Java as a Programming Language

Programming Languages

- Imperative ("HOW")
  - Procedural
  - Object-Oriented

- Declarative ("WHAT")
  - Logic-Based
  - Functional

Compiled Languages:
(e.g. C)

Virtual Machine Languages:
(e.g. Java)
Java as a Programming Language

Imperative Programming

• Imperative program: Sequence of statements
  (especially memory) of computer system

<table>
<thead>
<tr>
<th>cell nr</th>
<th>cell name</th>
<th>cell content</th>
</tr>
</thead>
<tbody>
<tr>
<td>1123</td>
<td>plato</td>
<td>false</td>
</tr>
<tr>
<td>1124</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1125</td>
<td>horst</td>
<td>101</td>
</tr>
<tr>
<td>1126</td>
<td>heiner</td>
<td>0</td>
</tr>
<tr>
<td>1127</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1128</td>
<td>fritz</td>
<td>0</td>
</tr>
</tbody>
</table>

boolean plato;
int horst;
int heiner;
int fritz;
plato = false;
horst = 101;
heiner = 2;
fritz = horst + heiner;
horst = 2000;

Procedural Programming

• Group sequences of instructions into named „procedures“ („functions“, „methods“, „sub-routines“ etc.)

```java
int doSelfSumSquare(int someNumber){
    int a;
    a = someNumber + someNumber;
    a = a * a;
    return a;
}
```

• Advantages
  • no copying of instruction sequences
  • better testing
  • modularity (e.g. code change inside function doesn’t affect caller)
  • code re-use
  • etc.
Java as a Programming Language

```java
int horst;
int heiner;
horst = 101;
heiner = 2;
heiner = doSelfSumSquare(horst);
heiner = doSelfSumSquare(117);
horst = horst + 2;

int doSelfSumSquare(int someNumber){
    int a;
    a = someNumber + someNumber;
    a = a * a;
    return a;
}
```

- In the example: Control flow is transferred to function, back to main program, back to function and back to main program

Object-oriented Programming

- Object-oriented programming:
  - Group data and functions into objects ↔ Models of state and behaviour of real world objects
  - State "fields"; behaviour "methods"

- Methods should mainly act on an object's fields
- Classes: Blueprints for objects → Objects: Instances of classes
- Advantages
  - Intuitive models
  - Information hiding
  - Increased modularity, locality etc.
  - Increased code re-use
  - etc.

Object-oriented Programming

- Object-oriented programming:
  - Group data and functions into objects ↔ Models of state and behaviour of real world objects
  - State "fields"; behaviour "methods"

- Methods should mainly act on an object's fields
- Classes: Blueprints for objects → Objects: Instances of classes
- Advantages
  - Intuitive models
  - Information hiding
  - Increased modularity, locality etc.
  - Increased code re-use
  - etc.
Object-oriented Programming

- Object-oriented programming:
  - Group data and functions into objects <-> Models of state and behaviour of real world objects
  - State „fields“; behaviour „methods“

- Methods should mainly act on an object’s fields
- Classes: Blueprints for objects → Objects: Instances of classes
- Advantages
  - Intuitive models
  - Information hiding
  - Increased modularity, locality etc.
  - Increased code re-use
  - etc.

Datenbanken

- Object-oriented programming:
  - Group data and functions into objects <-> Models of state and behaviour of real world objects
  - State „fields“; behaviour „methods“

- Methods should mainly act on an object’s fields
- Classes: Blueprints for objects → Objects: Instances of classes
- Advantages
  - Intuitive models
  - Information hiding
  - Increased modularity, locality etc.
  - Increased code re-use
  - etc.
Java

```java
public class SomeCode {
    public static void main(String[] args) {
        Professor prof2125 = new Professor("Sokrates", "C4", 226);
        Professor russelTheOldDad = new Professor("Russel", "C4", 232);
        Professor kopernikus = new Professor("Kopernikus", "C3", 310);
        Professor gustav = new Professor("Augustinus", "C3", 309);
        Professor oldMary = new Professor("Curia", "C4", 36);
        Professor prof2144 = new Professor("Kant", "C4", 7);
        ... 
    }
}
```
Java as a Programming Language

```java
class BicycleDemo {
    public static void main(String[] args) {
        // Create two different Bicycle objects
        Bicycle bike1 = new Bicycle();
        Bicycle bike2 = new Bicycle();

        // Invoke methods on these objects
        bike1.changeCadence(50);
        bike1.speedUp(10);
        bike1.changeGear(2);

        bike2.changeCadence(50);
        bike2.speedUp(10);
        bike2.changeGear(2);
        bike2.changeCadence(40);
        bike2.speedUp(10);
        bike2.changeGear(3);
    }
}
```

