Programmieren II

More Inheritance & Strings

Alexander Fraser

fraser@cl.uni-heidelberg.de

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

May 15, 2014

Outline

1. Recap: Unified Modeling Language (UML)
2. Recap: Inheritance
   - Inheritance
   - Initialization
   - Hiding variables
   - Overwriting methods
3. Multiple Inheritance
4. The class `Object` and reading the Java API
5. Strings

Outline

1 Recap: Unified Modeling Language (UML)

2 Recap: Inheritance
- Inheritance
- Initialization
- Hiding variables
- Overwriting methods

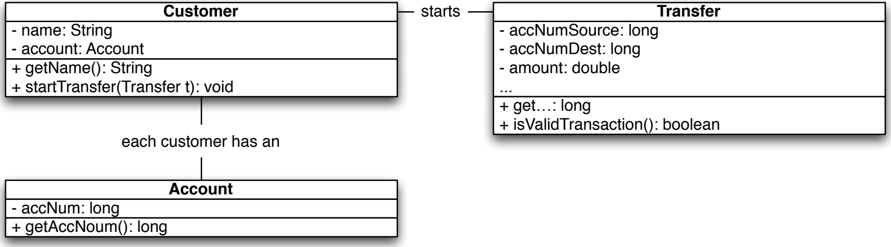
3 Multiple Inheritance

4 The class `Object` and reading the Java API

5 Strings

UML diagrams – example

Modeling associations (simplified)



Customer
- name: String
- account: Account

+ getName(): String
+ startTransfer(Transfer t): void

starts

Transfer
- accNumSource: long
- accNumDest: long
- amount: double

...

+ get…: long
+ isValidTransaction(): boolean

each customer has an

Account
- accNum: long

+ getAccNoum(): long

Recap – Inheritance

1. Constructors and Inheritance
2. Hiding variables
3. Overwriting methods
4. Use of packages and `package-private`
5. `protected`, `final`

Outline

1 Recap: Unified Modeling Language (UML)

2 Recap: Inheritance

- Inheritance
- Initialization
- Hiding variables
- Overwriting methods

3 Multiple Inheritance

4 The class `Object` and reading the Java API

5 Strings

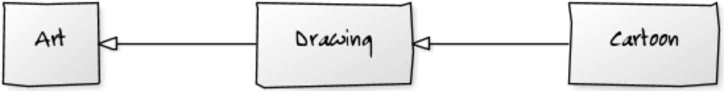
Class hierarchies

- Subclasses extend their superclass: they implement more specific or additional properties and behaviors
- Properties that are common to multiple classes are implemented in the super class
- **Advantage:** avoids duplicate code
  - less redundancy
  - changes at one place
  - effective for all subclasses (by inheritance. . .)

Effects of Inheritance

- Subclass inherits functionality of the super class
  - Variables (fields)
  - Methods
- only visible elements are inherited!

Demonstrating constructor calls I



Class hierarchy for Art – Drawing – Cartoon

Art.java

```
/** Base class */
class Art {
    /** default constructor */
    Art() {
        System.out.println("Art constructor");
    }
}
```

code/Art.java

Demonstrating constructor calls II

Drawing.java

```
class Drawing extends Art {
    /** default constructor */
    Drawing() {
        System.out.println("Drawing constructor");
    }
}
```

code/Drawing.java

Demonstrating constructor calls III

Cartoon.java

```
public class Cartoon extends Drawing {
    /** default constructor */
    public Cartoon() {
        System.out.println(”Cartoon constructor”);
    }
    public static void main(String[] args) {
        Cartoon x = new Cartoon();
    }
}
```

code/Cartoon.java

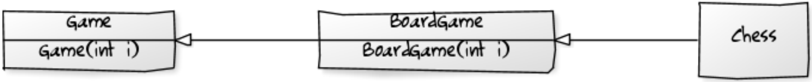
Constructor calls

- Construction happens from the base “outward”
- Default constructor in `Cartoon` is created that calls base constructors

