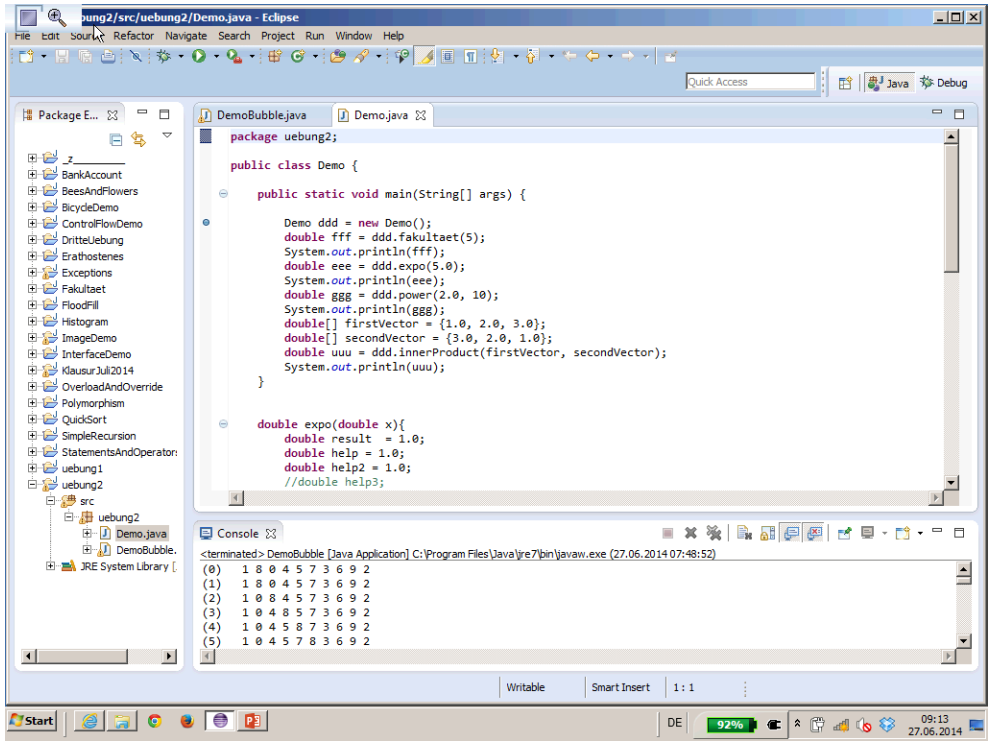
Script **generated by TTT**

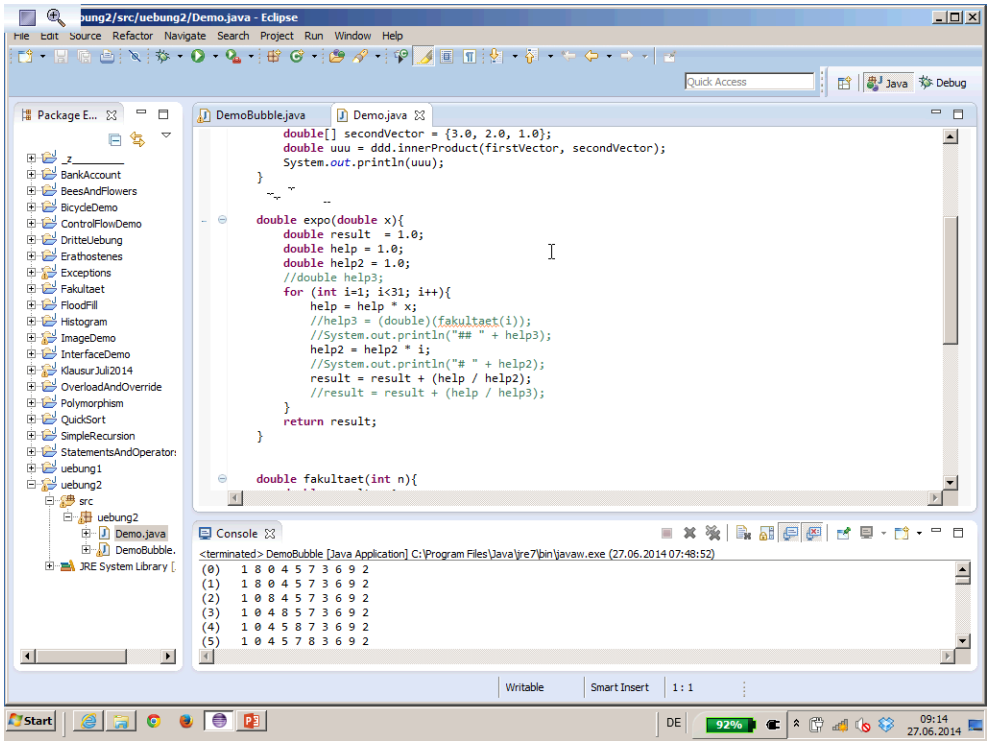
Title: groh: profile1 (27.06.2014)

