Technische Hochschule Rosenheim
Technical University of Applied Sciences

Programming Basics – WiSe21/22

Classes

Prof. Dr. Silke Lechner-Greite

Table of contents – planned topics

Technische Hochschule Rosenheim

1. Introduction
2. Fundamental language concepts
3. Control structures
4. Methods
5. Arrays
6. Object orientation
7. Classes
8. Packages
9. Characters and Strings
10. Unit Testing
11. Exceptions
12. I/O

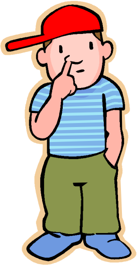
Chapter 7: Classes

5.1 Definition of terms and characteristic features of classes

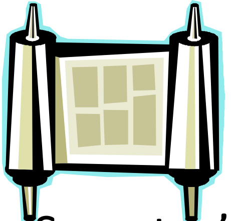
5.2 Programming classes in Java

Similar objects

- Some objects are somehow similar
- Can be grouped together



Peter



Socrates' scroll

Peter's bike

Tina

Luther's book

Lara

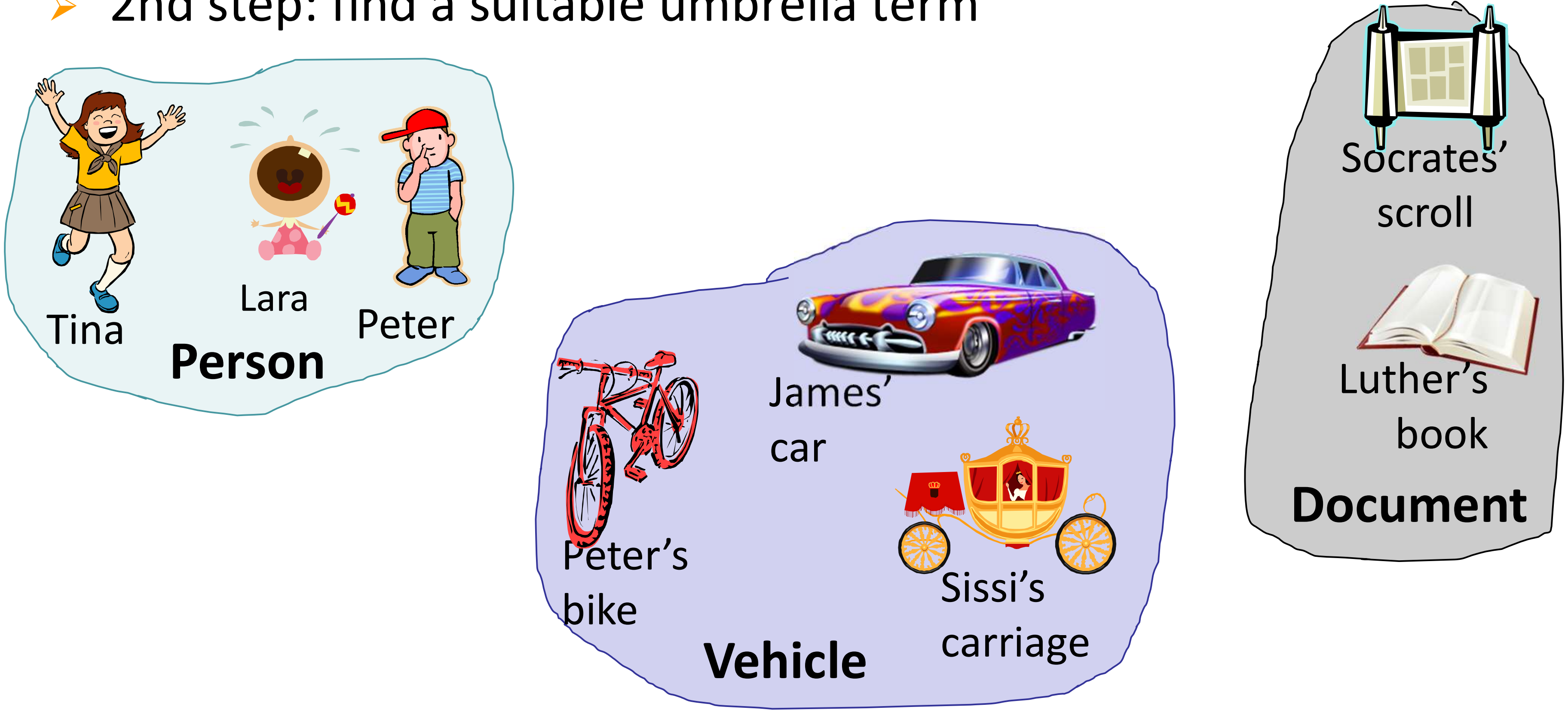
Sissi's carriage

James' car

What groupings do you recognise?

Grouping similar objects

- 1st step: group similar objects together
- 2nd step: find a suitable umbrella term



Tina

Lara

Peter

Person

Peter's bike

James' car

Sissi's carriage

Vehicle

Socrates' scroll

Luther's book

Document

Why objects? – Classes in Java, the better type

- Types in Java are declared (defined) by class.
  - In addition to primitive data types, classes also contain methods and can be inherited.

- Advantages:
  - Consistency is ensured by exclusively using the methods and restricting access to primitive types of a class.
  - Redundant programme code can be drastically reduced by skilful inheritance.

Classes and objects

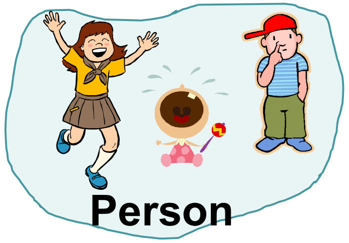
A ***class*** is a general description of things that can occur in different forms/versions, but all have a ***common structure*** and ***common behaviour***. It is a ***blueprint*** for the creation of individual specific versions. These versions are referred to as ***objects or instances*** of the class.

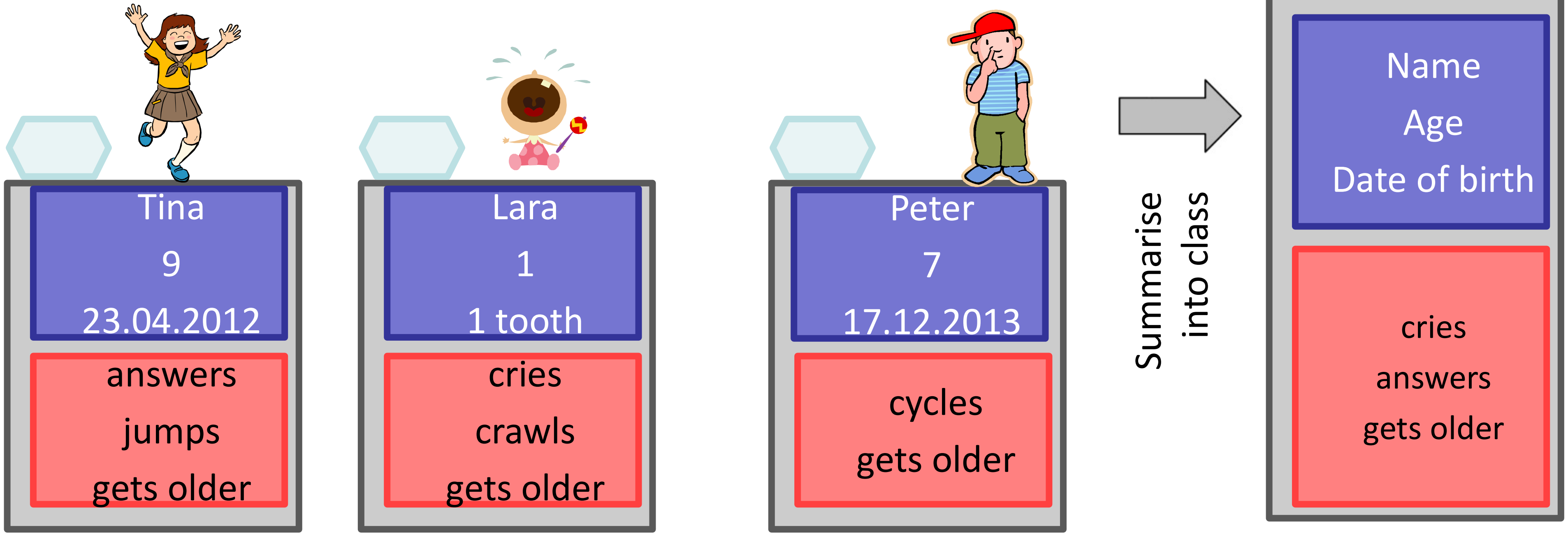
Source: D. Abts, *Grundkurs Java*, 9th edition, Springer

More examples from everyday life of classes and objects?

