合肥學院
HEFEI UNIVERSITY

OOP with Java

14. Objects, Instance Variables, and New

Thomas Weise · 汤卫思

tweise@hfuu.edu.cn · http://iao.hfuu.edu.cn

Hefei University, South Campus 2
Faculty of Computer Science and Technology
Institute of Applied Optimization
230601 Shushan District, Hefei, Anhui, China
Econ. & Tech. Devel. Zone, Jinxiu Dadao 99

合肥学院 南艳湖校区/南2区
计算机科学与技术系
应用优化研究所
中国 安徽省 合肥市 蜀山区 230601
经济技术开发区 锦绣大道99号

Outline

Introduction

- We now can create variables of primitive types (like `int`), of strings, and arrays of them

Introduction

- We now can create variables of primitive types (like `int`), of strings, and arrays of them
- Sometimes, we want more complex data structures, we want to combine several variables to a group

Introduction

- We now can create variables of primitive types (like `int`), of strings, and arrays of them
- Sometimes, we want more complex data structures, we want to combine several variables to a group
- We can do this with classes, objects, and instance variables

Terminology

- A `class` is a structured data type

Terminology

- A `class` is a structured data type
- An object is a concrete instance of a class

Terminology

- A `class` is a structured data type
- An object is a concrete instance of a class (like `int j` creates the instance `j` of the type `int`)
- Classes *define* instance variables

Terminology

- A `class` is a structured data type
- An object is a concrete instance of a class (like `int j` creates the instance `j` of the type `int`)
- Classes *define* instance variables
- Each instance of a class has one set of these instance variables

Terminology

- A `class` is a structured data type
- An object is a concrete instance of a class (like `int j` creates the instance `j` of the type `int`)
- Classes *define* instance variables
- Each instance of a class has one set of these instance variables
- We can have multiple, independent instances of the same class

Terminology

- A `class` is a structured data type
- An object is a concrete instance of a class (like `int j` creates the instance `j` of the type `int`)
- Classes *define* instance variables
- Each instance of a class has one set of these instance variables
- We can have multiple, independent instances of the same class
- Classes are instantiated with the keywork `new` followed by the class name and parentheses

Terminology

- A `class` is a structured data type
- An object is a concrete instance of a class (like `int j` creates the instance `j` of the type `int`)
- Classes *define* instance variables
- Each instance of a class has one set of these instance variables
- We can have multiple, independent instances of the same class
- Classes are instantiated with the keywork `new` followed by the class name and parentheses
- (Actually, you have already seen this when creating arrays. Arrays are special objects and so are Strings.)

A Class to Represent a Person

Listing: A Class to Represent a Person

```java
/** A class representing a person. */
public class Person {

  /** the family name of the person */
  String familyName;
  /** the given name of the person */
  String givenName;

  /** The main routine
   *  @param args
   *          we ignore this parameter */
  public static final void main(String[] args) {
    Person weise = new Person(); // create person object
    weise.familyName = "Weise"; // set the family name of object weise //$NON-NLS-1$
    weise.givenName = "Thomas"; // set the given name of object weise //$NON-NLS-1$

    Person chan = new Person(); // create person object
    chan.givenName = "Jacky"; // set the given name of object chan //$NON-NLS-1$
    chan.familyName = "Chan"; // set the family name of object chan //$NON-NLS-1$

    System.out.println(weise.givenName); // print the givenName "Thomas" of weise
    System.out.println(weise.familyName); // print the familyName "Weise" of weise

    System.out.println(chan.familyName); // print the familyName "Chan" of chan
    System.out.println(chan.givenName); // print the given name "Jacky" of chan
  }
}
```

Constructors

- When creating an object of a class (called "instantiating the class"), we actually call special method, the *constructor*

Constructors

- When creating an object of a class (called "instantiating the class"), we actually call special method, the *constructor*
- In the previous example, we did not specify one, so the object is just created and no special action is taken

Constructors

- When creating an object of a class (called "instantiating the class"), we actually call special method, the *constructor*
- In the previous example, we did not specify one, so the object is just created and no special action is taken
- But we can add some special actions that should be performed upon creation by specifying a constructor

Constructors

- When creating an object of a class (called "instantiating the class"), we actually call special method, the *constructor*
- In the previous example, we did not specify one, so the object is just created and no special action is taken
- But we can add some special actions that should be performed upon creation by specifying a constructor
- A constructor is a special method with the same name as the class

Constructors

- When creating an object of a class (called "instantiating the class"), we actually call special method, the *constructor*
- In the previous example, we did not specify one, so the object is just created and no special action is taken
- But we can add some special actions that should be performed upon creation by specifying a constructor
- A constructor is a special method with the same name as the class
- It can have parameters like a normal method, but has no return value