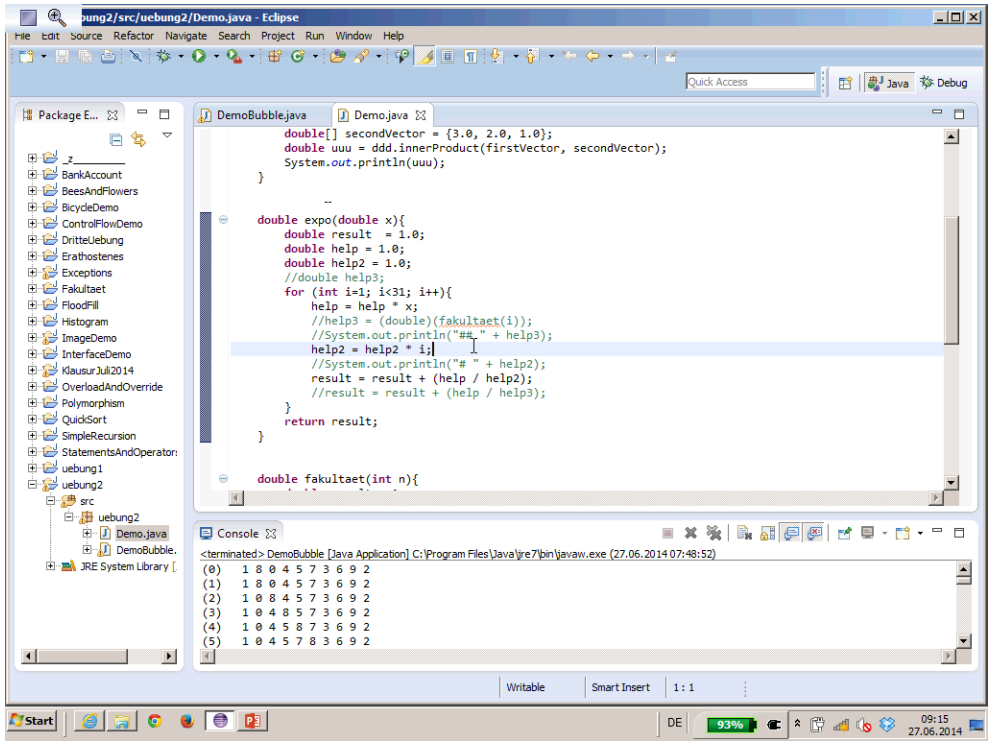
Date: Fri Jun 27 09:13:25 CEST 2014

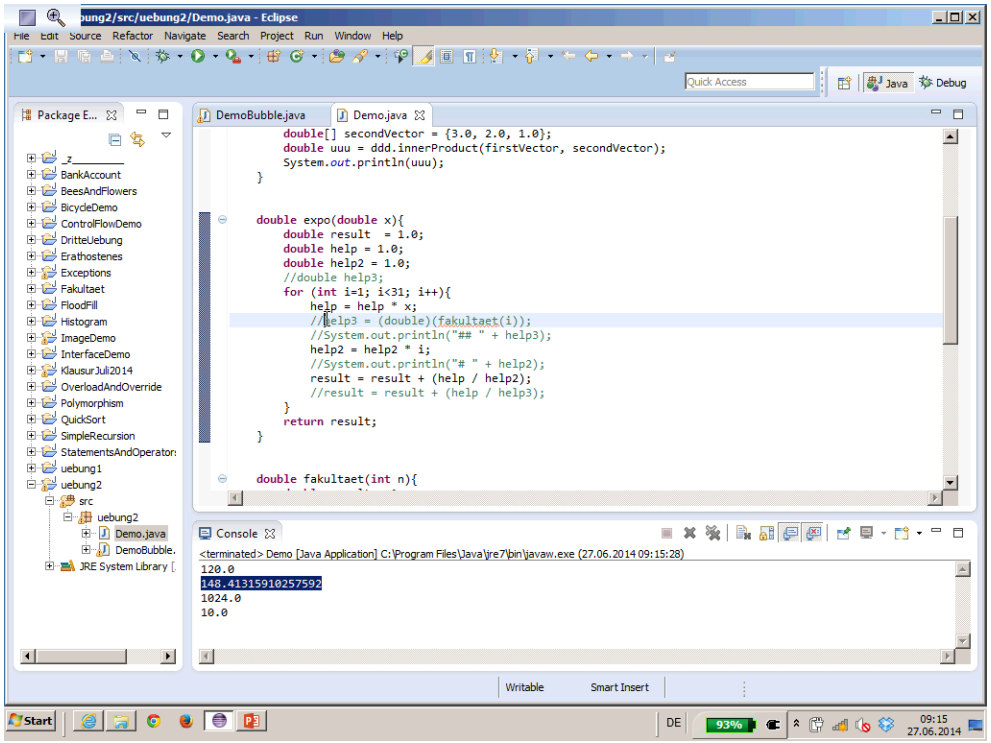
Duration: 90:15 min

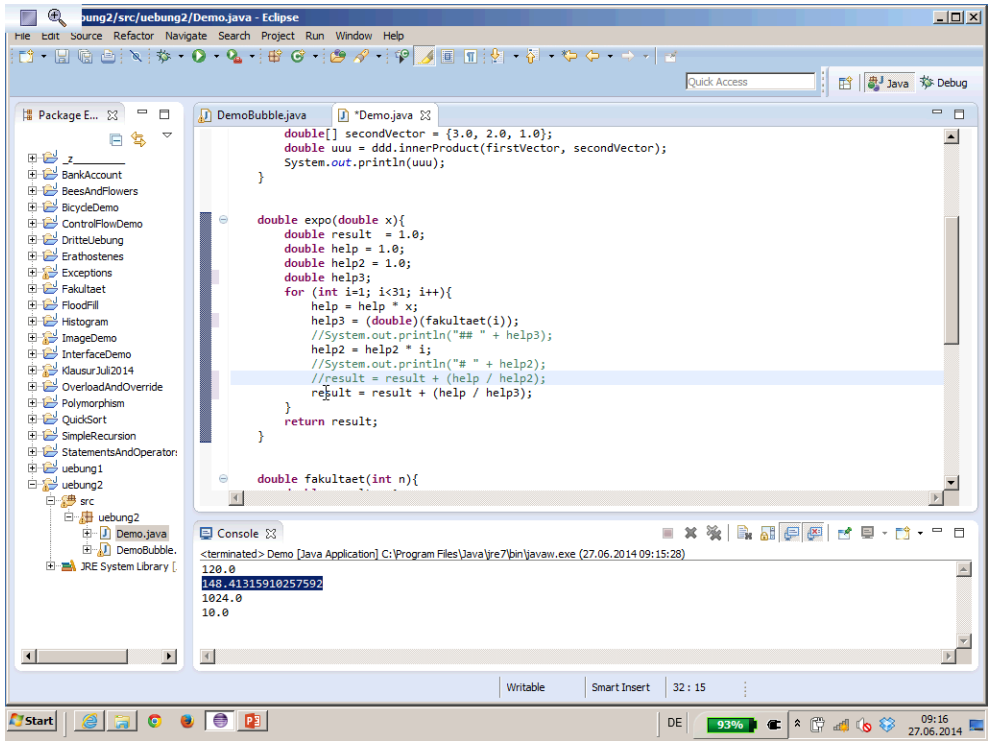
Pages: 73

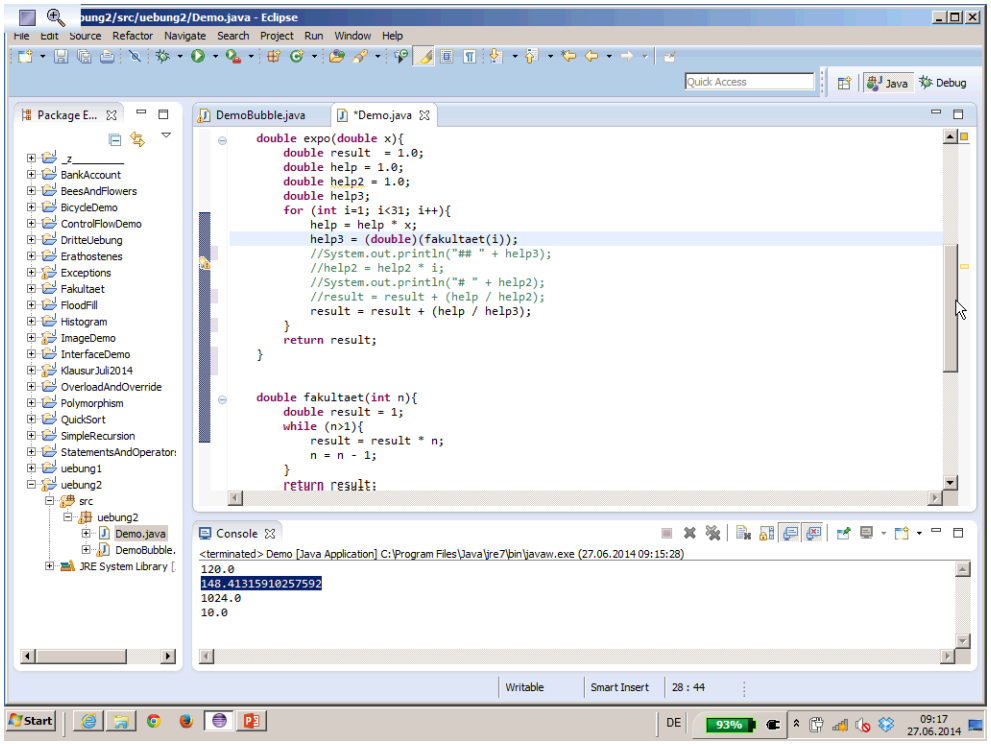


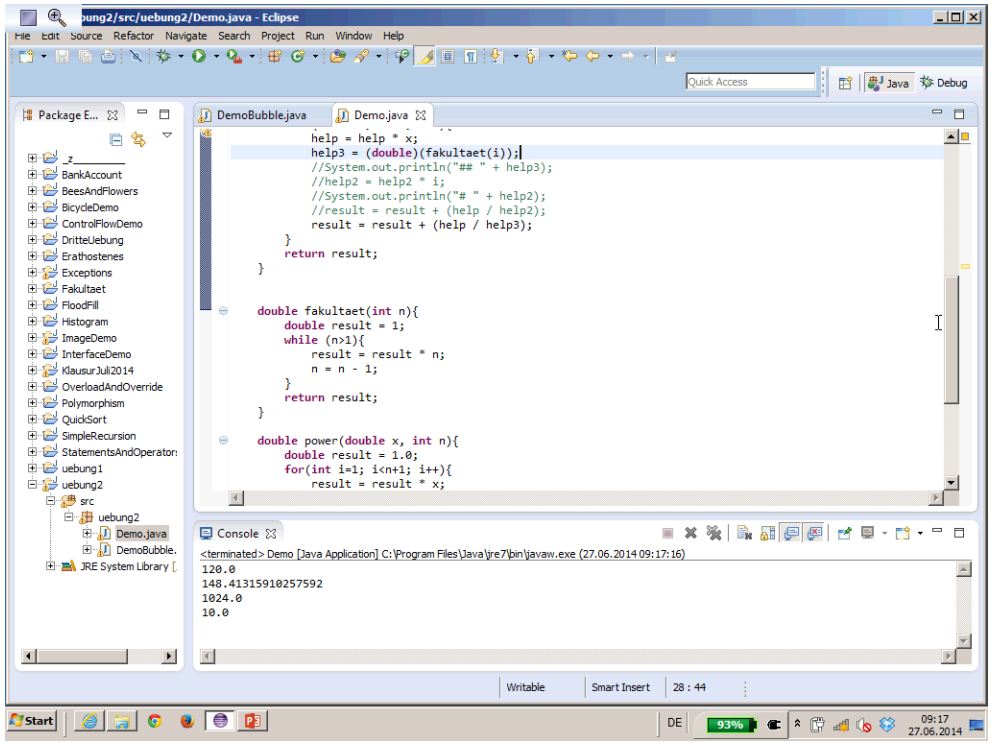


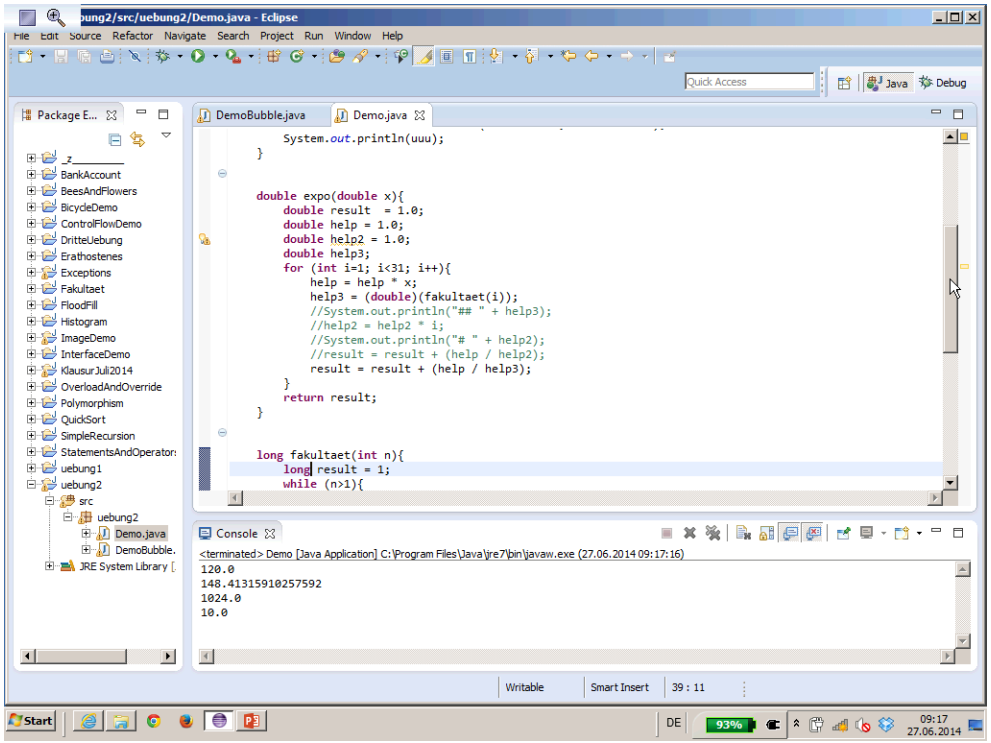


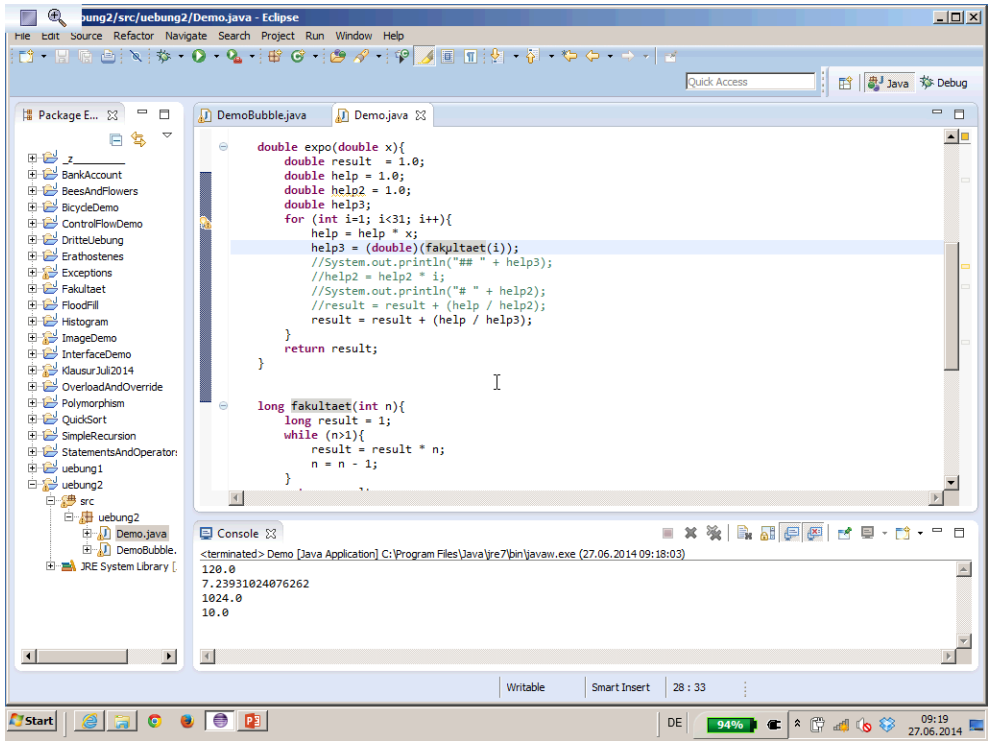

ung2/src/uebung2/Demo.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
Java
Debug
Package E...
DemoBubble.java
Demo.java
double[] secondVector = {3.0, 2.0, 1.0};
double uuu = ddd.innerProduct(firstVector, secondVector);
System.out.println(uuu);
}
double expo(double x){
double result = 1.0;
double help = 1.0;
double help2 = 1.0;
//double help3;
for (int i=1; i<31; i++){
help = help * x;
//help3 = (double)(fakultaet(i));
//System.out.println("## " + help3);
help2 = help2 * i;
//System.out.println("# " + help2);
result = result + (help / help2);
//result = result + (help / help3);
}
return result;
}
double fakultaet(int n){
Console