Constructors with arguments I

- If the base class does not have a default constructor or
- If you want to call another base class constructor

→ explicit call to base class constructor via `super(...)`



Constructors with arguments II

Game.java

```
class Game {
    /** Constructor with arguments (e.g. number of players) */
    Game(int i) {
        System.out.println("Game constructor");
    }
}
```

code/Game.java

Constructors with arguments III

BoardGame.java

```
class BoardGame extends Game {
    BoardGame(int i) {
        super(i);
        System.out.println("BoardGame constructor");
    }
}
```

code/BoardGame.java

Constructors with arguments IV

Chess.java

```
public class Chess extends BoardGame {

    Chess() {
        super(2);
        System.out.println("Chess constructor");
    }
    public static void main(String[] args) {
        Chess x = new Chess();
    }
}
```

code/Chess.java

Constructor with parameters

Explicitly calling the base constructor

- Without `super`: compiler error. Why?
- Default constructor does not exist in `Game.java`
- → explicit constructor call necessary

Constructor with parameters

Explicitly calling the base constructor

- Without `super`: compiler error. Why?
- Default constructor does not exist in `Game.java`
- → explicit constructor call necessary

Inheritance

```java
public class Person {
  private String name;

  public Person( String name ) {
    this.name = name;
  }

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

Person.java

Inheritance

```java
public class FaultyTeacherPatched extends Person {
    private int office;
    private String name; // hides Person's name

    public FaultyTeacherPatched(String name, int office) {
        super( name );
        this.name = "Prof. " + name;
        this.office = office;
    }

    public static void main( String[] args ) {
        FaultyTeacherPatched t = new FaultyTeacherPatched( "XY"
            , 505 );
        System.out.println( t.getName() ); // prints Person's
            name
        System.out.println( t.name ); // prints FTP's name
    }
}
```

FaultyTeacherPatched.java – evil!

Inheritance

Hiding variables

- Instance variable `name` from `Person` not visible in `TeacherHiding`
- `TeacherHiding` hides `name` by re-defining it
- Each instance of `TeacherHiding` has two `name`s!

Inheritance

```java
public class OverridingTeacher extends Person {
  private int office;

  public OverridingTeacher(String name, int office) {
    super( name );
    this.office = office;
  }

  @Override
  public String getName() {
    return "Prof. " + super.getName();
  }

  public static void main( String[] args ) {
    OverridingTeacher t = new OverridingTeacher("XY",505);
    System.out.println(t.getName());
  }
}
```

OverridingTeacher.java

Inheritance

- `super.` ...allows access to (visible) elements of the super-class
- "Overriding": methods can be overwritten in a sub-class
- Overriding is not Hiding:
  - Hidden fields can be made visible again by type casting
  - Overwritten methods remain associated with the object of the sub-class
- Hiding is rarely (never?) necessary – instead: new variable and/or overriding

Packages & `protected`

- Use of packages and `package-private`
- `protected`, `final`

Example: good practice I

Source: Thinking in Java, 3^{rd} ed., Ch. 6

Example: Villain.java

```java
class Villain {
    private String name;
    protected void set(String nm) { name = nm; }
    public Villain(String name) { this.name = name; }
    public String toString() {
        return "I’m a Villain and my name is " + name;
    }
}
```

code/Villain.java

Example: good practice II

Example: Orc.java