Constructors

- When creating an object of a class (called "instantiating the class"), we actually call special method, the *constructor*
- In the previous example, we did not specify one, so the object is just created and no special action is taken
- But we can add some special actions that should be performed upon creation by specifying a constructor
- A constructor is a special method with the same name as the class
- It can have parameters like a normal method, but has no return value
- It will be executed when creating an instance and you will then provide the parameters in the `new` statement

Constructors

- When creating an object of a class (called "instantiating the class"), we actually call special method, the *constructor*
- In the previous example, we did not specify one, so the object is just created and no special action is taken
- But we can add some special actions that should be performed upon creation by specifying a constructor
- A constructor is a special method with the same name as the class
- It can have parameters like a normal method, but has no return value
- It will be executed when creating an instance and you will then provide the parameters in the `new` statement
- You can have multiple constructors in class, but they then need to take parameters of different types

Constructors

- When creating an object of a class (called "instantiating the class"), we actually call special method, the *constructor*
- In the previous example, we did not specify one, so the object is just created and no special action is taken
- But we can add some special actions that should be performed upon creation by specifying a constructor
- A constructor is a special method with the same name as the class
- It can have parameters like a normal method, but has no return value
- It will be executed when creating an instance and you will then provide the parameters in the `new` statement
- You can have multiple constructors in class, but they then need to take parameters of different types
- One constructor can invoke another one as its very first command, like a function

The Special Variable `this`

- Inside an object's constructor (and any method of the object, see Lesson 15: *Instance Methods*), we can refer to the object itself via the `this` keyword

The Special Variable `this`

- Inside an object's constructor (and any method of the object, see Lesson 15: *Instance Methods*), we can refer to the object itself via the `this` keyword
- If we want to read the instance variable `bla` of the current object from its constructor, we can use `this.bla`

The Special Variable `this`

- Inside an object's constructor (and any method of the object, see Lesson 15: *Instance Methods*), we can refer to the object itself via the `this` keyword
- If we want to read the instance variable `bla` of the current object from its constructor, we can use `this.bla`
- Setting the instance variable `bla` to the value of expression `blubb` goes via `this.bla = blubb;`

A Class with Constructor to Represent a Person

Listing: A Class with Constructor to Represent a Person

```
/** A class representing a person. */
public class PersonWithConstructor {

  /** the family name of the person */
  String familyName;
  /** the given name of the person */
  String givenName;

  /** create a person record and set its name */
  PersonWithConstructor(String _familyName, String _givenName) {
    this.familyName = _familyName;
    this.givenName = _givenName;
  }

  /** The main routine
   *  @param args we ignore this parameter */
  public static final void main(String[] args) {

    PersonWithConstructor weise = new PersonWithConstructor(// create person object by
        "Weise", "Thomas"); //$NON-NLS-1$//$NON-NLS-2$      // calling the constructor

    PersonWithConstructor chan = new PersonWithConstructor(// create person object by
        "Chan", "Jacky"); //$NON-NLS-1$//$NON-NLS-2$       // calling the constructor

    System.out.println(weise.givenName); // print the givenName of Thomas
    System.out.println(weise.familyName); // print the familyName of Weise

    System.out.println(chan.familyName); // print the familyName of Chan
    System.out.println(chan.givenName); // print the given name of Jacky
  }
}
```

Objects and References

- There is a difference between an *object* and a *reference*

Objects and References

- There is a difference between an *object* and a *reference*
- Objects, such as Strings, arrays, and instances of our own classes, are basically a chunk of memory containing the instance variables (and some hidden management data)

Objects and References

- There is a difference between an *object* and a *reference*
- Objects, such as Strings, arrays, and instances of our own classes, are basically a chunk of memory containing the instance variables (and some hidden management data)
- A variable of (class) type `Person` is not the object itself

Objects and References

- There is a difference between an *object* and a *reference*
- Objects, such as Strings, arrays, and instances of our own classes, are basically a chunk of memory containing the instance variables (and some hidden management data)
- A variable of (class) type `Person` is not the object itself
- Instead, it *points* to the object, it is a reference, a pointer

Objects and References

- There is a difference between an *object* and a *reference*
- Objects, such as Strings, arrays, and instances of our own classes, are basically a chunk of memory containing the instance variables (and some hidden management data)
- A variable of (class) type `Person` is not the object itself
- Instead, it *points* to the object, it is a reference, a pointer
- If we do `Person A = new Person();` and then `Person B = A;`

Objects and References

- There is a difference between an *object* and a *reference*
- Objects, such as Strings, arrays, and instances of our own classes, are basically a chunk of memory containing the instance variables (and some hidden management data)
- A variable of (class) type `Person` is not the object itself
- Instead, it *points* to the object, it is a reference, a pointer
- If we do `Person A = new Person();` and then `Person B = A;`,
  - `A` and `B` point to the same object

Objects and References

- There is a difference between an *object* and a *reference*
- Objects, such as Strings, arrays, and instances of our own classes, are basically a chunk of memory containing the instance variables (and some hidden management data)
- A variable of (class) type `Person` is not the object itself
- Instead, it *points* to the object, it is a reference, a pointer
- If we do `Person A = new Person();` and then `Person B = A;`,
  - `A` and `B` point to the same object,
  - changes to the instance variables of the object references by `A` also appear in `B` (it is the same object!)