<terminated> Demo [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (27.06.2014 09:15:28)
120.0
148.41315910257592
1024.0
10.0
Writable
Smart Insert
Start
DE
93%
09:15
27.06.2014

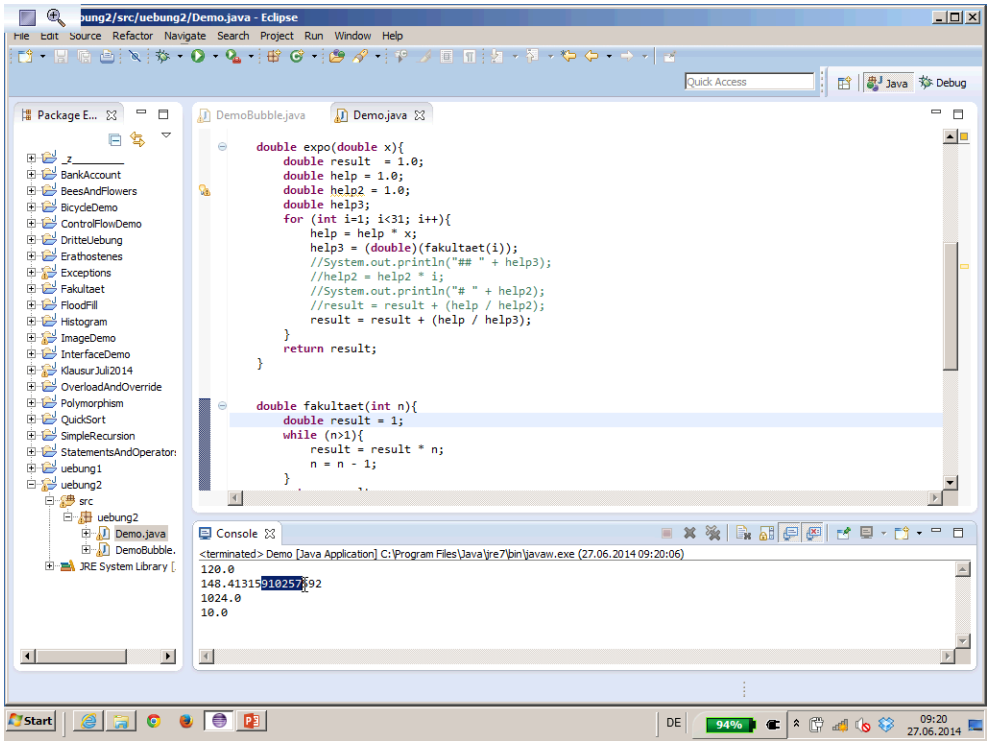

ung2/src/uebung2/Demo.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
Java
Debug
Package E...
DemoBubble.java
*Demo.java
double[] secondVector = {3.0, 2.0, 1.0};
double uuu = ddd.innerProduct(firstVector, secondVector);
System.out.println(uuu);
}
double expo(double x){
double result = 1.0;
double help = 1.0;
double help2 = 1.0;
double help3;
for (int i=1; i<31; i++){
help = help * x;
help3 = (double)(fakultaet(i));
//System.out.println("## " + help3);
help2 = help2 * i;
//System.out.println("# " + help2);
//result = result + (help / help2);
result = result + (help / help3);
}
return result;
}
double fakultaet(int n){
Console
<terminated> Demo [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (27.06.2014 09:15:28)
120.0
148.41315910257592
1024.0
10.0
Writable
Smart Insert
32 : 15
Start
DE
93%
09:16
27.06.2014


