



# DM537

# Object-Oriented Programming

Peter Schneider-Kamp

[petersk@imada.sdu.dk](mailto:petersk@imada.sdu.dk)

<http://imada.sdu.dk/~petersk/DM537/>

# **TYPE CASTS & FILES & EXCEPTION HANDLING**

# Type Conversion

- Java uses *type casts* for converting values
- `(int) x`: converts `x` into an integer
  - Example 1: `((int) 127) + 1 == 128`
  - Example 2: `((int) -3.999) == -3`
- `(double) x`: converts `x` into a float
  - Example 1: `((double) 42) == 42.0`
  - Example 2: `(double) "42"` gives compilation error
- `(String) x`: views `x` as a string
  - Example: `Object o = "Hello World!";`  
`String s = (String) o;`

# Catching Exceptions

- type conversion operations are error-prone
- Example: `Object o = new Integer(23);`  
`Strings s = (String) o;`
- good idea to avoid type casts
- sometimes necessary, e.g. when implementing `equals` method
- use try-catch statement to handle error situations
- Example I: `String s;`  
`try {`  
`s = (String) o;`  
`} catch (ClassCastException e) {`  
`s = "ERROR"; }`

# Catching Exceptions

- use try-catch statement to handle error situations
- Example 2:

```
try {  
    double x;  
    x = Double.parseDouble(str);  
    System.out.println("The number is " + x);  
} catch (NumberFormatException e) {  
    System.out.println("The number sucks.");  
}
```

# Arrays

- array = built-in, mutable list of fixed-length
- type declared by adding “[]” to base type
- Example: `int[] speedDial;`
  
- creation using same “new” as for objects
- size declared when creating array
- Example: `speedDial = new int[20];`
  
- also possible to fill array using “{}” while creating it
- then length determined by number of filled elements
- Example: `speedDial = {65502327, 55555555};`

# Arrays

- array = built-in, mutable list of fixed-length
- access using “[index]” notation (both read and write, 0-based)
- size available as attribute “.length”
- Example:

```
int[] speedDial = {65502327, 55555555};  
for (int i = 0; i < speedDial.length; i++) {  
    System.out.println(speedDial[i]);  
    speedDial[i] += 100000000;  
}  
for (int i = 0; i < speedDial.length; i++) {  
    System.out.println(speedDial[i]);  
}
```

# Command Line Arguments

- command line arguments given as array of strings
- Example:

```
public class PrintCommandLine {  
    public static void main(String[] args) {  
        int len = args.length;  
        System.out.println("got "+len+" arguments");  
        for (int i = 0; i < len; i++) {  
            System.out.println("args["+i+"] = "+args[i]);  
        }  
    }  
}
```



# Reading from Files

- done the same way as reading from the user
- i.e., using the class `java.util.Scanner`
- instead of `System.in` we use an object of type `java.io.File`
- Example (reading a file given as first argument):

```
import java.util.Scanner; import java.io.File;
public class OpenFile {
    public static void main(String[] args) {
        File infile = new File(args[0]);
        Scanner sc = new Scanner(infile);
        while (sc.hasNext()) {
            System.out.println(sc.nextLine());
        } } }
```

# Reading from Files

- Example (reading a file given as first argument):

```
import java.util.Scanner; import java.io.*;
public class OpenFile {
    public static void main(String[] args) {
        File infile = new File(args[0]);
        try {
            Scanner sc = new Scanner(infile);
            while (sc.hasNext()) { System.out.println(sc.nextLine()); }
        } catch (FileNotFoundException e) {
            System.out.println("Did not find your strange "+args[0]);
        } } }
```

# Writing to Files

- done the same way as writing to the screen
- i.e., using the class `java.io.PrintStream`
- `System.out` is a predefined `java.io.PrintStream` object
- Example (copying a file line by line):

```
import java.io.*; import java.util.Scanner;
public class CopyFile {
    public static void main(String[] args) throws new
FileNotFoundException {
        Scanner sc = new Scanner(new File(args[0]));
        PrintStream target = new PrintStream(new File(args[1]));
        while (sc.hasNext()) { target.println(sc.nextLine()); }
        target.close(); } }
```

