Programmieren II

Modifiers & Overloading

Alexander Fraser

fraser@cl.uni-heidelberg.de

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

May 8, 2014

Outline

1. Recap
   - Syntax of a class definition
   - Object declaration vs. instantiation
2. Constructors
3. Modifiers
   - static
   - private, public etc.
4. Overloading methods

Outline

1. Recap
   - Syntax of a class definition
   - Object declaration vs. instantiation
2. Constructors
3. Modifiers
   - static
   - private, public etc.
4. Overloading methods

What is an object?

- Everything is an object
- Objects store data, provide methods
- Classes are used to instantiate objects of the same type
- You should know the difference between declaration, creation and assignment

Object Oriented Design

Coming up with classes

- Idea: a problem is separated into multiple, independent components that are self-contained
- Ideally: one component == one class
- OOP is no exact science, there are always multiple acceptable solutions!

Hints

- Objects should be treated as a “Black Box” (encapsulation)
- Communication between objects with well-defined interfaces (methods). No direct access to instance variables
- Interaction between objects should be minimized

Syntax of a class definition

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

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

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

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

listing-06.java

Constructors

Constructors

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

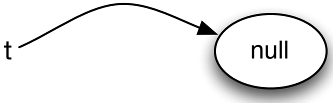
- same name as the class name
- have parameters like regular methods
- do not have an explicit return type or return value
- used to initialize objects (instance variables)

Declaration vs. instantiation

Declaration, instantiation, assignment

- Class: `Triangle.java`
- Reference variable: same type as the class: `Triangle t;` (**declaration**)
- Instantiation of a new object: `new Triangle();`
- Assigning the object to the reference variable: `t = new Triangle();`

Variable declaration

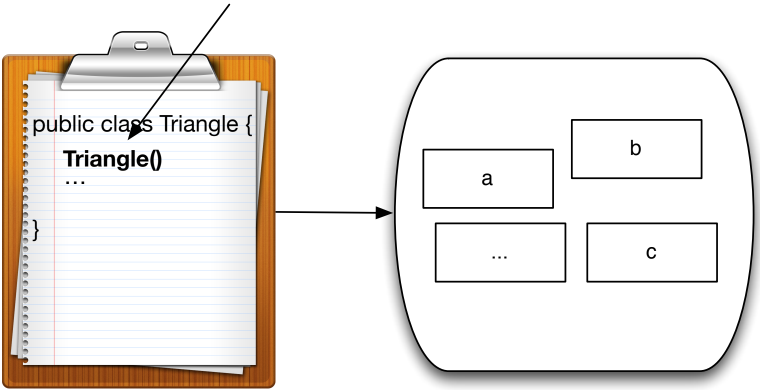
Statement	Result
Triangle t;	

Declaration of a new variable of type "Triangle". Does not point to any object at this point.

Object instantiation

Statement new Triangle();

Result



```
public class Triangle {
    Triangle()
    ...
}
```

class definition

instantiated object

What happens during instantiation?

Initializing an object

- Memory (on the heap) is reserved for all instance variables
- Instance variables are set to default values (in two slides)
- Instance initialization blocks are executed
- Constructor is executed
- Initialized object is returned

Objects on the heap

Objects on the heap

- Head First Java (2^{nd} ed.), p. 58.
- Exercise: p. 64

Default values for instance variables

- If the value of an instance variable is not explicitly stated upon instantiation, a default value is assigned

Default values

Data type	default value
byte,short,int,long	0
float,double	0.0
char	'u0000'
boolean	false
reference types	null

