Programmieren II

Intro Classes & Class Elements & Intro to OOP

Alexander Fraser

fraser@cl.uni-heidelberg.de

(Contains material from T. Bögel, K. Spreyer, S. Ponzetto, M. Hartung)

May 7, 2014

Outline

Outline

Control structures

Overview

- `if-else`
- Loops
  - `while`
  - `for`
  - (“for-each”)
- Others
  - `switch`
  - `do-while`

if-else

if-else cascades

- Syntax:

```
if ( expr1 )
    statement1;
else if ( expr2)
    statement2;
else if ( expr3)
    statement3;
. . .
else
    statements;
```

- Expressions are evaluated in order. Execute corresponding statement if an expression is `true`.

Loops: `while`

Loops: multiple executions of statements

Syntax

```
while ( expr )     ⇐ expr needs to be a boolean
     statement;
```

Meaning

1. Condition expr is evaluated
2. If it is `true`, `statement` is executed repeatedly
3. Otherwise: exit the loop

Loops: for

Syntax

```
for ( init; expr; update)
     statement
```

Meaning

1. Execute assignment `init` (usually: declaration & assignment)
2. if expr is `true`
   a. Execute `statement`
   b. Execute update
   c. Jump back to step 2
3. Otherwise: exit the loop

For-loop

Components

- **Initialization**: Initializes the loop; executed once at the beginning of the loop
- **Termination**: Loop is repeated until termination expression evaluates to `false`
- **Increment**: Invoked after each iteration.

- Scope of variables declared in `for`-loop: up to the end of the block.
- All three components of the `for`-loop are optional.

For-loop

A special kind of for-loop. . .

```
for ( ; ; ) {
        // this code is so beautiful that it should
        // be repeated forever...
}
```

Special for loop ("for-each")

Example

```
double[] scores = {1.2, 3.0, 0.8};
double sum = 0.;

for (double d : scores) {  // d gets successively each
                           // value in scores
    sum += d;
}
```

Jump statements

Jump statements

alter the execution flow within a loop

- `break` exits the surrounding loop immediately
- `continue` re-evaluates the condition of the loop (skips the remaining part of the loop)

switch

- Similar to cascaded `if-else`
- Multi-branched selection depending on the value of an expression
- You must not forget to break!
- Syntax:

```
switch ( expr )   ⇐ expr type byte, int, short, char or String!
      case value01:
            statement1;
      case value02:
            statement2;
...
[ default:  //optional !
            statement3; ]
```

Outline

Classes

Data types

- primitive data types (int, char, boolean, . . .)
- Arrays
- Classes

Classes

- A class defines a new data type
- A class is used to create new objects of this type
- Objects are instances of a class

Object-Oriented Cat

Object-Oriented Cat

Motivation

Motivation for object-oriented design

- Cats: different breeds, names, individual aspects
- But: common attributes and methods / states and services
- Classes define blueprints for objects
- Instantiating a class creates a new object
- $\rightarrow$ multiple individual cat objects, *one* class

Classes

```
class Cat {
   . . .
}

Cat c = new Cat();
```

Classes

```
class Cat {
    . . .
}

Cat c = new Cat();
```

⇒ Keyword for class definitions

Classes

```
class Cat {
   ...
}

Cat c = new Cat();
```

⇒ Class name (Convention: starts with a capital letter)

Classes

```
class Cat {
   . . .
}

Cat c = new Cat();
```

⇒ Instance variables and methods . . .

