

Objects and Classes

- Reference types and their characteristics
- Class Definition
- Constructors and Object Creation
- Special objects: Strings and Arrays

OOAD 1999/2000
Claudia Niederée, Joachim W. Schmidt
Software Systems Institute

ajc@tu-harburg.de

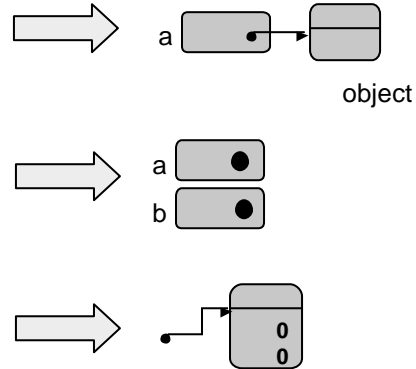
<http://www.sts.tu-harburg.de/AJC>

Reference Types

- Predefined or user-defined Classes
- String
- Array
- The values of reference types are not manipulated directly.
- Instead handles (references) to the actual values are used.

Reference Type Variables

- A variable of a reference type contains a reference to an anonymous object in storage.
- An uninitialized variable contains a null-reference.
`Point a,b;`
- Objects are created at run-time.
`new Point(0,0);`
- Objects are anonymous.
- They have an immutable identity.

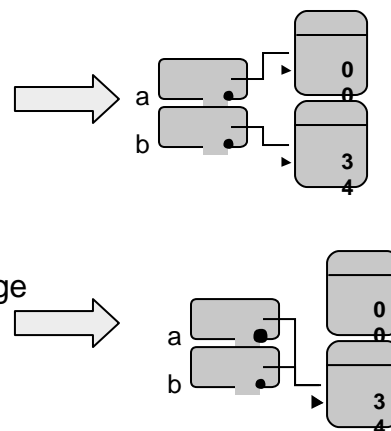


OOAD99/00-ST5-Objects and Classes

3

Binding Reference Type Variables

- Assigning an object to a variable, the variable gets a reference to the object as its value.
`a = new Point(0,0);`
`b = new Point(3,4);`
- Assigning values of other variables means assigning the references! The value of the object does not change
`a = b;`
- Maybe a pitfall, as the object may change

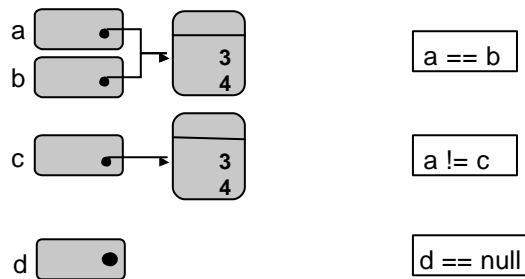


OOAD99/00-ST5-Objects and Classes

4

Comparing Objects

- Comparing two reference type variables means comparing the references they contain and, thus, comparing the identity of the referenced objects.



OOAD99/00-ST5-Objects and Classes

5

Comparing the Contents of Objects

- The contents of objects belonging to pre-defined classes can be compared using the method *equals*.
- If you define a class of your own, *equals* by default only compares references, as `==` does.
 - => You have to override *equals* for newly defined classes.

OOAD99/00-ST5-Objects and Classes

6

Creating New Data Types: class

- Class keyword introduces definition of new data type

```
class ATypeName { /* class body goes here */ }
ATypeName a = new ATypeName();
```

- Data members

```
class DataOnly {
    int i;
    float f;
    boolean b;
}
```

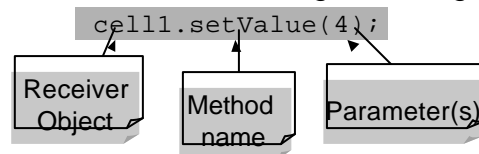
- Each instance of **DataOnly** gets its own copy of the data members
- In a class, primitives get default values.

Methods, Arguments and Return Values

- Methods: “how you get things done in an object”
 - May inspect and change the state of the object
 - Traditionally called “functions”
 - Can only be defined inside classes

```
returnType methodName(/* argument list */) {
    /* Method body */
}
```

- Example method call = „message sending“:



The Argument List

- Types of the objects to pass in to the method
- Name (identifier) to use for each one
- Whenever you seem to be passing objects in Java, you're actually passing handles

```
void reset(Cell c) {  
    c.setValue(0);  
}  
Cell cell1 = new Cell();  
cell1.setValue(4);  
reset(cell1);  
cell1.getValue();
```

Inside the Method Body

- Variable declaration and assignments
- Operations on primitive data types
- Object creation
- Message sending
- Iteration
- Conditionals
- **return** Statement

this: Handle to Current Object

```
public class Leaf {
    private int i = 0;
    Leaf increment() {
        i++;
        return this;
    }
    void print() {
        System.out.println("i = " + i);
    }
    public static void main(String args[]) {
        Leaf x = new Leaf();

        x.increment().increment().increment().print();
    }
}
```

