Constructors (1)

Technische Hochschule Rosenheim

- Goal: automatically create a useful initial state for new objects
  - Example `Rational`: $\frac{0}{1}$ useful, $\frac{1}{0}$ or $\frac{0}{0}$ useless
- Idea: constructors are special methods, which are called automatically when creating objects by calling `new`
- Declaration of a constructor as with normal methods, except …
  - the same name as the class (method name = class name)
  - no result type of `void` (or other result type)
- If a constructor isn't specified, a default constructor is added by the Java Framework. All instance variables are initialized with 0 / null default values/references.

Constructors (2)

Technische Hochschule Rosenheim

- Declaration of a constructor: header, parameter list and body as with other methods
- Example:

```
class Rational {
  …
  Rational() {
    numer = 0;
    denom = 1;
   }
  …
}
```

Default constructor =
constructor with empty parameter list

- `new` automatically calls constructor with fitting parameter list:

```
Rational r = new Rational(); // calls Rational()
r.print();                   // sets numer to 0 and denom to 1
```

Constructors (3)

Technische Hochschule Rosenheim

- Basic constructor: constructor with non-empty parameter list
- ➔ multiple parameters are expected

- Example:

```
class Rational {
   …
   Rational(int n, int d) {
    numer = n;
    denom = d;
   }
   …
}
```

- In the `new` call, appropriate values must be specified

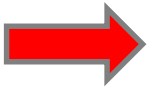
```
Rational r = new Rational(2,3);   // calls Rational(int,int)
r.print();                        // sets numer to 2 and denom to 3
```

Constructors (4)

Technische Hochschule Rosenheim

- Default values of instance variables
  - instance variables are automatically initialised with default values
  - Default values depend on the type

Type	Default value
`int`	`0`
`double`	`0.0`
`boolean`	`false`
`char`	`\u0000`
Reference type	`null`



Without a constructor, `Rational` objects start with $\frac{0}{0}$
(unusable due to 0 in the denominator)

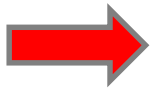
- Difference to local variables: they are not automatically initialised!

Constructors (5)

Technische Hochschule Rosenheim

- Automatically defined constructor
  - Class without explicitly defined constructor: Compiler automatically generates default constructor
  - Body is empty
  - Any explicit declaration prevents automatic declaration

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



Class has only custom constructor
and no default constructor

```
new Rational(2,3);   //OK
new Rational();   // Error
```

- Recommendation: each class always has at least one constructor

Constructors (6)

Technische Hochschule Rosenheim

Copy constructor

- Creates a copy of an already existing object
- Template (= original object) is passed as parameter `that`

```
class Rational {
  Rational(Rational that) {
    numer = that.numer;
    denom = that.denom;
   }
    ...
}
```

Parameters of the same class

- Call with object as "copy template"

```
Rational original = new Rational(2,3);
Rational copy = new Rational(original);
copy.print();                                  // outputs 2/3
```

Constructors (7)

Technische Hochschule Rosenheim

Constructor chaining

- Constructors usually have more tasks than just assigning values to instance variables, e.g. pre-testing, preprocessing of parameters, creating log outputs, etc.
- If multiple overloaded constructors are defined:
  - Danger: the same code is copied in each constructor → bad maintainability, error-prone!
  - Better: code only in one constructor, which is also used by all other constructors though concatenated calls