Options for initializing instance variables

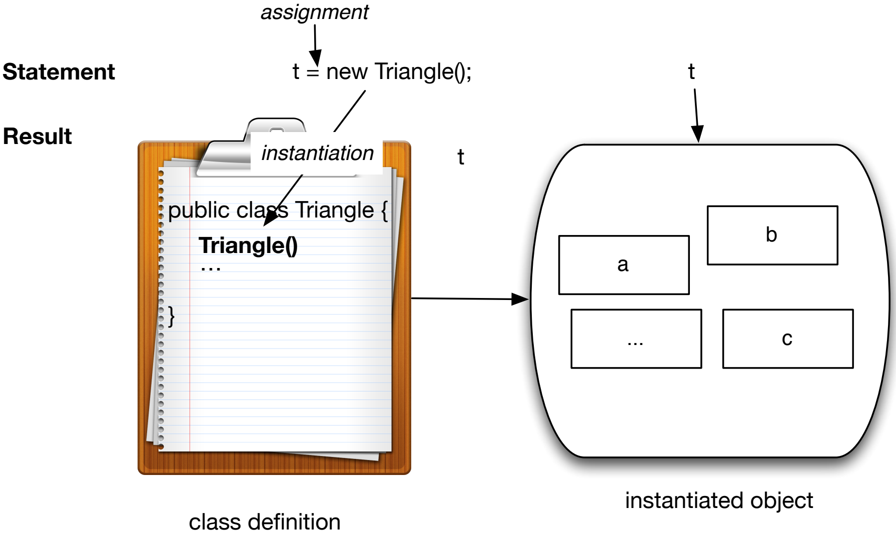
Summary

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

1. Constructor: set values directly or use parameters
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

⇒ example: Moodle

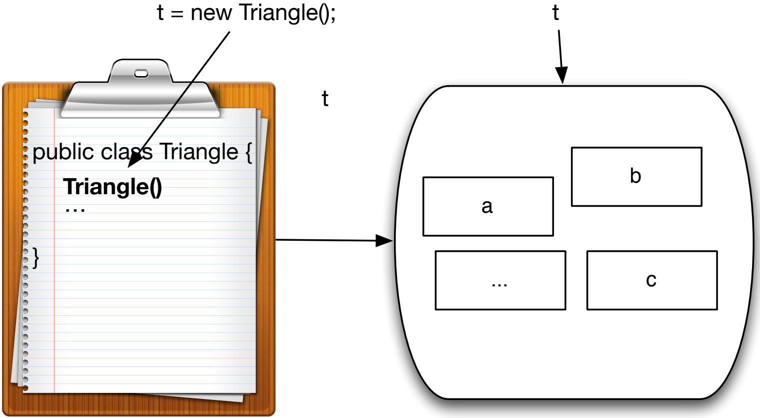
Instantiation and assignment in one step



Reference variables

Statement

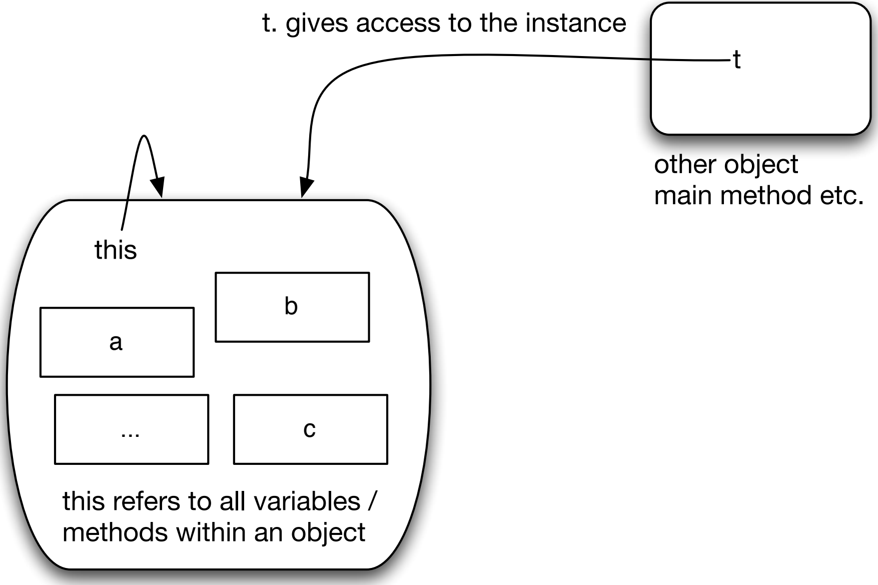
Result



class definition

instantiated object

This vs. reference variable



t. gives access to the instance

other object
main method etc.

this refers to all variables /
methods within an object

This

this

- this is used to refer to the object itself
- this is a reference variable 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 ⇒ Parameter name