# Throwing Exceptions

- Java uses `throw` (comparable to `raise` in Python)
- Example (method that receives unacceptable input):

```
static double power(double a, int b) {  
    if (b < 0) {  
        String msg = "natural number expected";  
        throw new IllegalArgumentException(msg);  
    }  
    result = 1;  
    for (; b > 0; b--) { result *= a; }  
    return result;  
}
```

# OBJECT ORIENTATION

# Objects, Classes, and Instances

- class = description of a class of objects
- Example: a **Car** is defined by model, year, and colour
- object = concrete *instance* of a class
- Example: a silver Audi A4 from 2013 is an instance of **Car**
- Example (**Car** as Java class):

```
public class Car {  
    public String model, colour;  
    public int year;  
    public Car(String model, int year, String colour) {  
        this.model = model; this.year = year; this.colour = colour;  
    }  
}
```

# Attributes

- attributes belonging to each object are *member variables*
- they are declared by giving their types inside the class
- Example:

```
public class Car {  
    public String model, colour;  
    public int year;  
    ...  
}
```

- visibility can be **public**, **protected**, package or **private**
- for now only **public** or **private**:
  - **public** = usable (read and write) for everyone
  - **private** = usable (read and write) for the class

# Getters and Setters

- getter = return value of a **private** attribute
- setter = change value of a **private** attribute
- Example:

```
public class Car {  
    private String model;  
    public String getModel() {  
        return this.model;  
    }  
    public void setModel(String model) {  
        this.model = model;  
    } ...  
}
```



# Getters and Setters

- very useful to abstract from internal representation
- Example:

```
public class Car { // built after 1920
    private byte year;
    public int getYear() {
        return this.year >= 20 ? this.year + 1900 : this.year + 2000;
    }
    public void setYear(int year) {
        this.year = (byte) year % 100;
    } ...
}
```

# Static Attributes

- attributes belonging to the class are *static attributes*
- declaration by `static` and giving their types inside the class
- Example:

```
public class Car {  
    private static int number = 0;  
    public Car(String model, int year, String colour) {  
        this.model = model; this.year = year; this.colour = colour;  
        Car.number++;  
    }  
    public int getNumberOfCars() { return number; }  
}
```

# Initializing Global and Local Variables

- local variable = variable declared in a block
- global variable = member variable or static attribute
- all local and all global variables can be initialized
- Example:

```
public class Car {  
    private static int number = 0;  
    public String model = "Skoda Fabia";  
    public Car(String model, int year, String colour) {  
        boolean[] wheelOk = new boolean[4];  
    }  
}
```

# Constructors

- objects are created by using “new”
- Example: `Car mine = new Car("VW Passat", 2003, "black");`
- Execution:
  - Java Runtime Environment reserves memory for object
  - constructor with matching parameter list is called
- constructor is a special method with no (given) return type
- Example:

```
public class Car {  
    public Car(String model, int year, String colour) {  
        this.model = model; this.year = year; this.colour = colour;  
    } ...  
}
```

# Constructors

- more than one constructor possible (different parameter lists)
- constructors can use each other in first line using “`this(...);`”
- Example:

```
public class Car {  
    public Car(String model, int year, String colour) {  
        this.model = model; this.year = year; this.colour = colour;  
    }  
    public Car(String model, byte year, String colour) {  
        this(model, year > 20 ? 1900+year : 2000+year, colour);  
    }  
    ...  
}
```

# Overloading

- overloading = more than one function of the same name
- allowed as long as parameter lists are different
- different return types is **not** sufficient!
- Example:

```
public class Car {  
    ...  
    public void setColour(String colour) { this.colour = colour; }  
    public void setColour(String colour, boolean dark) {  
        if (dark) { colour = "dark"+colour; }  
        this.colour = colour;  
    }  
}
```