Classes

```
class Cat {
    . . .
}

Cat c = new Cat();
```

⇒ Variable of type Cat

Classes

```
class Cat {
   ...
}

Cat c = new Cat();
```

⇒ new operator instantiates `Cat` class

Classes

```
class Cat {
    . . .
}

Cat c = new Cat();
```

⇒ ... by calling the Cat constructor

Class elements

Instance variables (fields)

- data associated with objects
- state of an object
- e.g. `Cat`: `name, yearOfBirth, breed, mood`

Methods

- Code associated with objects
- implement behavior of objects, e.g. `purr()`
- manipulation of the state, e.g. `setName()`
- information about current state, e.g. `getName()`

Each object has its own copy of class elements

Class elements

```java
import java.io.*;

public class Cat {

    // instance variable of Cat objects
    String name;
    int yearOfBirth;
    String breed;
    String mood;

    /**
     *  Constructs a Cat object with the given name and
         yearOfBirth.
     */
    Cat( String name, int yearOfBirth ) {
        this.name = name;  /* "this" refers to object being
            created */
        this.yearOfBirth = yearOfBirth;
    }
```

Cat.java

Class elements

```
    /**
     *  Makes the Cat object "purr" to stdout.
     */
    void purr() {
        System.out.println( "Purr!" );
    }

    public static void main( String[] args ) {

        // call constructor to create new Cat object
        Cat gracie = new Cat( "Gracie", 2013 );

        // access the class members w/ the dot operator
        System.out.println( gracie.name );
        gracie.purr();
    }
}
```

Cat.java

Class elements

Class elements of the class `Cat`

- Instance variables: `name, yearOfBirth`
- Methods: `void purr()`

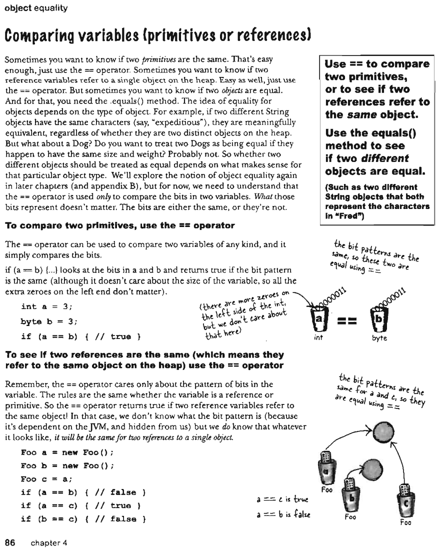
Accessing class elements

- using the dot operator
- for instance variables:
  `gracie.name; this.name = “Gracie”`
- and for methods:
  `gracie.purr(); System.out.println( “Purr!” )`

Classes/OOP in more detail: Goals

- Conceptual understanding of objects & classes
- Applying OOP to real-world problems
- How to write methods & instantiate new objects
- Learn about constructors and instance initialization
- Learn about common modifiers (`static`, `public`, `private`)

Comparing primitive and reference types



Comparing variables (primitives or references)

Sometimes you want to know if two *primitives* are the same. That's easy enough, just use the == operator. Sometimes you want to know if two reference variables refer to a single object on the heap. Easy as well, just use the == operator. But sometimes you want to know if two *objects* are equal. And for that, you need the .equals() method. The idea of equality for objects depends on the type of object. For example, if two different String objects have the same characters (say, "expeditious"), they are meaningfully equivalent, regardless of whether they are two distinct objects on the heap. But what about a Dog? Do you want to treat two Dogs as being equal if they happen to have the same size and weight? Probably not. So whether two different objects should be treated as equal depends on what makes sense for that particular object type. We'll explore the notion of object equality again in later chapters (and appendix B), but for now, we need to understand that the == operator is used *only* to compare the bits in two variables. *What* those bits represent doesn't matter. The bits are either the same, or they're not.

Use == to compare two primitives, or to see if two references refer to the *same* object.

Use the equals() method to see if two *different* objects are equal.