What is a class?

- Class = description of a set of objects with common attributes and behaviours
  - Bundles similar objects in a schema
  - Summarises relevant properties
  - Defines possible behaviour





Tina
9
23.04.2012
answers
jumps
gets older

Lara
1
1 tooth
cries
crawls
gets older

Peter
7
17.12.2013
cycles
gets older

Summarise into class

Person
Name
Age
Date of birth
cries
answers
gets older

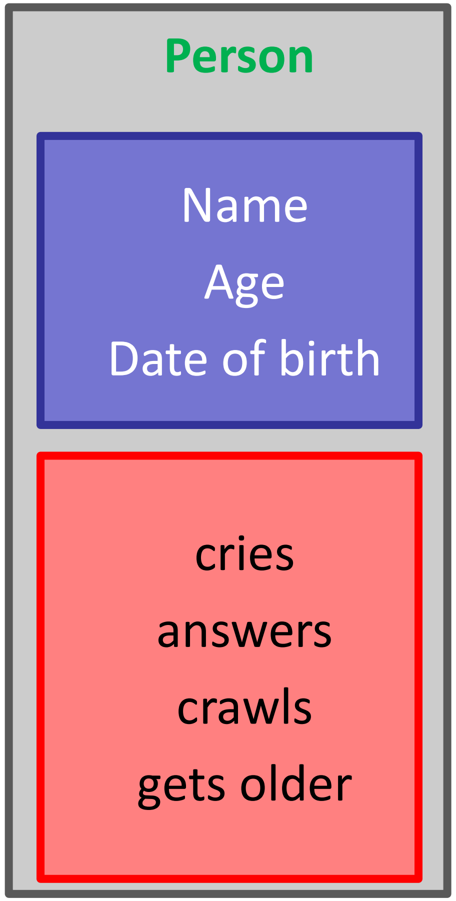
Characteristics of a class - schema

Concept **Class** bundling, abstraction **Real world**

Class name -
uniquely names the class

State -
describes an object of the class with relevant values (properties and relationships)

Behaviour -
Operations that an object in the class can perform



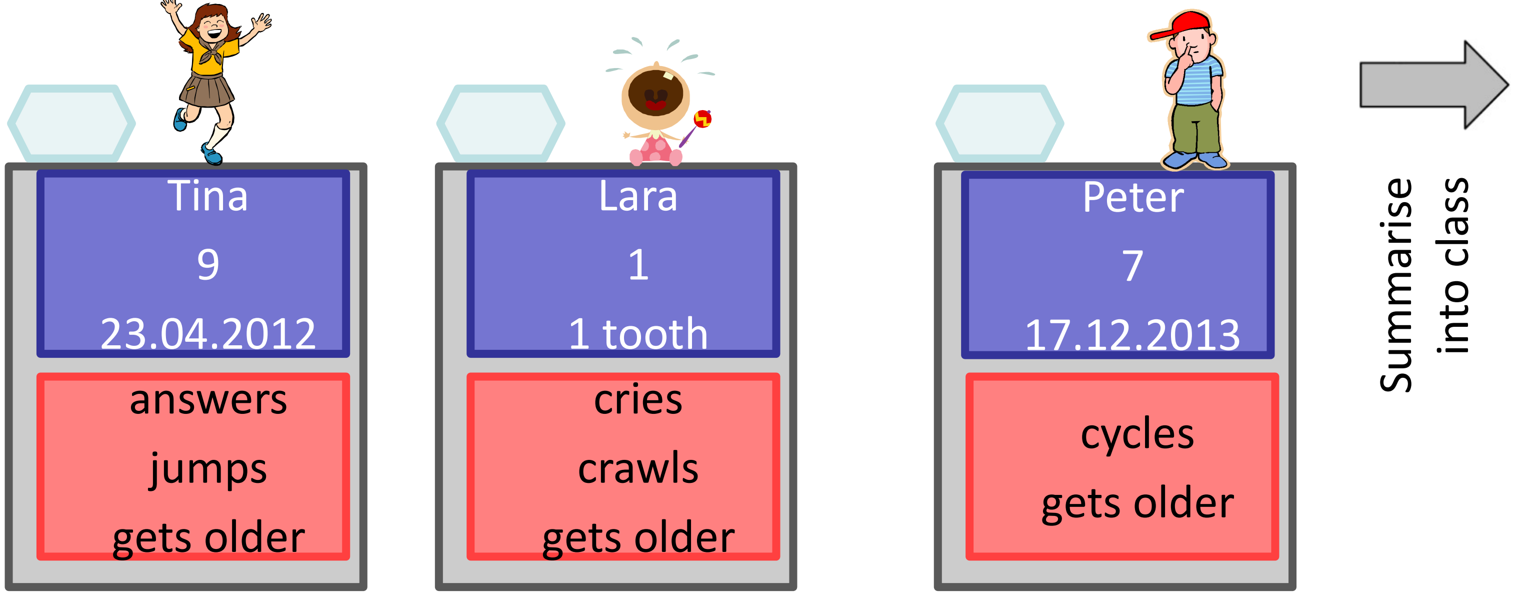
Person

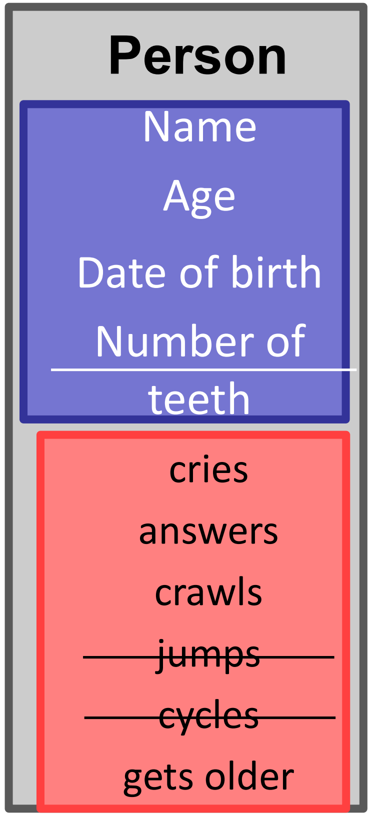
Name
Age
Date of birth

cries
answers
crawls
gets older

Procedure for forming classes

- Hybrid approach
  - Form set union (intersection)
  - Then thin out selectively
    - Properties / behaviour possible anywhere?
    - Properties / behaviour relevant everywhere?