ung2/src/uebung2/Demo.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
Java
Debug
Package E...
DemoBubble.java
*Demo.java
double expo(double x){
double result = 1.0;
double help = 1.0;
double help2 = 1.0;
double help3;
for (int i=1; i<31; i++){
help = help * x;
help3 = (double)(fakultaet(i));
//System.out.println("## " + help3);
//help2 = help2 * i;
//System.out.println("# " + help2);
//result = result + (help / help2);
result = result + (help / help3);
}
return result;
}
double fakultaet(int n){
double result = 1;
while (n>1){
result = result * n;
n = n - 1;
}
return result;
Console
<terminated> Demo [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (27.06.2014 09:15:28)
120.0
148.41315910257592
1024.0
10.0
Writable
Smart Insert
28 : 44
Start
DE
93%
09:17
27.06.2014


ung2/src/uebung2/Demo.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
Java
Debug
Package E...
DemoBubble.java
Demo.java
help = help * x;
help3 = (double)(fakultaet(i));
//System.out.println("## " + help3);
//help2 = help2 * i;
//System.out.println("# " + help2);
//result = result + (help / help2);
result = result + (help / help3);
}
return result;
}
double fakultaet(int n){
double result = 1;
while (n>1){
result = result * n;
n = n - 1;
}
return result;
}
double power(double x, int n){
double result = 1.0;
for(int i=1; i<n+1; i++){
result = result * x;
Console
<terminated> Demo [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (27.06.2014 09:17:16)
120.0
148.41315910257592
1024.0
10.0
Writable
Smart Insert
28 : 44
Start
DE
93%
09:17
27.06.2014







3 Classes, Objects, Inheritance

```
class Bicycle {
```

- Class definition (general form):
  modifier `class MyClass extends MySuperClass implements YourInterface1, ..., YourInterfaceN`
  `{ // fields, constructors, methods }`
- *(Access) modifier* (for classes):
  certain combinations of { `public, protected, private, static, final, abstract` }

```
public class MountainBike extends Bicycle {
    public int seatHeight;

    public MountainBike(int startHeight, int startCadence,
                        int startSpeed, int startGear)
    {
        super(startCadence, startSpeed, startGear);
        seatHeight = startHeight;
    }

    public void setHeight(int newValue) {
        seatHeight = newValue;
    }
}
```

Source: [JTutorial]

3 Classes, Objects, Inheritance

```
class Bicycle {
    public int cadence = 0;
    public int speed = 0;
    public int gear = 1;
    ...
}

public class MountainBike extends Bicycle {
    public int seatHeight;
    public MountainBike(int startHeight, int startCadence,
                        int startSpeed, int startGear)
    {
        super(startCadence, startSpeed, startGear);
        seatHeight = startHeight;
    }
    public void setHeight(int newValue) {
        seatHeight = newValue;
    }
}
```

- Field declaration (general form):
  modifier type name ;
- *(Access) modifier* (for fields):
  certain combinations of { `public, protected, private, static, final` }
- *type*: Any primitive or reference type

Source: [JTutorial]

3 Classes, Objects, Inheritance


```
class Bicycle {
    public int cadence = 0;
    public int speed = 0;
    public int gear = 1;
    ...
}

public class MountainBike extends Bicycle {
    public int seatHeight;
    public MountainBike(int startHeight, int startCadence,
                        int startSpeed, int startGear)
    {
        super(startCadence, startSpeed, startGear);
        seatHeight = startHeight;
    }
    public void setHeight(int newValue) {
        seatHeight = newValue;
    }
}
```

- Field declaration (general form):
  modifier type name ;
- *(Access) modifier* (for fields):
  certain combinations of { `public, protected, private, static, final` }
- *type*: Any primitive or reference type

Source: [JTutorial]

3 Classes, Objects, Inheritance


```
class Bicycle {
    public int cadence = 0;
    public int speed = 0;
    public int gear = 1;
    ...
}

public class MountainBike extends Bicycle {
    public int seatHeight;
    public MountainBike(int startHeight, int startCadence,
                        int startSpeed, int startGear)
    {
        super(startCadence, startSpeed, startGear);
        seatHeight = startHeight;
    }
    public void setHeight(int newValue) {
        seatHeight = newValue;
    }
}
```

- Method declaration (general form):
  modifier typeOfReturnValue name (parameter) throwsClause {*
  *statement**
  }
- *(Access) modifier* (for methods):
  certain combinations of { `public, protected, private, static, final, abstract` }
- *typeOfReturnValue*: Any primitive or reference type
- *parameter**: (later)
- *throwsClause**: (later)
- *statement**: statement(s) to execute

Source: [JTutorial]

3 Classes, Objects, Inheritance


```
class Bicycle {
    public int cadence = 0;
    public int speed = 0;
    public int gear = 1;
    ...
}

public class MountainBike extends Bicycle {
    public int seatHeight;
    public MountainBike(int startHeight, int startCadence,
                        int startSpeed, int startGear)
    {
        super(startCadence, startSpeed, startGear);
        seatHeight = startHeight;
    }
    public void setHeight(int newValue) {
        seatHeight = newValue;
    }
}
```

- Method declaration (general form):
  modifier typeOfReturnValue name (parameter) throwsClause {*
  *statement**
  }
- *(Access) modifier* (for methods):
  certain combinations of { `public, protected, private, static, final, abstract` }
- *typeOfReturnValue*: Any primitive or reference type
- *parameter**: (later)
- *throwsClause**: (later)
- *statement**: statement(s) to execute

Source: [JTutorial]

```
class Bicycle {
    public int cadence = 0;
    public int speed = 0;
    public int gear = 1;

    public Bicycle(int startCaden
        gear = startGear;
        cadence = startCadence;
        speed = startSpeed;
    }

    public void changeCadence(int
        cadence = newValue;
    }

    public void chang
        gear = newVa
    }

    public void spee
        speed = spe
    }

    public void apply
        speed = spee
    }
}
```

```
public clas
    public
    public MountainBike(int startHeight, int startCadence,
                        int startSpeed, int startGear)
    {
        super(startCadence, startSpeed, startGear);
        seatHeight = startHeight;
    }

    public void setHeight(int newValue) {
        seatHeight = newValue;
    }
}
```

- Constructor declaration (general form):
  modifier `MyClass` (*parameter**) *throwsClause*
  {
  *statement**
  }
- *(Access) modifier*:
  certain combinations of { `public`, `protected`, `private` }
- *parameter**: (later)
- *throwsClause**: (later)
- *statement**: statement(s) to execute

Source: [JTutorial]

3 Classes, Objects, Inheritance


```
class Bicycle {
    public int cadence = 0;
    public int speed = 0;
    public int gear = 1;

    public Bicycle(int startCaden
        gear = startGear;
        cadence = startCadence;
        speed = startSpeed;
    }

    public void changeCadence(int
        cadence = newValue;
    }

    public void chang
        gear = newVa
    }

    public void spee
        speed = spe
    }

    public void apply
        speed = spee
    }
}
```

```
public clas
    public
    public MountainBike(int startHeight, int startCadence,
                        int startSpeed, int startGear)
    {
        super(startCadence, startSpeed, startGear);
        seatHeight = startHeight;
    }

    public void setHeight(int newValue) {
        seatHeight = newValue;
    }
}
```

- Constructor declaration (general form):
  modifier `MyClass` (*parameter**) *throwsClause*
  {
  *statement**
  }
- *(Access) modifier*:
  certain combinations of { `public`, `protected`, `private` }
- *parameter**: (later)
- *throwsClause**: (later)
- *statement**: statement(s) to execute

Source: [JTutorial]

3 Classes, Objects, Inheritance

Why do we need constructors?