Source: [Tutorial]

### Datenbanken

```java
public class SomeCode {
    public static void main(String[] args) {
        Professor prof1 = new Professor("Socrates", "C4", 226);
        Professor prof2 = new Professor("Russe1", "C3", 232);
        Professor prof3 = new Professor("Kopernikus", "C3", 320);
        Professor prof4 = new Professor("Popper", "C3", 52);
        Professor prof5 = new Professor("Augustinus", "C3", 300);
        Professor prof6 = new Professor("Curtis", "C4", 56);
        Professor prof7 = new Professor("Kant", "C4", 7);
    }
}
```

### Java

```java
public class Professor {
    public String name;
    public String range;
    public int room;

    public Professor(String name, String range, int room){
        this.name = name;
        this.range = range;
        this.room = room;
    }

    public void teach(){
        System.out.println("... now teaching something :-)!");
    }
}
```

### Inheritance

Inheritance:

Define new, more specialized classes from existing classes

#### Bicycle

- cadence: int
- speed: int
- gear: int
- changeGear(int value)
- speedUp(int increment)
- applyBrakes(int decrement)

#### RoadBike

- diag: boolean
- races: boolean
- putOnRaceTires()

#### TandemBike

- numberOfDrivers: int

```java
class RoadBike extends Bicycle {
    // additional fields and methods
    // that define a road bike
    // go here
}
```
Java as a Programming Language

```java
class BicycleDemo {
    public static void main(String[] args) {
        // Create two different Bicycle objects
        Bicycle bike1 = new Bicycle();
        Bicycle bike2 = new Bicycle();

        // Invoke methods on these objects
        bike1.changeCadence(50);
        bike1.speedUp(10);
        bike1.changeGear(2);

        bike2.changeCadence(50);
        bike2.speedUp(10);
        bike2.changeGear(2);
        bike2.changeCadence(40);
        bike2.speedUp(10);
        bike2.changeGear(3);
    }
}
```

Source: [Tutoria](#)

Java as a Programming Language

Inheritance

- Define new, more specialized classes from existing classes

```
interface IBicycle {
    void changeCadence(int newValue);

    void changeGear(int newValue);

    void speedUp(int increment);

    void applyBrakes(int decrement);
}
```

class Bicycle implements IBicycle {
    // remainder of this class implemented as before
    // except that above methods must be public
}

class RoadBike extends Bicycle {
    // additional fields and methods
    // that define a road bike
    // go here
}

class MountainBike {
    int seatHeight;
    setHeight();
    getDropperSeat();
    putOnFrameTest();
}

Java as a Programming Language

Interfaces

- Specify in an abstract way what a class implementing that interface should exhibit as behaviours (create blueprint for blueprints)

```
interface IBicycle {
    void changeCadence(int newValue);

    void changeGear(int newValue);

    void speedUp(int increment);

    void applyBrakes(int decrement);
}
```

class Bicycle implements IBicycle {
    // remainder of this class implemented as before
    // except that above methods must be public
}

see: [Tutoria](#)

Java as a Programming Language

Interfaces

```
interface IBicycle {
    void changeCadence(int newValue);

    void changeGear(int newValue);

    void speedUp(int increment);

    void applyBrakes(int decrement);
}
```

class Bicycle implements IBicycle {
    // remainder of this class implemented as before
    // except that above methods must be public
}

see: [Tutoria](#)
**Interfaces**

**Interface:**
Specify in an abstract way what a class implementing that interface should exhibit as behaviours (create blueprint for blueprints)

```java
interface IBicycle {
    void changeCadence(int newValue);
    void changeGear(int newValue);
    void speedUp(int increment);
    void applyBrakes(int decrement);
}
```

```java
class Bicycle implements IBicycle {
    // remainder of this class implemented as before
    // except that above methods must be public
}
```

**Inheritance**

**Inheritance:**
Define new, more specialized classes from existing classes

```java
interface IBicycle {
    void changeCadence(int newValue);
    void changeGear(int newValue);
    void speedUp(int increment);
    void applyBrakes(int decrement);
}
```

```java
class RoadBike extends Bicycle {
    // additional fields and methods
    // that define a road bike
    // go here
}
```
Interfaces