# Printing Objects

- printing objects does not give the desired result

- Example:

```
System.out.println(new Car("Audi A1", 2011, "red"));
```

- method “`public String toString()`” (like `__str__` in Python)

- Example:

```
public class Car {  
    ...  
    public String toString() {  
        return this.colour+" "+this.model+" from "+this.year;  
    }  
}
```

# PROJECT PART I



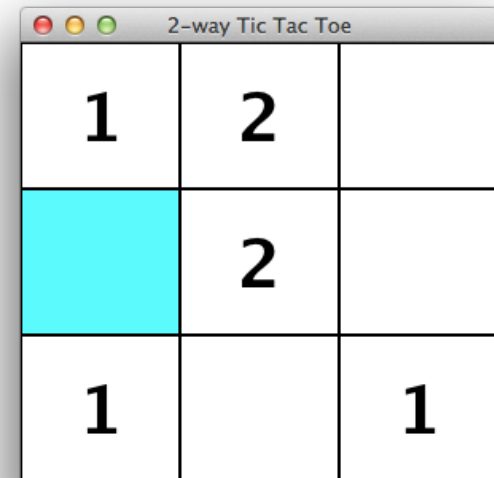
# Organizational Details

- exam project consisting of 2 parts
- both parts have to be passed to pass the course
- projects must be done individually, so no co-operation
- you may talk about the problem and ideas how to solve them
- deliverables:
  - written 4 page report as specified in project description
  - handed in electronically as a SINGLE PDF file
  - deadline: Friday, December 6, 23:59
- ENOUGH - now for the FUN part ...

# Board Games: Tic Tac Toe & Co

- Tic Tac Toe: simple 2 player board game played on a 3 x 3 grid

- extended rules for n-way Tic Tac Toe:
  - n players
  - $(n+1) \times (n+1)$  grid
  - 3 marks in a row, column, diagonal



A screenshot of a window titled "2-way Tic Tac Toe" showing a 3x3 grid. The grid contains the following marks:

1	2	
	2	
1		1

The cell at row 2, column 1 is highlighted in cyan.

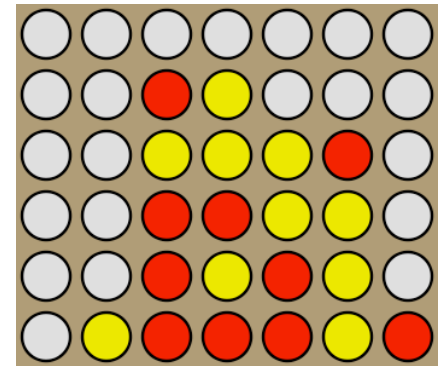
- **Goal:** complete an implementation of n-way Tic Tac Toe
- **Challenges:** Interfaces, GUI, Array Programming

# Board Games: Tic Tac Toe & Co

- Task 0: Preparation
  - download and understand existing framework
  - need to describe design in your report!
- Task 1: Bounding and Shifting Coordinates
  - implement check whether position on board or not
  - implement shift with given differential vector
- Task 2: Implementing the Board
  - get mark for a position or check if it is free
  - record the move of a player
  - check whether there are any moves left
  - check the winning condition

# Board Games: Tic Tac Toe & Co

- Task 3: Testing the Game
  - test game play for standard 2 player 3 x 3 Tic Tac Toe
  - test game play for n-way Tic Tac Toe with  $n > 2$
- Task 4 (optional): Connect Four
  - different simple board game
  - can be implemented similar to Tic Tac Toe
- Task 5 (optional): Go
  - rich board game in a league with chess
  - can be implemented like this, too
  - more challenging!



