elements of an array must be of the same type.
- Arrays are allocated dynamically using the **`new`** operator.

```
Circle circles[] = new Circle[5];    // array of 5 circles

int[] numbers = new int[5];          // array of 5 integers

int l = numbers.length;              // length of array
for (int i=0; i<l; i++) {
    numbers[i] = i;
}
```

Arrays

Multdimensional Arrays

- Are realized as arrays of arrays (not necessarily rectangular).
- May be initialized during allocation (static initializers).

Declaration and allocation

```
int[][] a      = new int[3][2];      // 3x2 array
int[]   a[]    = new int[3][2];      //  - " -
int     a[][]  = new int[3][2];      //  - " -
```

Declaration, allocation and initialization

```
int[][] m;
m = new int[3][2];
for(int i=0; i<m.length; i++)
  for(int j=0; j<m[i].length; j++)
     m[i][j] = 2*i+j+1;
```

```
// static initializer
int[][] m = {{1,2},{3,4}};
```

Arrays

Multdimensional Arrays

- When allocating multidimensional arrays at least the size of the first dimension must be specified, while the size of remaining dimension can be omitted.

```
int a[][][] = new int[3][][2];     // error
int b[][][] = new int[3][2][];
```

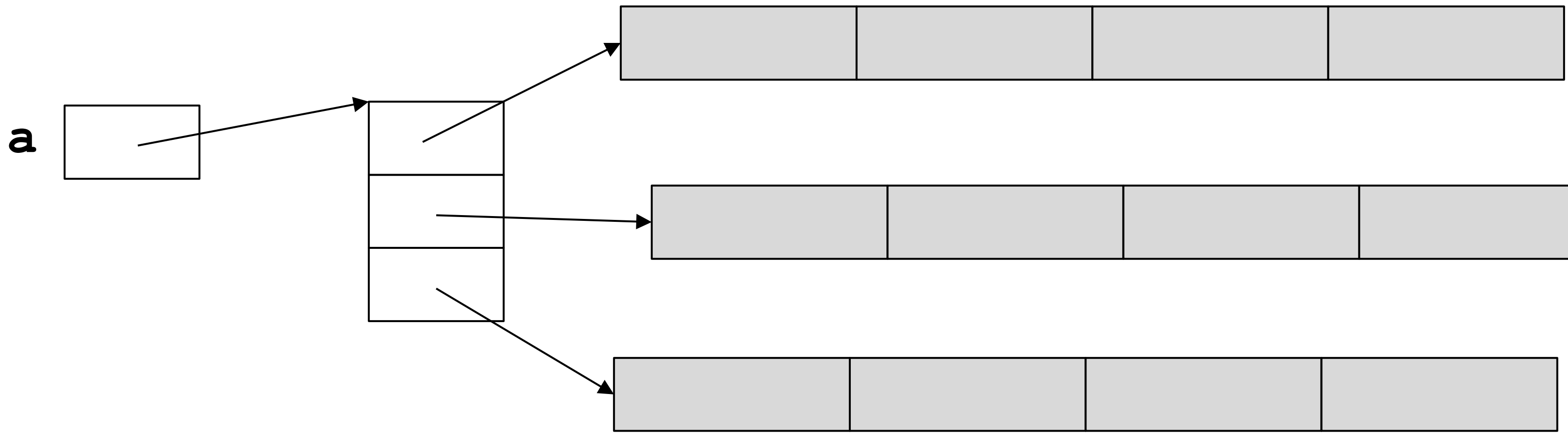
- An n-dimensional array is assignment compatible to an n-dimensional array of a compatible type, i.e. Java array types are covariant.

```
Student[] sa = new Student[10];
Person[] ps;
ps = sa;
```

Arrays

Multdimensional Arrays – Memory Layout

```
final int N=3, M=4;
double[][] a = new double[N][M];
```




```
for (int i=0; i<N; i++)
    for (int j=0; j<M; j++)
        a[i][j] = i*N+M;
```