**Interface:**
Specify in an abstract way what a class implementing that interface should exhibit as behaviours (create blueprint for blueprints)

```java
interface IBicycle {
    void changeCadence(int newValue);
    void changeGear(int newValue);
    void speedUp(int increment);
    void applyBrakes(int decrement);
}
```

class Bicycle implements IBicycle {
    // remainder of this class implemented as before
    // except that above methods must be public
}

Example:

```
// Diagram showing class relationships and method implementations.
```

See: [Tutorial]
public class littletree {
    double collectedPollen = 0;
    void string() {
        System.out.print("Wink!");
    }
}
public class BeeDemo {
    public static void main(String[] args) {
        LittleBean l1 = new LittleBean();
        l1.setVolume(100);
        l1.openFlower();
    }
}

public class LittleBean {
    public String voice; // volume
    public void voiceUp() {
        this.voice = this.voice + 1;
    }
    public void openFlower() {
        System.out.println("Little bean - voice: "+ this.voice);
    }
}

public class BeeDemo {
    public static void main(String[] args) {
        LittleBean l1 = new LittleBean();
        l1.setVolume(100);
        l1.openFlower();
    }
}

public class LittleBean {
    public String voice; // volume
    public void voiceUp() {
        this.voice = this.voice + 1;
    }
    public void openFlower() {
        System.out.println("Little bean - voice: "+ this.voice);
    }
}

public class BeeDemo {
    public static void main(String[] args) {
        LittleBean l1 = new LittleBean();
        l1.setVolume(100);
        l1.openFlower();
    }
}

public class LittleBean {
    public String voice; // volume
    public void voiceUp() {
        this.voice = this.voice + 1;
    }
    public void openFlower() {
        System.out.println("Little bean - voice: "+ this.voice);
    }
}
public class Littlebees {
    public static void main(String[] args) {
        Littlebees will = new Littlebees();
        will.willcollectedPollen = 0;
        will.greenPollen = 10;
        will.willow = 15;
        will.collectionPollen = collectedPollen + will.willow;
        will.willow = new Flower();
        will.greenPollen = new Flower();
    }
}

public class Littlebees {
    double collectedPollen = 0;
    void intel() {
        System.out.println("Willow!");
    }
    void source() {
        System.out.println("Source!");
    }
    void add() {
        System.out.println("Add!");
    }
    void collectPollen(Flower fly) {
        double amount = 0;
        collectedPollen + collectedPollen + amount;
        System.out.println("I'll do it as soon as the amount + " + "my pollen is maximum!");
    }
    void fly() { System.out.println();

public class flyingInsect {
    void fly() {
        System.out.println("Flying");
    }
}

doctorClass {
    void prescribe() {
        System.out.println("Prescribing");
    }
}

public class pollinator {
    void collectPollen() {
        double collectedPollen = 0;
        void sting() {
            System.out.println("Stung");
        }
        void source() {
            System.out.println("Sourced");
        }
        void collectPollen() {
            double amount = 10.0;
            collectedPollen += amount;
            System.out.println("Got " + amount + " mg of Pollen");
        }
        void pollinate() {
            System.out.println("Pollinated");
        }
    }
}

public class honeyGathering {
    void collectHoney() {
        double collectedHoney = 0;
        void sting() {
            System.out.println("Stung");
        }
        void source() {
            System.out.println("Sourced");
        }
        void collectHoney() {
            double amount = 10.0;
            collectedHoney += amount;
            System.out.println("Got " + amount + " kg of Honey");
        }
        void pollinate() {
            System.out.println("Pollinated");
        }
    }
}
2 Language Basics

Deepening readings:

http://java.sun.com/docs/books/tutorial/java/nutsandbolts/variables.html
http://java.sun.com/docs/books/tutorial/java/nutsandbolts/datatypes.html
http://java.sun.com/docs/books/tutorial/java/nutsandbolts/arrays.html
http://java.sun.com/docs/books/tutorial/java/nutsandbolts/expressions.html
http://java.sun.com/docs/books/tutorial/java/nutsandbolts/while.html
http://java.sun.com/docs/books/tutorial/java/nutsandbolts/for.html
http://java.sun.com/docs/books/tutorial/java/nutsandbolts/branch.html
2 Language Basics – Variables