Objects and References

- There is a difference between an *object* and a *reference*
- Objects, such as Strings, arrays, and instances of our own classes, are basically a chunk of memory containing the instance variables (and some hidden management data)
- A variable of (class) type `Person` is not the object itself
- Instead, it *points* to the object, it is a reference, a pointer
- If we do `Person A = new Person();` and then `Person B = A;`,
  - `A` and `B` point to the same object,
  - changes to the instance variables of the object references by `A` also appear in `B` (it is the same object!), and
  - changes to the instance variables of the object references by `B` also appear in `A` (it is the same object!)

Objects and References

- There is a difference between an *object* and a *reference*
- Objects, such as Strings, arrays, and instances of our own classes, are basically a chunk of memory containing the instance variables (and some hidden management data)
- A variable of (class) type `Person` is not the object itself
- Instead, it *points* to the object, it is a reference, a pointer
- If we do `Person A = new Person();` and then `Person B = A;`,
  - `A` and `B` point to the same object,
  - changes to the instance variables of the object references by `A` also appear in `B` (it is the same object!), and
  - changes to the instance variables of the object references by `B` also appear in `A` (it is the same object!), since
  - the `=` operator does not copy/assign *objects* but *references*!

Objects Lifecycle

- OK, so an object is created by invoking its constructor

Objects Lifecycle

- OK, so an object is created by invoking its constructor
- And then? Do they live on forever?

Objects Lifecycle

- OK, so an object is created by invoking its constructor
- And then? Do they live on forever?
  - If this was true, programs would consume more and more memory the longer they run.

Objects Lifecycle

- OK, so an object is created by invoking its constructor
- And then? Do they live on forever?
  - If this was true, programs would consume more and more memory the longer they run.
  - Think web server programs accepting connections from web browsers ... the longer the run, the more memory they would consume.

Objects Lifecycle

- OK, so an object is created by invoking its constructor
- And then? Do they live on forever?
  - If this was true, programs would consume more and more memory the longer they run.
  - Think web server programs accepting connections from web browsers ... the longer the run, the more memory they would consume.
  - Eventually, they would crash because all available memory was used.

Objects Lifecycle

- OK, so an object is created by invoking its constructor
- And then? Do they live on forever?
  - If this was true, programs would consume more and more memory the longer they run.
  - Think web server programs accepting connections from web browsers ... the longer the run, the more memory they would consume.
  - Eventually, they would crash because all available memory was used.
- No. The memory occupied by objects will be freed when they are no longer needed.

Objects Lifecycle

- OK, so an object is created by invoking its constructor
- And then? Do they live on forever?
  - If this was true, programs would consume more and more memory the longer they run.
  - Think web server programs accepting connections from web browsers ... the longer the run, the more memory they would consume.
  - Eventually, they would crash because all available memory was used.
- No. The memory occupied by objects will be freed when they are no longer needed.
- We do not need to worry about that

Objects Lifecycle

- OK, so an object is created by invoking its constructor
- And then? Do they live on forever?
  - If this was true, programs would consume more and more memory the longer they run.
  - Think web server programs accepting connections from web browsers ... the longer the run, the more memory they would consume.
  - Eventually, they would crash because all available memory was used.
- No. The memory occupied by objects will be freed when they are no longer needed.
- We do not need to worry about that:
  - In Java, a Garbage Collector (GC) is sometimes automatically executed in the background.

Objects Lifecycle

- OK, so an object is created by invoking its constructor
- And then? Do they live on forever?
  - If this was true, programs would consume more and more memory the longer they run.
  - Think web server programs accepting connections from web browsers ... the longer the run, the more memory they would consume.
  - Eventually, they would crash because all available memory was used.
- No. The memory occupied by objects will be freed when they are no longer needed.
- We do not need to worry about that:
  - In Java, a Garbage Collector (GC) is sometimes automatically executed in the background.
  - It frees objects which are no longer referenced.

Objects Lifecycle

- OK, so an object is created by invoking its constructor
- And then? Do they live on forever?
  - If this was true, programs would consume more and more memory the longer they run.
  - Think web server programs accepting connections from web browsers ... the longer the run, the more memory they would consume.
  - Eventually, they would crash because all available memory was used.
- No. The memory occupied by objects will be freed when they are no longer needed.
- We do not need to worry about that:
  - In Java, a Garbage Collector (GC) is sometimes automatically executed in the background.
  - It frees objects which are no longer referenced.
  - It runs from time to time and not always frees all disposable objects at once (to be time-efficient).