Instance vs. local variables

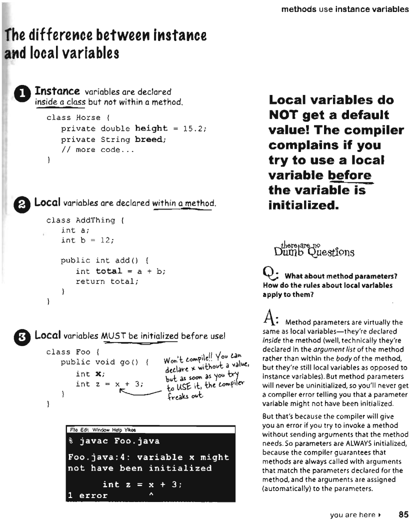
Local variables

- Declared within a method
- Do **NOT** get a default value
- → Need to be initialized

Instance variables

- Declared inside a class but not within a method
- Instance variables are initialized automatically

Instance vs. local variables



methods use instance variables

The difference between instance and local variables

1 **Instance** variables are declared inside a class but not within a method.

```
class Horse {
   private double height = 15.2;
   private String breed;
   // more code...
}
```

2 **Local** variables are declared within a method.

```
class AddThing {
   int a;
   int b = 12;

   public int add() {
      int total = a + b;
      return total;
   }
}
```

3 **Local** variables MUST be initialized before use!

```
class Foo {
   public void go() {
      int x;
      int z = x + 3;
   }
}
```

Won't compile!! You can declare x without a value, but as soon as you try to USE it, the compiler freaks out

File Edit Window Help Yikes

```
% javac Foo.java
Foo.java:4: variable x might
not have been initialized
         int z = x + 3;
1 error          ^
```

Local variables do NOT get a default value! The compiler complains if you try to use a local variable before the variable is initialized.

there are no Dumb Questions

Q: **What about method parameters? How do the rules about local variables apply to them?**

A: Method parameters are virtually the same as local variables—they're declared *inside* the method (well, technically they're declared in the *argument list* of the method rather than within the *body* of the method, but they're still local variables as opposed to instance variables). But method parameters will never be uninitialized, so you'll never get a compiler error telling you that a parameter variable might not have been initialized.

But that's because the compiler will give you an error if you try to invoke a method without sending arguments that the method needs. So parameters are ALWAYS initialized, because the compiler guarantees that methods are always called with arguments that match the parameters declared for the method, and the arguments are assigned (automatically) to the parameters.

you are here ▸ 85

HF, p. 85

Summary

Objects

- What is a class? What's an object?
- What are the elements of a class?
- Declaration, creation and assignment
- Constructors
- Instance initializers

Goals of this session

Constructors

- More on constructors

Modifiers

- Learn about `static` methods and variables
- Learn about visibility of methods and instance variables
- Learn how to overload (same name, different parameter list) methods

Outline

1. Recap
   - Syntax of a class definition
   - Object declaration vs. instantiation

2. Constructors

3. Modifiers
   - static
   - private, public etc.

4. Overloading methods

Constructors

- Normally, we don't want to explicitly initialize all variables for each created instance
- Instead: use a constructor to automatically initialize an object upon its creation
- Constructors do not have any explicit return value
- Constructors have the same name as their class

Constructors with parameters

```java
class Triangle2 {
    double a,b,c;
    double alpha,beta,gamma;

    Triangle2(double alpha, double beta, double gamma){
        if ( checkAngularSum(alpha, beta, gamma) ){
            this.alpha = alpha;
            this.beta = beta;
            this.gamma = gamma;
        } else {
            System.out.println("This cannot be a triangle..");
            System.exit( 1 );
        }
    }

    ...
}
```

Triangle2.java

Calling the constructor – example

```java
public class TriangleDemo4 {

    public static void main(String[] args) {

        Triangle t = new Triangle( 60, 60, 60);
        System.out.println( t.alpha );
        // prints 60
    }
}
```

TriangleDemo4.java

Constructors

Default constructor

- Is provided automatically for each class
- No parameter (`<classname>()`)
- No class-specific initialization

Caution

The default constructor is only provided, if no other constructors are defined!