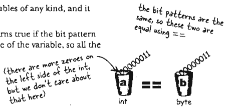
(Such as two different String objects that both represent the characters in "Fred")

To compare two primitives, use the == operator

The == operator can be used to compare two variables of any kind, and it simply compares the bits.

if (a == b) {...} looks at the bits in a and b and returns true if the bit pattern is the same (although it doesn't care about the size of the variable, so all the extra zeroes on the left end don't matter).

```
int a = 3;
byte b = 3;
if (a == b) { // true }
```

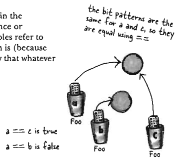


the bit patterns are the same, so these two are equal using ==
(there are more zeroes on the left side of the int, but we don't care about that here)
00000011
00000011
a
==
b
int
byte

To see if two references are the same (which means they refer to the same object on the heap) use the == operator

Remember, the == operator cares only about the pattern of bits in the variable. The rules are the same whether the variable is a reference or primitive. So the == operator returns true if two reference variables refer to the same object! In that case, we don't know what the bit pattern is (because it's dependent on the JVM, and hidden from us) but we *do* know that whatever it looks like, *it will be the same for two references to a single object.*

```
Foo a = new Foo();
Foo b = new Foo();
Foo c = a;
if (a == b) { // false }
if (a == c) { // true }
if (b == c) { // false }
```



the bit patterns are the same for a and c, so they are equal using ==
a == c is true
a == b is false
a
Foo
b
Foo
c
Foo

Source: Head First Java, p. 86.

Class declaration in general

Usually, classes consist of data (instance variables, “fields”) and code (methods). Instance variables and methods are often called elements of a class.

```
class <class name> {
  <type> <instance variable>;
  <type> <instance variable>;

  <type> <method name>(<list of parameters>){
    //...
  }

  <type> <method name>(<list of parameters>){
        //...
    }
}
```

listing-06.java

this

`this` refers to the reference of the object in which `this` is evaluated.
`this` is used to

- explicitly use instance variables in an object, if other, local variables of the same name are used.

  ```
  this.name = name;
  ```

- call methods for the same object

  ```
  int perimeter = this.getPerimeter();
  ```

- call constructors from other constructors

  ```
  this( name );
  this.nickName = nickName;
  ```

⇒ `this` is used to explicitly state that an object should use its *own* variables/methods

Methods of a class

- declaration in general:

```java
<type> method_name(<parameter list>) {
  <method body>
  return <value>;
}
```

listing-10.java

- `type` refers to the return type returned by the method (for methods without return values: `void`)
- `parameter list` is a comma-separated list of pairs of data type and variable name
- `return <value>` returns the value of type `<type>`

Methods

Methods

```
void purr( String sound ) {
   System.out.println( sound );
}
```

- have a name
- have parameters (0 or more). Their type & name is specified after the method name
- have a return value = type of the return value
- return value void means that no value is returned
- have a body, where parameter values can be used

Methods

Methods

```
void purr( String sound ) {
   System.out.println( sound );
}
```

- have a name
- have parameters (0 or more). Their type & name is specified after the method name
- have a return value = type of the return value
- return value void means that no value is returned
- have a body, where parameter values can be used

Methods

Methods

```
void purr( String sound ) {
   System.out.println( sound );
}
```

- have a name
- have parameters (0 or more). Their type & name is specified after the method name
- have a return value = type of the return value
- return value void means that no value is returned
- have a body, where parameter values can be used

Methods

Methods

```
void purr( String sound ) {
   System.out.println( sound );
}
```

- have a name
- have parameters (0 or more). Their type & name is specified after the method name
- have a return value = type of the return value
- return value void means that no value is returned
- have a body, where parameter values can be used

Methods

Methods

```
void purr( String sound ) {
   System.out.println( sound );
}
```

- have a name
- have parameters (0 or more). Their type & name is specified after the method name
- have a return value = type of the return value
- return value void means that no value is returned
- have a body, where parameter values can be used

Methods

Methods

```
void purr( String sound ) {
   System.out.println( sound );
}
```

- have a name
- have parameters (0 or more). Their type & name is specified after the method name
- have a return value = type of the return value
- return value void means that no value is returned
- have a body, where parameter values can be used

Methods

Returning values

- `return` statement: `return <expr>;`
- `return` exits the method body

Example: getter methods

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

→ are often defined for all instance variables of a class to avoid direct access

Constructors

Constructors

```
Cat( String name, int yearOfBirth ) {
   this.name = name;
   this.yearOfBirth = yearOfBirth;    this.mood =
"grumpy"; }
```

