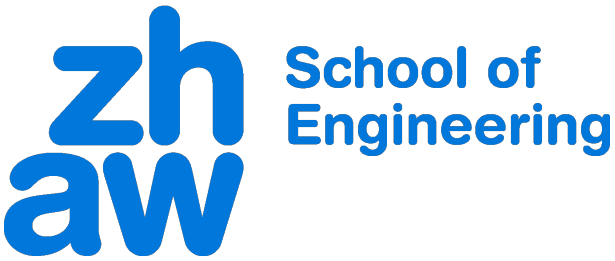
ZHAW Zurich University of Applied Sciences
Winterthur

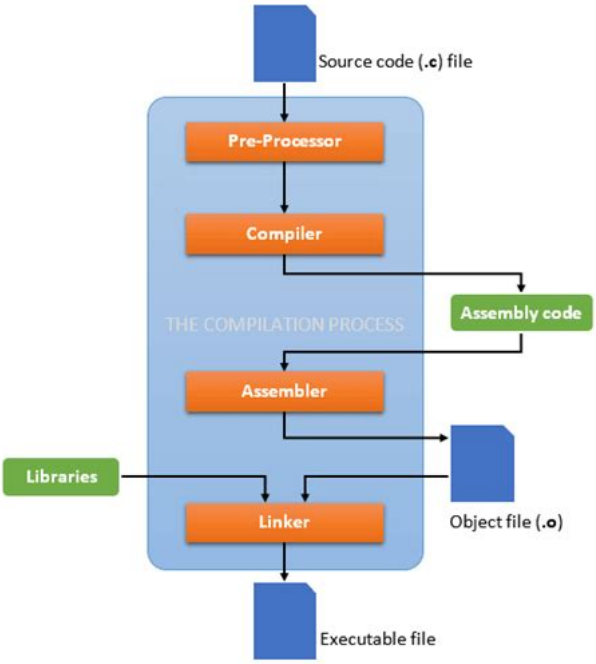


Zusammenfassung INF2
Studienwochen 1-14

Written by: Severin Sprenger
(w/ inputs from lienhyan & hofmaal2)
October 13, 2025
Zf. INF2 SW 1-14

1 C

1.1 Compiler



`#include<...>` → Lib file from standard folder
`#include"..."` → Lib file from cwd
`#ifdef/#ifndef ... #endif` → Conditionals for pre-processor

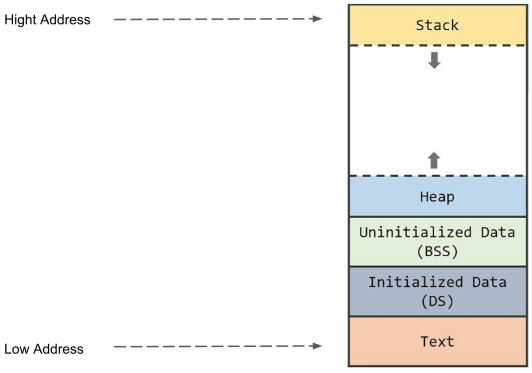
1.1.1 Constants

`__FILE__`	Name of the currently processed file
`__DATE__`	Date of translation
`__TIME__`	Time of translation
`__LINE__`	Line that is being processed
`__STDC__`	Defined when compiler in C mode
`__cplusplus`	Defined when compiler in C++ mode

1.1.2 Example macro

```
#define ALARM(text) \
    printf("******* "); \
    printf("%s", text); \
    printf("******* ");
```

1.2 Dynamic memory (`<stdlib.h>`)



Code: Program code
Data: Global and static variables
Heap: Dynamically allocated memory
Stack: Local data (Variables in functions), Parameters for functions

1.3 Escape sequences

Sequence	ASCII	Description
`\n`	10	new line
`\r`	13	carriage return
`\t`	09	horizontal tab
`\v`	11	vertical tab
`\f`	12	form feed (Curser to start of next page)
`\b`	08	backspace
`\a`	07	bell
`\'`	39	apostrophe
`\"`	34	quote
`\\`	92	backslash
`\nnn`		char value in octal (n = 0..7)
`\xhh`		char value in hex

1.4 Format

Sequence	Output
%d or %i	int
%c	character
%e or %E	double in format [-] d.ddd e±dd
%f	double in format [-] ddd.ddd
%ot	int as octal
%s	string
%p	As pointer address
%u	unsigned int
%x pr %X	int as hex
%%	%