Relationship between class and object (1)

Classes and objects in software systems

- At development time
  - Software system consists of a family of classes
  - Class is named programme unit/module
  - Class structure of a software system is static

Class =
static programme element

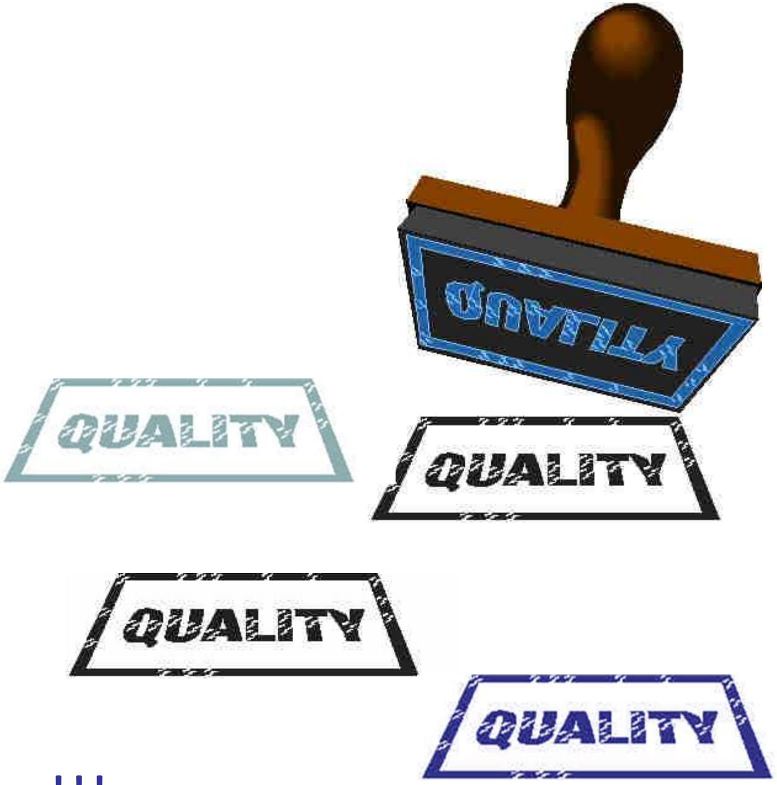
- At runtime
  - Objects perform the behaviour of the software system during runtime
  - Objects are derived from classes as needed
  - Object encapsulates data
  - Access to this data via methods of the associated class
  - Objects determine the dynamics of the programme sequence

Objects =
dynamic programme elements

Relationship between class and object (2)

- Class is pattern / template / blueprint

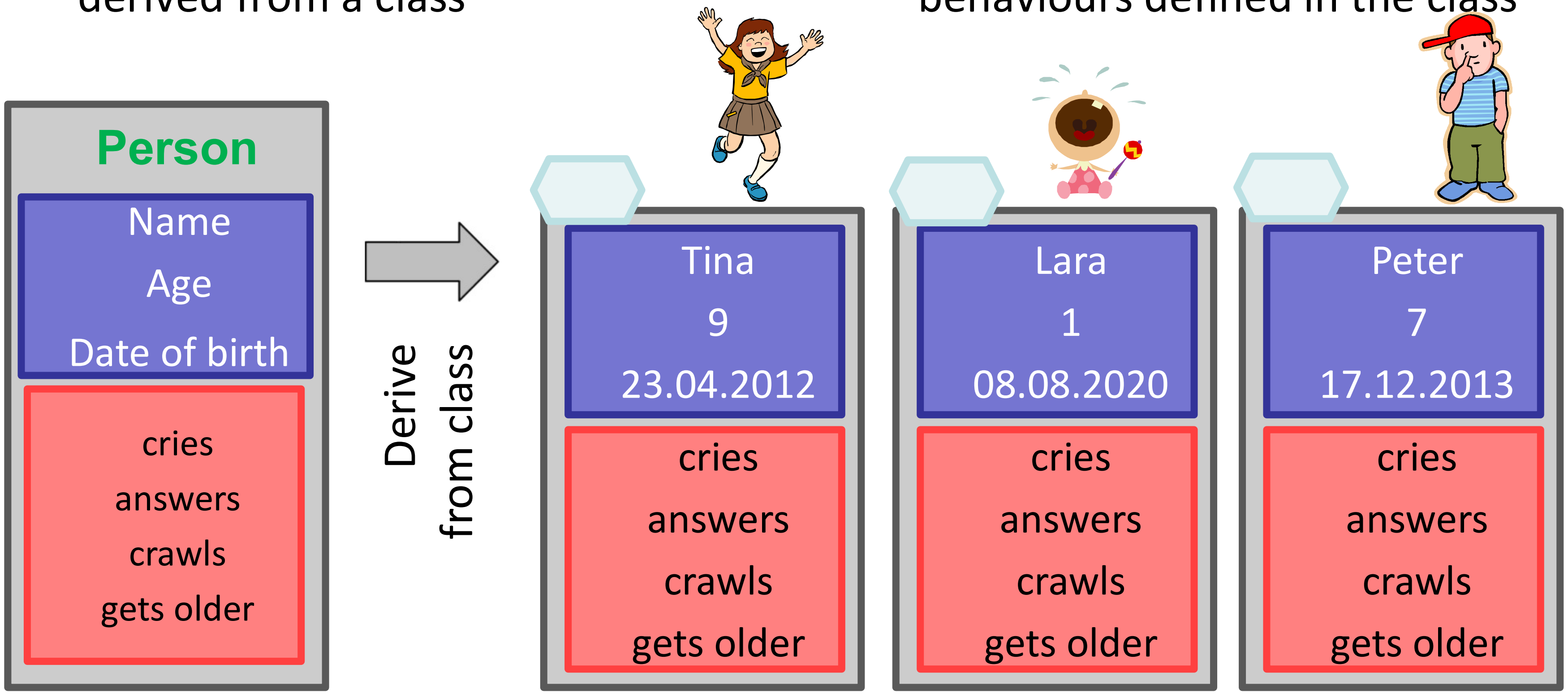
- Object is the stamped image



Object does not change its class at runtime!!!

Relationship between class and object (3)

- Class == abstraction
  - Pattern / blueprint for objects
  - Any number of objects can be derived from a class
- Object == concretisation
  - Derived from class (instance)
  - Has exactly the properties / behaviours defined in the class



Person

Name

Age

Date of birth

cries

answers

crawls

gets older

Derive from class

Tina

9

23.04.2012

cries

answers

crawls

gets older

Lara

1

08.08.2020

cries

answers

crawls

gets older

Peter

7

17.12.2013

cries

answers

crawls

gets older

How is this drawn in a standardised manner?

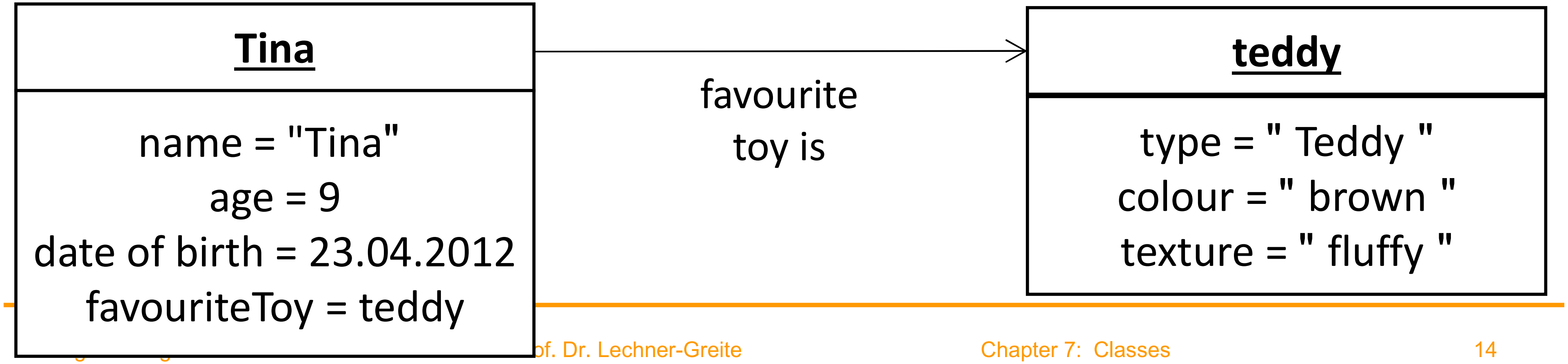
- Reminder: UML object diagram from the last chapter