# **ADVANCED OBJECT-ORIENTATION**

# Object-Oriented Design

- classes often do not exist in isolation from each other
- a vehicle database might have classes for cars and trucks
- in such situation, having a common superclass useful
- Example:

```
public class Vehicle {  
    public String model;  
    public int year;  
    public Vehicle(String model, int year) {  
        this.model = model; this.year = year;  
    }  
    public String toString() {return this.model+" from "+this.year;}  
}
```

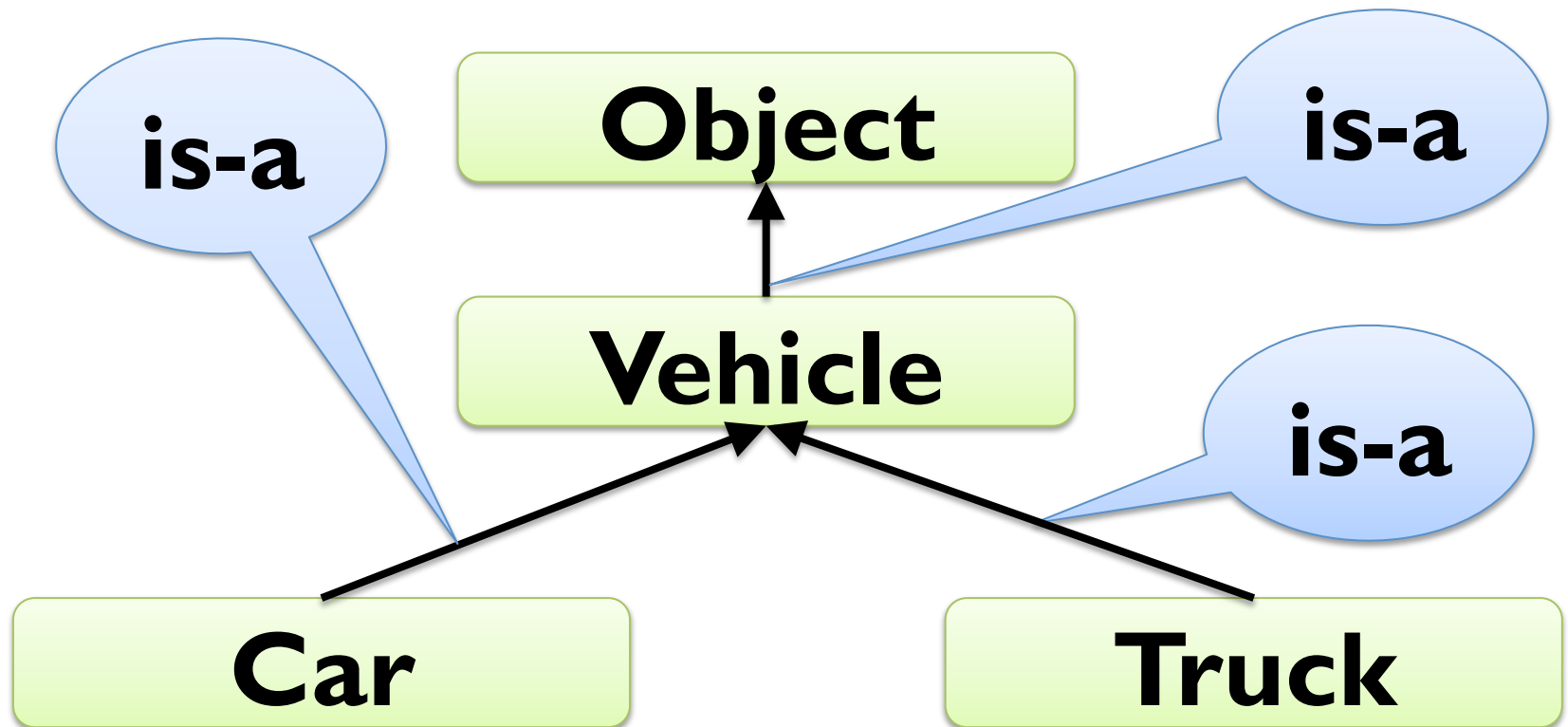
# Extending Classes

- `Car` and `Truck` then *extend* the `Vehicle` class
- Example:

```
public class Car extends Vehicle {  
    public String colour;  
    public Car(string model, int year, String colour) {  
        this.colour = colour;    // this makes NO SENSE  
    }  
    public String toString() { return this.colour; }  
}  
  
public class Truck extends Vehicle {  
    public double maxLoad;  
    ... }  
}
```