- Ensure **complete** and **consistent** initialization after object creation
- Access (non-default) superclass constructors:
  Construct object according to definition of superclass, then add specifics
- Provide additional constructors for varying use-cases

```
class Bicycle {
  int cadence;
  int speed;
  int gear;

  Bicycle(int c, int s, int g) {
    cadence = c;
    speed = s;
    gear = g;
  }

  Bicycle(int g) {
    cadence = 0;
    speed = 0;
    gear = g;
  }
}
```

```
class Tandem extends Bicycle {
  int numberOfDrivers;

  Tandem(int c, int s, int g, int n) {
    super(c, s, g);
    numberOfDrivers = n;
  }
}
```

3 Classes, Objects, Inheritance

Example:

```
class Person {
  String firstName;
  String lastName;
  long   taxIdent;                          // must be unique!
}
```

```
// Manual initialization, easy to make a mistake (e.g. what about `taxIdent`?)
Person p1     = new Person();
p1.firstName  = "Max";
p1.lastName   = "Mustermann";
p1.taxIdent   = 12345;

Person p2     = new Person();
p2.firstName  = "Fabienne";
p2.lastName   = "Fabelhaft";
p2.taxIdent   = 12345;          // oops!
```

Example:

```
class Person {
  String firstName;
  String lastName;
  long   taxIdent;                                // must be unique!

  Person(String fName, String lName, long tIdent) {
    firstName = fName;
    lastName = lName;

    // Use the given tax identifier `tIdent` only if we can make sure it is unique:
    if (isUniqueTaxIdentifier(tIdent)) {
      taxIdent = tIdent;
    } else {
      System.err.println("Not unique!");
    }
  }
}
```

```
// Complete and consistent.
Person p1 = new Person("Max", "Mustermann", 12345);
Person p2 = new Person("Fabienne", "Fabelhaft", 67890);
```

3 Classes, Objects, Inheritance

Example:

```
class Person {
  String firstName;
  String lastName;
  long   taxIdent;                                // must be unique!

  Person(String fName, String lName, long tIdent) {
    firstName = fName;
    lastName = lName;

    // Use the given tax identifier `tIdent` only if we can make sure it is unique:
    if (isUniqueTaxIdentifier(tIdent)) {
      taxIdent = tIdent;
    } else {
      System.err.println("Not unique!");
    }
  }
}
```

```
// Complete and consistent.
Person p1 = new Person("Max", "Mustermann", 12345);
Person p2 = new Person("Fabienne", "Fabelhaft", 67890);
```

3 Classes, Objects, Inheritance

Example:

```
class Person {
  String firstName;
  String lastName;
  long   taxIdent;                                // must be unique!

  Person(String fName, String lName, long tIdent) {
    firstName = fName;
    lastName = lName;

    // Use the given tax identifier `tIdent` only if we can make sure it is unique:
    if (isUniqueTaxIdentifier(tIdent)) {
      taxIdent = tIdent;
    } else {
      System.err.println("Not unique!");
    }
  }
}
```

```
// Complete and consistent.
Person p1 = new Person("Max", "Mustermann", 12345);
Person p2 = new Person("Fabienne", "Fabelhaft", 67890);
```

3 Classes, Objects, Inheritance

Example:

```
class Person {
  String firstName;
  String lastName;
  long   taxIdent;                                // must be unique!

  Person(String fName, String lName, long tIdent) {
    firstName = fName;
    lastName = lName;

    // Use the given tax identifier `tIdent` only if we can make sure it is unique:
    if (isUniqueTaxIdentifier(tIdent)) {
      taxIdent = tIdent;
    } else {
      System.err.println("Not unique!");
    }
  }
}
```

```
// Complete and consistent.
Person p1 = new Person("Max", "Mustermann", 12345);
Person p2 = new Person("Fabienne", "Fabelhaft", 67890);
```

Example:

```
class Person {
  String firstName;
  String lastName;
  long   taxIdent;                              // must be unique!

  Person(String fName, String lName, long tIdent) {
    firstName = fName;
    lastName = lName;

    // Use the given tax identifier `tIdent` only if we can make sure it is unique:
    if (isUniqueTaxIdentifier(tIdent)) {
      taxIdent = tIdent;
    } else {
      System.err.println("Not unique!");
    }
  }

  Person(String fName, String lName) {
    firstName = fName;
    lastName = lName;

    // A unique tax identifier is created as a side-effect of this constructor:
    taxIdent = createUniqueTaxIdentifier();
  }
}
```

```
// Complete, consistent, convenient ☺
Person p1 = new Person("Max", "Mustermann", 12345);  // first constructor is called
Person p2 = new Person("Fabienne", "Fabelhaft");     // second constructor is called
```

3 Classes, Objects, Inheritance

Example:


```
class Person {
  String firstName;
  String lastName;
  long   taxIdent;                              // must be unique!

  Person(String fName, String lName, long tIdent) {
    firstName = fName;
    lastName = lName;

    // Use the given tax identifier `tIdent` only if we can make sure it is unique:
    if (isUniqueTaxIdentifier(tIdent)) {
      taxIdent = tIdent;
    } else {
      System.err.println("Not unique!");
    }
  }

  Person(String fName, String lName) {
    firstName = fName;
    lastName = lName;

    // A unique tax identifier is created as a side-effect of this constructor:
    taxIdent = createUniqueTaxIdentifier();
  }
}
```

```
// Complete, consistent, convenient ☺
Person p1 = new Person("Max", "Mustermann", 12345);  // first constructor is called
Person p2 = new Person("Fabienne", "Fabelhaft");     // second constructor is called
```


3 Classes, Objects, Inheritance

Example:


```
class Person {
  String firstName;
  String lastName;
  long   taxIdent;                              // must be unique!

  Person(String fName, String lName, long tIdent) {
    firstName = fName;
    lastName = lName;

    // Use the given tax identifier `tIdent` only if we can make sure it is unique:
    if (isUniqueTaxIdentifier(tIdent)) {
      taxIdent = tIdent;
    } else {
      System.err.println("Not unique!");
    }
  }
}
```


```
// Complete and consistent.
Person p1 = new Person("Max", "Mustermann", 12345);
Person p2 = new Person("Fabienne", "Fabelhaft", 67890);
```

3 Classes, Objects, Inheritance

Example:


```
class Person {
  String firstName;
  String lastName;
  long   taxIdent;                              // must

  Person(String fName, String lName, long tIdent) {
    firstName = fName;
    lastName = lName;

    // Use the given tax identifier `tIdent` only if we
    if (isUniqueTaxIdentifier(tIdent)) {
      taxIdent = tIdent;
    } else {
      System.err.println("Not unique!");
    }
  }

  Person(String fName, String lName) {
    firstName = fName;
    lastName = lName;

    // A unique tax identifier is created as a side-effe
    taxIdent = createUniqueTaxIdentifier();
  }
}
```


Which constructor gets called is determined by the number and type of parameters


```
// Complete, consistent, convenient ☺
Person p1 = new Person("Max", "Mustermann", 12345);  // first constructor is called
Person p2 = new Person("Fabienne", "Fabelhaft");     // second constructor is called
```

Example:

```
class Person {
  String firstName;
  String lastName;
  long   taxIdent;                                  // must

  Person(String fName, String lName, long tIdent) {
    firstName = fName;
    lastName = lName;

    // Use the given tax identifier `tIdent` only if we
    if (isUniqueTaxIdentifier(tIdent)) {
      taxIdent = tIdent;
    } else {
      System.err.println("Not unique!");
    }
  }

  Person(String fName, String lName) {
    firstName = fName;
    lastName = lName;

    // A unique tax identifier is created as a side-effe
    taxIdent = createUniqueTaxIdentifier();
  }
}
```

Which constructor gets called is determined by the number and type of parameters

```
// Complete, consistent, convenient ☺
Person p1 = new Person("Max", "Mustermann", 12345);  // first constructor is called
Person p2 = new Person("Fabienne", "Fabelhaft");     // second constructor is called
```


Parameters

- *parameter list*: Passing parameters to methods or constructors

```
int doSomething(int primitiveParameter1,
                double primitiveParameter2,
                SomeClass referenceParameter)
{
        int someInt = 17 + 9;
        primitiveParameter1 = 0;
        referenceParameter = null;
        return someInt;
}
```

body

- Passing primitive type parameters: **Call By Value**
  Changes to parameter have no effect outside of method or constructor

```
int x = 1;
SomeClass someObject = new SomeClass();
int y = doSomething(x, 2.345, someObject);
// At this point, x still has value 1.
```


Parameters

- *parameter list*: Passing parameters to methods or constructors

```
int doSomething(int primitiveParameter1,
                double primitiveParameter2,
                SomeClass referenceParameter)
{
        int someInt = 17 + 9;
        primitiveParameter1 = 0;
        referenceParameter = null;
        return someInt;
}
```

body

- Passing primitive type parameters: **Call By Value**
  Changes to parameter have no effect outside of method or constructor

```
int x = 1;
SomeClass someObject = new SomeClass();
int y = doSomething(x, 2.345, someObject);
// At this point, x still has value 1.
```


Parameters

- *parameter list*: Passing parameters to methods or constructors

```
int doSomething(int primitiveParameter1,
                double primitiveParameter2,
                SomeClass referenceParameter)
{
        int someInt = 17 + 9;
        primitiveParameter1 = 0;
        referenceParameter = null;
        return someInt;
}
```

body

- Passing reference type parameters: **ALSO Call By Value (!!)**
  Changes to parameter have no effect outside of method or constructor

```
int x = 1;
SomeClass someObject = new SomeClass();
int y = doSomething(x, 2.345, someObject);
// At this point, someObject still references
// the same object (someObject != null).
```