Objects Lifecycle

- OK, so an object is created by invoking its constructor
- And then? Do they live on forever?
  - If this was true, programs would consume more and more memory the longer they run.
  - Think web server programs accepting connections from web browsers ... the longer the run, the more memory they would consume.
  - Eventually, they would crash because all available memory was used.
- No. The memory occupied by objects will be freed when they are no longer needed.
- We do not need to worry about that:
  - In Java, a Garbage Collector (GC) is sometimes automatically executed in the background.
  - It frees objects which are no longer referenced.
  - It runs from time to time and not always frees all disposable objects at once (to be time-efficient).
- If an object is created in a method and no reference to it is returned, it also becomes subject to disposal when the method returns.

Using Objects: Allocation, Assignment, Disposal

Listing: Using Objects: Allocation, Assignment, Disposal

```java
/** Using class PersonWithConstructor representing a person:
 *  allocation, member variable setting, null, disposal. */
public class PersonWithConstructorUsage {

  /** The main routine
   *  @param args we ignore this parameter */
  public static final void main(String[] args) {
    PersonWithConstructor weise = new PersonWithConstructor("Weise", "Thomas"); //$NON-NLS-1$//$NON-NLS-2$
    PersonWithConstructor chan = new PersonWithConstructor("Chan", "Jacky"); //$NON-NLS-1$//$NON-NLS-2$

    System.out.println(weise.givenName); // print the givenName of weise: "Thomas"
    System.out.println(weise.familyName); // print the familyName of weise: "Weise"

    weise.givenName = chan.givenName; // weise.givenName now points to same String object as chan.givenName
    System.out.println(weise.givenName); // print the givenName of weise: "Jacky"
    System.out.println(weise.familyName); // print the familyName of weise: "Weise"

    chan.givenName = "Kong-sang"; // change the given name of chan, given name of weise stays unchanged //$NON-NLS-1$
    System.out.println(weise.givenName); // print the givenName of weise: "Jacky"
    System.out.println(chan.givenName); // print the given name of Chan: "Kong-sang"

    weise = chan; // variables weise and chan now point to same object. this is NOT a copy
                  // the original weise object can eventually be disposed by GC, since it is no longer needed
    System.out.println(weise.givenName); // print the givenName of weise: "Kong-sang"
    System.out.println(weise.familyName); // print the familyName of weise: "Chan"

    chan.givenName = "Jacky"; // change givenName of object pointed to by chan (and weise) //$NON-NLS-1$
    System.out.println(chan.givenName); // print the givenName of chan: "Jacky"
    System.out.println(weise.givenName); // print the givenName of weise: "Jacky"

    chan = weise; // nothing changes, both variables still point to same object
    System.out.println(chan.familyName); // print the familyName of Chan: "Chan"
    System.out.println(chan.givenName); // print the given name of Chan: "Jacky"
  }
}
```

The Special Constant `null`

- (Object) variables point to objects

The Special Constant `null`

- (Object) variables point to objects
- If we want that a variable `v` points no longer to any object, we set it to `null`, i.e., `v=null`

The Special Constant `null`

- (Object) variables point to objects
- If we want that a variable `v` points no longer to any object, we set it to `null`, i.e., `v=null`
- We can compare values with `null`, i.e., do `if(v == null)` to see if `v` points to any object

The Special Constant `null`

- (Object) variables point to objects
- If we want that a variable `v` points no longer to any object, we set it to `null`, i.e., `v=null`
- We can compare values with `null`, i.e., do `if(v == null)` to see if `v` points to any object
- In your programs, if you do no longer need an object, it may make sense to set the variables referencing it explicitly to `null`

The Special Constant `null`

- (Object) variables point to objects
- If we want that a variable `v` points no longer to any object, we set it to `null`, i.e., `v=null`
- We can compare values with `null`, i.e., do `if(v == null)` to see if `v` points to any object
- In your programs, if you do no longer need an object, it may make sense to set the variables referencing it explicitly to `null`
- Obviously, an expression/variable with value `null` does not point to any object and you cannot access the instance variables of that...

The Special Constant `null`

- (Object) variables point to objects
- If we want that a variable `v` points no longer to any object, we set it to `null`, i.e., `v=null`
- We can compare values with `null`, i.e., do `if(v == null)` to see if `v` points to any object
- In your programs, if you do no longer need an object, it may make sense to set the variables referencing it explicitly to `null`
- Obviously, an expression/variable with value `null` does not point to any object and you cannot access the instance variables of that...
  ...In lesson Lesson 25: *Exceptions*, we will learn what happens if you try to do that anyway.