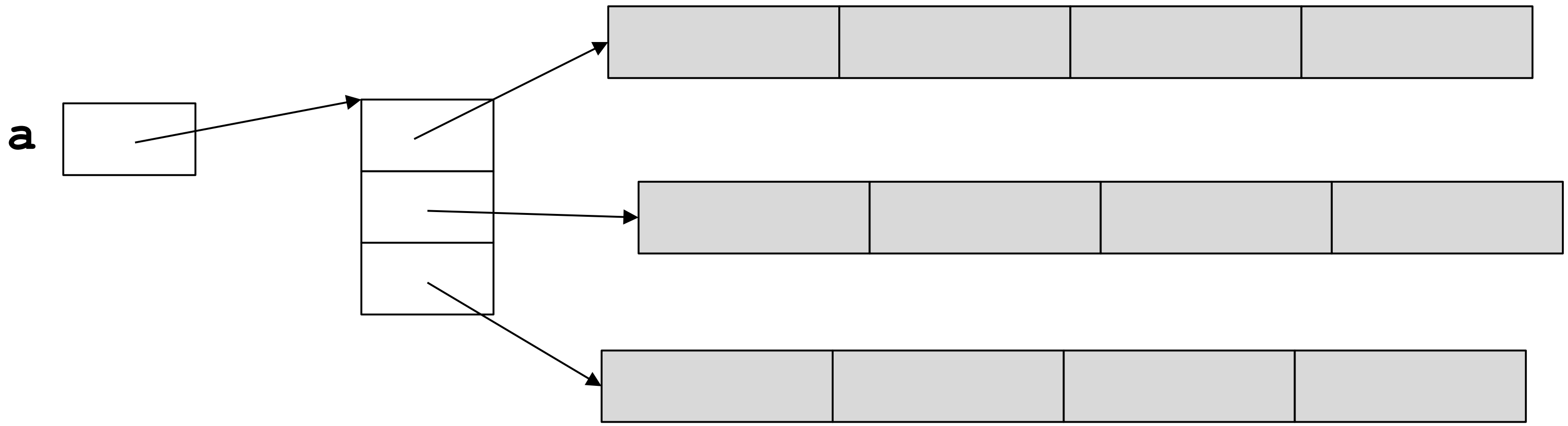
???

```
for (int j=0; j<M; j++)
    for (int i=0; i<N; i++)
        a[i][j] = i*N+M;
```

Arrays

Multdimensional Arrays – Memory Layout

```
final int N=3, M=4;
double[][] a = new double[N][M];
```



N = 100000, M = 1000;

```
for (int i=0; i<N; i++)
    for (int j=0; j<M; j++)
        a[i][j] = i*N+M;
```

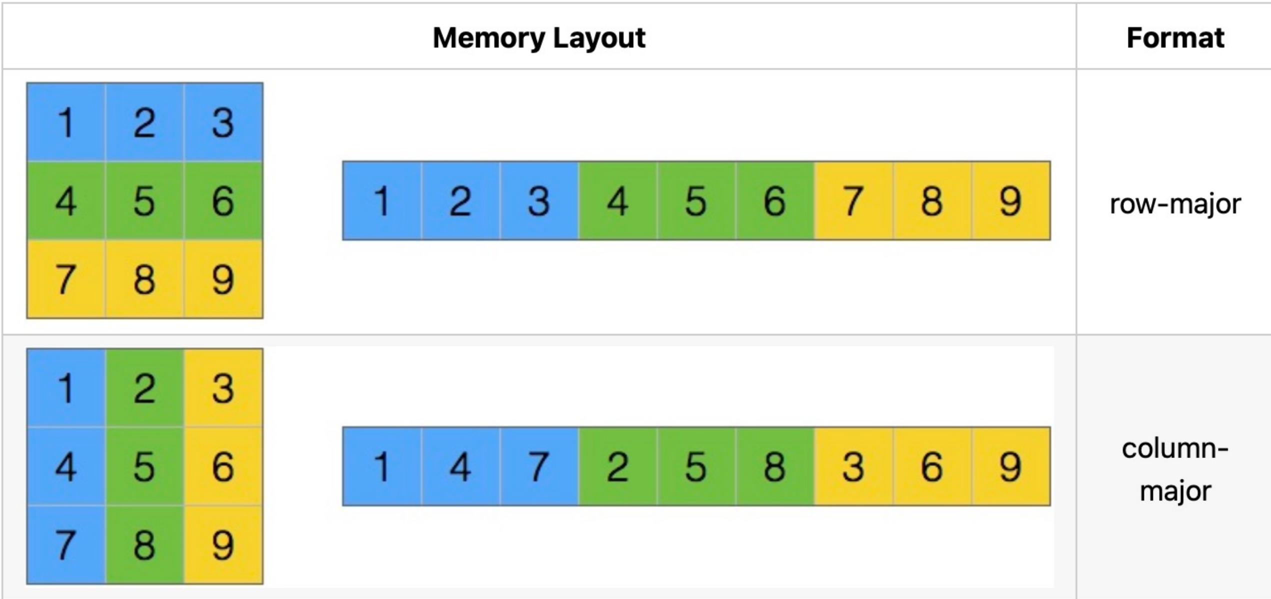
Runtime: 0,08 secs

N = 100000, M = 1000;

```
for (int j=0; j<M; j++)
    for (int i=0; i<N; i++)
        a[i][j] = i*N+M;
```

Runtime: 2,1 secs (26X)

Arrays – Other Languages



- **C/C++**
  - Multidimensional arrays are stored row-major order (contiguously).
- **Fortran**
  - Multidimensional arrays are stored column-major order (contiguously).
- **Python/NumPy**
  - Storage order may be defined by user.