Constructor chaining == calling another constructor of the same class

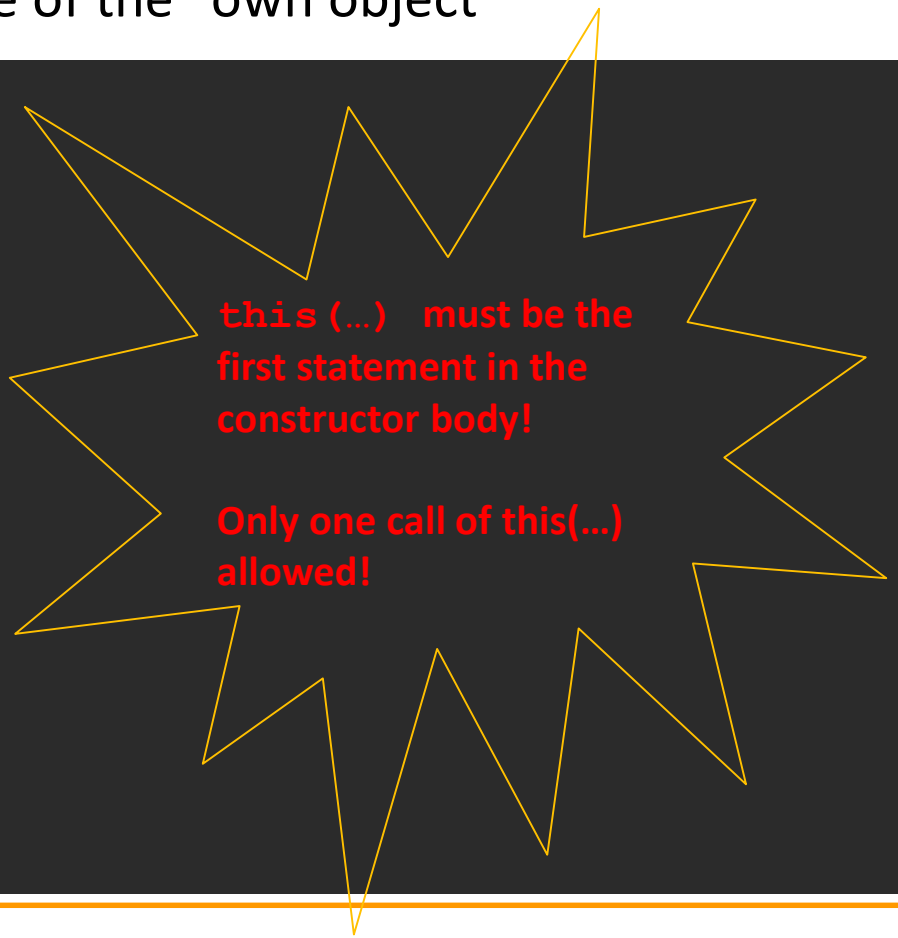
Constructors (8)

Technische Hochschule Rosenheim

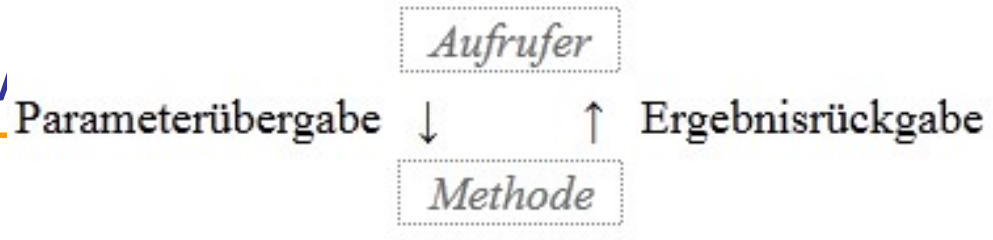
Constructor chaining

- Syntax: `this` as representative of the "own object"