Listing: Using Objects: Allocation, Assignment, Disposal, and null

```java
/** Using class PersonWithConstructor representing a person:
 *  allocation, member variable setting, null, disposal. */
public class PersonWithConstructorUsageNull {

  /** The main routine
   *  @param args
   *          we ignore this parameter */
  public static final void main(String[] args) {
    PersonWithConstructor weise = new PersonWithConstructor("Weise", "Thomas"); //$NON-NLS-1$//$NON-NLS-2$
    PersonWithConstructor chan = new PersonWithConstructor("Chan", "Jacky"); //$NON-NLS-1$//$NON-NLS-2$

    System.out.println(weise.givenName); // print the givenName of weise: "Thomas"
    System.out.println(weise.familyName); // print the familyName of weise: "Weise"

    weise.givenName = chan.givenName; // weise.givenName now points to same String object as chan.givenName
    System.out.println(weise.givenName); // print the givenName of weise: "Jacky"
    System.out.println(weise.familyName); // print the familyName of weise: "Weise"

    chan.givenName = "Kong-sang"; // change the given name of chan, given name of weise stays unchanged //$NON-NLS-1$
    System.out.println(weise.givenName); // print the givenName of weise: "Jacky"
    System.out.println(chan.givenName); // print the given name of Chan: "Kong-sang"

    weise = chan; // variables weise and chan now point to same object. this is NOT a copy: original weise object can be disposed by GC
    System.out.println(weise.givenName); // print the givenName of weise: "Kong-sang"
    System.out.println(weise.familyName); // print the familyName of weise: "Chan"

    chan.givenName = "Jacky"; // change givenName of object pointed to by chan (and weise) //$NON-NLS-1$
    System.out.println(chan.givenName); // print the givenName of chan: "Jacky"
    System.out.println(weise.givenName); // print the givenName of weise: "Jacky"

    chan = weise; // nothing changes, both variables still point to same object
    System.out.println(chan.familyName); // print the familyName of Chan: "Chan"
    System.out.println(chan.givenName); // print the given name of Chan: "Jacky"

    chan = null; // variable chan now does not point to an object anymore, but original chan object still referenced by weise
    System.out.println(weise.givenName); // print the givenName of weise: "Jacky"
    System.out.println(weise.familyName); // print the familyName of weise: "Chan"

    weise.givenName = null; // given name of weise is now null (btw, the string "Kong-sang" can eventually be disposed by GC)
    System.out.println(weise.givenName); // print the givenName of weise: null

    weise = null; // variable weise now does not point to an object anymore, original chan object no longer used, will eventually be disposed
    // System.out.println(weise.givenName); // this would crash: we do not point to any object
    // System.out.println(weise.familyName); // this would crash: we do not point to any object
  }
}
```

Objects in Expressions

- When creating a new class, we have created a new *type*

Objects in Expressions

- When creating a new class, we have created a new *type*
- We can use this type as if it was a "native" Java type

Objects in Expressions

- When creating a new class, we have created a new *type*
- We can use this type as if it was a "native" Java type
- We can have variables of the type and assign values to them

Objects in Expressions

- When creating a new class, we have created a new *type*
- We can use this type as if it was a "native" Java type
- We can have variables of the type and assign values to them
- We can have expressions returning an instance of the type

Objects in Expressions

- When creating a new class, we have created a new *type*
- We can use this type as if it was a "native" Java type
- We can have variables of the type and assign values to them
- We can have expressions returning an instance of the type
- We can use the type for parameters of methods

Objects in Expressions

- When creating a new class, we have created a new *type*
- We can use this type as if it was a "native" Java type
- We can have variables of the type and assign values to them
- We can have expressions returning an instance of the type
- We can use the type for parameters of methods
- We can use it also as return type for functions

Objects in Expressions

- When creating a new class, we have created a new *type*
- We can use this type as if it was a "native" Java type
- We can have variables of the type and assign values to them
- We can have expressions returning an instance of the type
- We can use the type for parameters of methods
- We can use it also as return type for functions
- Let us explore this power with a slightly larger example

A class for representing complex numbers, i.e., $\mathbb{C}$

Listing: A class for representing complex numbers, i.e., $\mathbb{C}$

```
/** a class representing a complex number z∈ℂin rectangular form z=α+βi */
public class ComplexNumber {

  double realPart; // the real part α of the complex number
  double imaginaryPart; // the imaginary part β

  /** create a new complex number, sets real and imaginary part to 0d */
  public ComplexNumber() {
  }

  /** create a new complex number setting the real part, leaving imaginary part 0 */
  public ComplexNumber(final double _realPart) {
    this(); // optional: first invoke the parameterless first constructor
    this.realPart = _realPart;
  }

  /** create a new complex number setting both real and imaginary part */
  public ComplexNumber(final double _realPart, final double _imaginaryPart) {
    this(_realPart); // first invoke the one-parameter constructor setting real part
    this.imaginaryPart = _imaginaryPart;
  }
}
```