- same name as the class name
- have parameters like regular methods
- do not have an explicit return type or return value
- used to initialize objects (instance variables)

Constructors

Constructors

```
Cat( String name, int yearOfBirth ) {
   this.name = name;
   this.yearOfBirth = yearOfBirth;    this.mood =
"grumpy"; }
```

- same name as the class name
- have parameters like regular methods
- do not have an explicit return type or return value
- used to initialize objects (instance variables)

Constructors

Constructors

```
Cat( String name, int yearOfBirth ) {
   this.name = name;
   this.yearOfBirth = yearOfBirth;    this.mood =
"grumpy"; }
```

- same name as the class name
- have parameters like regular methods
- do not have an explicit return type or return value
- used to initialize objects (instance variables)

Constructors

Constructors

```
Cat( String name, int yearOfBirth ) {
   this.name = name;
   this.yearOfBirth = yearOfBirth;    this.mood =
"grumpy"; }
```

- same name as the class name
- have parameters like regular methods
- do not have an explicit return type or return value
- used to initialize objects (instance variables)

Constructors

Constructors

```
Cat( String name, int yearOfBirth ) {
   this.name = name;
   this.yearOfBirth = yearOfBirth;    this.mood =
"grumpy"; }
```

- same name as the class name
- have parameters like regular methods
- do not have an explicit return type or return value
- used to initialize objects (instance variables)

Constructors

this

- this is used to refer to the object itself
- e.g. `this.name = “Gracie”; this.name = name;`
- distinguishes between instance variables and parameter names or local variables:

```
Cat( String name, int yearOfBirth ) {
    this.name = name;
    . . .
}
```

Constructors

`this`

- `this` is used to refer to the object itself
- e.g. `this.name = “Gracie”; this.name = name;`
- distinguishes between instance variables and parameter names or local variables:

  ```
  Cat( String name, int yearOfBirth ) {
      this.name = name;
      . . .
  }
  ```

⇒ Instance variable name

Constructors

`this`

- `this` is used to refer to the object itself
- e.g. `this.name = “Gracie”; this.name = name;`
- distinguishes between instance variables and parameter names or local variables:

```
Cat( String name, int yearOfBirth ) {
    this.name = name;
    . . .
}
```

$\Rightarrow$ Parameter name

Detour: Instance initialization

- **Initializer**: alternative to constructors
    - Instance variable initializers
    - Instance initializers (instance initialization block)

Instance variable initializers

- Consists of an equals sign and one expression

Example: initializing a student (constructor)

```
public class StudentInitializer {
    private String major;
    public StudentInitializer() {
        this.major = "CoLi";
    }
}
```

Alternative: instance variable initializer

```
public class StudentInitializerInstInit {
    private String major = "CoLi";
    // no constructor...
}
```

Instance initialization block

- Same idea as instance initializers
- Allows more complex operations for initialization
- No forward reference

This will not compile!

```
public class Restaurant {
    private int chairs = tables * 4;
    private int tables = 10;
}
```

Options for initializing instance variables

Summary

- There are three ways to initialize default values for new objects:

1. Constructor: set values directly or use parameters (next section)
2. Variable initializer: simple assignment of values to instance variables where they are declared in a class
3. Initialization block: more complex block that is executed prior to the constructor

$\Rightarrow$ example: Moodle

Pass by value

Pass by value

- What happens with a variable if you call a method with the variable as a parameter?
- Head First Java (2^{nd} ed.), p. 77.