```java
public class Orc extends Villain {
    private int orcNumber;
    public Orc(String name, int orcNumber) {
        super(name);
        this.orcNumber = orcNumber;
    }
    public void change(String name, int orcNumber) {
        set(name); // Available because it's protected
        this.orcNumber = orcNumber;
    }
    public String toString() {
        return "Orc " + orcNumber + ": " + super.toString();
    }
    public static void main(String[] args) {
        Orc orc = new Orc("Limburger", 12);
        System.out.println(orc);
        orc.change("Bob", 19);
        System.out.println(orc);
    }
}
```

Exercises

Overloading methods

Create a class with a method that is overloaded three times. Inherit a new class, add a new overloading of the method, and show that all four methods are available in the derived class.

Constructors and Inheritance

Create a base class with only a nondefault constructor, and a derived class with both a default (no-arg) and nondefault constructor. In the derived-class constructors, call the base-class constructor

Outline

1 Recap: Unified Modeling Language (UML)

2 Recap: Inheritance
- Inheritance
- Initialization
- Hiding variables
- Overwriting methods

3 **Multiple Inheritance**

4 The class `Object` and reading the Java API

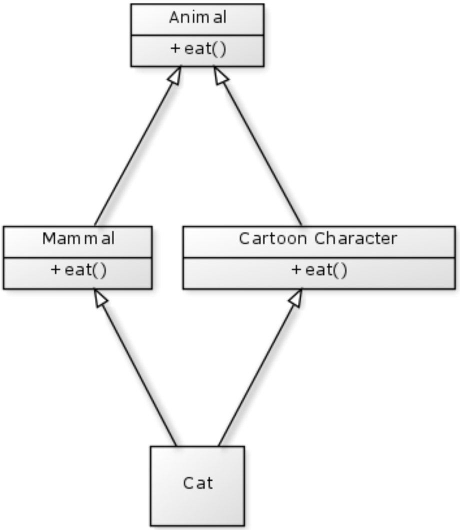
5 Strings

Multiple Inheritance I

- Class extends more than one super class
- `Cat` could inherit from *Cartoon character* and *Pet* and *Mammal*
- **You can't do that in Java!**
- If you want to do multiple inheritance: learn C++! ;)

Deadly diamond of death

Deadly diamond of death



- If `Cat` calls `eat()`, which method is actually called?
- Paradox!

Outline

1 Recap: Unified Modeling Language (UML)

2 Recap: Inheritance
- Inheritance
- Initialization
- Hiding variables
- Overwriting methods

3 Multiple Inheritance

4 The class `Object` and reading the Java API

5 Strings

The class `Object`

- Each class in Java has a super class
- But we've seen classes without `extends` ...
- Class `Object` is the root of all class hierarchies in Java
  - Only class without a super class
  - Indirect super class of all classes
  - Direct super class of classes without explicit `extends`

Object methods

- `boolean equals( Object obj )`
- `String toString()`
- `String getClass()`
- `int hashCode()`
- `Object clone()`

- and some others...

Object methods

```
boolean equals( Object obj )
```

- implements an equivalence relation for objects
- *intended* meaning: "structural" equality, not necessarily same object identity
- But: `equals` method in `Object` tests for object identity:

  `o1.equals(o2)` $\Leftrightarrow$ `o1==o2` for `Object` instances `o1, o2`
- to define useful equivalence relations, `equals` is almost always overwritten (alongside the hashCode method . . .)

Object methods

```
boolean equals( Object obj )
```

- equals is reflexive
- equals is symmetric
- equals is transitive
- equals is consistent: multiple calls: same result
- For x $\neq$ null, `x.equals( null )` always returns `false`

Object methods

`String toString()`

- returns a “textual” representation of an object
- should always be overwritten
  - generic implementation in `Object` quite useless, e.g. `Person@635b9e68`
  - informative string representation useful (sometimes even necessary) for testing

Object methods

- `clone()` returns a (shallow) copy of the object (i.e. a new object)
- `hashCode()` defines a mapping from objects to numbers
- We'll come back to `hashCode()` for Collections (especially hash maps ≈ dictionaries)

- http://docs.oracle.com/javase/7/docs/api/lang/Object.html

Outline

1 Recap: Unified Modeling Language (UML)

2 Recap: Inheritance
- Inheritance
- Initialization
- Hiding variables
- Overwriting methods

3 Multiple Inheritance

4 The class `Object` and reading the Java API

5 **Strings**

The class String

- Objects of the class java.lang.String (standard library)
- can be written as literals