Parameters

- *parameter list*: Passing parameters to methods or constructors

```
int doSomething(int primitiveParameter1,
                double primitiveParameter2,
                SomeClass referenceParameter)
{
        int someInt = 17 + 9;
        primitiveParameter1 = 0;
        referenceParameter = null;
        return someInt;
}
```

body

- Passing reference type parameters: **ALSO Call By Value (!!)**
  Changes to parameter have no effect outside of method or constructor

```
int x = 1;
SomeClass someObject = new SomeClass();
int y = doSomething(x, 2.345, someObject);
// At this point, someObject still references
// the same object (someObject != null).
```

3 Classes, Objects, Inheritance

Parameters

- *parameter list*: Passing parameters to methods or constructors

```
int doSomething(int primitiveParameter1,
                double primitiveParameter2,
                SomeClass referenceParameter)
{
        int someInt = 17 + 9;
        primitiveParameter1 = 0;
        referenceParameter = null;
        return someInt;
}
```

body

- Passing reference type parameters: **ALSO Call By Value (!!)**
  Changes to parameter have no effect outside of method or constructor

```
int x = 1;
SomeClass someObject = new SomeClass();
int y = doSomething(x, 2.345, someObject);
// At this point, someObject still references
// the same object (someObject != null).
```

3 Classes, Objects, Inheritance

Parameters

- However, passing reference type parameters can be used to modify objects or arrays with a lasting effect:

```
void doSomethingElse(int[] refParameter) {
  for (int i=0; i<refParameter.length; i++) {
    refParameter[i] = 47;
  }
}
```

```
// Somewhere else...
int[] someArray = { 1, 2, 3, 4, 5 };
doSomethingElse(someArray);
for (int i=0; i<someArray.length; i++) {
  System.out.print("#" + i + ": " + someArray[i]);
}
```

⇨ output will be: `#0: 47 #1: 47 #2: 47 #3: 47 #4: 47`

3 Classes, Objects, Inheritance

Parameters

- However, passing reference type parameters can be used to modify objects or arrays with a lasting effect:

```
void doSomethingElse(int[] refParameter) {
  for (int i=0; i<refParameter.length; i++) {
    refParameter[i] = 47;
  }
}
```

```
// Somewhere else...
int[] someArray = { 1, 2, 3, 4, 5 };
doSomethingElse(someArray);
for (int i=0; i<someArray.length; i++) {
  System.out.print("#" + i + ": " + someArray[i]);
}
```

⇨ output will be: `#0: 47 #1: 47 #2: 47 #3: 47 #4: 47`

Parameters

- However, passing reference type parameters can be used to modify objects or arrays with a lasting effect:

```
void doSomethingElse(int[] refParameter) {
  for (int i=0; i<refParameter.length; i++) {
    refParameter[i] = 47;
  }
}
```

```
// Somewhere else...
int[] someArray = { 1, 2, 3, 4, 5 };
doSomethingElse(someArray);
for (int i=0; i<someArray.length; i++) {
  System.out.print("#" + i + ": " + someArray[i]);
}
```

⇨ output will be: `#0: 47 #1: 47 #2: 47 #3: 47 #4: 47`

3 Classes, Objects, Inheritance

Parameters

- However, passing reference type parameters can be used to modify objects or arrays with a lasting effect:

```
void doSomethingElse(int[] refParameter) {
  for (int i=0; i<refParameter.length; i++) {
    refParameter[i] = 47;
  }
}
```

```
// Somewhere else...
int[] someArray = { 1, 2, 3, 4, 5 };
doSomethingElse(someArray);
for (int i=0; i<someArray.length; i++) {
  System.out.print("#" + i + ": " + someArray[i]);
}
```

⇨ output will be: `#0: 47 #1: 47 #2: 47 #3: 47 #4: 47`

3 Classes, Objects, Inheritance

Parameters

- However, passing reference type parameters can be used to modify objects or arrays with a lasting effect:

```
void doSomethingElse(int[] refParameter) {
  for (int i=0; i<refParameter.length; i++) {
    refParameter[i] = 47;
  }
}
```

```
// Somewhere else...
int[] someArray = { 1, 2, 3, 4, 5 };
doSomethingElse(someArray);
for (int i=0; i<someArray.length; i++) {
  System.out.print("#" + i + ": " + someArray[i]);
}
```

⇨ output will be: `#0: 47 #1: 47 #2: 47 #3: 47 #4: 47`

3 Classes, Objects, Inheritance

Parameters

- However, passing reference type parameters can be used to modify objects or arrays with a lasting effect:

```
void doSomethingElse(int[] refParameter) {
  for (int i=0; i<refParameter.length; i++) {
    refParameter[i] = 47;
  }
}
```

```
// Somewhere else...
int[] someArray = { 1, 2, 3, 4, 5 };
doSomethingElse(someArray);
for (int i=0; i<someArray.length; i++) {
  System.out.print("#" + i + ": " + someArray[i]);
}
```

⇨ output will be: `#0: 47 #1: 47 #2: 47 #3: 47 #4: 47`

3 Classes, Objects, Inheritance

The special value `null` :

- `null` points to "nothing"

```
Bicycle bike1 = new Bicycle();

Bicycle sameBike = bike1;

sameBike = null;
// Has no effect on bike1.
```

memory (simplified model)		
cell nr	cell name	cell content
...	...	...
1149	bike1	<1150>
1150	bike1.cadence	0
1151	bike1.speed	0
1152	bike1.gear	1
...	...	...
1327	sameBike	null
...	...	...

3 Classes, Objects, Inheritance

Returning values

- Methods may return a value (corresponding to declared return type, which may also be `void`) :

```
long faculty(int n) {
  long result = 1;
  for (int i = 2; i <= n; i++) {
    result = result * i;
  }
  return result;
}
```

```
// Somewhere else...
long x = faculty(5);
System.out.println("Faculty of 5 is " + x + ".");
```

- General form: `return` *expression*;

  Returns the **value** of *expression*

3 Classes, Objects, Inheritance

Returning values

- Methods may return a value (corresponding to declared return type, which may also be `void`) :

```
long faculty(int n) {
  long result = 1;
  for (int i = 2; i <= n; i++) {
    result = result * i;
  }
  return result;
}
```

```
// Somewhere else...
long x = faculty(5);
System.out.println("Faculty of 5 is " + x + ".");
```

- General form: `return` *expression*;

  Returns the **value** of *expression*

3 Classes, Objects, Inheritance

Returning values

- Methods may return a value (corresponding to declared return type, which may also be `void`) :

```
long faculty(int n) {
  long result = 1;
  for (int i = 2; i <= n; i++) {
    result = result * i;
  }
  return result;
}
```

```
// Somewhere else...
long x = faculty(5);
System.out.println("Faculty of 5 is " + x + ".");
```

- General form: `return` *expression*;

  Returns the **value** of *expression*

Returning values

- Aside from primitive types, references can be returned as well:

```
Bicycle goGetABike() {
  if (checkForSufficientFunds()) {
    return new Bicycle();
  } else {
    return null;
  }
}

// Call the method from somewhere else...
Bicycle bike = goGetABike();
```

- Corresponding objects/arrays are not "destroyed" (Remember: Reference type variables hold references to the objects, not the objects themselves!)

3 Classes, Objects, Inheritance

Returning values

- Aside from primitive types, references can be returned as well:

```
Bicycle goGetABike() {
  if (checkForSufficientFunds()) {
    return new Bicycle();
  } else {
    return null;
  }
}

// Call the method from somewhere else...
Bicycle bike = goGetABike();
```

- Corresponding objects/arrays are not "destroyed" (Remember: Reference type variables hold references to the objects, not the objects themselves!)

3 Classes, Objects, Inheritance

Returning values

- Aside from primitive types, references can be returned as well:

```
Bicycle goGetABike() {
  if (checkForSufficientFunds()) {
    return new Bicycle();
  } else {
    return null;
  }
}

// Call the method from somewhere else...
Bicycle bike = goGetABike();
```

- Corresponding objects/arrays are not "destroyed" (Remember: Reference type variables hold references to the objects, not the objects themselves!)

3 Classes, Objects, Inheritance

Returning values

- Aside from primitive types, references can be returned as well:

```
Bicycle goGetABike() {
  if (checkForSufficientFunds()) {
    return new Bicycle();
  } else {
    return null;
  }
}

// Call the method from somewhere else...
Bicycle bike = goGetABike();
```

- Corresponding objects/arrays are not "destroyed" (Remember: Reference type variables hold references to the objects, not the objects themselves!)