Tina

favourite

toy is

Teddy



Tina

name = "Tina"
age = 9
date of birth = 23.04.2012
favouriteToy = teddy

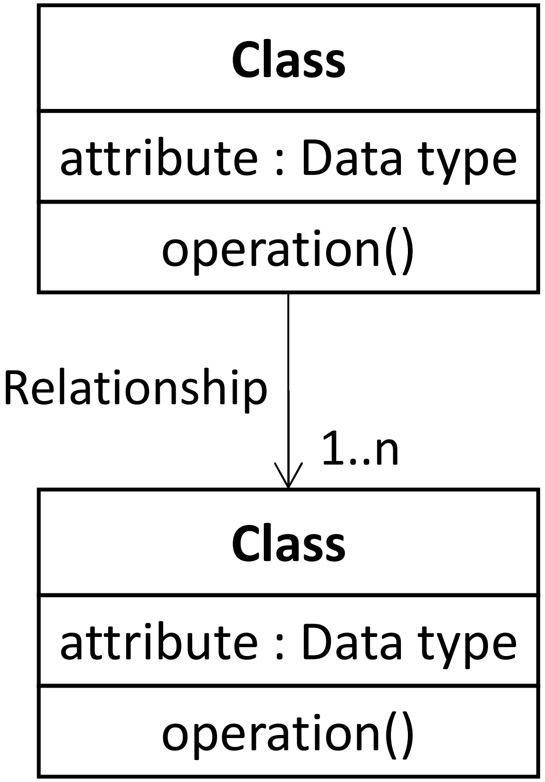
favourite
toy is

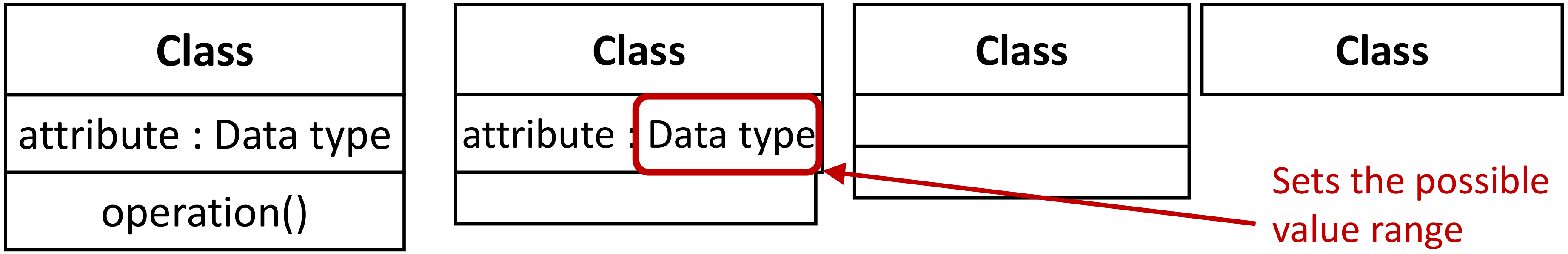
teddy

type = " Teddy "
colour = " brown "
texture = " fluffy "

Class diagram

- Representation of classes in UML
  - Rectangle with three areas
    - Class name
    - Attributes
    - Operations
  - Relationships to other classes possible
    - Multiplicity specifies the number of combinable objects
- Variants: leave individual areas out

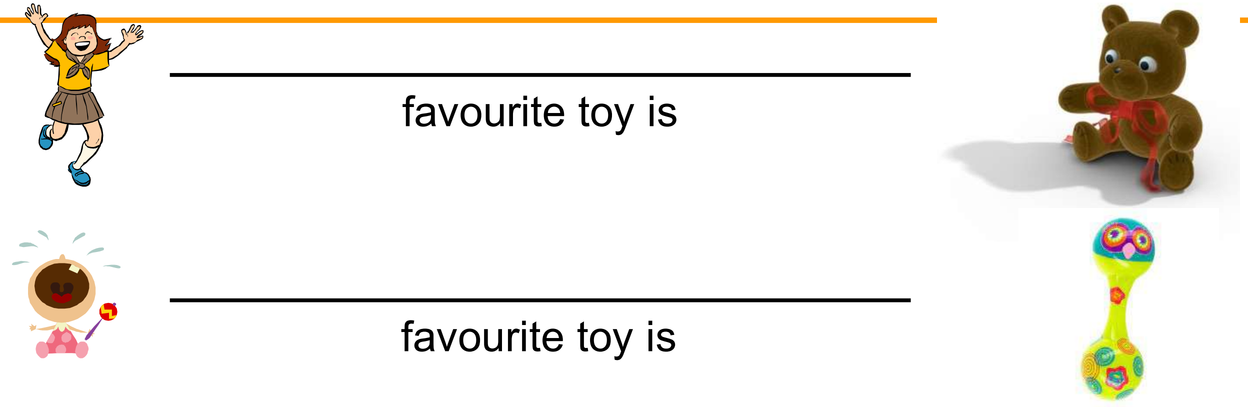




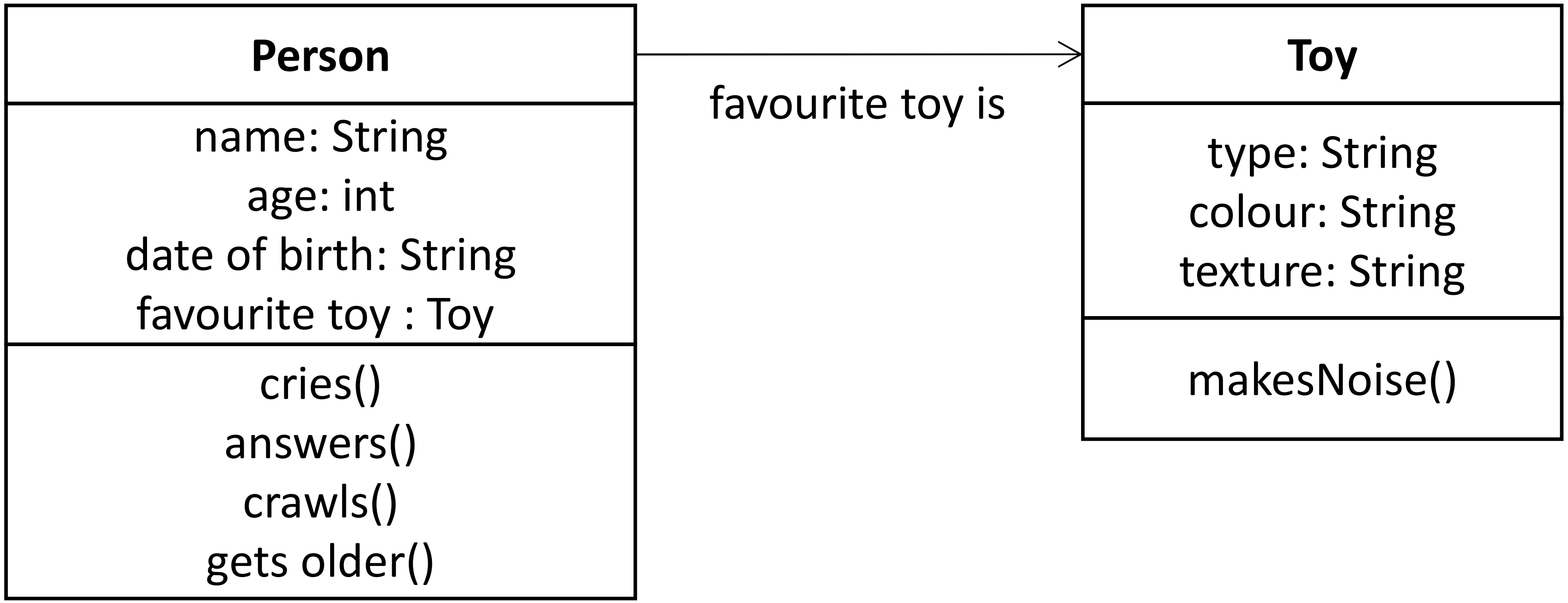
Sets the possible value range

Class diagram - example

Objects



Classes



Person

name: String
age: int
date of birth: String
favourite toy : Toy

cries()
answers()
crawls()
gets older()

favourite toy is

Toy

type: String
colour: String
texture: String

makesNoise()