Strings

Class **`String`**

- for unmodifiable (constant) strings
- Some methods: **`length()`**, **`equals()`**, **`indexOf()`**, **`substring()`**, **`toLowerCase()`**, ...

Class **`StringBuffer`**

- for unmodifiable (constant) strings, that can be changed at runtime
- Some methods: **`insert()`**, **`setChar()`**, **`append()`**, **`reverse()`**, ...

Strings

Declaration

```
String       s  = "abcd";                    // constant
StringBuffer sb = new StringBuffer(s);       // modifiable
```

Comparison

```
if (s.equals("abcd")) ...                  // compare
if (s.equals(sb))  ...                     // doesn't work
if (s.equals(sb.toString())) ...           // sb -> String
if (s.startsWith("ab")) ...                // compare prefix
```

Concatenation

```
s  = s + i + 1 + "xy";                      // concatenate
sb = sb.append("efgh"+i);                   // append
```

In expressions like e.g., **`s+3`**, numbers are implicitly converted to strings.

Strings

String conversion – primitive data types

```
// String -> int
int j = new Integer("123").intValue();
int i = Integer.parseInt("4711");

// int -> String
int drei = 3;
String s = String.valueOf(0815);
String s = Integer.toString(drei);

// String -> float
float f = new Float("0.1").floatValue();

// float -> String
String s = String.valueOf(f);
String s = Float.toString(f);
```

Strings

String conversion – objects

- If an object is part of a string conversion, the **`toString()`** method will be called.

```
System.out.println(new Date());
```

- **toString()** is inherited from Object and can be overridden.

```
class Circle {
    ...
    public String toString() {
        return new String("I am a Circle");
    }
}

Circle c = new Circle();
System.out.println(c);
```

Strings

Example: Reverse a String

```
String s = "was it a car or a cat I saw";
String reverse = newStringBuffer(s).reverse().toString();
```

String vs. StringBuffer

```
final int N = 1000000;
String sb="";

for (int i=0; i<N; i++)
      sb.append("x");
```

???

```
final int N = 1000000;
String s="";

for (int i=0; i<N; i++)
      s = s + "x";
```

Runtime: 0,035 secs

Runtime: 55 secs (1571X).

Class Organization - Summary

Class Body

Class declaration

Variables

Constructor

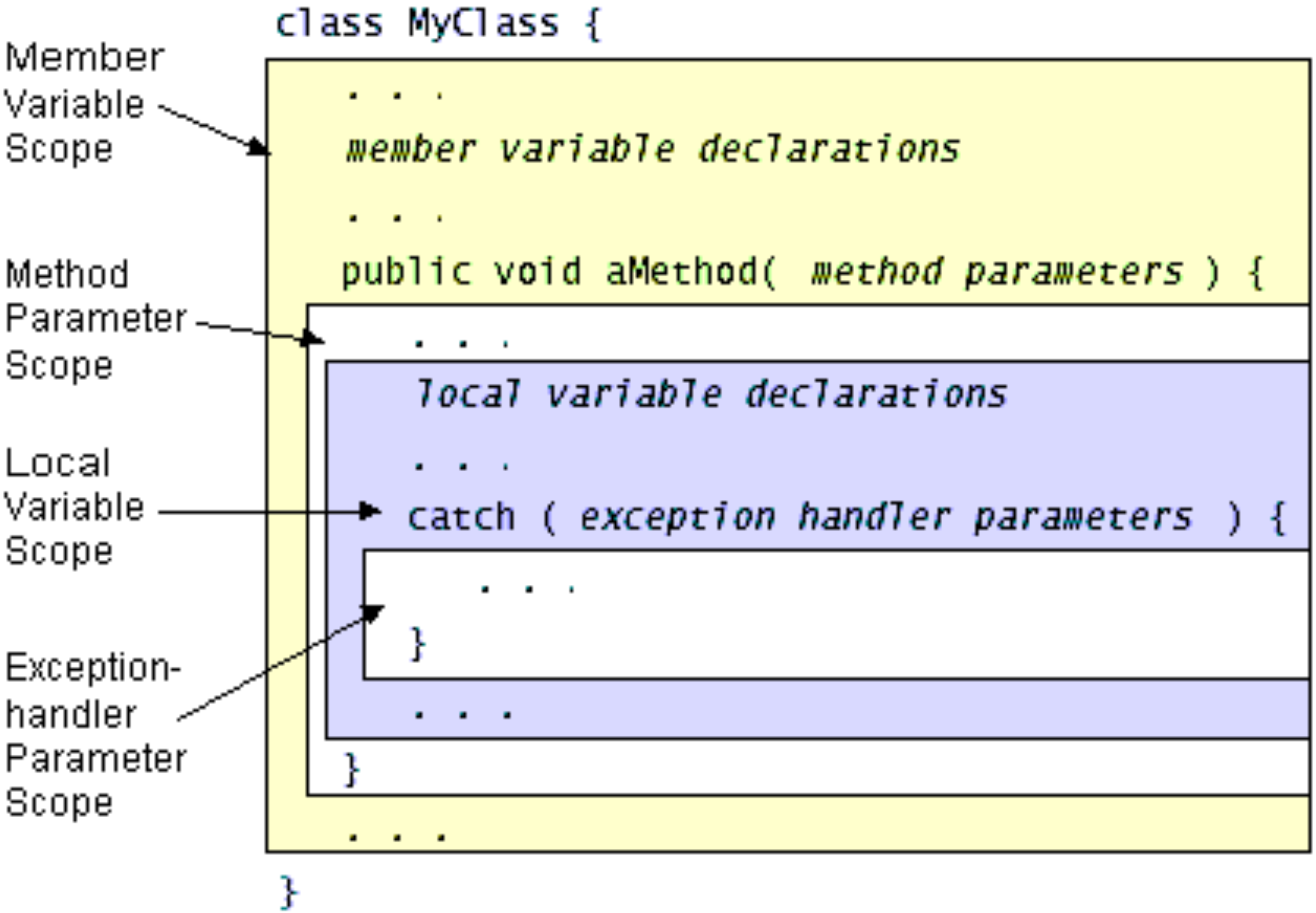
Methods

```
public class Person {
    private String name;
    private String svnr;

    public Person(String name, String svnr) {
        this.svnr = svnr;
        this.name = name;
    }

    public String getSvnr() {
        return svnr;
    }
    public String getName() {
        return name;
    }
}
```