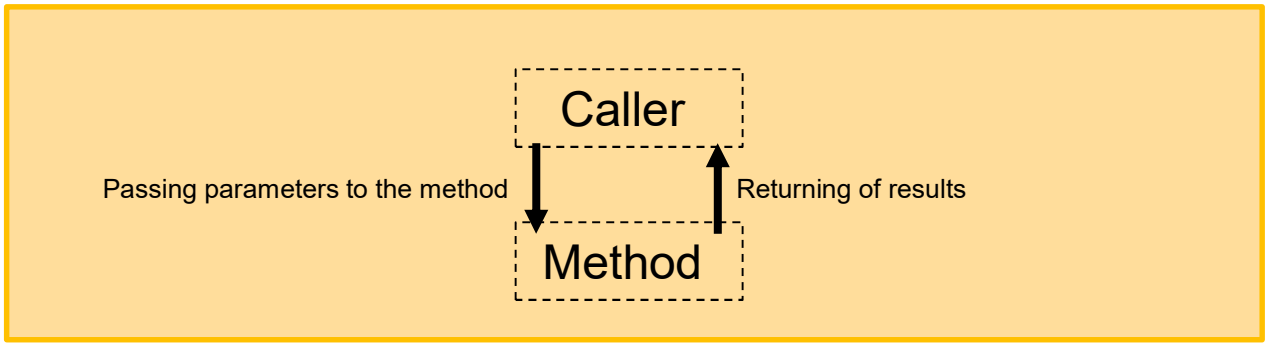
```
public class Rational {
   // Instance variables
   int numer;
   int denom;

   Rational() {
      this(0);          //calls Rational(int)
   }

   Rational(int n) {
      this(n, 1);       //calls Rational(int, int)
   }

   Rational(int n, int d) {
      // 1. Check whether d != 0
      // 2. Reduce n and d
      // ...
      numer = n;
      denom = d;
   }
}
```



The return v



- Idea:
  - By passing parameters to the method, we transport information from the caller to the method.
  - The return value of a method bounces information from the method back to the caller



  - A method can accept any number of parameter values, but can only return **one result value**

The return value of a method (2)

Technische Hochschule Rosenheim

- Two coupled measures:
  1. Type of the result value in the method **header**
  2. `return` statement in the method **body**
- Schema:

```
type methodName(…)
{
   ...
  return expression;
}
```

`type` in method header must be compatible with type of `expression` in `return` statement!

- Example:

```
int getNumer() {     //int method
  return numer;
}
```

The return value of a method (3)

Technische Hochschule Rosenheim

- Other examples:

```
class Rational {
   …

  int getDenom() {     //int method
    return denom;
    }

  double getReal() {     //double method
    return ((double)numer)/denom;
    }

  void reduce() {     //void method
  …
    }
}
```

The return value of a method (4)

Technische Hochschule Rosenheim

- Multiple `return` statements are allowed inside the body
  - Method returns as soon as the first `return` is reached during runtime
  - Static sequence of the `return` statements is irrelevant, the specific sequence during runtime is decisive

```
class Rational {
   …
  int signum() {
    if(numer * denom > 0)
      return 1;
    else
      if(numer == 0)
        return 0;
      else
        return -1;
}
```

The return value of a method (5)

Technische Hochschule Rosenheim

- Methods without result
  - Return without result: specification of the pseudo type `void` (= without any value)
  - Automatic return at the end of the method body
  - Return in the middle of the body with `return` statement without expression

```
class Rational {
   …
  void reduce() {
    int gcd = …;
    if (gcd == 1)
      return;          // already reduced, do nothing

  numer = numer / gcd;
  denom = denom / gcd;
   //automatic return
}
```

The return value of a method (6)

Technische Hochschule Rosenheim

- `return` statement is not allowed in constructors
- The returned value is automatically fixed == new object
- There is no possibility to return from the middle of the body (except the scenarios shown on previous slides, i.e. inside if-statement)
- The constructors are defined without result type, also not `void`.
  The constructor always delivers an instance of the class.
  Therefore, it is not needed to declare the return type.

Exercise – Programme Java class

Technische Hochschule Rosenheim

- Live exercise
  - Complete Task 3 & 4 on the live exercises sheet “Class declaration and use”
  - You have 10 minutes.

Coarse phases of typical object-oriented programmes

Technische Hochschule Rosenheim

- **Creating objects in the `main` method**
  - new operator
  - initialise the object appropriately by handing over parameters to the constructor → constructor takes care of proper initialization (duty by the programmer!)