Advice for constructors

- Constructors should be simple: small number of parameters, primitive whenever possible
- Consider whether to use static factory methods (later) instead of constructors
- Use the same name for constructor parameters and properties, if appropriate
- Do *minimal* work in the constructor!
- If useful: add a default constructor

Outline

1 Recap
- Syntax of a class definition
- Object declaration vs. instantiation

2 Constructors

3 Modifiers
- static
- private, public etc.

4 Overloading methods

Modifiers: `static`

Static variables

- In general, variables of a class are assigned to objects the class instantiates (e.g. the variable name of the class `Cat`).
- Sometimes it makes sense to define variables for the class itself, e.g. if the value is identical for all objects of a class

⇒ modifier `static`

Modifiers: static

Static variables

```java
public class Circle {

    int radius;
    static double pi = 3.14159;

    public Circle (int r) {
      radius = r;
    }

    public getArea () {
        return (radius * radius * pi);
    }
}
```

code/Circle.java

Modifiers: `static`

Static variables

- can be accessed directly via the class itself without instantiating an object

```
System.out.println( Circle.pi );
```

Modifiers: static

Static variables

- can be accessed directly via the class itself without instantiating an object

```
System.out.println( Circle.pi );
```

Modifiers: `static`

Static variables

- can be accessed directly via the class itself without instantiating an object

```
System.out.println( Circle.pi );
```

Modifiers: static + final

Static variables vs. constants

- static variables are not constants: they can be changed and the changed values applies to all objects of the class
- to define variables as read-only (i.e. as constants), they need to be declared as final

  ```
  static final double PI = 3.14159d; // static + final
  final int yearOfBirth = 1999;      // only final
  ```

- Convention: constants are written in capital letters

Modifiers: static

Static methods

- are also accessed directly via the class itself
- do not have access to non-static elements (why?)

Example

```
public class Circle {
...
   public static double getPi () {
      return pi;
   }
...
}
```

Modifiers: `public` and `private`

Visibility

- `public` and `private` (and `protected` – later) specify the visibility of class elements (variables, methods)
- visibility determines who is able to access an element

Modifiers: `public` and `private`

`public`

- `public` elements are visible everywhere
- `public` variables can be accessed and changed from outside the class
- `public` methods can be called from outside the class

⇒ `public` elements are interfaces of an object to the outside world

Modifiers: public and private

private

- private elements are visible to objects of the class only
- private variables are accessed and assigned within the class
- private methods can only be accessed in methods within the same class

⇒ private allows "hiding" details of the implementation
⇒ private declaration protects from unregulated access to class details

Modifiers: public and private

Variables that can't be used?

- just because an instance variable is not directly accessible, does not mean it can't be used
- use public accessors ("getters")

```
public double getAlpha() {
   return this.alpha;
}
```

- and "setters"

```
public void setAlpha( double alpha ) {
   this.alpha = alpha;
}
```

Modifiers: `public` and `private`

Advantages of `private` variable + `public` setter/getter

- Check for inconsistencies
- Implementation independent from interface

Modifiers

Other modifiers

- We'll cover these later:
  - `protected`
  - `final`
  - `abstract`
- not covered in this lecture
  - `native`, `strict`, `synchronized`, `transient`, `volatile`

Real developers encapsulate!

Without encapsulation…

- Reference variable: `Cat myCat = new Cat();`
- This would be ok: `myCat.height = 27;`
- This would be disastrous: `myCat.height = -1;`

→ we need to protect the cat (and all other objects) from invalid size (and other variable) changes!

Encapsulation

- **Always** choose the most restrictive visibility possible
- To allow changes: use setters and getters
- This process is called **encapsulation**
- Mark instance variables **private**
- Mark getters/setters **public**

Outline

1. Recap
   - Syntax of a class definition
   - Object declaration vs. instantiation
2. Constructors
3. Modifiers
   - static
   - private, public etc.
4. Overloading methods

Overloading

- Java allows to declare multiple methods with the same name within one class, if their parameter list differs
- This is called overloading of methods
- Distinction between different versions of an overloaded method is done by the type and number of parameters
- Overloaded methods can have a different return type!
- Constructors can also be overloaded

Overloading

- Multiple methods with the same name but different parameters
- Parameters differ in number and/or type

- Often used for constructors:

```
Cat() { ... }
Cat( int age ) { ... }
Cat( int age, boolean isGrumpy ) { ... }
Cat( int age, boolean isGrumpy, double weight ) { ... }
```

Overloading: example

- Imagine three different alternatives for purring in the Cat class

```
public void purr(String s) { ... }
public void purr(Sound s) { ... }
public void purr() { ... }
```

- Depending on the provided parameters, the appropriate method is called and executed
- E.g. catVar.purr(); vs. catVar.purr("Purr!")

