DM537
Object-Oriented Programming

Peter Schneider-Kamp

petersk@imada.sdu.dk

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

PROJECT PART I

Organizational Details

- exam project consisting of 2 parts
- both parts have to be passed to pass the course

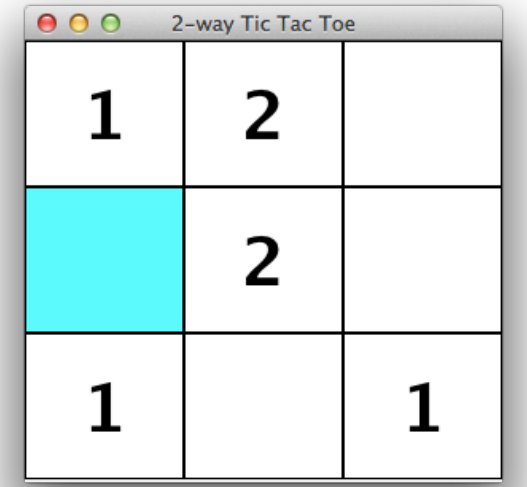
- projects must be done individually, so no co-operation
- you may talk about the problem and ideas how to solve them

- 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) x (n+1) grid
  - 3 marks in a row, column, diagonal



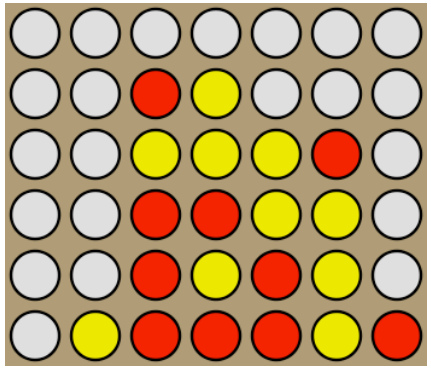
- **Goal:** complete an implementation of n-way Tic Tac Toe
- **Challenges:** Interfaces, GUI, Array Programming

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;}
}
```

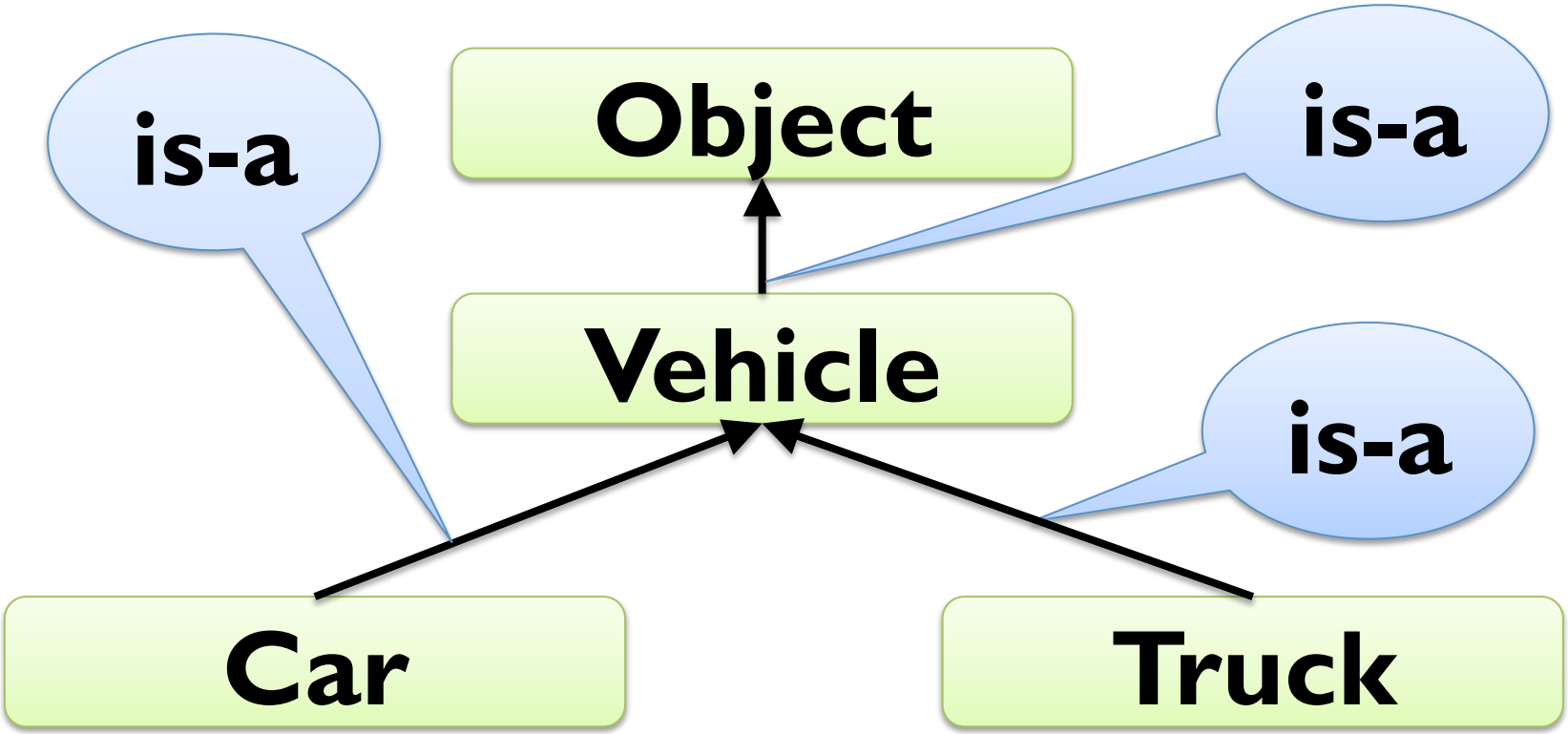
Extending Classes

- 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; }
    … }
```