- ***Linking the objects***
  - either via parameters of the constructor
  - or by calling a method available of that object

- ***Working phase***
  - objects work together → exchange of information: a method is triggered, results might be sent back

- ***Destruction phase***
  - destruction of the objects
  - Java automatically detects which objects are no longer linked and deletes them **(garbage collection)**

Example

Technische Hochschule Rosenheim

```
class Person {

   // instance variables
   String name;
   int age;
   Toy favouriteToy;

   // constructors
   Person(String name, int age, Toy toy) {
      this.name = name;
      this.age = age;
      this.favouriteToy = toy;
   }

   // methods
   void getsOlder() {   this.age += 1;    }
   String answers() {   return "My name is " + this.name + " and I am " + age + " years old.";   }
   void setMyFavouriteToy(Toy toy) {   this.favouriteToy = toy;   }
}
```

Example cont.

```
public class Toy {

   // instance variables
   String type;
   String color;
   String texture;

   // constructors
   Toy(String type) {
      this.type = type;
      this.color = "n.a.";
      this.texture = "n.a.";
   }

   Toy(String type, String color) {
      this(type);
      this.color = color;
      this.texture = "n.a.";
   }

   Toy(String type, String color, String texture) {
      this(type, color);
      this.texture = texture;
   }

   // methods
   String makesNoise() {    return (type.equals("Teddy")) ? "Beah!!!" : "Silence is golden.";   }
   void setTexture(String texture) {   this.texture = texture;   }
}
```

Example cont.

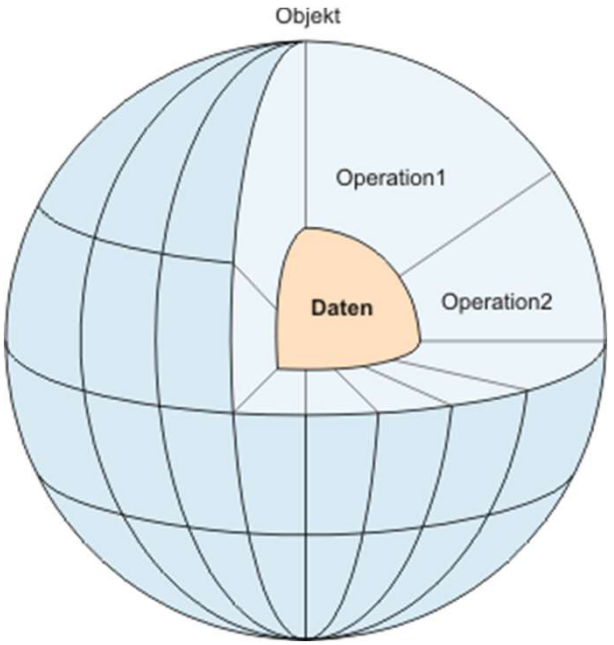
Technische Hochschule Rosenheim

```
public class Application {

   public static void main(String[] args) {

      // we initialize new objct(s)
      Toy teddy = new Toy("Teddy", "brown", "fluffy");
      Person tina = new Person("Tina", 9, teddy);

      // we work with the objects:
      tina.answers();
      teddy.setTexture("rough");
      teddy.makesNoise();
      tina.setMyFavouriteToy(teddy);
      tina.getsOlder();
      tina.answers();

   }

}
```

Basic principles of object orientation Part 1

Technische Hochschule Rosenheim

- **Abstraction**
  - Extract from the real world
  - Relevant objects
  - Relevant, characteristic properties of objects.
- **Modularity**
  - Partitioning into smaller, less complex units
  - Structuring by objects, classes and packages
- **Data encapsulation ("information hiding")**
  - Summarising data and behaviour.
  - Hiding the implementation behind an ***interface.***
  - Access only via the interface, so that internal data remains consistent.