Standardised representation in UML

- Class is abstraction / blueprint
- Object is concretisation

Person

name : String
age : int
date of birth : String
favourite toy : Toy

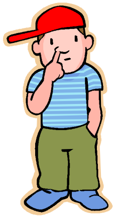
cries()
answers()
crawls()
gets older()

tina : Person

name = " Tina "
age = 9
date of birth = " 23.04.2012 “
favourite toy = teddy

lara : Person

name = " Lara "
age = 1
date of birth = " 08.08.2020 “
favourite toy = rattle

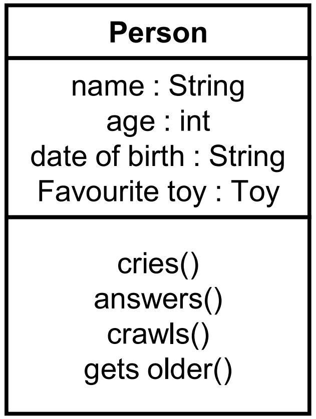


peter : Person

name = " Peter "
age = 7
date of birth = " 17.12.2013 “
favourite toy = ball

Important similarities / differences

- Class diagram
  - Identifier



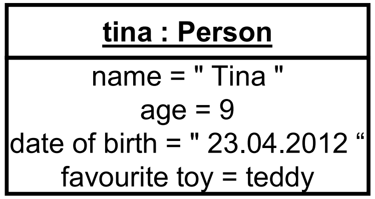
Person

name : String
age : int
date of birth : String
Favourite toy : Toy

cries()
answers()
crawls()
gets older()

  - Attributes
    - Specify data types
  - Behaviour
    - Only represented for classes

- Object diagram
  - Identifier
    - Underlined
    - Usually refers to class names



tina : Person

name = " Tina "
age = 9
date of birth = " 23.04.2012 “
favourite toy = teddy

  - Attributes
    - Set specific values
  - Behaviour
    - Not explicitly specified
    - Results from the corresponding class!

No relationship arrows between class and derived objects!

Chapter 7: Classes

5.1 Definition of terms and characteristic features of classes

5.2 Programming classes in Java

Declaration of classes (1)

- Classes define new data types
  - `int`, `double`, `boolean` are predefined types
  - Fractions, for example, are not predefined ($\frac{1}{2}, \frac{3}{4}, \frac{7}{13}, \ldots$)
- Syntax:

```
class ClassName {
      ...
}
```

- Example:

```
class Rational {
      ...
}
```

Convention:

– Each class declaration in its own source code file
  => `Rational.java`

Declaration of classes (2)

- Attributes of a fraction: numerator, denominator
- List the attributes in class declaration:

```
class Rational {
  int numer;
  int denom;
   ...
}
```

Instance variable for the numerator

Instance variable for the denominator

Any number and sequence!

- Individual components = instance variable
  - Same syntax as previously used for variables (local variables => statements in methods)
  - Only different location of the declaration (=> element of a class)

Exercise – Declare Java class

- Live exercise
  - Complete Task 1 on the
    live exercises sheet “Class declaration and use”
  - You have 5 minutes.

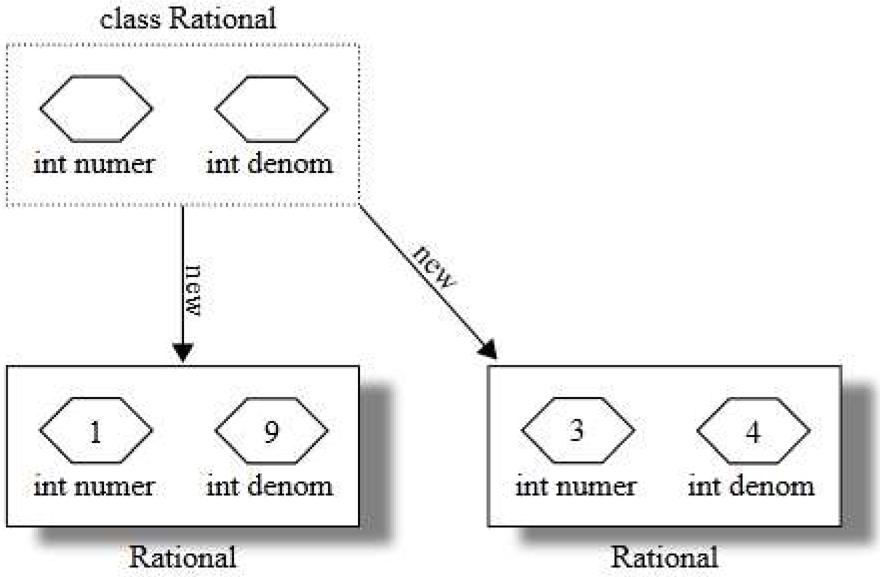
Declaration of classes (3)

- Comparison of primitive types versus non-primitive (reference) types

Primitive types	Non-primitive (reference) types
Assortment fixed; cannot be redefined	Can be redefined for specific problems
Atomic, internal structure of a value does not matter	Consists of different components (instance variables); can be addressed and processed individually
`int, double, boolean, …`	`Rational, Customer, Person, …`
	Some non-primitive (reference) types are already predefined in Java: `String` for strings or `System` for input and output data streams

Objects (1)

- Class declaration = blueprint, design specification
- Objects of the class must be explicitly created
- Example: two `Rational` objects with the values $\frac{1}{9}$ *und* $\frac{3}{4}$



1 class declaration

↓

any number of objects

Object = instance of the class

Objects (2)

- Creating a new object = instantiation
  (also "construction", "allocation")
- With operator `new`:

```
new Rational()
```

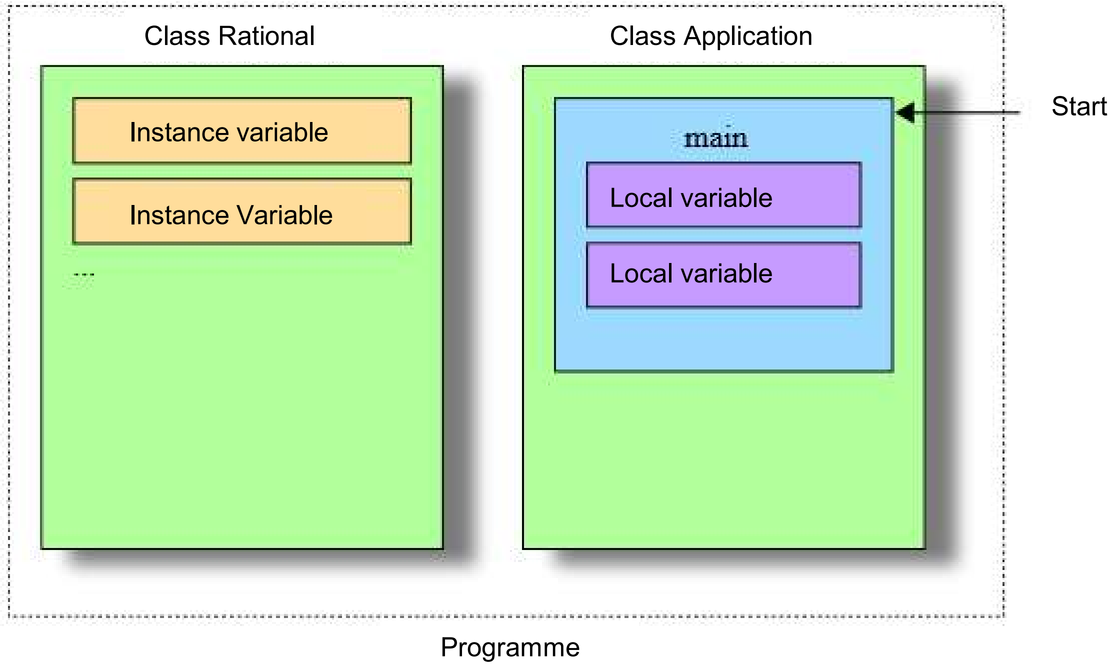
- `new` produces a single, new object of this class from a class declaration
- Multiple objects => multiple calls of `new`
- `new` calls a special method of the class (constructor - details later)

Objects (3)

- Java programme with classes consists of:
  - Declaration of the class
  - Use of the class

(usually) in different source code files

- Structure:



Objects (4)

- Example:

```
class Rational {
  int numer;
  int denom;
   ...
}
```

```
class Application {
  public static void main (String[] args){
    … new Rational() …
  }
}
```

In Java, there is no code outside of classes!
However, the class `Application` merely serves as a "container" for the main-method.
It mimics a use case of how the application reacts dynamically.