Overloading

Overloading constructors

```
Cat( int age ) { this.age = age; }

Cat( int age, boolean isGrumpy ) {
   this.age = age;
   this.isGrumpy = isGrumpy;
}

Cat( int age, boolean isGrumpy, double weight ) {
   this.age = age;
   this.isGrumpy = isGrumpy;
   this.weight = weight;
}
```

Avoid duplicate code!!

Overloading

Overloading constructors

```
Cat( int age ) { this.age = age; }

Cat( int age, boolean isGrumpy ) {
   this.age = age;
   this.isGrumpy = isGrumpy;
}

Cat( int age, boolean isGrumpy, double weight ) {
   this.age = age;
   this.isGrumpy = isGrumpy;
   this.weight = weight;
}
```

Avoid duplicate code!!

Overloading

Overloading constructors

```
Cat( int age ) { this.age = age; }

Cat( int age, boolean isGrumpy ) {
   this.age = age;
   this.isGrumpy = isGrumpy;
}

Cat( int age, boolean isGrumpy, double weight ) {
   this.age = age;
   this.isGrumpy = isGrumpy;
   this.weight = weight;
}
```

Avoid duplicate code!!

Overloading

- Duplicating initialization code in multiple methods should be avoided
- Instead: constructor chaining
- Call simple constructors from more complex ones
- Syntax: `this( ... );`
- Needs to be the first statement in a constructor!

Overloading

```
Cat( int age ) { this.age = age; }

Cat( int age, boolean isGrumpy ) {
   this.age = age;

   this.isGrumpy = isGrumpy;
}

Cat( int age, boolean isGrumpy, double weight ) {
   this.age = age;
   this.isGrumpy = isGrumpy;

   this.weight = weight;
}
```

Overloading

```
Cat( int age ) { this.age = age; }

Cat( int age, boolean isGrumpy ) {
   this.age = age;
   this.isGrumpy = isGrumpy;
}

Cat( int age, boolean isGrumpy, double weight ) {
   this.age = age;
   this.isGrumpy = isGrumpy;
   this.weight = weight;
}
```

Overloading

```
Cat( int age ) { this.age = age; }

Cat( int age, boolean isGrumpy ) {
   this( age );
   this.isGrumpy = isGrumpy;
}

Cat( int age, boolean isGrumpy, double weight ) {
   this.age = age;
   this.isGrumpy = isGrumpy;
   this.weight = weight;
}
```

Overloading

```
Cat( int age ) { this.age = age; }

Cat( int age, boolean isGrumpy ) {
   this( age );

   this.isGrumpy = isGrumpy;
}

Cat( int age, boolean isGrumpy, double weight ) {
   this.age = age;
   this.isGrumpy = isGrumpy;

   this.weight = weight;
}
```

Overloading

```
Cat( int age ) { this.age = age; }

Cat( int age, boolean isGrumpy ) {
   this( age );

   this.isGrumpy = isGrumpy;
}

Cat( int age, boolean isGrumpy, double weight ) {
   this( age, isGrumpy );


   this.weight = weight;
}
```

What you should know after this session

- What does `static` mean?
- What's the difference between `private` and `public`?
- Why do we need `private` variables and methods?
- How and why do we overload methods? (...and what does overloading actually mean)

Literature

- Sierra, K. & Bates, B.
  Head First Java. (Mostly Ch. 3 & 4)
  O'Reilly Media, 2005.

- Ullenboom, Ch.
  Java ist auch eine Insel. (Ch. 5)
  Galileo Computing, 2012.

- Eckel, B. (for reference)
  Thinking in Java. (Ch. 2 & 4)
  Prentice Hall, 2006.