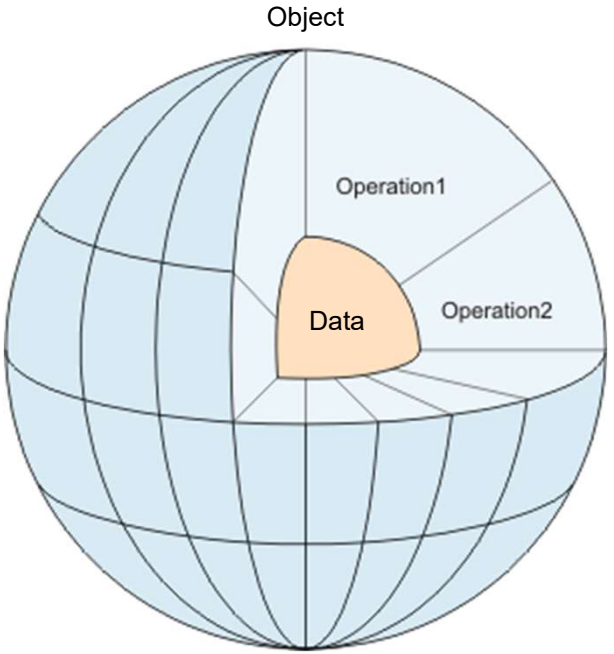


Polymorphism & inheritance (course OOP in Semester 2)

Data encapsulation – information hiding

Technische Hochschule Rosenheim

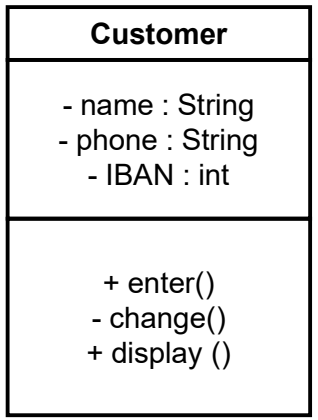
- Object encapsulates state (**data**) and operations (**behaviour**)
  - Data can only be read and changed using the operations
  - Representation of the data hidden to the outside
- Object realises information hiding
  - Secure and controlled access to the attributes
- Access to internal state is realized by access identifiers



Implementation in UML and/or Java

Technische Hochschule Rosenheim

- UML
  - Minus sign: attribute and/or operation is not visible or accessible from the outside
  - Plus sign: attribute and/or operation is visible and accessible from the outside



- Java
  - Visibility of attributes and of operations are set by keywords
  - Publicly visible: `public`
  - Not visible from the outside: `private`

Data encapsulation (1)

Technische Hochschule Rosenheim

- Idea:
  - Important tool in the construction of programmes = modularisation
    - Programme is broken down into parts that can be treated individually and independently
    - Programme part is called a module (usually a class in Java)
  - The fewer the dependencies between the modules, the easier it is to individually design, implement, exchange, extend, test, correct, ... the source code
  - Data encapsulation = measure for reducing dependencies (*data hiding*)

Data encapsulation (2)

Technische Hochschule Rosenheim

- Data and operations
  - Data encapsulation = hiding data behind operations
  - Access to data only via operations
  - Java:
    - Data <-> Instance variables
    - Operations <-> Methods
  - Users of a class: may call methods, but may not (directly) address instance variables

Data encapsulation (3)

Technische Hochschule Rosenheim

- Access protection
  - Modifier `private` limits the visibility of a declaration to its own class
  - Access from the "outside" is not granted

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

instance variables are not visible for class users!

```
class Application {
  public static void main(String[] args) {
    Rational r = new Rational();
    r.numer = 1;
        //Error - 'numer' has private access in Rational.java
}
```

Data encapsulation (4)

Technische Hochschule Rosenheim

- `private` methods
  - cannot be called from the outside, only by methods of the own class
  - useful for helper methods that the user should not use but which are useful to further structure the source code inside the class.
- Example:

```
class Rational {
    …
  void reduce() {
    final int gcd = gcd(numer, denom);
    numer /= gcd;
    denom /= gcd;
  }

  private int gcd(int a, int b) {…}
}
```

Visibility at a glance

Technische Hochschule Rosenheim

- The *visibility* can be restricted by **modifiers**
  - **`public`** (UML: +)
    - Access from outside the class possible with **"."** operator.
  - **`private`** (UML: -)
    - No access from outside the class.
    - Access is only possible within methods of the same class.
  - ***no specification / package visible*** (UML: ~)
    - Is present if you do not specify visibility.
    - All classes of the same ***package*** (→ later) have access
  - **`protected`** (UML: #)
    - Visible in the own class, as well as in all derived classes and all classes of the package.