Pass by value vs. pass by reference – Litmus test

- Aim: testing whether a programming language performs pass-by-value (Java!) or pass-by-reference

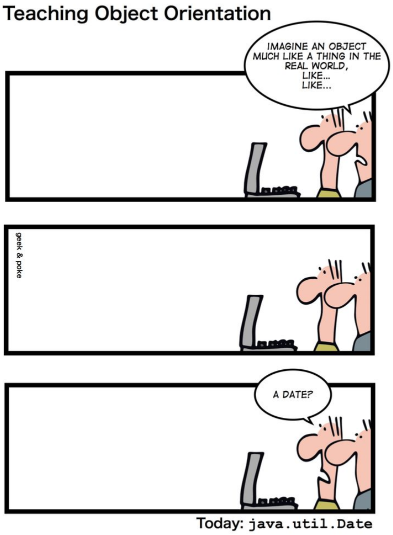
Litmus test

```
swap(Type arg1, Type arg2) {
    Type temp = arg1;
    arg1 = arg2;
    arg2 = temp;
}
```

- If the values of `arg1` and `arg2` are swappable: **pass-by-reference** (e.g. C++)
- You cannot do this in Java!

Outline

What is an object?


Teaching Object Orientation
IMAGINE AN OBJECT MUCH LIKE A THING IN THE REAL WORLD, LIKE... LIKE...
geek & poke
A DATE?
Today: java.util.Date

source: `http://geek-and-poke.com/2012/08/teaching-oo.html`

Everything is an object

Everything is an object. Think of an object as a fancy variable; it stores data, but you can 'make requests' to that object, asking it to perform operations on itself. In theory, you can take any conceptual component in the problem you're trying to solve (dogs, buildings, services, etc.) and represent it as an object in your program.

Source: Thinking in Java, 3rd Ed.

Definition of a program

A program is a bunch of objects telling each other what to do by sending messages. To make a request of an object, you ‘send a message’ to that object. More concretely, you can think of a message as a request to call a method that belongs to a particular object.

Source: Thinking in Java, 3rd Ed.

Objects and complexity

Each object has its own memory made up of other objects. Put another way, you create a new kind of object by making a package containing existing objects. Thus, you can build complexity into a program while hiding it behind the simplicity of objects.

Source: Thinking in Java, 3rd Ed.

Objects and data types

Every object has a type. Using the parlance, each object is an instance of a class, in which 'class' is synonymous with 'type'. The most important distinguishing characteristic of a class is "What messages can you send to it?"

Source: Thinking in Java, 3rd Ed.

Objects and abstraction

All objects of a particular type can receive the same messages. This is actually a loaded statement, as you will see later. Because an object of type ‘circle’ is also an object of type ‘shape,’ a circle is guaranteed to accept shape messages. This means you can write code that talks to shapes and automatically handle anything that fits the description of a shape. This substitutability is one of the powerful concepts in OOP.

Source: Thinking in Java, 3rd Ed.

A simple example of a class

Modeling a triangle

- What fields do we need?
- What methods could be useful?

```java
class Triangle{
  double a,b,c;
  double alpha,beta,gamma;

  boolean checkAngularSum(){
    double asum;
    asum = alpha + beta + gamma;
    if (asum == 180.0)
      return true;
    else
      return false;
  }

  double getPerimeter(){
    return a + b + c;
  }
}
```

Triangle.java

Creating objects

```
public class TriangleDemo1 {

    public static void main(String[] args) {
        // Creation of a new  object of the type Triangle
        Triangle t = new Triangle();
    }
}
```

TriangleDemo1.java

The new operator instantiates a new object.
Variable `t` refers to an instance of an object of type `Triangle` with all instance variables and methods