```
String s1 = "I’m a String object";
char[] chars = { ’h’, ’e’, ’r’, ’o’, ’e’, ’s’ };
String s2 = new String( chars );
String s3 = new String( s1 );
boolean eqTest = s3 == s1; // false!!
```

- Strings – like all objects – are compared with equals (not with ==)

The class String

- Objects of the class `java.lang.String` (standard library)
- can be written as literals

```
String s1 = "I’m a String object”;
char[] chars = { ’h’, ’e’, ’r’, ’o’, ’e’, ’s’ };
String s2 = new String( chars );
String s3 = new String( s1 );
boolean eqTest = s3 == s1; // false!!
```

- Strings – like all objects – are compared with `equals` (not with `==`)

The class String

- Objects of the class java.lang.String (standard library)
- can be written as literals

```
String s1 = "I’m a String object";
char[] chars = { ’h’, ’e’, ’r’, ’o’, ’e’, ’s’ };
String s2 = new String( chars );
String s3 = new String( s1 );
boolean eqTest = s3 == s1; // false!!
```

- Strings – like all objects – are compared with equals (not with ==)

The class String

- Objects of the class java.lang.String (standard library)
- can be written as literals

```
String s1 = "I'm a String object";
char[] chars = { 'h', 'e', 'r', 'o', 'e', 's' };
String s2 = new String( chars );
String s3 = new String( s1 );
boolean eqTest = s3 == s1; // false!!
```

- Strings – like all objects – are compared with equals (not with ==)

The class String

- Objects of the class java.lang.String (standard library)
- can be written as literals

```
String s1 = "I'm a String object";
char[] chars = { 'h', 'e', 'r', 'o', 'e', 's' };
String s2 = new String( chars );
String s3 = new String( s1 );
boolean eqTest = s3 == s1; // false!!
```

- Strings – like all objects – are compared with equals (not with ==)

The class String

- Operator for string concatenations: +:

  $$\texttt{"Hello" + " " + "World"} \rightsquigarrow \texttt{"Hello World"}$$

- Can operators be overloaded by the user? No.

The class String

- Objects of the class java.lang.String (standard library)
- can be written as literals

```
String s1 = "I’m a String object";
char[] chars = { ’h’, ’e’, ’r’, ’o’, ’e’, ’s’ };
String s2 = new String( chars );
String s3 = new String( s1 );
boolean eqTest = s3 == s1; // false!!
```

- Strings – like all objects – are compared with equals (not with ==)

The class String

- Objects of the class java.lang.String (standard library)
- can be written as literals

```
String s1 = ”I’m a String object”;
char[] chars = { ’h’, ’e’, ’r’, ’o’, ’e’, ’s’ };
String s2 = new String( chars );
String s3 = new String( s1 );
boolean eqTest = s3 == s1; // false!!
```

- Strings – like all objects – are compared with equals (not with ==)

The class String

- Objects of the class java.lang.String (standard library)
- can be written as literals

```
String s1 = ”I’m a String object”;
char[] chars = { ’h’, ’e’, ’r’, ’o’, ’e’, ’s’ };
String s2 = new String( chars );
String s3 = new String( s1 );
boolean eqTest = s3 == s1; // false!!
```

- Strings – like all objects – are compared with equals (not with ==)

The class String

- Objects of the class java.lang.String (standard library)
- can be written as literals

```
String s1 = "I’m a String object";
char[] chars = { ’h’, ’e’, ’r’, ’o’, ’e’, ’s’ };
String s2 = new String( chars );
String s3 = new String( s1 );
boolean eqTest = s3 == s1; // false!!
```

- Strings – like all objects – are compared with equals (not with ==)

The class String

- Objects of the class java.lang.String (standard library)
- can be written as literals