`%ni` output as int / flush right / n characters wide
`%0ni` output as int / flush right / n characters wide filled with 0
`%.nf` output as float / n digits after comma
`%+i` output as int / forces plus or minus sign
`%#x` int as hex / forces `0x` before number

1.5 Allocate memory

```
void *malloc (size_t size)
```

1.6 Allocate memory and zero

```
void *calloc (size_t num, size_t size)
```

1.7 Reallocate memory

```
void *realloc (void *ptr, size_t size)
```

1.8 Free memory

```
void free (void *ptr)
```

1.9 Weird operators

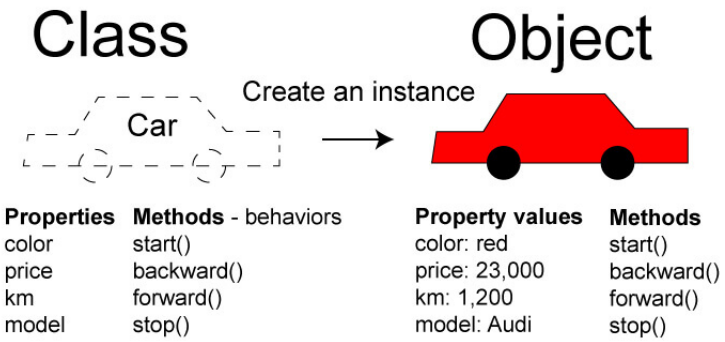
`i++` → Value is returned before and then incremented
`++i` → Value is first incremented and then returned

2 Java

2.1 What is Java?

Java is a cross-platform object oriented programming language. The language is cross-platform because of the use of the JVM (Java Virtual Machine). When Java is compiled during development, it is compiled into byte code that can be executed on the JVM.

2.2 What is a object?



2.3 Basic functions

2.3.1 Main function

```
public static void main(String[] args) {}
```

2.3.2 Constructor

The constructor is a "method" on a class that is called when a new instance of that class is created. The constructor needs to have the same name as the class itself and has no return value. A class can have multiple constructors with different types of arguments.

```
public myTestClass () {}
```

2.3.3 Class structure

```
public class SomeRandomClass {
    /** Any object properties */

    public SomeRandomClass() {
        /** This is the default constructor */
    }

    /** Some object methods */
}
```

2.3.4 Date class

```
Date now = new Date();
String nowStr = now.toString();
```

2.3.5 Final

A variable that is defined `final` can't be reassigned.

2.3.6 Static

For executing a static method of a class, no instance is needed and static functions can't modify data on the class.

2.3.7 This

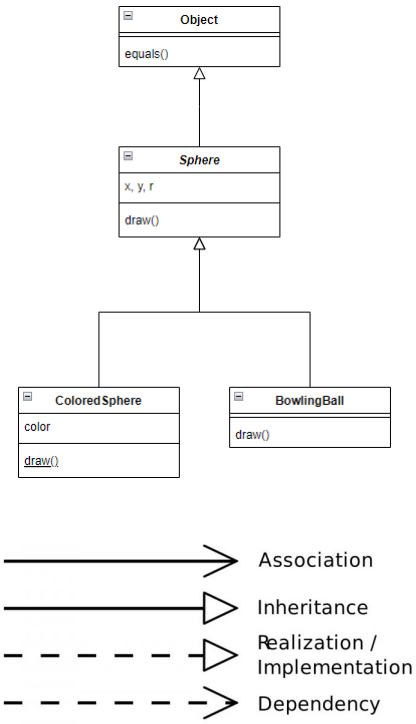
`this` is a reference to the current object and can be used to assign variables on the object.

```
this.gugus = gugus;
```

2.4 Inheritance

A new class can inherit properties and methods from another "template" class. To use a class as a "template" for another class the following syntax is needed. (Note: All classes in Java inherit from a class named Object)

```
public class SomeNewClass extends YourTemplateClass {
    /** Add constructors, methods and properties to the new class */
}
```



2.4.1 Variable visibility

Modifier	Same class	Sub class	Other classes
`private`	yes	no	no
`protected`	yes	yes	no
`public`	yes	yes	yes

2.4.2 Abstract

A method declared in a root class can be marked as `abstact`, this forces the implementation of that function in sub classes. If one method is marked as `abstact` in a class, the whole class needs to me marked with `abstact` as well. This has the result that this class can't be instantiated on its own.