- no **this** in static methods

OOAD99/00-ST5-Objects and Classes

11

Nuance

- We can deduce meaning from context

“Wash the shirt”

“Wash the car”

“Wash the dog”

Not

“shirtWash the shirt”

“carWash the car”

“dogWash the dog”

OOAD99/00-ST5-Objects and Classes

12

Method Overloading

```
class SuperWasher{  
    void wash(Shirt s) { ... }  
    void wash(Car c) { ... }  
    void wash(Dog d) { ... }  
    ...  
}
```

- One word, many meanings:
overloaded
- Unique argument type combinations
distinguish overloaded methods



Overloading on Return Values

- Why not also use return values in method
overloading?

```
void f() {...}  
int f() {...}
```

- Then what would this mean?

```
f();
```



Default Constructor

- Constructors are needed to create instances (objects) of a class
- Compiler provides one for you if you write no constructor

```
class Bird {
    int i;
}

public class DefaultConstructor {
    public static void main(String args[]) {
        Bird nc = new Bird(); // default!
    }
}
```

Constructor Definition

```
class Rock {
    Rock() { // This is the constructor
        System.out.println("Creating Rock");
    }
}

public class SimpleConstructor {
    public static void main(String args[]) {
        for(int i = 0; i < 10; i++)
            new Rock();
    }
}
```

Constructor Overloading

- Like methods constructors may be overloaded

```
class Tree {
    int height;
    Tree() {
        System.out.println("A seedling");
        height = 0;
    }
    Tree(int i) {
        System.out.println("A new Tree, "
            + i + " feet tall");
        height = i;
    }
}
```

this in Constructors

- A very common kind to use **this** is in constructors to initialize data members with the constructor's arguments

```
public class Animal {
    private int numberOfLegs;

    Animal(int numberOfLegs) {
        this.numberOfLegs = numberOfLegs;
    }
}
```

Arrays and Array Initialization

- Arrays are objects

```
int a1[]; // this...
int[] a1; // is the same as this!
```

Creates a handle, not the array. Can't size it.

- To create an array of primitives:

```
int [] a1 = { 1, 2, 3, 4, 5 };
```

- Bounds are checked, **length** produces size of the array `a1.length`

- If you do anything wrong either the compiler will catch it or an exception will be thrown

Arrays of Objects

- An array of class objects:

```
Animal[] a = new Animal[20];
System.out.println(a.length + " animals");
for(int i = 0; i < a.length; i++)
    a[i] = new Animal((i % 2 + 1) * 2);
```

- Can also use bracketed list (The size is then fixed at compile-time)

```
Integer[] a = {
    new Integer(1),
    new Integer(2),
    new Integer(3),
};
```

Multi-dimensional Arrays

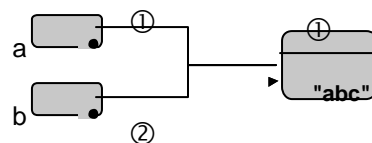
- It is possible to define multi-dimensional arrays.
`int[][] a;`
- The brackets even may be distributed between type and identifier.
`int[] a[];`
- Initialization can be done directly
`int[][] a = { { 1, 2, 3 }, { 5, 6 } };`
- It can also be done by nested iterations over the array and its components.

Strings

- Strings are immutable objects of the class **String**.
- String literals are zero, one or more characters included within double quotes.
- When a binding to a string literal is executed for the first time, a new String object is created.
- If any other bindings to this literal appear, the respective variables will hold reference to the same object.

```
String a = "abc";  
String b = "abc";
```

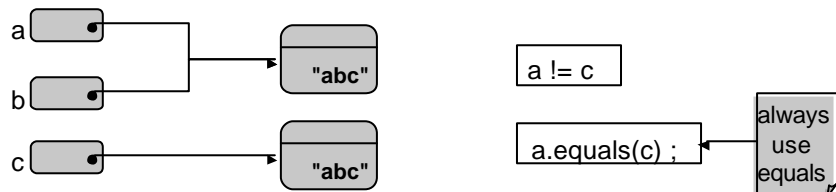
`a == b`



String Constructors

- String objects can also be created by calling a constructor.
- String constructors create new objects whenever they are called.

```
String c = new String("abc");
```



- There are several constructors defined for strings.

OOAD99/00-ST5-Objects and Classes

23

String Concatenation

- Strings can be concatenated by using +.
- The concatenation also creates a new String object.
- Values of other types can be concatenated to strings, too.
- They are implicitly converted to String.

```
String n = "Number " + 49;
```

OOAD99/00-ST5-Objects and Classes

24