```
String s1 = "I’m a String object";
char[] chars = { ’h’, ’e’, ’r’, ’o’, ’e’, ’s’ };
String s2 = new String( chars );
String s3 = new String( s1 );
boolean eqTest = s3 == s1; // false!!
```

- Strings – like all objects – are compared with equals (not with ==)

The class `String`

- Operator for string concatenations: `+`:

  `"Hello" + " " + "World"` $\rightsquigarrow$ `"Hello World"`

- Can operators be overloaded by the user? (E.g. use the + operator to concatenate lists) – No.

The class String

```
charAt       indexOf      contains
startsWith   endsWith     length
toLowerCase  toUpperCase  trim
substring    replace      split
. . .
```

http://download.oracle.com/javase/7/docs/api/java/lang/String.html

The class String

```
char charAt( int index )
```

- returns the character at position index of the string
- throws an exception if index is negative or larger than the length of the string

```
char c = "abc".charAt(1); // c == 'b'
char x = "abc".charAt(3); // IndexOutOfBoundsException!!
```

The class String

```
char charAt( int index )
```

- returns the character at position index of the string
- throws an exception if index is negative or larger than the length of the string

```
char c = "abc".charAt(1); // c == 'b'
char x = "abc".charAt(3); // IndexOutOfBoundsException!!
```

The class String

```
int indexOf( int ch )
```

- returns the position of the first occurrence of character ch in the string
- or -1 if ch does not occur at all

```
int i = "abc".indexOf('a');
char c = "abc".charAt(i); // c == 'a'

int j = "abc".indexOf('d');
```

The class String

```
int indexOf( int ch )
```

- returns the position of the first occurrence of character ch in the string
- or -1 if ch does not occur at all

```
int i = "abc".indexOf('a');
char c = "abc".charAt(i); // c == 'a'

int j = "abc".indexOf('d');
```

The class String

- indexOf is overloaded:

```
int indexOf( int ch )
int indexOf( int ch, int fromIndex )
int indexOf( String str)
int indexOf( String str, int fromIndex )
```

```
int i = "abcba".indexOf('a', 2); // i == 4
int j = "abcba".indexOf("bc");   // j == 1
int k = "abcba".indexOf("a", 2); // k == 4
```

The class String

- indexOf is overloaded:

  ```
  int indexOf( int ch )
  int indexOf( int ch, int fromIndex )
  int indexOf( String str)
  int indexOf( String str, int fromIndex )
  ```

```
int i = "abcba".indexOf('a', 2); // i == 4
int j = "abcba".indexOf("bc");   // j == 1
int k = "abcba".indexOf("a", 2); // k == 4
```

The class String

`boolean contains( CharSequence s )`

- [CharSequence is a super type of String]
- tests whether a character sequence s occurs in the string

```
String sub = "bcb";
String sup = "abcba";
boolean b = sup.contains( sub ); // true
boolean d = sub.contains( sup ); // false
```

The class String

```
boolean contains( CharSequence s )
```

- [CharSequence is a super type of String]
- tests whether a character sequence s occurs in the string

```
String sub = "bcb";
String sup = "abcba";
boolean b = sup.contains( sub ); // true
boolean d = sub.contains( sup ); // false
```

The class String

```
boolean startsWith( String prefix )
boolean endsWith( String suffix )
```

- tests, whether a string starts with `prefix` (`startsWith`) or ends with `suffix` (`endsWith`)

```
”abc”.startsWith( ”a” ) // true
”abc”.startsWith( ”abc” ); // true
”abc”.endsWith( ”bc” ); // true
”abc”.endsWith( ”a” ) // false
```

The class String

```
boolean startsWith( String prefix )
boolean endsWith( String suffix )
```

- tests, whether a string starts with `prefix` (`startsWith`) or ends with `suffix` (`endsWith`)

```
"abc".startsWith( "a" ) // true
"abc".startsWith( "abc" ); // true
"abc".endsWith( "bc" ); // true
"abc".endsWith( "a" ) // false
```

The class String

```
int length( )
```

- returns the length of a string (= number of characters)

```
"".length()    // 0
"abc".length() // 3
"aba".length() // 3
```

The class String