Accessing variables of an object

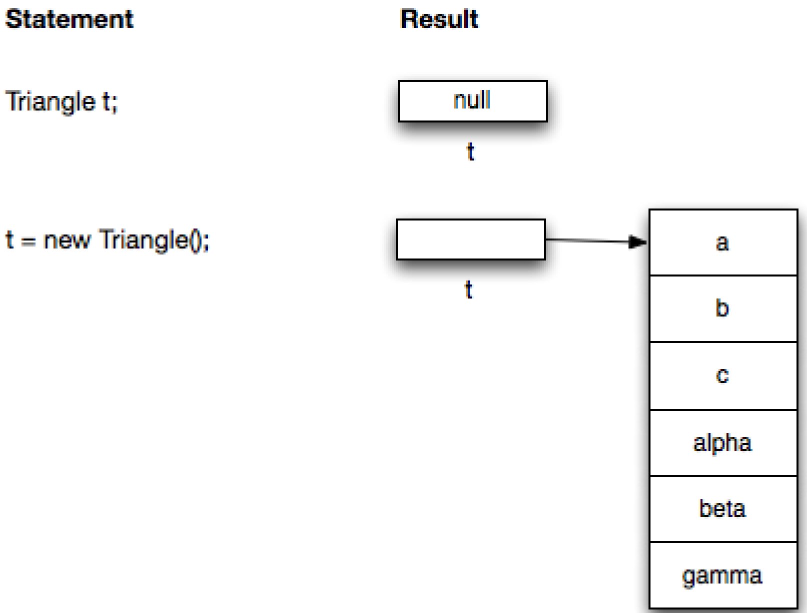
```
public class TriangleDemo2 {

    public static void main(String[] args) {
        Triangle t = new Triangle();
        // assign a value to t's instance variable a
        t.a = 2;
        System.out.println( t.a );
    }
}
```

TriangleDemo2.java

Variables of a class can be accessed with the "." operator (if they are not declared as `private`). Alternatively: separate methods (getters).

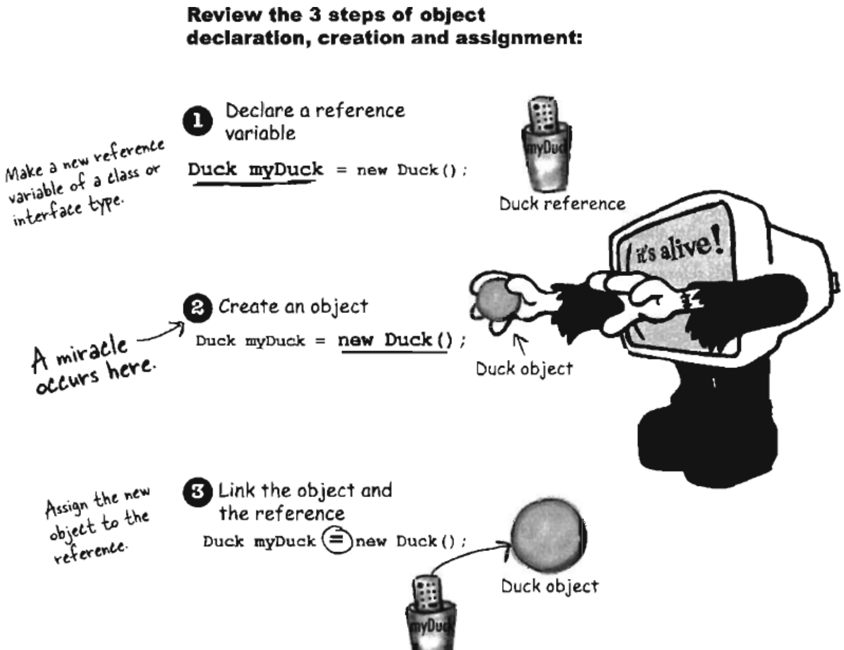
Declaration vs. object creation



The new operator

- The declaration of a variable (`Triangle t`) declares the variable but does not allocate memory
- The new operator instantiates a new object of the specified type and allocates memory
- in general:
  `<object reference variable> = new <class name>() ;`
- `<class name>()` calls the default constructor

Declaration, Creation, Assignment (HFJ, p. 240)



Review the 3 steps of object declaration, creation and assignment:

Make a new reference variable of a class or interface type.

1 Declare a reference variable

```
Duck myDuck = new Duck();
```

A miracle occurs here.