A class implementing mathematical operations over $\mathbb{C}$

Listing: A class implementing mathematical operations over $\mathbb{C}$

```
/** a calculator for complex numbers in ℂ */
public class ComplexNumberCalculator {

  /** add two complex numbers, return new complex number with result (αx+αy) + (βx+βy)i */
  static ComplexNumber add(ComplexNumber x, ComplexNumber y) {
    return new ComplexNumber((x.realPart + y.realPart), (x.imaginaryPart + y.imaginaryPart));
  }

  /** subtract two complex numbers (x-y), return new complex number with result (αx-αy) + (βx-βy)i */
  static ComplexNumber subtract(ComplexNumber x, ComplexNumber y) {
    return new ComplexNumber(//
        (x.realPart - y.realPart), //
        (x.imaginaryPart - y.imaginaryPart));
  }

  /** multiply two complex numbers (x*y), return new complex number with result (αxαy-βxβy) + (αxβy+βxαy)i */
  static ComplexNumber multiply(ComplexNumber x, ComplexNumber y) {
    double a1 = x.realPart, b1 = x.imaginaryPart;
    double a2 = y.realPart, b2 = y.imaginaryPart;

    return new ComplexNumber(((a1 * a2) - (b1 * b2)), //
        ((a1 * b2) + (b1 * a2)));
  }

  /** divide two complex numbers (x/y), return new complex number with result (αxαy+βxβy)/(αy²+βy²) + (αyβx-βyαx)/(αy²+βy²)i */
  static ComplexNumber divide(ComplexNumber x, ComplexNumber y) {
    double a1 = x.realPart, b1 = x.imaginaryPart;
    double a2 = y.realPart, b2 = y.imaginaryPart;

    return new ComplexNumber((((a1 * a2) + (b1 * b2)) / ((a2 * a2) + (b2 * b2))), //
        (((a2 * b1) - (b2 * a1)) / ((a2 * a2) + (b2 * b2))));
  }

  /** print a complex number to stdout */
  static void println(ComplexNumber x) {
    System.out.print(x.realPart);
    System.out.print(" + "); //$NON-NLS-1$
    System.out.print(x.imaginaryPart);
    System.out.println('i');
  }
}
```

A class testing these mathematical operations

Listing: A class testing these mathematical operations

```
/** testing the complex number calculator */
public class ComplexNumberTest {

  /** The main routine
   *  @param args
   *          we ignore this parameter */
  public static final void main(String[] args) {
    ComplexNumber a, b, res;

    ComplexNumberCalculator.println(// print the result of...
        a = new ComplexNumber(20d)); //... the construction of a real-valued complex number
    ComplexNumberCalculator.println(// print the result of...
        b = new ComplexNumber(1d, -2d)); //...the construction of a complex number of value 1 − 2i

    ComplexNumberCalculator.println(// print the result of (20 ∗ (1 − 2i)) − (1 − 2i) = 19 ∗ (1 − 2i) = 19 − 38i
        res = ComplexNumberCalculator.subtract(ComplexNumberCalculator.multiply(a, b), b));

    ComplexNumberCalculator.println(// print the result of (19−38i)/(1−2i) = 19 = 19 − 0i
        ComplexNumberCalculator.divide(res, b));

    ComplexNumberCalculator.println(// print the result of
        ComplexNumberCalculator.divide(// ((19−38i)∗(19−38i))/((19−38i)∗(1−i)) = (19−38i)/(1−i)
            ComplexNumberCalculator.multiply(res, res), // = 28.5 − 9.5i
            ComplexNumberCalculator.multiply(res, new ComplexNumber(1d, -1d)))); // using new in expression
  }
}
```

Objects Identity

- When comparing object variables/expressions using `==`, we do not compare the values of the object variables but only the references

Objects Identity

- When comparing object variables/expressions using `==`, we do not compare the values of the object variables but only the references
- Two objects may have the exact same values in their fields (member variables), but `==` returns `false` since they are not the same object

Objects Identity

- When comparing object variables/expressions using `==`, we do not compare the values of the object variables but only the references
- Two objects may have the exact same values in their fields (member variables), but `==` returns `false` since they are not the same object
- We distinguish same and equal

Objects Identity

- When comparing object variables/expressions using `==`, we do not compare the values of the object variables but only the references
- Two objects may have the exact same values in their fields (member variables), but `==` returns `false` since they are not the same object
- We distinguish same and equal:
  - Two cars of the same brand, color, and with identical specification may be equal

Objects Identity

- When comparing object variables/expressions using `==`, we do not compare the values of the object variables but only the references
- Two objects may have the exact same values in their fields (member variables), but `==` returns `false` since they are not the same object
- We distinguish same and equal:
  - Two cars of the same brand, color, and with identical specification may be equal
  - Yet, they may belong to two different people, i.e., they are not the same

Objects Identity

- When comparing object variables/expressions using `==`, we do not compare the values of the object variables but only the references
- Two objects may have the exact same values in their fields (member variables), but `==` returns `false` since they are not the same object
- We distinguish same and equal:
  - Two cars of the same brand, color, and with identical specification may be equal
  - Yet, they may belong to two different people, i.e., they are not the same
- Two objects may have the same field values, but the two objects are located at different places in memory.