2.4.3 Interface

The keyword `class` can be replaced with `interface`. This has the result that the whole class is interpreted as abstract, so no methods can be implemented in a interface class. Interface classes can be understood as pure template classes.

2.4.4 Implements

To implement a interface class the keyword `implements` needs to be used. The keyword `implements` can be paired with the `extends` keyword. If any attributes are defined in the interface, they are automatically defined as `static final`. Interfaces can also extend other classes.

```
public class gugus (extends hihi) implements mülleimer ...
public interface gugus extends hihi
```

2.4.5 Nested classes

Classes can have other classes defined in them. These can be either defined as `private` or `protected`.

2.4.6 Anonyms classes

A anonyms class is a class without a class name, for example used to implement a action listened or other callbacks.

```
SomeSourceObject.addActionListener(new ActionListener() {
    public void actionPerformed (ActionEvent e) {
        /** Implement a event handler */
    }
});
```

2.4.7 Adapter classes

A adapter class describes a class that implements a interface with mostly empty (default implementations) of a method. If a method needs to do more than the default the developer can implement these requirements in the class that extends the adapter class.

2.4.8 Constructors

The class that inherits from a "template" class can have its own constructor, in this constructor the constructor of the "template" class can be called using `super(...)`. If the constructor of the "template" class doesn't require any parameters, the constructor will be called automatically.

2.4.9 Casting

In Java, class casting is the process of treating an object of one type as if it were another type. This is commonly used when dealing with inheritance, where objects of subclasses can be treated as objects of their superclass (upcasting) or vice versa (downcasting).

- **Upcasting:** This is the process of casting a subclass object to a superclass reference. It's always safe and doesn't require an explicit cast.
- **Downcasting:** This is the process of casting a superclass reference to a subclass object. It requires an explicit cast and can lead to a ClassCastException if the object is not actually an instance of the subclass.

2.5 Events (Java GUI's)

In java events are handele in a event loop. If an events was triggered, its event handler (developer implemented) is called.

You can add a event listener using the following syntax. The following code snipped registers the current class as the event handler.

```
SomeSourceObject.addActionListener(this);
```

If an event is triaged by a source object, the method `actionPerformed` is called and the source object is passed as a argument.

```
public void actionPerformed(ActionEvent evt) {
    /** Handle events */
    repaint();
}
```

2.6 Basic GUI code syntax

```
public class SomeGuiClass extends JFrame implements ActionListener {
    public static void main(String[] args) {
        // Set look and feel from system look and feel
        try {
            UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
        } catch (Exception e) {
            return;
        }

        SomeGuiClass window = new SomeGuiClass();

        // Set window attributes
        window.setTitle("Some window title");
        window.setSize(WINDOW_WIDTH, WINDOW_HEIGHT);

        // Initialize all components and make the window visible
        window.initComponents();
        window.setVisible(true);
    }

    private void initComponents() {
        JPanel panel = (JPanel) this.getContentPane();

        // Set the layout to be used
        panel.setLayout(new FlowLayout());
    }


    private void initComponents() {}

    public void actionPerformed(ActionEvent e) {}
}
```

2.7 Window event

A application can react to window event (focus changes and so on). For that the application class need to implement the WindowListener interface.

```
class SomeAppClass extends JFrame implements WindowListener, ActionListener
```

2.7.1 Avaliable events

```
public void windowOpened (WindowEvent event)
public void windowClosing (WindowEvent event)
public void windowClosed (WindowEvent event)
public void windowIconified (WindowEvent event)
public void windowDeiconified (WindowEvent event)
public void windowActivated (WindowEvent event)
public void windowDeactivated (WindowEvent event)
```

2.8 Layout managers

A layout manager defines how the content added to a window is organized. The following layouts are available:

2.8.1 FlowLayout

```
FlowLayout() // centered, left to right
FlowLayout(FlowLayout.LEFT) // left to right
FlowLayout(FlowLayout.RIGHT) // right to left
panel.add(SomeComponent);
```

2.8.2 BorderLayout

```
BorderLayout() // 5 areas with no borders
BorderLayout(int hg, int vg) // 5 areas with pixel borders
BorderLayout.(NORTH, EAST, WEST, SOUTH, CENTER) // Available locations
panel.add(SomeComponent, BorderLayout.XXX);
```