```
int length( )
```

- returns the length of a string (= number of characters)

```
"".length()    // 0
"abc".length() // 3
"aba".length() // 3
```

The class String

```
String toLowerCase()
String toUpperCase()
```

- returns a copy of a String where all characters are converted to lower case (toLowerCase) or upper case (toUpperCase)

```
"abc".toUpperCase() // "ABC"
"aBc".toUpperCase() // "ABC"
"aBc".toLowerCase() // "abc"
```

The class String

```
String toLowerCase()
String toUpperCase()
```

- returns a copy of a String where all characters are converted to lower case (toLowerCase) or upper case (toUpperCase)

```
"abc".toUpperCase() // "ABC"
"aBc".toUpperCase() // "ABC"
"aBc".toLowerCase() // "abc"
```

The class `String`

`String trim()`

- returns a copy of the String where all white spaces at the begin and end are removed

```
"  a".trim()          // "a"
"Bla bla\n".trim()    // removes "\n"
" yada yada ".trim() // "yada yada"
```

The class String

`String trim()`

- returns a copy of the String where all white spaces at the begin and end are removed

```
"  a".trim()          // "a"
"Bla bla\n".trim()    // removes "\n"
" yada yada ".trim() // "yada yada"
```

The class String

```
String substring( int beginIndex)
String substring( int beginIndex, int endIndex)
```

- returns a new String which is a sub-string of the original string
- Sub-string specified by position of the first (and optionally last) character of the original string

```
String sub1 = ”abcba”.substring( 2 );    // sub1 == ”cba”
String sub2 = ”abcba”.substring( 2, 3 ); // sub2==”c”
String sub3 = ”abcba”.substring( 2, 5 ); // sub3==”cba”
String sub4 = ”abcba”.substring( 2, 6 );
                                // IndexOutOfBoundsException
```

The class String

```
String substring( int beginIndex)
String substring( int beginIndex, int endIndex)
```

- returns a new String which is a sub-string of the original string
- Sub-string specified by position of the first (and optionally last) character of the original string

```
String sub1 = "abcba".substring( 2 );    // sub1 == "cba"
String sub2 = "abcba".substring( 2, 3 ); // sub2=="c"
String sub3 = "abcba".substring( 2, 5 ); // sub3=="cba"
String sub4 = "abcba".substring( 2, 6 );
                              // IndexOutOfBoundsException
```

The class String

```
String replace( char oldChar, char newChar )
String replace( CharSequence old, CharSequence repl )
```

- replaces each occurrence of `oldChar / old` by `newChar / repl`
- Replacement from left to right

```
String s1 = ”abcba”;
String s2 = s1.replace( ’a’, ’x’ ); // s2 == ”xbcbx”
s2 = s1.replace( ”a”, ”x” );        // same as with chars
s2 = ”aaa”.replace( ”aa”, ”b” );    // what is s2 now?
```

The class `String`

```
String replace( char oldChar, char newChar )
String replace( CharSequence old, CharSequence repl )
```

- replaces each occurrence of `oldChar` / `old` by `newChar` / `repl`
- Replacement from left to right

```
String s1 = "abcba";
String s2 = s1.replace( 'a', 'x' ); // s2 == "xbcbx"
s2 = s1.replace( "a", "x" );        // same as with chars
s2 = "aaa".replace( "aa", "b" );    // what is s2 now?
```

The class String

```
String[] split( String regex)
```

- splits a string around each occurrence of regex

```
"boo:and foo".split( ":" );    // { "boo", "and foo" }
"boofoobar".split( "oo" );    // { "b", "f", "bar" }
```

The class String

```
String[] split( String regex)
```

- splits a string around each occurrence of regex

```
"boo:and foo".split( ":" );     // { "boo", "and foo" }
"boofoobar".split( "oo" );     // { "b", "f", "bar" }
```

The class String