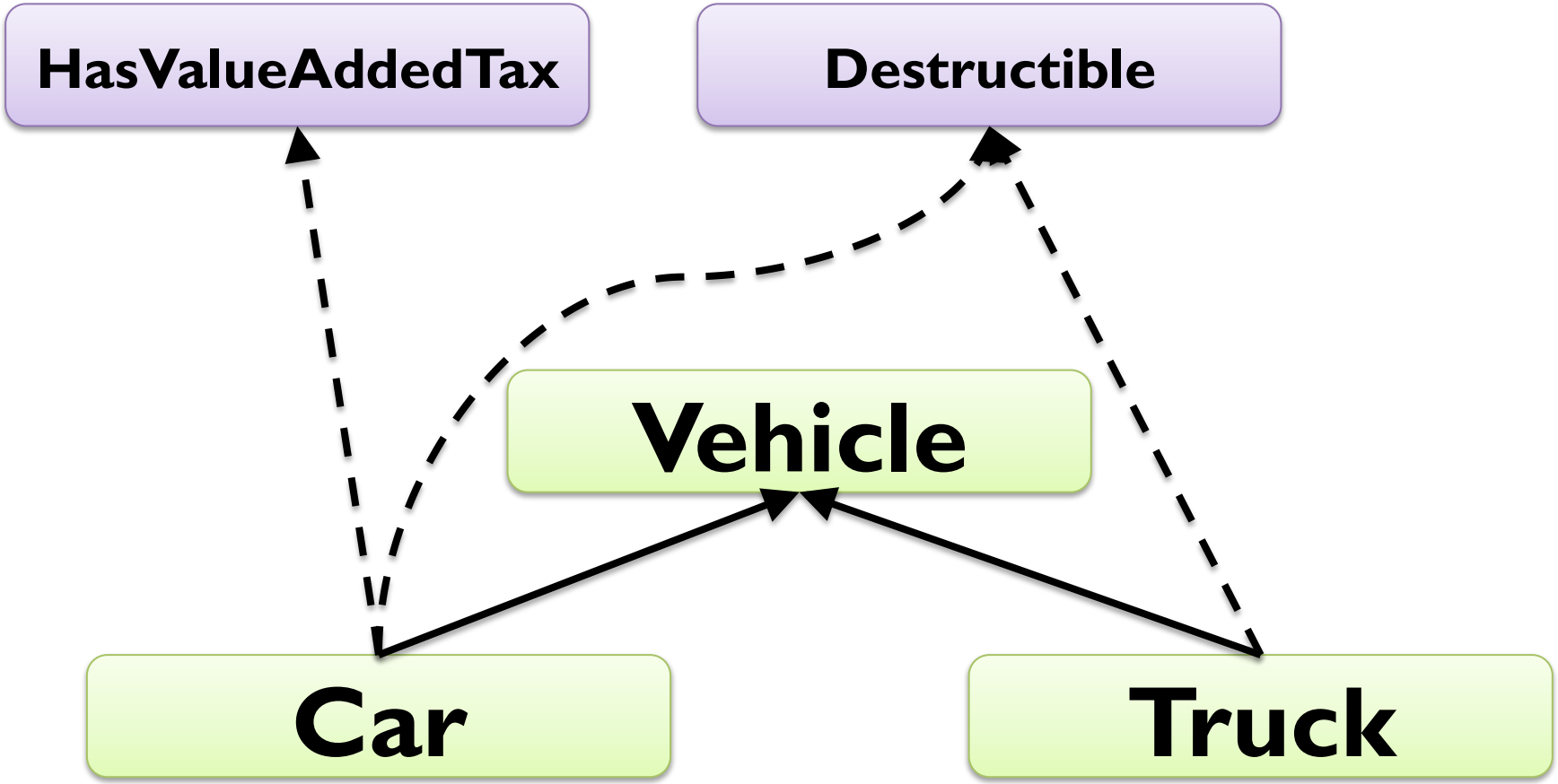
Interfaces

- 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;
```

Creating a Window

- windows are represented by objects of class JFrame
- constructor gets title displayed at top of window
- Example:

```
JFrame window = new JFrame("My first window!");

window.setSize(400, 250);      // set size of window to 700x400
window.setLocation(50, 50);   // top-left corner at (50, 50)

// exit program when window is closed
window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

window.setVisible(true);       // show window on the screen
```

Creating Content

- content is placed in objects of class JPanel
- on these we can either
  - draw directly on it using the paintComponent method
  - add ready-made components using the add method
- every window has a JPanel as its main "content pane"
- Example 1 (draw directly):

```
public class MyPanel extends JPanel {
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        g.drawString("My first panel!", 100, 100);
    }
}
```

Creating Content

- content is placed in objects of class JPanel
- on these we can either
  - draw directly on it using the paintComponent method
  - add ready-made components using the add method
- every window has a JPanel as its main "content pane"
- Example 2 (add a button):

```
JButton button = new JButton("My first button!");
button.addActionListener(new ButtonHandler());
JPanel panel = new JPanel();
panel.add(button);
window.setContentPane(panel);
window.pack();
```

Listeners and Events

- events = changes in the user interface
- mouse movement, key pressed, button clicked, …

- listeners = objects that respond to events
- Example (ActionListener for button from previous slide):

```
import java.awt.*;
import java.awt.event.*;
public class ButtonHandler implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        System.exit(0);
    }
}
```