      → See chapter on inheritance.

Visibility at a glance

Technische Hochschule Rosenheim

(UML) Modifier	Class	Package	Subclass	World
`(+) public`	Yes	Yes	Yes	Yes
`(#) protected`	Yes	Yes	Yes	No
(~) no modifier	Yes	Yes	No	No
`(-) private`	Yes	No	No	No

http://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html

- Attributes are usually `private`
  - …except if there's a good reason for `protected` or `public`
- Methods are usually `public`
  - …except if there's a good reason for `protected` or `private`

Visibilities: miscellaneous

Technische Hochschule Rosenheim

- **Private constructor** only makes sense in exceptional cases.

- **Visibility of classes**
  - If classes are declared as private, all attributes and methods of the class are automatically private.
  - Actually only useful in connection with ***inner classes*** (see in OOP, 2nd semester)

Data encapsulation (5)

Technische Hochschule Rosenheim

- getter methods
  - `private` instance variable not reachable from the outside
  - still make value accessible: offer getter method
  - declaration of a getter to an instance variable

```
private type name;
```

according to the pattern

```
type getName()
{
   return this.name;
}
```

```
class Rational {
   …
  int getNumer()  {
    return numer;
  }

  int getDenom()  {
    return denom;
  }
 …
}
```

Data encapsulation (6)

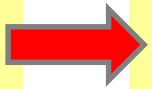
Technische Hochschule Rosenheim

- setter methods
  - `private` instance variable in modifiable classes: no external access, hence, a modification is not possible
  - still allow modifications: setter methods
  - declaration of a setter to an instance variable

```
private type name;
```

according to the pattern

```
void setName(type name) {
    this.name = name;
}
```



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

  void setNumer(int numer) {
    this.numer = numer;
   }

  void setDenom(int numer) {
    this.denom = denom;
  }
}
```

Data encapsulation (5)

Technische Hochschule Rosenheim

- Rules for getter and setter methods:
  - A so-called `get` or `set` method is programmed for each variable in the class, which must be accessed from outside the class.
  - The `get` method reads a single variable value. Method identifier should be `getAttributeName`.
  - The `set` method assigns a value to a variable. Method identifier should be `setAttributeName`.

Data encapsulation (6)

Technische Hochschule Rosenheim

- Interfaces
  - Interface of a class =
    - signatures of public (= non-private) methods +
    - public (= non-private) instance variables
  - Implementation: everything else

  - In other words:
    - The interface describes what a class offers
    - The implementation determines how it is realized
  - The user only needs to know the interface of a class (what is offered by the class) in order to use it

Exercise – Programme Java class

Technische Hochschule Rosenheim

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

Immutable classes (1)

Technische Hochschule Rosenheim

- Problem: free access to instance variables so that the state of the object can be changed from the outside.

```
class Rational {
   …
  void mult(Rational that) {
    …
    that.denom = 0;      // foreign object is changed!!!
    }
}
```

- Solution idea: immutable classes do not allow any changes to instance variables through other instances
  - Reading information from objects is OK
  - Changes are not possible

Immutable classes

Technische Hochschule Rosenheim

- **Definition**:
  - A class is ***immutable*** if the state of an object does not change after it is initialized.
- **Classes should be immutable as far as possible!!!**
  - The state is already defined "at birth". Simplifies application in data structures
  - Thread safety
  - No implementation of the `clone` method is necessary (see in later courses).
- **How do we make a class *immutable* in Java ?**
  - Declare the class as `final`
    - Prevents derivation from the class (key word inheritance).
    - Declare all attributes as `private` and `final`
  - No methods that change attributes
    - Exception: constructor

Immutable classes (2)

Technische Hochschule Rosenheim

- `final` instance variables
  - Modifier `final` for instance variables blocks changes
  - `final` instance variable must be assigned with a value once
  - Assignment of the value optionally …
    - at the declaration
    - in a constructor
    - in a chained constructor

Immutable classes (3)

Technische Hochschule Rosenheim

- `final` instance variables - examples