Non-primitive (reference) variables (1)

- == variables of non-primitive (reference) types
- Declaration of non-primitive (reference) variables is analogous to variable declaration of primitive data type (primitive variables)
- Example:

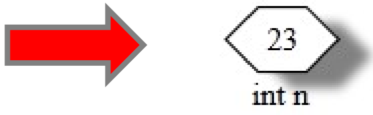
```
Rational r;
```

Declaration of a variable `r` of the non-primitive (reference) type `Rational`

Non-primitive (reference) variables (2)

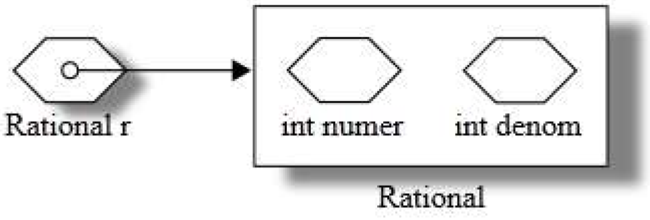
- Value assignment: fundamental difference between primitive variables and non-primitive (reference) variables
  - Primitive variable

```
int n = 23;
```



Variable and memory space for value coincide

  - Non-primitive (reference) variable
    - Variable and value (= object) exist independently and separately

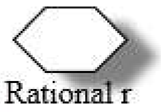


Non-primitive (reference) variables (3)

- Individual steps when initialising a non-primitive (reference) variable

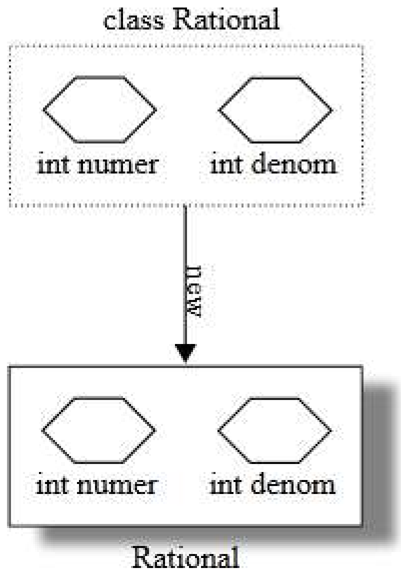
1. Declare non-primitive (reference) variable

```
Rational r;
```



2. Allocate new object in memory with `new`

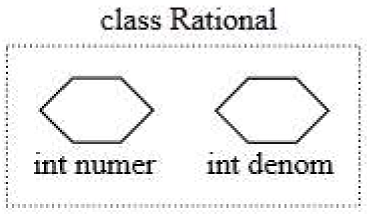
```
new Rational()
```

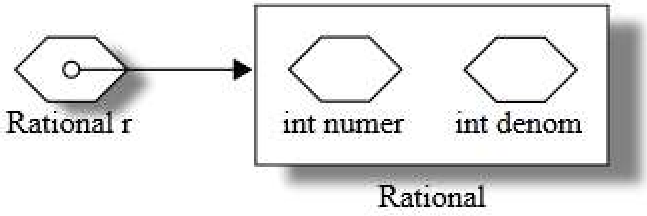


Non-primitive (reference) variables (4)

3. Assign object to reference variable

```
r = new Rational();
```





Variable references object

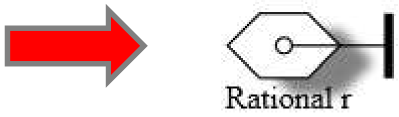
- All steps together: initialise a non-primitive (reference) variable during declaration

```
Rational r = new Rational();
```

Non-primitive (reference) variables (5)

- `null` non-primitive (reference)
  - `null` stands for no object
  - `null` can be assigned to any non-primitive (reference) variable

```
Rational r = null;
```



  - `null` is a well-defined value, can be compared

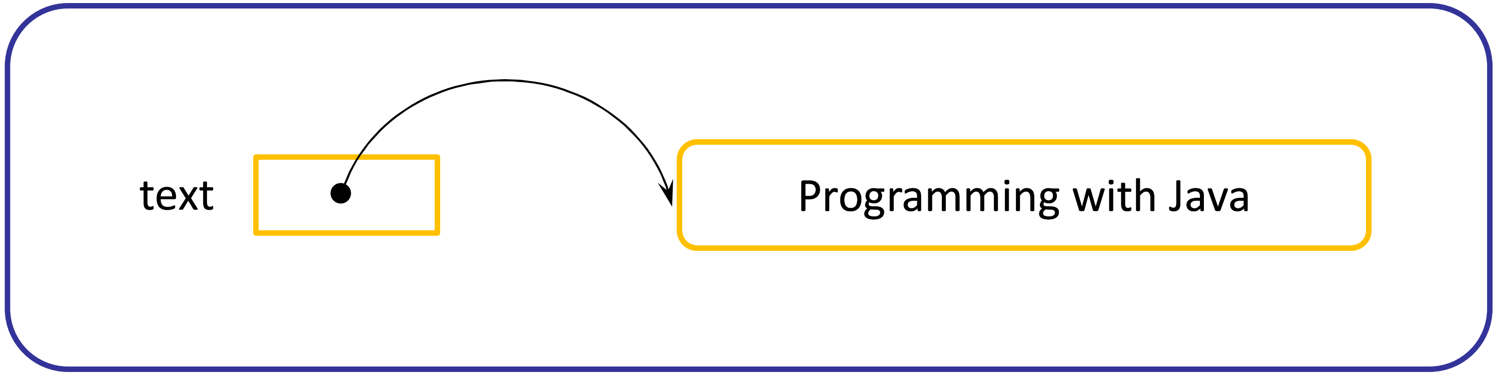
```
if (r == null)
   System.out.println("no object");
```

Always assign `null` to a non-primitive (reference) variable if there is no object!

Example `String (1)`

- Create and reference to the String object:

```
String text;
text = new String("Programming with Java");
```



- Short form (and preferred version):

```
String text;
text ="Programming with Java";
```

Technische Hochschule Rosenheim

Example `String`

- Comparison of strings (test for equal contents)
  - `boolean equals(Object anObject)`
  - Compares character-by-character and returns `true` if equal

```
String       s1 = "Hello";
String       s3 = new String("Hello");
boolean b3 = s3.equals(s1); // b3 is true
```

⚠ Operator `==` checks the identity of `String` objects, not the contents!

Instance variables (1)

- Access
  - Instance variables are declared outside a method definition.
  - Each class contains the instance variables specified in the class declaration
  - Instance variables of an object can be addressed individually: element access
  - Object to which element access is directed: target object
  - Syntax:

```
targetObject.objectVariable
```

Instance variables (2)

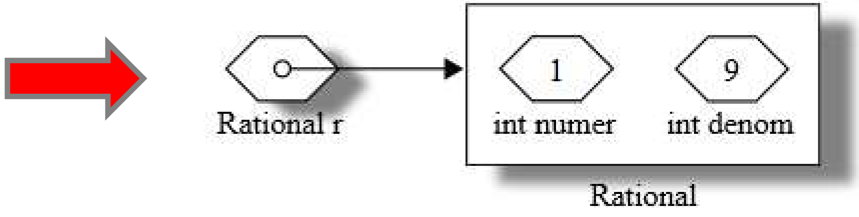
- Example: create a `Rational` object with value $\frac{1}{9}$

1. Create `Rational` object and assign to variable

```
Rational r = new Rational();
```

2. Assign the numerator and denominator of the target object `r` individually

```
r.numer = 1;
r.denom = 9;
```



Instance variables (3)

- Handling
  - Only access qualifiers show the difference between instance variables and local variables
  - Same use as local variables
  - Examples:

```
int i = 10 – r.numer * 5;
```

```
r.numer = r.numer + 8;
```

```
r.numer++;
```