```
String[] split( String regex, int limit)
```

- `limit` > 0: Maximal length of returned array is `limit`
- `limit` $<= 0$: Split at each occurrence (no limit)
- `limit` $= 0$: Empty Strings at the end of the array are discarded

```
"boo:and foo".split( "o", 2 );  // { "b", "o:and foo" }
"boo:and foo".split( "o", -1 ); // { "b", "", ":and f", "", "" }
"boo:and foo".split( "o" );     // { "b", "", ":and f" }
```

- Slightly confusing. . . Thus: remove at the beginning or the end (first: remove, then: split)

The class `String`

- Apropos:
  Command line arguments in `main`'s `String[] args`
- `> java Bla` one 2 anotherArg 4.5
- Arguments: one, 2, anotherArg, 4.5
- `args` ≈ `"one 2 anotherArg 4.5".split( "\\s+" )`

⇒ String parameter of the `split` method can be any regular expression

The class `String`

- Apropos:
  Command line arguments in `main`'s `String[] args`
- `> java Bla` one 2 anotherArg 4.5
- Arguments: one, 2, anotherArg, 4.5
- `args` ≈ `"one 2 anotherArg 4.5".split( "\\s+" )`

⇒ String parameter of the `split` method can be any regular expression

The class `String`

- Apropos:
  Command line arguments in `main`'s `String[] args`
- `> java Bla` one 2 anotherArg 4.5
- Arguments: one, 2, anotherArg, 4.5
- `args` ≈ `"one 2 anotherArg 4.5".split( "\\s+" )`

⇒ String parameter of the `split` method can be any regular expression

The class `String`

- once created, a String object cannot be changed
- Concatenating 2 Strings results in a new (3.) String object
- Strings live in the so-called String pool of the JVM
- Strings are recycled – true duplicates only by `String( String orig )`

The class String

```
String s = “0”;
for ( int x = 1; x < 10; x++ ) {
   s = s + x;
}
```

⇒ after the exexution of the loop, the String pool contains 10 objects: `"0", "01", ..., "0123456789"`

The class String

```
static String methodA( String s ) {
    s += "x";
    return s;
}
```

⇒ explicit assignment (`s = methodA(s)`) circumvents the problem

What happens, if return value is needed for other parameters?

The class String

```
static Message methodB( String s ) {
   // create Message object using part of s
   Message obj = . . . ;
   // “update” s
   return obj;
}
```

⇒ Update of s within method methodB. . .

The class String

- StringBuilder (not thread-safe, somewhat faster)
- StringBuffer (thread-safe, but less efficient)
- Methods:
  - like String: charAt, length, substring
  - in addition (mutating):

```
StringBuffer delete( int start, int end)
StringBuffer append( . . . stuff. . . )
StringBuffer insert( int offset, . . . stuff. . . )
StringBuffer replace(int start, int end, String s)
...
```

The class String

```
static Message methodB( StringBuilder s ) {
   // create Message object using part of s
   Message obj = . . . ;
   // “update” s, e.g.,
   s.delete( 0, 2 );

   return obj;
}
```

⇒ Object that is used as a parameter, is changed – update also outside of `methodB`

The class String

```
static Message methodB( StringBuilder s ) {
   // create Message object using part of s
   Message obj = . . . ;
   // “update” s, e.g.,
   s.delete( 0, 2 );

   return obj;
}
```

⇒ Object that is used as a parameter, is changed – update also outside of `methodB`

Exercises

Doubling characters

> *Given a string, return a string where for every char in the original, there are two chars.*

`doubleChar(“The”)` → `“TThhee”`

Summing digits

> *Given a string, return the sum of the digits 0-9 that appear in the string, ignoring all other characters. Return 0 if there are no digits in the string.*

`sumDigits(“aa11b33”)` → `8`

Literature

📄 The Java tutorials
`http://docs.oracle.com/javase/tutorial/java/data/strings.html`

📕 Ullenboom, Ch.
Java ist auch eine Insel. (Chapter 4)
Galileo Computing, 2012.