Objects Identity

- When comparing object variables/expressions using `==`, we do not compare the values of the object variables but only the references
- Two objects may have the exact same values in their fields (member variables), but `==` returns `false` since they are not the same object
- We distinguish same and equal:
  - Two cars of the same brand, color, and with identical specification may be equal
  - Yet, they may belong to two different people, i.e., they are not the same
- Two objects may have the same field values, but the two objects are located at different places in memory.
- They can be equal, but they are not the same

Objects Identity

- When comparing object variables/expressions using `==`, we do not compare the values of the object variables but only the references
- Two objects may have the exact same values in their fields (member variables), but `==` returns `false` since they are not the same object
- We distinguish same and equal:
  - Two cars of the same brand, color, and with identical specification may be equal
  - Yet, they may belong to two different people, i.e., they are not the same
- Two objects may have the same field values, but the two objects are located at different places in memory.
- They can be equal, but they are not the same
- `==` return `true` only if two variables are the same

Objects Identity

- When comparing object variables/expressions using `==`, we do not compare the values of the object variables but only the references
- Two objects may have the exact same values in their fields (member variables), but `==` returns `false` since they are not the same object
- We distinguish same and equal:
  - Two cars of the same brand, color, and with identical specification may be equal
  - Yet, they may belong to two different people, i.e., they are not the same
- Two objects may have the same field values, but the two objects are located at different places in memory.
- They can be equal, but they are not the same
- `==` return `true` only if two variables are the same
- Iff two variables point to the same object, `==` returns `true`

A class testing object identity via ==

Listing: A class testing object identity via ==

```java
/** test objects whether they are the same via == */
public class IdentityTest {

  /** The main routine
   *  @param args
   *          we ignore this parameter */
  public static void main(String[] args) {
    PersonWithConstructor personA = new PersonWithConstructor("Weise", "Thomas"); //$NON-NLS-1$ //$NON-NLS-2$
    System.out.println(personA.familyName + '␣' + personA.givenName);

    PersonWithConstructor personB = new PersonWithConstructor("Weise", "Thomas"); //$NON-NLS-1$ //$NON-NLS-2$
    System.out.println(personB.familyName + '␣' + personB.givenName);

    System.out.println(personA == personB); // false: the variables hold same data, but are different objects!

    personB = personA; // now personB and personA reference the same object
    System.out.println(personA == personB); // true: both variables now reference the same instance

    ComplexNumber c1 = new ComplexNumber(0d); // create a complex number 0+0i
    ComplexNumber c2 = new ComplexNumber(0d, 0d); // create a complex number 0+0i

    System.out.println(c1 == c2); // false: the two variables hold the same data, but are different objects!
    System.out.println((c1 = c2) == c2); // true: both variables now reference the same instance
  }
}
```

Arrays of Objects

- We can create arrays of objects, in the same way we did before with primitive types

Arrays of Objects

- We can create arrays of objects, in the same way we did before with primitive types

Listing: An array of PersonWithConstructor objects

```
/** An array of instances of class PersonWithConstructor class representing a person. */
public class PersonWithConstructorArray {

  /** The main routine
   *  @param args   we ignore this parameter */
  public static final void main(String[] args) {

    PersonWithConstructor[] array = { // create and initialize an array
        new PersonWithConstructor("Weise", "Thomas"), //$NON-NLS-1$//$NON-NLS-2$
        new PersonWithConstructor("Chan", "Jacky"), //$NON-NLS-1$//$NON-NLS-2$
        new PersonWithConstructor("Onegin", "Eugene"), //$NON-NLS-1$//$NON-NLS-2$
    };

    for (PersonWithConstructor element : array) { // fast read-only iteration
      System.out.println(element.familyName);
    } // Weise \n Chan \n Onegin
  }
}
```

Using `null` In Code

- Just a quick example for using `null` and a test for `null` in an array

Using `null` In Code

- Just a quick example for using `null` and a test for `null` in an array

Listing: An array of PersonWithConstructor objects with `null` element

```
/** An array of instances of class PersonWithConstructor class representing a person. */
public class PersonWithConstructorArrayWithNull {

  /** The main routine
   *  @param args
   *           we ignore this parameter */
  public static final void main(String[] args) {

    PersonWithConstructor[] array = { // create and initialize an array
        new PersonWithConstructor("Weise", "Thomas"), //$NON-NLS-1$//$NON-NLS-2$
        new PersonWithConstructor("Chan", "Jacky"), //$NON-NLS-1$//$NON-NLS-2$
        null, // nothing
        new PersonWithConstructor("Onegin", "Eugene"), //$NON-NLS-1$//$NON-NLS-2$
    };

    for (PersonWithConstructor element : array) { // fast read-only iteration
      if (element != null) { // check for null, we would get an error when trying null.familyName...
        System.out.println(element.familyName);
      } else { // ok, null -> print something elese
        System.out.println("Missing element!");//$NON-NLS-1$
      }
    } // Weise \n Chan \n Missing element! \n Onegin
  }
}
```