Mouse Events

- interface MouseListener for mouse events
- needs to be added using addMouseListener methods
- often component class implementing the interface itself
- Example (panel that changes color during click):

```
public class Clicky extends JPanel implements MouseListener {
    public Clicky() {  this.addMouseListener(this);  }
    public void mousePressed(MouseEvent event) {
        this.setBackground(Color.RED);
    }
    public void mouseReleased(MouseEvent evt) {
        this.setBackground(Color.GRAY);
    }  …  }
```

Colors

- colors are represented by objects of class Color
- define by RGB values or use pre-defined constants
- Example:

```
import java.awt.*;
…
JPanel panel = new JPanel(new BorderLayout());
JPanel panelA = new JPanel();
panelA.setBackground(new Color(192, 64,128)); // strange color
JPanel panelB = new JPanel();
panelB.setBackground(Color.RED));
panel.add(panelA, BorderLayout.NORTH);
panel.add(panelB, BorderLayout.SOUTH);
```

Labels

- simple component to display strings or images
- labels are objects of class JLabel
- text, colors, fonts etc. can be changed during runtime
- Example:

```
JLabel label = new JLabel("My first label!", JLabel.CENTER);
…
label.setText("something more interesting");
label.setForeground(Color.BLUE);
label.setBackground(Color.YELLOW);
label.setOpaque(true);                    // background filled
label.setFont(new Font("Serif", Font.ITALIC, 15));
```

Fonts

- fonts represented by objects of class Font
- constructor takes name, style, and point size
- see Java API documentation for more examples
- Example:

```
import java.awt.*;
…
Font font = new Font("Arial", Font.BOLD, 42);
JButton button = new JButton("Click me!");
button.setFont(font);
…
```

Borders

- borders are represented by objects of class Border
- borders can be added to any component
- typically created using static methods in BorderFactory
- Example:

```
JPanel panel = new JPanel(new GridLayout(3,3));
for (int i = 0; i < 9; i++) {
    JPanel subPanel = new JPanel();
    subPanel.setBorder(BorderFactory.createLineBorder(
        Color.BLACK));
    panel.add(subPanel);
}
```

Panel Layout

- layout = spatial organization of components
- components can be either
  - organized by absolute coordinates
  - organized by an object of class LayoutManager
- Example 1 (layout with BorderLayout):

```
JPanel panel = new JPanel(new BorderLayout());
panel.add(new JButton("North"), BorderLayout.NORTH);
panel.add(new JButton("Center"), BorderLayout.CENTER);
panel.add(new JButton("West"), BorderLayout.WEST);
panel.add(new JButton("South"), BorderLayout.SOUTH);
panel.add(new JButton("East"), BorderLayout.EAST);
```

Panel Layout

- layout = spatial organization of components
- components can be either
  - organized by absolute coordinates
  - organized by an object of class LayoutManager
- Example 2 (layout with GridLayout):

```
JPanel panel = new JPanel(new GridLayout(2,3));
panel.add(new JButton("North"));
panel.add(new JButton("Center"));
panel.add(new JButton("West"));
panel.add(new JButton("South"));
panel.add(new JButton("East"));
```

Basic Components

- buttons represented by objects of class JButton
- Example (disabled button with text label):

```
JButton button = new JButton("Big, bad, and ugly!");
button.addActionListener(new MyButtonHandler());
button.setEnabled(false);
```

- check boxes represented by objects of class JCheckBox
- Example (initially selected two-state check box):

```
JCheckBox checkBox = new JCheckBox("more money!", true);
…
boolean wantsMore = checkBox.isSelected();
```

Basic Components

- selectable options represented by objects of class JComboBox
- Example (select from a list of numbers):

```
Object[] options = {1, 2, 3, 4, 5, 10, 23, 42};
JComboBox optionBox = new JComboBox(options);
optionBox.setSelected(6);
optionBox.addActionListener(new MySelectionHandler());
…
int selectedInt = (Integer) optionBox.getSelectedItem();
```

Basic Components

- selection on a range of values by objects of class JSlider
- Example (select percentage from 0 to 100, initally 50):

```
JSlider percent = new JSlider(0, 100, 50);
percent.setMajorTickSpacing(25);
percent.setMinorTickSpacing(5);
percent.setPaintTicks(true);
percent.setPaintLabels(true);
percent.addChangeListener(new MyChangeHandler());
```

Text Components

- text fields represented by objects of class JTextField
- Example (text field for email input):

```
JTextField email = new JTextField();
…
String userEmail = checkRFC5322(email.getText());
```

- text areas represented by objects of class JTextArea
- Example (full-window scrollable editable text entry area):

```
JTextArea entryArea = new JTextArea(5, 20);
textArea.setEditable(true);
JScrollPane scrollPane = new JScrollPane(textArea);
window.getContentPane().add(scrollPane);
```

Menus

- menus represented by JMenuBar, JMenu, and JMenuItem
- Example (menu bar with a single file menu with three items):

```
JMenu file = new JMenu("File");        // create drop down menu
JMenuItem open = new JMenuItem("Open");
file.add(open);          open.addActionListener(this);
JMenuItem save = new JMenuItem("Save");
file.add(save);          save.addActionListener(this);
JMenuItem saveas = new JMenuItem("Save as ...");
file.add(saveas);        saveas.addActionListener(this);
JMenuBar menuBar = new JMenuBar();       // menu bar
menuBar.add(file);
```

Menus

- menus represented by JMenuBar, JMenu, and JMenuItem
- Example (menu bar with a single file menu with three items):

```
public class MyMenu implements ActionListener {
    public MyMenu() {
        …          // see previous slide
    }
    public void actionPerformed(ActionEvent e) {
        …          // check which menu item was clicked and react
    }
}
```

ABSTRACT DATATYPES

Abstract Datatype (ADT)

- abstract datatype = data + operations on the data
- **Idea:** encapsulate data + operations with uniform interface

- operations of a datatype
  - at least one constructor
  - modifiers / setters
  - readers / getters
  - computations

- ADTs typically specified by interfaces in Java

Abstract Datatype (ADT)

- abstract datatype = data + operations on the data
- when specifying an ADT, we describe
  - the data and its *logical* organization
  - which operations we want to be able to perform
  - what the results of the operations should be
- we do NOT describe
  - where and how the data is stored
  - how the operations are performed
- ADTs are independent of the implementation (& language)
- one ADT can have many different implementations!