Variables

- **Variables have a type**
  - **Primitive type**
  - **Reference type**

<table>
<thead>
<tr>
<th>Definition</th>
<th>Declaration</th>
<th>Instantiation</th>
<th>Manipulation</th>
<th>Equality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primitive</td>
<td>int a;</td>
<td>a = 117;</td>
<td>a = b + 42;</td>
<td>a == b;</td>
</tr>
<tr>
<td>Reference</td>
<td>class Student { // Fields and // methods ... }</td>
<td>Student heiner;</td>
<td>heiner.age = 21; heiner.yawn( );</td>
<td>heiner.equals( sabine );</td>
</tr>
</tbody>
</table>

Variables have a type

- **Primitive type**
- **Reference type**

<table>
<thead>
<tr>
<th>Definition</th>
<th>Declaration</th>
<th>Instantiation</th>
<th>Manipulation</th>
<th>Equality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primitive</td>
<td>int a;</td>
<td>a = 117;</td>
<td>a = b + 42;</td>
<td>a == b;</td>
</tr>
<tr>
<td>Reference</td>
<td>class Student { // Fields and // methods ... }</td>
<td>Student heiner;</td>
<td>heiner.age = 21; heiner.yawn( );</td>
<td>heiner.equals( sabine );</td>
</tr>
</tbody>
</table>

Deepening readings:

http://java.sun.com/docs/books/tutorial/java/nutsandbolts/variables.html
http://java.sun.com/docs/books/tutorial/java/nutsandbolts/datatypes.html
http://java.sun.com/docs/books/tutorial/java/nutsandbolts/arrays.html
http://java.sun.com/docs/books/tutorial/java/nutsandbolts/opssummary.html
http://java.sun.com/docs/books/tutorial/java/nutsandbolts/expressions.html
http://java.sun.com/docs/books/tutorial/java/nutsandbolts/if.html
http://java.sun.com/docs/books/tutorial/java/nutsandbolts/while.html
http://java.sun.com/docs/books/tutorial/java/nutsandbolts/for.html
http://java.sun.com/docs/books/tutorial/java/nutsandbolts/branch.html
2 Language Basics – Variables

Variables

- Variables have a type
  - Primitive type
  - Reference type

<table>
<thead>
<tr>
<th>Definition</th>
<th>Declaration</th>
<th>Instantiation</th>
<th>Manipulation</th>
<th>Equality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primitive</td>
<td>predefined</td>
<td>int a;</td>
<td>a = 117;</td>
<td>a == b;</td>
</tr>
<tr>
<td>Reference</td>
<td>class Student { // Fields and // methods ... }</td>
<td>Student heiner;</td>
<td>heiner.age = 21; heiner.yawn();</td>
<td>heiner.equals(sabine);</td>
</tr>
</tbody>
</table>

2 Language Basics – Variables

Variables

- Variables have a type
  - Primitive type
  - Reference type

<table>
<thead>
<tr>
<th>Definition</th>
<th>Declaration</th>
<th>Instantiation</th>
<th>Manipulation</th>
<th>Equality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primitive</td>
<td>predefined</td>
<td>int a;</td>
<td>a = 117;</td>
<td>a == b;</td>
</tr>
<tr>
<td>Reference</td>
<td>class Student { // Fields and // methods ... }</td>
<td>Student heiner;</td>
<td>heiner.age = 21; heiner.yawn();</td>
<td>heiner.equals(sabine);</td>
</tr>
</tbody>
</table>

memory (simplified model)

<table>
<thead>
<tr>
<th>cell no.</th>
<th>cell name</th>
<th>cell content</th>
</tr>
</thead>
<tbody>
<tr>
<td>1123</td>
<td>horst</td>
<td>101</td>
</tr>
<tr>
<td>1124</td>
<td>heiner</td>
<td>235638465</td>
</tr>
<tr>
<td>1125</td>
<td></td>
<td>83746845</td>
</tr>
<tr>
<td>1150</td>
<td>bike1.cadence</td>
<td>0</td>
</tr>
<tr>
<td>1151</td>
<td>bike1.speed</td>
<td>0</td>
</tr>
<tr>
<td>1152</td>
<td>bike1.gear</td>
<td>3</td>
</tr>
<tr>
<td>1330</td>
<td>bike2.cadence</td>
<td>0</td>
</tr>
<tr>
<td>1331</td>
<td>bike2.speed</td>
<td>0</td>
</tr>
<tr>
<td>1332</td>
<td>bike2.gear</td>
<td>1</td>
</tr>
<tr>
<td>1333</td>
<td>bike2.seatHeight</td>
<td>15</td>
</tr>
<tr>
<td>4027</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4028</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4029</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4035</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

int horst = 101;
long heiner;
heiner = 235638465837465845;