```
class Immu {
   final int n = 1;
    Immu(){}
}
```

Value assignment at declaration

```
class Immu {
    final int n;
   Immu(){
      n = 1;
    }
}
```

Value assignment in a constructor

Value assignment in a chained constructor

```
class Immu {
    final int n;
   Immu(){
    this(1);
    }
   Immu(int n) {
    this.n = n;
    }
}
```

Class variables (1)

Technische Hochschule Rosenheim

- So far: instance variables and methods refer to a specific object (= target object)
- Class variables (class attributes, static attributes) are assigned to an entire class, not an individual object
- Class variables exist independently of objects
- Class variables can be used by any method of the class

Class variables (2)

Technische Hochschule Rosenheim

- Declaration of a class variable:
  - Same as normal instance variable + modifier `static`

```
class Rational {
  static int count;
}
```

- Accessing class variables
  - With class names instead of target objects

```
Rational.count = 0;
```

- Example: mathematical constant π

```
Math.PI
```

== Class variable `PI` in the `Math` class

Class variables (3)

Technische Hochschule Rosenheim

- Initialisation
  - Takes place at the declaration, as with other instance variables

```
class Rational {
  static int count = 0;
   ...
}
```

  - Without explicit initialisation: default value, dependent on the type

- Lifetime of a class variable: total programme runtime, independent of objects

Class variables (4)

Technische Hochschule Rosenheim

- Example: object counter
  - Class variables are available for methods just like instance variables
  - But: only one instance for all objects
  - Demo: count the number of `Rational` objects

```
class Rational {
 static int count = 0;
 private final int numer;
 private final int denom;

  Rational() {
    numer = 0;
    denom = 1;
    count++;
}
```

Increment counter in constructor

Class variables (5)

Technische Hochschule Rosenheim

- Example: serial number
  - Unique serial number for each object

```
class Rational {
   private static int nextSerial = 0;   // next free number
   private final int serial;            // number of this object
   …
   Rational() {
      serial = nextSerial;
      nextSerial++;
      …
   }
   int getSerial() {
      return serial;
   }
   …
}
```

Class methods (1)

Technische Hochschule Rosenheim

- Class methods (static methods): address an entire class, not a specific object
- Declaration with modifier `static`

```
class Rational {
  static int getCount() {…}
}
```

- Calling with class names instead of target objects

```
System.out.println(Rational.getCount());
```

Class methods (2)

Technische Hochschule Rosenheim

- Already know: static method `main` == main programme

```
class SomeClass {
    public static void main(String[] args) {
        ...
  }
}
```

  - Before `main` there is no object yet: `main` must be static

Class methods (3)

Technische Hochschule Rosenheim

- Static helper methods
  - Helper methods that are independent of objects in the class
  - Example: calculation of the `ggT` in the class `Rational`

```
class Rational {
   static int gcd(int a, int b) {...}
}
```

usable without `Rational` objects:

```
System.out.println(Rational.gcd(221, 255));
```

Class methods (4)

Technische Hochschule Rosenheim

- Limitations:
  - Access only to class variables
  - Only call other class methods / static methods
  - `this` not available
  - Is statically bound, not dynamic

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

    static int gcd() {
        int a = numer;   // Error - numer is instance variable
        int b = denom;   // Error - denom is instance variable
        ...
    }
    ...
}
```

enum classes (1)