2 Create an object

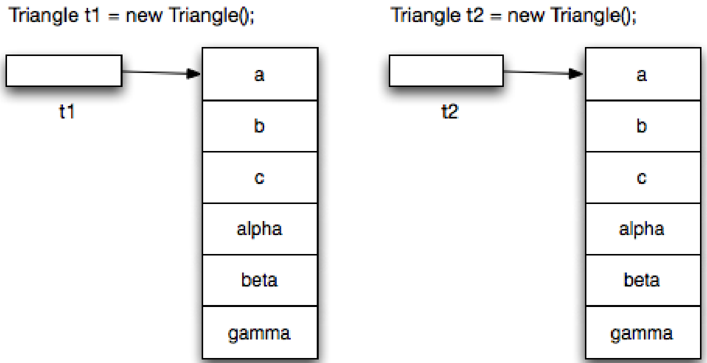
```
Duck myDuck = new Duck();
```

Assign the new object to the reference.

3 Link the object and the reference

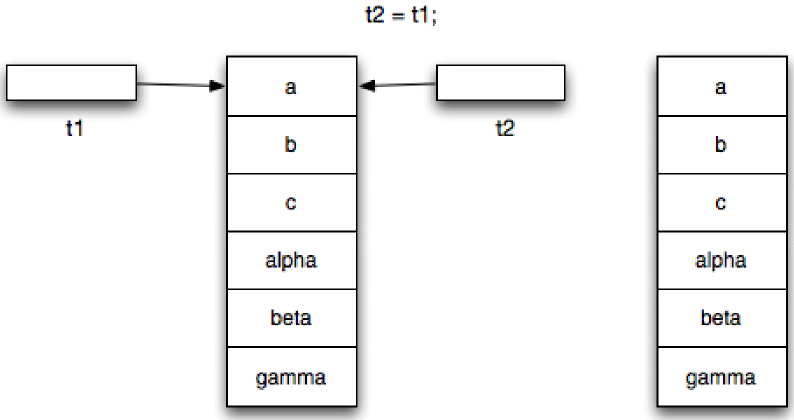
```
Duck myDuck = new Duck();
```

Multiple instances of a class



- t1 and t2 refer to different objects, i.e. instances derived from the same class
- Instance variables of both objects can have different values!

Assigning object reference variables



What does this mean for the instance variables of `t1` and `t2`?

Assigning object reference variables – example

```java
class TriangleDemo{
  public static void main(String args[]){
    Triangle t1 = new Triangle();
    Triangle t2 = new Triangle();

    t1.alpha = 45.0;
    t1.beta = 45.0;
    t1.gamma = 90.0;

    t2.alpha = 30.0;
    t2.beta = 105.0;
    t2.gamma = 45.0;

    t2 = t1;

    t1.a = 5.0;
  }
}
```

listing-09.java

Exercise

Object declaration vs. instantiation

- Exercise:
- Head First Java (2^{nd} ed.), p. 63.

Methods of the class Triangle

```java
class Triangle{
  double a,b,c;
  double alpha,beta,gamma;

  boolean checkAngularSum(){
    double asum;
    asum = alpha + beta + gamma;
    if (asum == 180.0)
      return true;
    else
      return false;
  }

  double getPerimeter(){
    return a + b + c;
  }
}
```

Triangle.java

Accessing methods of a class

```
public class TriangleDemo3 {

    public static void main(String[] args) {
        Triangle t1 = new Triangle();
        t1.alpha = 60;
        t1.beta = 60;
        t1.gamma = 60;
        boolean correct = t1.checkAngularSum();
        System.out.println("Correct angular sum: "+correct);
    }
}
```

TriangleDemo3.java

Methods of a class are accessed with the (already known) "." operator

Literature

- Sierra, K. & Bates, B.
  Head First Java. (Ch. 2, 4)
  O'Reilly Media, 2005.
- Ullenboom, Ch.
  Java ist auch eine Insel. (Ch. 3)
  Galileo Computing, 2012.
- Eckel, B. (more as a comprehensive reference)
  Thinking in Java. (Ch. 2 & 4)
  Prentice Hall, 2006.