- Primitive type

int horst = 101;
long heiner;
heiner = 235638465837465845;

- Reference type

Bicycle bike1 = new Bicycle();
bike1.gear = 3;

MountainBike bike2 = new MountainBike();
## Language Basics – Variables

### Primitive Types

- **Primitive** types (numeric):

<table>
<thead>
<tr>
<th>Type</th>
<th>byte</th>
<th>short</th>
<th>int</th>
<th>long</th>
<th>float</th>
<th>double</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>8 bit</td>
<td>16 bit</td>
<td>32 bit</td>
<td>64 bit</td>
<td>32 bit</td>
<td>64 bit</td>
</tr>
</tbody>
</table>

- Examples:

```java
byte flags = 63;
short bbb = 10133;
int heiner = 234103234;
long dong = -83628735682345;
float fff = 5464.00345;
float ggg = -345545.34534E-12f;
double sss = 3245343455.5555567;  // = 3245343455.55555 * 10^67
```

### Primitive Types

- **Primitive** types (numeric, boolean, character):

<table>
<thead>
<tr>
<th>Type</th>
<th>byte</th>
<th>short</th>
<th>int</th>
<th>long</th>
<th>float</th>
<th>double</th>
<th>boolean</th>
<th>char</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>8 bit</td>
<td>16 bit</td>
<td>32 bit</td>
<td>64 bit</td>
<td>32 bit</td>
<td>64 bit</td>
<td>1 bit</td>
<td>16 bit</td>
</tr>
<tr>
<td>Values</td>
<td>[-2^7, 2^7-1]</td>
<td>[-2^16, 2^16-1]</td>
<td>[-2^32, 2^32-1]</td>
<td>[-2^64, 2^64-1]</td>
<td>[-1.4<em>10^10, +1.4</em>10^10]</td>
<td>[-3.4<em>10^38, +3.4</em>10^38]</td>
<td>true, false</td>
<td>\text{UTF-16}</td>
</tr>
</tbody>
</table>

- Examples:

```java
byte flags = 63;
short bbb = 10133;
int heiner = 234103234;
long dong = -83628735682345;
float fff = 5464.00345;
float ggg = -345545.34534E-12f;
double sss = 3245343455.5555567;  // = 3245343455.55555 * 10^67
```
### 2 Language Basics – Variables

#### More examples:

```java
byte flags = 63;
short bbb = 10133;
int heiner = 234103234;
long dng = -83628735682345;
float fff = 5464.00345f;
float qgg = -345545.34534E-12f;
double sss = 3245343455.555867d;
char ccc = 'm';
char ccc2 = '\n';
boolean isCool = true;
```

#### Reference type

```java
Bicycle bike1 = new Bicycle();
bike1.gear = 3;
MountainBike bike2 = new MountainBike();
```

#### Variables have a type

##### Primitive type

```java
int horst = 101;
long heiner;
heiner = 235638465837465845;
```

##### Reference type

```java
Bicycle bike1 = new Bicycle();
bike1.gear = 3;
MountainBike bike2 = new MountainBike();
```

### 2 Language Basics – Variables

#### Reference Type Variables

##### Reference type variables "point" to an object of the reference type

```java
bikel = new Bicycle();
bike2 = new Bicycle();

boolean c;
c = bike1.equals(bike2); // c == true
c = (bikel == bike2); // c == false
```
2 Language Basics – Variables

Reference Type Variables

- Reference type variables "point" to an object of the reference type

```java
bikel = new Bicycle();
bike2 = new Bicycle();

bikel.gear = 3;
bikel = bike2;
```

```java
boolean c;
c = bikel.equals(bike2); // c == true
c = (bikel == bike2); // c == true
```

- Reference type variables "point" to an object of the reference type

```java
bikel = new Bicycle();
bike2 = new Bicycle();

bikel.gear = 3;
bikel = bike2;
```

```java
boolean c;
c = bikel.equals(bike2); // c == true
c = (bikel == bike2); // c == true
```
2 Language Basics – Variables

Reference Type Variables

- **Reference** type variables "point" to an object of the reference type

```java
bikel = new Bicycle();
bike2 = new Bicycle();
bikel.gear = 3;
bikel = bike2;
```

```java
boolean c;
c = bikel.equals(bike2);
// c == true
```