Technische Hochschule Rosenheim

- Enumeration types
- Meaning
  - Often data types with specific values are needed
  - Neither numbers nor truth values
  - Examples
    - colours: red, green, yellow
    - days of the week: Mon, Tue, Wed, Thur, Fri, Sat, Sun
    - player positions: goalkeeper, pivot, defender, winger
- Type definition with limited number of values
  - Explicitly lists the desired values of the data type
  - Synonyms: enumeration

enum classes (2)

Technische Hochschule Rosenheim

- Definition schema for enum types:

```
enum enumtype {enumelemen, enumelement, …}
```

Type — List of possible values

- Examples:

```
enum Colour {Red, Green, Blue, Yellow}
enum Day {Mon, Tue, Wed, Thu, Fri, Sat, Sun}
enum ChessPiece {Pawn, Rook, Knight, Bishop, Queen, King}
```

enum classes (3)

Technische Hochschule Rosenheim

- enum types
  - Equal rights with other types
  - Example: declaration of a variable

```
Colour c;
```

  - Example: assignment of a value

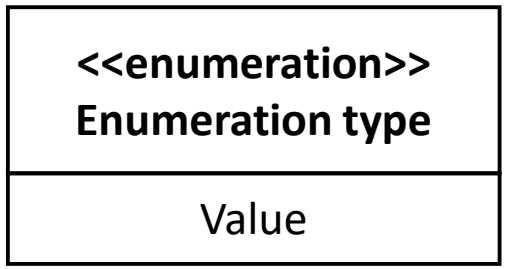
```
c = Colour.Red;
```

  - Comparison of a value:

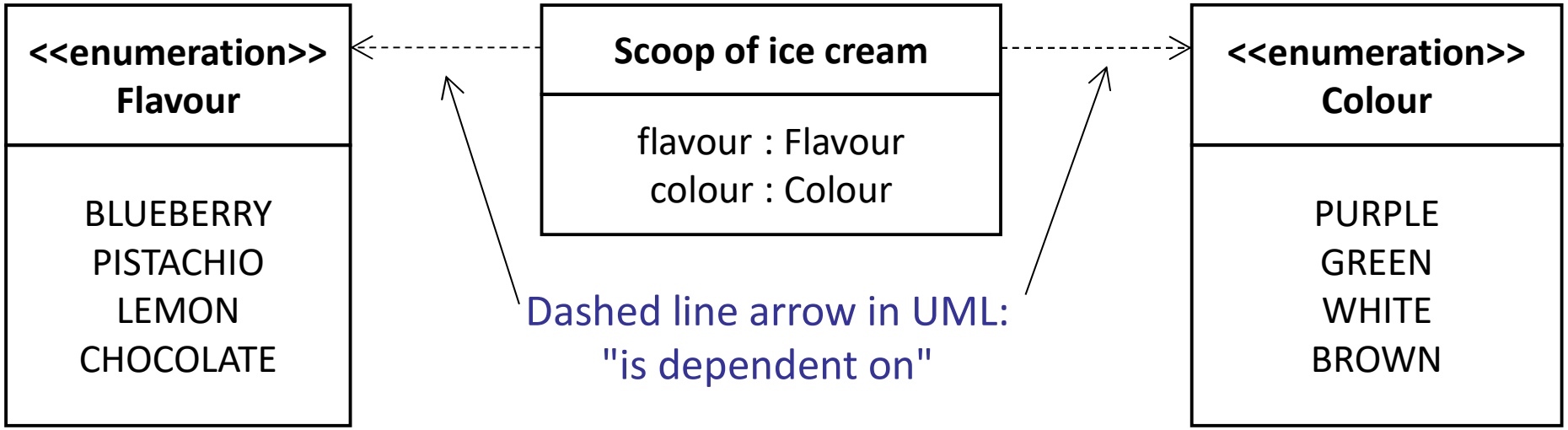
```
if (c == Colour.Yellow) …
```

Enumeration type in the class diagram

- Representation
  - Variant of the symbol for class
  - Stereotype <<enumeration>>
  - No operations
  - Explicit listing of the data values in the attributes area



- Example:



Example: Enumeration type in the class diagram

Technische Hochschule Rosenheim