# Class Hierarchy

- class hierarchies are parts of class diagrams
- for our example we have:





# Abstract Classes

- often, superclasses should not have instances
- in our example, we want no objects of class `Vehicle`
- can be achieved by declaring the class to be *abstract*
- Example:

```
public abstract class Vehicle {  
    public String model;  
    public int year;  
    public Vehicle(string model, int year) {  
        this.model = model; this.year = year;  
    }  
    public String toString() {return this.model+" from "+this.year;}  
}
```

# Accessing Attributes

- attributes of superclasses can be accessed using “this”
- Example:

```
public class Car extends Vehicle {  
    public String colour;  
    public Car(string model, int year, String colour) {  
        this.model = model; this.year = year; this.colour = colour;  
    }  
    public String toString() {  
        return this.colour+" "+this.model+" from "+this.year;  
    }  
}
```

# Accessing Superclass

- methods of superclasses can be accessed using “super”
- Example:

```
public class Car extends Vehicle {  
    public String colour;  
    public Car(string model, int year, String colour) {  
        this.model = model; this.year = year; this.colour = colour;  
    }  
    public String toString() {  
        return this.colour+" "+super.toString();  
    }  
}
```

# Superclass Constructors

- constructors of superclasses can be accessed using “super”
- Example:

```
public class Car extends Vehicle {  
    public String colour;  
    public Car(string model, int year, String colour) {  
        super(model, year);  
        this.colour = colour;  
    }  
    public String toString() {  
        return this.colour+" "+super.toString();  
    }  
}
```

# Abstract Methods

- abstract method = method declared but not implemented
- useful in abstract classes (and later interfaces)
- Example:

```
public abstract class Vehicle {  
    public String model;  
    public int year;  
    public Vehicle(string model, int year) {  
        this.model = model; this.year = year;  
    }  
    public String toString() {return this.model+" from "+this.year;}  
    public abstract computeResaleValue();  
}
```

# Interfaces

- different superclasses could have different implementations
- to avoid conflicts, classes can only extend one (abstract) class
- interfaces = abstract classes without implementation
- only contain **public abstract** methods (abstract left out)
- no conflict possible with different interfaces
- Example:

```
public interface HasValueAddedTax {  
    public double getValueAddedTax(double percentage);  
}
```

```
public class Car implements HasValueAddedTax {  
    public double getValueAddedTax(double p) { return 42000; }  
    ... }  
}
```