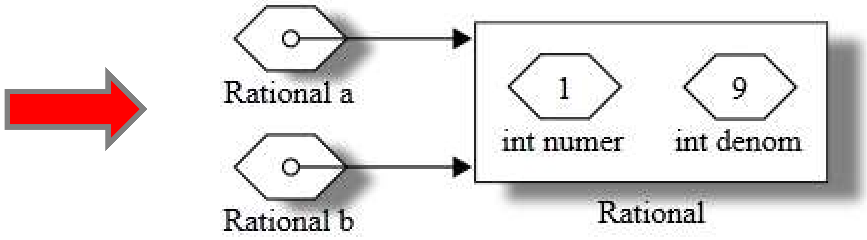
```
if (r.denom != 0) …
```

```
...
```

Instance variables (4)

- Value assignments for non-primitive (reference) types
  - Reference is duplicated, not the object!

```
Rational a = new Rational();
a.numer = 1;
a.denom = 9;
Rational b = a;
```



Both variables `a` and `b` reference the same object with value $\frac{1}{9}$

```
...
b.numer++;
System.out.println(a.numer);   //outputs 2!
```

Changes to an object are visible in both variables

Instance variables (5)

- Comparison of non-primitive (reference) types
  - Comparison with `==` checks the identity and ignores the content
    - `true` if both operands are one and the same object
    - `false` if the operands are different objects

```
Rational a = new Rational();
a.numer = 1;
a.denom = 9;
Rational b = new Rational();
b.numer = 1;
b.denom = 9;

if (a == b)    //false
   ...
```

```
Rational a = new Rational();
a.numer = 1;
a.denom = 9;
Rational b = a;

if (a == b)    //true
   ...
```

  - Content comparison of objects: compare instance variables in pairs

```
if (a.numer == b.numer && a.denom == b.denom)
   ...
```

Instance variables (6)

- Validity range (scope)
  - In the entire own class

- Lifetime
  - Coincides with the object they are part of
  - Are created as soon as an object is created
  - Are released when the object is no longer accessible

Exercise – Programme Java class

- Live exercise
  - Complete Task 2 on the live exercises sheet “Class declaration and use”
  - You have 5 minutes.

Methods (1)

- Describe the behaviour of objects
- Correspond to processes => named with (English) verbs

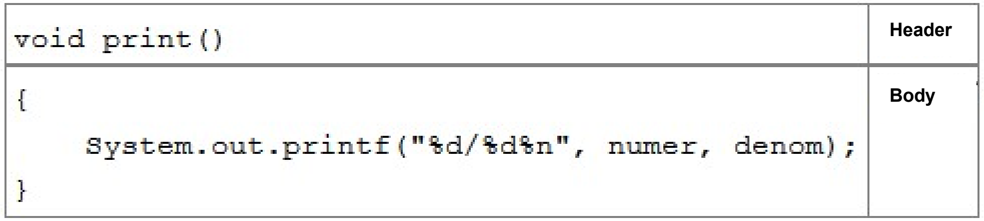
- Example:

```
class Rational {
  int numer;
  int denom;

  void print() {
    System.out.printf("%d/%d\n", numer, denom);
   }
}
```

Methods (2)

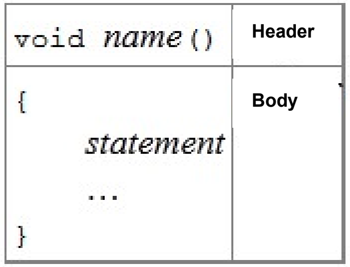
- Signature and body
  - Method declaration = (method) header + (method) body



`void print()`	**Header**

`{`
`    System.out.printf("%d/%d%n", numer, denom);`
`}`
	Body

  - General



void *name*()	**Header**

{
    statement
    ...
}
	Body

  - Note: Brackets in the body are mandatory, even with one (or without any) statement!

Methods (3)

Calling methods

- A method is called with target object
- Syntax similar to element access:

```
targetObject.methodName();
```

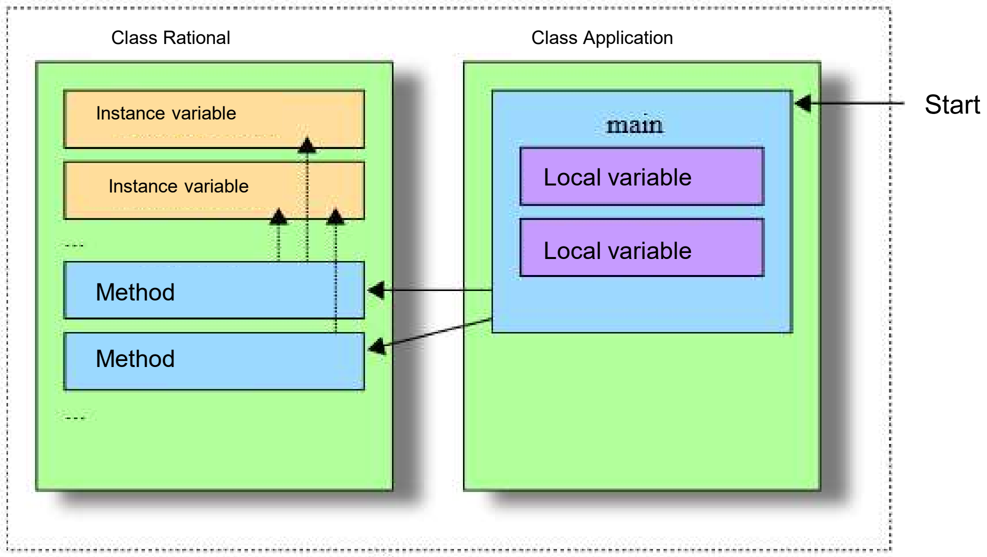
Round brackets mark method call

- Example:

```
Rational r = new Rational();
r.numer = 1;
r.denom = 8;
r.print();             //Method call outputs 1/8
```

Methods (4)

- Structure:
  - Programme consists of multiple classes,
  - Class consists of instance variables and methods

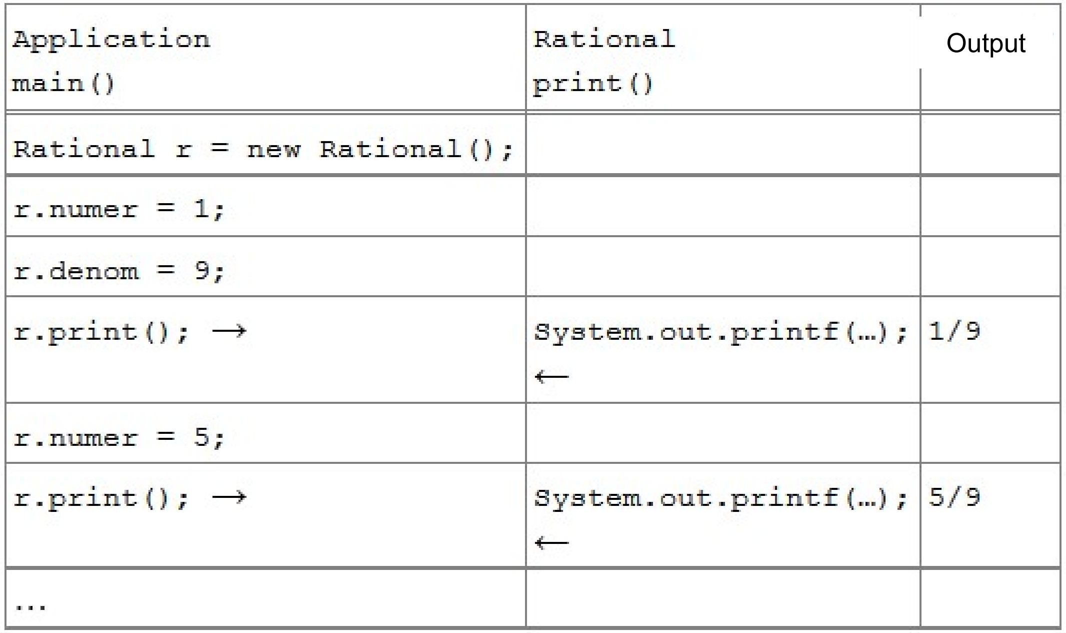


Programme

- Method declarations only allowed in classes,
  - not outside of a class declaration
  - not within another method declaration
- Any number, sequence and arrangement of method declarations in a class
- The application initializes objects from the classes and accesses them i.e. through method calls.