Returning values

- Aside from primitive types, references can be returned as well:

```
Bicycle goGetABike() {
  if (checkForSufficientFunds()) {
    return new Bicycle();
  } else {
    return null;
  }
}


// Call the method from somewhere else...
Bicycle bike = goGetABike();
```

- Corresponding objects/arrays are not "destroyed" (Remember: Reference type variables hold references to the objects, not the objects themselves!)

3 Classes, Objects, Inheritance

Parameters

- However, passing reference type parameters can be used to modify objects or arrays with a lasting effect:

```
void doSomethingElse(int[] refParameter) {
  for (int i=0; i<refParameter.length; i++) {
    refParameter[i] = 47;
  }
}
```

```
// Somewhere else...
int[] someArray = { 1, 2, 3, 4, 5 };
doSomethingElse(someArray);
for (int i=0; i<someArray.length; i++) {
  System.out.print("#" + i + ": " + someArray[i]);
}
```

⇒ output will be: `#0: 47 #1: 47 #2: 47 #3: 47 #4: 47`

3 Classes, Objects, Inheritance

Calling methods

- Methods can be called from inside and outside a class:

```
public class Bicycle {
  public int cadence = 0;

  public void changeCadence(int newCadence) {
    cadence = newCadence;                  // also: this.cadence
  }

  public void someOtherMethod() {
    changeCadence(5);                      // also: this.changeCadence
  }
}
```

```
public static void main(String[] args) {
  Bicycle bike = new Bicycle();

  bike.changeCadence(10);
  // bike.cadence == 10;

  bike.someOtherMethod();
  // bike.cadence == 5;
}
```

- If needed, objects may refer to themselves as **this**

3 Classes, Objects, Inheritance

Calling methods

- Methods can be called from inside and outside a class:

```
public class Bicycle {
  public int cadence = 0;

  public void changeCadence(int newCadence) {
    cadence = newCadence;                  // also: this.cadence
  }

  public void someOtherMethod() {
    changeCadence(5);                      // also: this.changeCadence
  }
}
```

```
public static void main(String[] args) {
  Bicycle bike = new Bicycle();

  bike.changeCadence(10);
  // bike.cadence == 10;

  bike.someOtherMethod();
  // bike.cadence == 5;
}
```

- If needed, objects may refer to themselves as **this**

Calling methods

- Methods can be called from inside and outside a class:

```
public class Bicycle {
  public int cadence = 0;

  public void changeCadence(int newCadence) {
    cadence = newCadence;                  // also: this.cadence
  }

  public void someOtherMethod() {
    changeCadence(5);                      // also: this.changeCadence
  }
}
```

```
public static void main(String[] args) {
  Bicycle bike = new Bicycle();

  bike.changeCadence(10);
  // bike.cadence == 10;

  bike.someOtherMethod();
  // bike.cadence == 5;
}
```

- If needed, objects may refer to themselves as **`this`**

3 Classes, Objects, Inheritance

Calling methods

- Methods can be called from inside and outside a class:

```
public class Bicycle {
  public int cadence = 0;

  public void changeCadence(int newCadence) {
    cadence = newCadence;                  // also: this.cadence
  }

  public void someOtherMethod() {
    changeCadence(5);                      // also: this.changeCadence
  }
}
```

```
public static void main(String[] args) {
  Bicycle bike = new Bicycle();

  bike.changeCadence(10);
  // bike.cadence == 10;

  bike.someOtherMethod();
  // bike.cadence == 5;
}
```

- If needed, objects may refer to themselves as **`this`**

Access Modifiers (`final`, `static`)

- Access modifiers:
  - **`static`**: field or method bound to class instead of object
    class-method, *class-variable* as opposed to *instance-method*, *instance-variable*
  - **`final`**:
    - fields: cannot be changed (constants)
    - methods: cannot be *overridden* (later)
    - classes: cannot be subclassed

```
final class MyClass {
        static int       sameValueForAllInstances = 3;
        final  int       constantValue = 5;
        static final int constantValueForAllInstances = 7;

        static void  methodOne() { /* ... */ }
        final void   methodTwo() { /* ... */ }
        static final void methodThree() { /* ... */ }
}
```


Access Modifiers & Packages

- Access modifiers:
  - **`public`**: Can be accessed / invoked by anybody
  - **`private`**: Can only be accessed / invoked from within same class
  - **`protected`**: Can only be accessed / invoked from within same class and its subclasses
  - <no modifier>: Can be accessed / invoked from within same package

- Packages:
  - Encapsulate a set of classes and interfaces
  - Hierarchical organization
  - Declaration: `package myfirstpackage;`
  - Examples: `java.math`, `de.tum.wzw`

3 Classes, Objects, Inheritance

Access Modifiers (`final`, `static`)

- Access modifiers:
  - `static`: field or method bound to class instead of object
    class-method, *class-variable* as opposed to *instance-method*, *instance-variable*
  - `final`:
    - fields: cannot be changed (constants)
    - methods: cannot be *overridden* (later)
    - classes: cannot be subclassed

```
final class MyClass {
        static int       sameValueForAllInstances = 3;
        final  int       constantValue = 5;
        static final int constantValueForAllInstances = 7;

        static void  methodOne() { /* ... */ }
        final void   methodTwo() { /* ... */ }
        static final void methodThree() { /* ... */ }
}
```


3 Classes, Objects, Inheritance

Access Modifiers (`final`, `static`)

- Access modifiers:
  - `static`: field or method bound to class instead of object
    class-method, *class-variable* as opposed to *instance-method*, *instance-variable*
  - `final`:
    - fields: cannot be changed (constants)
    - methods: cannot be *overridden* (later)
    - classes: cannot be subclassed

```
final class MyClass {
        static int       sameValueForAllInstances = 3;
        final  int       constantValue = 5;
        static final int constantValueForAllInstances = 7;

        static void  methodOne() { /* ... */ }
        final void   methodTwo() { /* ... */ }
        static final void methodThree() { /* ... */ }
}
```

3 Classes, Objects, Inheritance

Access Modifiers (`final`, `static`)

- Access modifiers:
  - `static`: field or method bound to class instead of object
    class-method, *class-variable* as opposed to *instance-method*, *instance-variable*
  - `final`:
    - fields: cannot be changed (constants)
    - methods: cannot be *overridden* (later)
    - classes: cannot be subclassed

```
final class MyClass {
        static int       sameValueForAllInstances = 3;
        final  int       constantValue = 5;
        static final int constantValueForAllInstances = 7;

        static void  methodOne() { /* ... */ }
        final void   methodTwo() { /* ... */ }
        static final void methodThree() { /* ... */ }
}
```

3 Classes, Objects, Inheritance

Access Modifiers (`final`, `static`)

- Access modifiers:
  - `static`: field or method bound to class instead of object
    class-method, *class-variable* as opposed to *instance-method*, *instance-variable*
  - `final`:
    - fields: cannot be changed (constants)
    - methods: cannot be *overridden* (later)
    - classes: cannot be subclassed

```
final class MyClass {
        static int       sameValueForAllInstances = 3;
        final  int       constantValue = 5;
        static final int constantValueForAllInstances = 7;

        static void  methodOne() { /* ... */ }
        final void   methodTwo() { /* ... */ }
        static final void methodThree() { /* ... */ }
}
```

Overriding, Hiding

- Overriding methods
  - Why?
    Let subclasses provide a more specialized version of an instance-method
  - How?
    Subclass defines an instance-method with same signature (name plus number and types of parameters) as defined by super-class

3 Classes, Objects, Inheritance

Overriding, Hiding

- Overriding methods
  - Let subclasses provide a more specialized version of an instance-method
  - Subclass defines an instance-method with same signature (name plus *number* and *types* of parameters) as defined by superclass

```
class Bicycle {
    int speed;
    public void speedUp(int increment) {
        speed = speed + increment;
        System.out.println("superclass instance-method");
    }
}
class MountainBike extends Bicycle {
    public void speedUp(int increment) {
        speed = speed + 2 * increment;
        System.out.println("subclass instance-method");
    }
}
```

```
MountainBike mb = new MountainBike();
mb.speedUp(10);                              // mb.speed == 20
```

⇨ output will be: **subclass instance-method**

3 Classes, Objects, Inheritance

Overriding, Hiding

- Overriding methods
  - Let subclasses provide a more specialized version of an instance-method
  - Subclass defines an instance-method with same signature (name plus *number* and *types* of parameters) as defined by superclass


```
class Bicycle {
    int speed;
    public void speedUp(int increment) {
        speed = speed + increment;
        System.out.println("superclass instance-method");
    }
}
class MountainBike extends Bicycle {
    public void speedUp(int increment) {
        speed = speed + 2 * increment;
        System.out.println("subclass instance-method");
    }
}
```

```
MountainBike mb = new MountainBike();
mb.speedUp(10);                              // mb.speed == 20
```