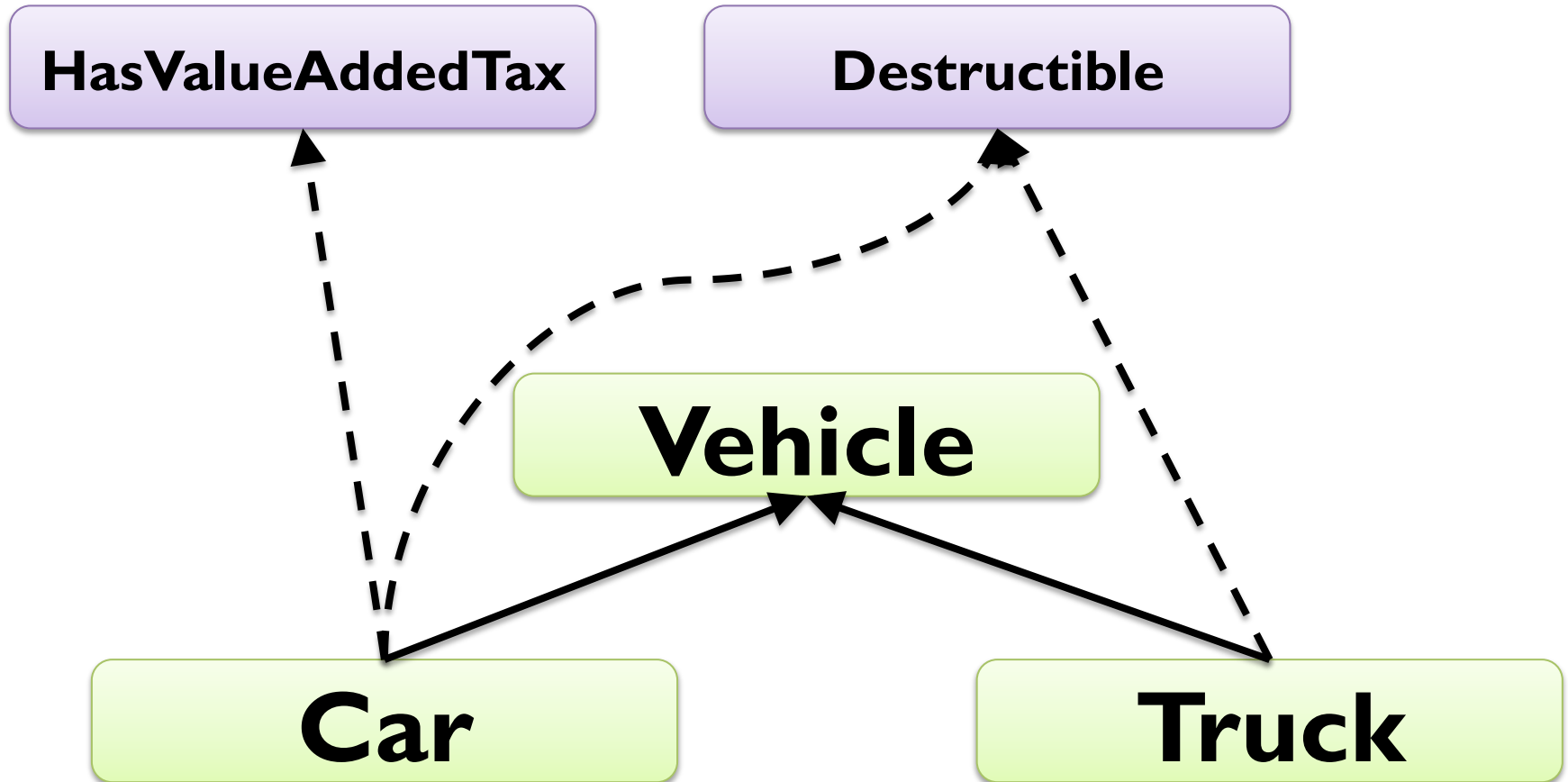
# Interfaces

- Example:

```
public interface HasValueAddedTax {  
    public double getValueAddedTax(double percentage);  
}  
  
public interface Destructible {  
    public void destroy();  
}  
  
public class Car implements HasValueAddedTax, Destructible {  
    public double getValueAddedTax(double p) { return 42000; }  
    public void destroy() { this.model = "BROKEN"; }  
    ...  
}
```

# Interface and Class Hierarchy

- interfaces outside normal class hierarchy





# **GRAPHICAL USER INTERFACES**

# HelloWorld Reloaded

- Java standard GUI package is Swing
- from popup message to professional user interface
- Example:

```
import javax.swing.*;
public class HelloWorldSimple {
    public static void main(String[] args) {
        JOptionPane.showMessageDialog(null, "Hello World!");
    }
}
```

- more challenging to do anything more complicated
- multi-threaded event-driven model-based UI design :-o

# Dialogs

- user dialogs are created using `JDialog` class
- basically like `JFrame` (next slide), but with a parent window
- often used via static `JOptionPane` methods
- Example:

```
Object[] options = {1, 2, 3, 4, 5, 10, 23, 42};
```

```
Object result = JOptionPane.showInputDialog(null,  
    "Select number", "Input",  
    JOptionPane.INFORMATION_MESSAGE, null,  
    options, options[0]);
```

```
int selectedInt = (Integer) result;
```