Classes - Scopes



Immutable Objects

- An object is considered immutable if its state cannot change after it is constructed.
- Maximum reliance on immutable objects is widely accepted as a sound strategy for creating simple, reliable code.
- Immutable objects are particularly useful in concurrent applications. Since they cannot change state, they cannot be corrupted by thread interference or observed in an inconsistent state.

See: https://docs.oracle.com/javase/tutorial/essential/concurrency/immutable.html

Local Variable Type Inference

- Local variables with non-null initializers, enhanced for-loop indexes, and index variables declared in traditional for loops, may be declared with **`var`**.
- The type of local variables declared with var is inferred automatically by the compiler.

```
var url = new URL("http://www.oracle.com/");
var conn = url.openConnection();
var reader = new BufferedReader(
    new InputStreamReader(conn.getInputStream()));
```

```
URL url = new URL("http://www.oracle.com/");
URLConnection conn = url.openConnection();
Reader reader = new BufferedReader(
    new InputStreamReader(conn.getInputStream()));
```

Strategy for Defining Immutable Objects

- Don't provide "setter" methods — methods that modify fields or objects referred to by fields.

- Make all fields final and private.

- Don't allow subclasses to override methods. The simplest way to do this is to declare the class as final. A more sophisticated approach is to make the constructor private and construct instances in factory methods.

- If the instance fields include references to mutable objects, don't allow those objects to be changed:
  - Don't provide methods that modify the mutable objects.
  - Don't share references to the mutable objects. Never store references to external, mutable objects passed to the constructor; if necessary, create copies, and store references to the copies. Similarly, create copies of your internal mutable objects when necessary to avoid returning the originals in your methods

See: https://docs.oracle.com/javase/tutorial/essential/concurrency/immutable.html

Records

- Records are classes that act as transparent carriers for immutable data.
- The header of a record describes its state (the types and names of its fields).
- As objects, instances of records also have an identity.
- As opposed to classes, records do not support decoupling the internal implementation from its API.
- As opposed to classes, records cannot be extended.

```
record Rectangle(double length, double width) { }
...
record r = new Rectangle(4,5);
```

See: https://openjdk.org/jeps/384

Records

- A record's fields are implicitly final because a record serves as a simple data carrier.
- The API for records is derived automatically including methods for construction, member access, equality, and display.

```
record Rectangle(double length, double width) { }

record r = new Rectangle(4,5);
System.out.println(r);
System.out.println("area: " + r.length() * r.width() );
```

```
Rectangle[length=4.0, width=5.0]
area: 20.0
```

Records

The previous record declaration is equivalent to the following class.

```
public final class Rectangle {
    private final double length;
    private final double width;

    public Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }

    double length() { return this.length; }
    double width()  { return this.width; }

    // Implementation of equals() and hashCode(), which specify
    // that two record objects are equal if they
    // are of the same type and contain equal field values.
    public boolean equals...
    public int hashCode...

    // An implementation of toString() that returns a string
    // representation of all the record class's fields,
    // including their names.
    public String toString() {...}
}
```