⇨ output will be: **subclass instance-method**

3 Classes, Objects, Inheritance

Overriding, Hiding

- Overriding methods
  - Let subclasses provide a more specialized version of an instance-method
  - Subclass defines an instance-method with same signature (name plus *number* and *types* of parameters) as defined by superclass


```
class Bicycle {
    int speed;
    public void speedUp(int increment) {
        speed = speed + increment;
        System.out.println("superclass instance-method");
    }
}
class MountainBike extends Bicycle {
    public void speedUp(int increment) {
        speed = speed + 2 * increment;
        System.out.println("subclass instance-method");
    }
}
```

```
MountainBike mb = new MountainBike();
mb.speedUp(10);                              // mb.speed == 20
```

⇨ output will be: **subclass instance-method**

Overriding, Hiding

- Overriding methods
  - Let subclasses provide a more specialized version of an instance-method
  - Subclass defines an instance-method with same signature (name plus *number* and *types* of parameters) as defined by superclass

```
class Bicycle {
    int speed;
    public void speedUp(int increment) {
        speed = speed + increment;
        System.out.println("superclass instance-method");
    }
}
class MountainBike extends Bicycle {
    public void speedUp(int increment) {
        speed = speed + 2 * increment;
        System.out.println("subclass instance-method");
    }
}
```

```
MountainBike mb = new MountainBike();
mb.speedUp(10);                              // mb.speed == 20
```

⇒ output will be: **`subclass instance-method`**

Overriding, Hiding

- Hiding class-methods
  - Let subclasses provide a more specialized version of a class-method
  - Subclass defines a class-method with same signature (name plus number and types of parameters) as defined by superclass

```
class Bicycle {
    public static void myClassMethod(int someInt) {
        System.out.println("superclass class-method");
    }
}
```

```
class MountainBike extends Bicycle {
    public static void myClassMethod(int someInt) {
        System.out.println("subclass class-method");
    }
}
```

```
Bicycle.myClassMethod(10);         // "superclass class-method"
MountainBike.myClassMethod(10);    // "subclass class-method"
```


Overriding, Hiding

- Hiding class-methods
  - Let subclasses provide a more specialized version of a class-method
  - Subclass defines a class-method with same signature (name plus number and types of parameters) as defined by superclass

```
class Bicycle {
    public static void myClassMethod(int someInt) {
        System.out.println("superclass class-method");
    }
}
```

```
class MountainBike extends Bicycle {
    public static void myClassMethod(int someInt) {
        System.out.println("subclass class-method");
    }
}
```

```
Bicycle.myClassMethod(10);         // "superclass class-method"
MountainBike.myClassMethod(10);    // "subclass class-method"
```


Polymorphism

- Polymorphism: subclass objects may be assigned to superclass variables

```
MountainBike mountainBike = new MountainBike();
Bicycle bicycle = mountainBike;
```

→ Essential feature of object oriented software

- Only methods and fields defined by the the superclass "portion" of the object may be accessed; and the overridden ("right") methods are called

```
bicycle.gear = 3;             // Ok, gear defined in class Bicycle
bicycle.seatHeight = 20;      // ERROR! seatHeight is not a field in class Bicycle
mountainBike.setHeight = 20; // Ok

mountainBike.speedUp(5);      // Overridden method in subclass MountainBike is used
bicycle.speedUp(10);          // Overridden method in subclass MountainBike is used

MountainBike.myClassMethod(99);    // "subclass class-method"
Bicycle.myClassMethod(99);         // "superclass class-method"
```

Polymorphism

- Polymorphism: subclass objects may be assigned to superclass variables

```
MountainBike mountainBike = new MountainBike();
Bicycle bicycle = mountainBike;
```

→ Essential feature of object oriented software

- Only methods and fields defined by the the superclass "portion" of the object may be accessed; and the overridden ("right") methods are called

```
bicycle.gear = 3;            // Ok, gear defined in class Bicycle
bicycle.seatHeight = 20;     // ERROR! seatHeight is not a field in class Bicycle
mountainBike.setHeight = 20; // Ok

mountainBike.speedUp(5);     // Overridden method in subclass MountainBike is used
bicycle.speedUp(10);         // Overridden method in subclass MountainBike is used

MountainBike.myClassMethod(99);    // "subclass class-method"
Bicycle.myClassMethod(99);         // "superclass class-method"
```

3 Classes, Objects, Inheritance

Polymorphism

- Polymorphism: subclass objects may be assigned to superclass variables

```
MountainBike mountainBike = new MountainBike();
Bicycle bicycle = mountainBike;
```

→ Essential feature of object oriented software

- Only methods and fields defined by the the superclass "portion" of the object may be accessed; and the overridden ("right") methods are called

```
bicycle.gear = 3;            // Ok, gear defined in class Bicycle
bicycle.seatHeight = 20;     // ERROR! seatHeight is not a field in class Bicycle
mountainBike.setHeight = 20; // Ok

mountainBike.speedUp(5);     // Overridden method in subclass MountainBike is used
bicycle.speedUp(10);         // Overridden method in subclass MountainBike is used

MountainBike.myClassMethod(99);    // "subclass class-method"
Bicycle.myClassMethod(99);         // "superclass class-method"
```

3 Classes, Objects, Inheritance

Polymorphism

- Polymorphism: subclass objects may be assigned to superclass variables

```
MountainBike mountainBike = new MountainBike();
Bicycle bicycle = mountainBike;
```

→ Essential feature of object oriented software

- Only methods and fields defined by the the superclass "portion" of the object may be accessed; and the overridden ("right") methods are called

```
bicycle.gear = 3;            // Ok, gear defined in class Bicycle
bicycle.seatHeight = 20;     // ERROR! seatHeight is not a field in class Bicycle
mountainBike.setHeight = 20; // Ok

mountainBike.speedUp(5);     // Overridden method in subclass MountainBike is used
bicycle.speedUp(10);         // Overridden method in subclass MountainBike is used

MountainBike.myClassMethod(99);    // "subclass class-method"
Bicycle.myClassMethod(99);         // "superclass class-method"
```

3 Classes, Objects, Inheritance

Polymorphism

- Polymorphism: subclass objects may be assigned to superclass variables

```
MountainBike mountainBike = new MountainBike();
Bicycle bicycle = mountainBike;
```

→ Essential feature of object oriented software

- Only methods and fields defined by the the superclass "portion" of the object may be accessed; and the overridden ("right") methods are called

```
bicycle.gear = 3;            // Ok, gear defined in class Bicycle
bicycle.seatHeight = 20;     // ERROR! seatHeight is not a field in class Bicycle
mountainBike.seatHeight = 20; // Ok

mountainBike.speedUp(5);     // Overridden method in subclass MountainBike is used
bicycle.speedUp(10);         // Overridden method in subclass MountainBike is used

MountainBike.myClassMethod(99);    // "subclass class-method"
Bicycle.myClassMethod(99);         // "superclass class-method"
```

Overriding, Hiding

- Overriding methods
  - Let subclasses provide a more specialized version of an instance-method
  - Subclass defines an instance-method with same signature (name plus *number* and *types* of parameters) as defined by superclass

```
class Bicycle {
    int speed;
    public void speedUp(int increment) {
        speed = speed + increment;
        System.out.println("superclass instance-method");
    }
}
class MountainBike extends Bicycle {
    public void speedUp(int increment) {
        super(2 * increment); // call overridden method from superclass
        System.out.println("subclass instance-method");
    }
}
```

```
MountainBike mb = new MountainBike();
mb.speedUp(10);                          // mb.speed == 20
```

⇨ output will be:

```
superclass instance-method
subclass instance-method
```

3 Classes, Objects, Inheritance

Polymorphism

- Polymorphism: subclass objects may be assigned to superclass variables

```
MountainBike mountainBike = new MountainBike();
Bicycle bicycle = mountainBike;
```

→ Essential feature of object oriented software

- Only methods and fields defined by the the superclass "portion" of the object may be accessed; and the overridden ("right") methods are called

```
bicycle.gear = 3;            // Ok, gear defined in class Bicycle
bicycle.seatHeight = 20;     // ERROR! seatHeight is not a field in class Bicycle
mountainBike.seatHeight = 20; // Ok

mountainBike.speedUp(5);     // Overridden method in subclass MountainBike is used
bicycle.speedUp(10);         // Overridden method in subclass MountainBike is used

MountainBike.myClassMethod(99);    // "subclass class-method"
Bicycle.myClassMethod(99);         // "superclass class-method"
```