Methods (5)

- Sequence of a method call in several individual steps = call sequence
  1. Interrupt the calling programme ("caller")
  2. Run through the method body
  3. Continue the caller after the call
- Example sequence:

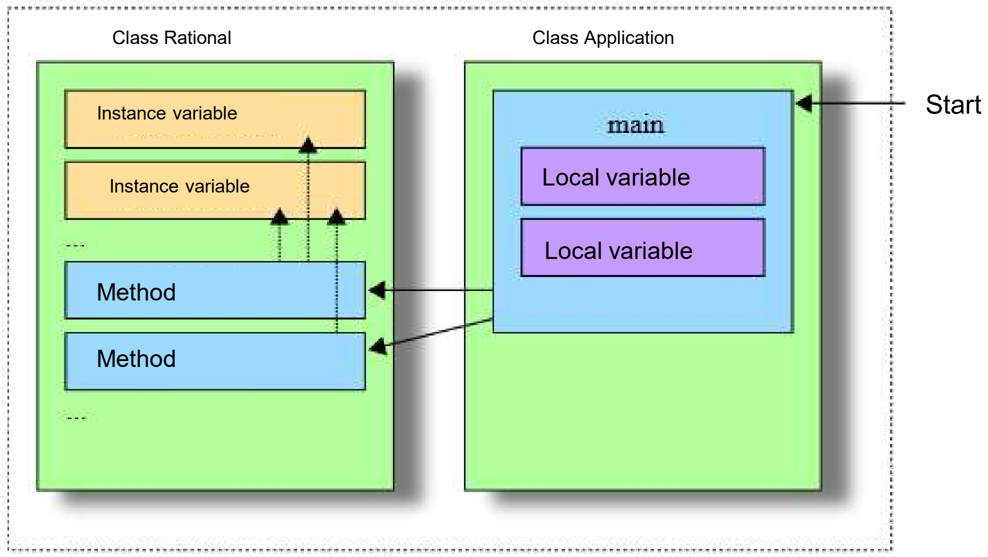


Application main()	Rational print()	Output
`Rational r = new Rational();`		
`r.numer = 1;`		
`r.denom = 9;`		
`r.print();` →	`System.out.printf(…);` ←	1/9
`r.numer = 5;`		
`r.print();` →	`System.out.printf(…);` ←	5/9
...		

Caller is interrupted every time

Methods (6)

- Method body = block
  - Any statements allowed (all control structures and declarations)
  - Validity range (scope) of local declarations
  - Lifetime of local variables: one call each (created on call; released again on return)



Programme

Programme consists of multiple classes, class consists of instance variables and methods, methods contain local variables

Methods (7)

- Example:
  - Method to reduce a fraction:

```
class Rational {
  int numer;
  int denom;

  void reduce() {
    int gcd = …;
    numer = numer /gcd;
    denom = denom / gcd;
   }
}
```

  - Call of `reduce`:

```
Rational r = new Rational();
r.numer = 6;
r.denom = 9;
r.print();   //outputs 6/9
r.reduce();
r.print();   //outputs 2/3
```

Local variable `gcd` only valid in the body, exists for one call at a time!

Methods (8)

- Access from a method body
  - To own instance variables without specifying a target object => instance variables can be addressed like local variables
  - Call of methods of the own class without specifying the target object

```
class Rational {
  int numer;
  int denom;

  void print() {…}

  void reduce() {…}

  void printReduced() {
    reduce();                 //Method of the own class
    print();                  //Method of the own class
    }
}
```

Methods (9)

- Access from a method body
  - To instance variables and methods of another object: specifying the target object

```
class Application {
  public static void main (String[] args){
   Rational r =new Rational();
   r.numer = 6;
   r.denom = 9;
   r.printReduced();
  }
}
```

Methods (10)

- Naming conflicts
  - Names of local variables and instance variables do not conflict

```
class DemoName {
  int numer = 8;                        //instance variable
   …
  void output() {
    int numer = 5;                     // local variable
    System.out.println(numer);     // outputs 5
  }
}
```

Local declaration "hides" instance variable

Technically possible but bad style!

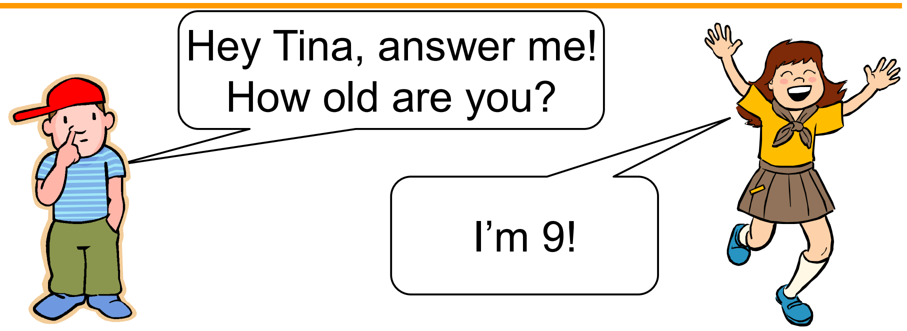
Methods (11)

- Self-reference `this`
  - Is automatically declared in every class, always available
  - Reserved keyword = self-reference to the current object
  - **Allows access to instance variables**

```
class DemoName {
  int numer = 8;                                     //instance variable
   …
  void output() {
    int numer = 5;                                 // local variable
    System.out.println(numer);                     // outputs 5
    System.out.println(this.numer);                // outputs 8
  }
}
```

Parameters (1)

- Parameters pass information from caller to methods
- Two language elements coupled:
  1. Method lists the required parameters
  2. Caller passes arguments for the required parameters



- A parameter list can be specified in the method header

```
void methodName(type1 name1, type2 name2, . . .)
```

- Any number of parameters allowed
  (previously: no parameters, empty list)

Parameters (2)

- Example:

```
class Rational {
   …
   void extend(int f) {       //header with parameter int f
      numer = numer * f;
      denom = denom * f;
   }
   …
}
```

- Caller must specify a compatible argument for each call:

```
r.extend(2);
```

When called, compiler checks if arguments and parameters match!

Parameters (3)

- Example with multiple parameters:

```
class Rational {
    …
  void set(int n, int d) {        //2 parameters
    numer = n;
    denom = d;
   }
  void setZero() {                //0 parameters
    numer = 0;
    denom = 1;
    }
}
```

- Correct calls:

```
Rational r = new Rational();
r.set(5,8);                         // r = 5/8
Rational s = new Rational();
s.setZero();                        // s = 0/1
```

Parameters (4)

- Primitive types as parameters
  - Hidden value assignment when passing parameters
  - Values of primitive types are copied
  - Implicit and explicit type conversions as for value assignments

```
Rational r = new Rational();
r.extend(5.18);                    // Error incorrect type!
r.extend((int)5.18);               // OK
```

- Local variables as parameters

```
Rational r = new Rational();
int x = 5;
r.extend(x);
```

Parameters (5)

- Non-primitive (reference) types as parameters
- Example:
  - Method `mult` expects another `Rational` object as a parameter

```
class Rational {
...
  void mult (Rational frac) {
    numer = numer * frac.numer;
    denom = denom * frac.denom;
   }
}
```

  - From the perspective of `mult`: `frac` is another reference variable
    - Addressing the own instance variables without a target object
    - Addressing the foreign instance variables with target object `frac`

Overloading (1)

- Overloading = multiple methods with the same name but different parameter lists
- Useful for related methods with a similar purpose
- Example: three methods `set` for specifying a fraction

```
class Rational {
  void set() {
    numer = 0;
    denom = 1;
   }
  void set(int n) {
    numer = n;
    denom = 1;
  }
  void set(int n, int d) {
    numer = n;
    denom = d;
    }
    . . .
}
```

Overloading with different number of parameters or different types of parameters or both

Overloading (2)

- Based on the argument list provided by the caller, the compiler decides which method is called.
-
```
r.set();          // calls set()
r.set(1);         // calls set(int)
r.set(1,2);       // calls set(int,int)
r.set(1,2,3);     // Error! No suitable method declared
```

- Overloaded methods lead toward polymorphism – another core property of object orientation.