`static` vs. instance variables

- In Lesson 12: *Static Variables*, we learned about `static` variables

`static` vs. instance variables

- In Lesson 12: *Static Variables*, we learned about `static` variables
- `static` variables are different from instance variables

`static` vs. instance variables

- In Lesson 12: *Static Variables*, we learned about `static` variables
- `static` variables are different from instance variables
- A `static` variable exists "once per class"

`static` vs. instance variables

- In Lesson 12: *Static Variables*, we learned about `static` variables
- `static` variables are different from instance variables
- A `static` variable exists "once per class"
- An instance variable exists "once per object"

Using `static` and instance variables

Listing: A program using both `static` and instance variables

```
/** A class representing a person with unique counted id. */
public class PersonWithIDStatic {

  /** the static variable counting the person objects */
  static int idCounter = 0;

  /** the family name of the person */
  String familyName;
  /** the given name of the person */
  String givenName;
  /** the id of the person */
  int id;

  /** create a person record and set its name */
  PersonWithIDStatic(String _familyName, String _givenName) {
    this.familyName = _familyName;
    this.givenName = _givenName;
    this.id = (++PersonWithIDStatic.idCounter); // increase the id counter and set the id of his object
  }

  /** The main routine
   *  @param args
   *           we ignore this parameter */
  public static final void main(String[] args) {

    System.out.println(idCounter); // print the id counter: 0

    PersonWithIDStatic weise = new PersonWithIDStatic("Weise", "Thomas"); //$NON-NLS-1$//$NON-NLS-2$

    System.out.println(idCounter); // print the id counter: 1

    PersonWithIDStatic chan = new PersonWithIDStatic("Chan", "Jacky"); //$NON-NLS-1$//$NON-NLS-2$

    System.out.println(weise.givenName); // print the givenName of weise
    System.out.println(weise.familyName); // print the familyName of weise
    System.out.println(weise.id); // print the id of weise: 1
    System.out.println(idCounter); // print the id counter: 2

    System.out.println(chan.familyName); // print the familyName of Chan
    System.out.println(chan.givenName); // print the given name of Chan
    System.out.println(chan.id); // print the id of chan: 2
    System.out.println(weise.id); // print the id of weise: 1
    System.out.println(idCounter); // print the id counter: 2
  }
}
```

Summary

- We have learned about objects, i.e., instances of classes

Summary

- We have learned about objects, i.e., instances of classes
- They can have instance variables

Summary

- We have learned about objects, i.e., instances of classes
- They can have instance variables
- They can have (multiple) constructors

Summary

- We have learned about objects, i.e., instances of classes
- They can have instance variables
- They can have (multiple) constructors
- We have learned what `this` and `null` are

Summary

- We have learned about objects, i.e., instances of classes
- They can have instance variables
- They can have (multiple) constructors
- We have learned what `this` and `null` are
- We have learned what references are and how `=` actually copies a reference, not an object

Summary

- We have learned about objects, i.e., instances of classes
- They can have instance variables
- They can have (multiple) constructors
- We have learned what `this` and `null` are
- We have learned what references are and how `=` actually copies a reference, not an object
- We have learned how we can pass as parameters to and return values from methods

Summary

- We have learned about objects, i.e., instances of classes
- They can have instance variables
- They can have (multiple) constructors
- We have learned what `this` and `null` are
- We have learned what references are and how `=` actually copies a reference, not an object
- We have learned how we can pass as parameters to and return values from methods
- We have discussed the life cycle of an object and what garbage collection is

Summary

- We have learned about objects, i.e., instances of classes
- They can have instance variables
- They can have (multiple) constructors
- We have learned what `this` and `null` are
- We have learned what references are and how `=` actually copies a reference, not an object
- We have learned how we can pass as parameters to and return values from methods
- We have discussed the life cycle of an object and what garbage collection is
- We have learned that all java arrays and strings are actually objects, too

Summary

- We have learned about objects, i.e., instances of classes
- They can have instance variables
- They can have (multiple) constructors
- We have learned what `this` and `null` are
- We have learned what references are and how `=` actually copies a reference, not an object
- We have learned how we can pass as parameters to and return values from methods
- We have discussed the life cycle of an object and what garbage collection is
- We have learned that all java arrays and strings are actually objects, too
- And we have learned that we can have arrays of objects

谢谢

Thank you

Thomas Weise [汤卫思]
tweise@hfuu.edu.cn
http://iao.hfuu.edu.cn

Hefei University, South Campus 2
Institute of Applied Optimization
Shushan District, Hefei, Anhui,
China

Caspar David Friedrich, “Der Wanderer über dem Nebelmeer”, 1818
http://en.wikipedia.org/wiki/Wanderer_above_the_Sea_of_Fog