2.8.3 GridLayout

```
GridLayout(int rows, int cols) // Grid
GridLayout(int rows, int cols, int hg, int vg) // Grid with pixel borders
panel.add(SomeComponent); // Filled from top left to bottom right
```

2.8.4 null

```
SomeComponent.setBounds(int x, int y, int width, int height);
SomeComponent.setLocation(int x, int y);
SomeComponent.setSize(int width, int height);
panel.add(SomeComponent);
```

2.8.5 Nested panels

```
JPanel nestedPanel = new JPanel(new FlowLayout())
nestedPanel.add(SomeComponent)
panel.add(nestedPanel);
```

2.9 Look and feel

Needs to be set in main method before window is opened. This changes the appearance of the gui application.

2.10 Menus

```
JMenuBar menuBar = new JMenuBar();
JMenu someMenuOption = new JMenu("gugus");

JMenuItem SomeMenuItem = new JMenuItem("gugus");
someMenuOption.add(SomeMenuItem);
SomeMenuItem.addActionListener(this);

menuBar.add(someMenuOption);

frame.setJMenuBar(menuBar);
```

2.10.1 Events

To capture menu events the application class needs to implement the ItemListener interface and also needs to implement the `itemStateChanged(ItemEvent e)` method.

2.11 Radio buttons

Default action handler used for all events. `boolean state = SomeRadioButtonInstance.isSelected();` is used to check for state of button.

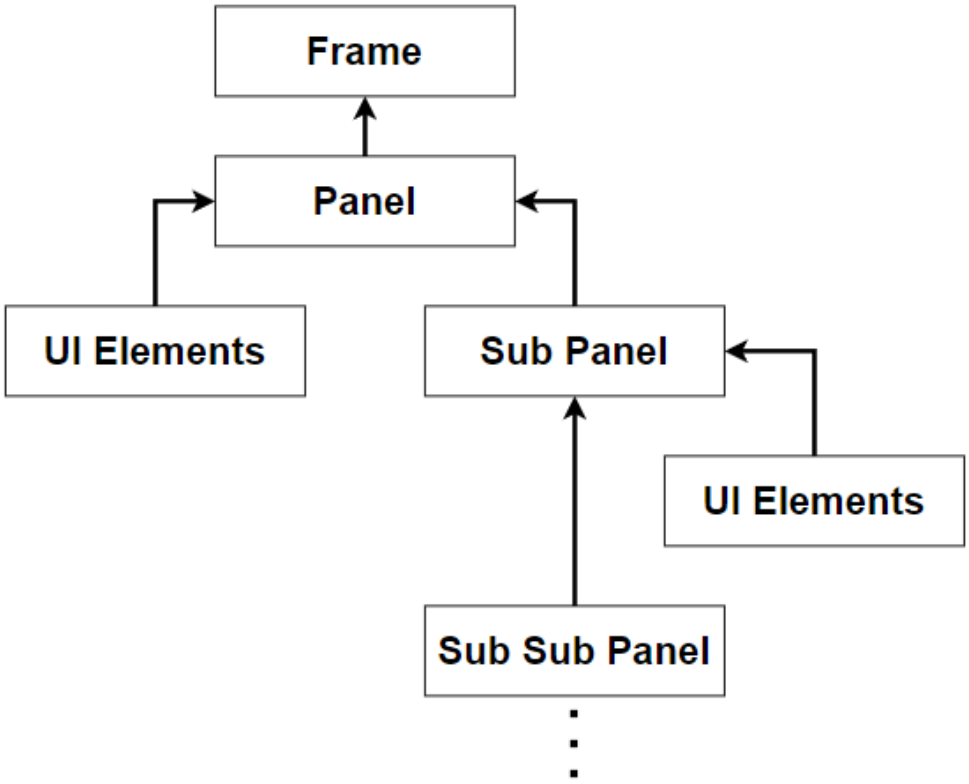
```
ButtonGroup group = new ButtonGroup()
JRadioButton one = new JRadioButton("one", true); // This is the default
JRadioButton two = new JRadioButton("two");
ButtonGroup group = new ButtonGroup();
group.add(one);
group.add(two);
panel.add(group);
```

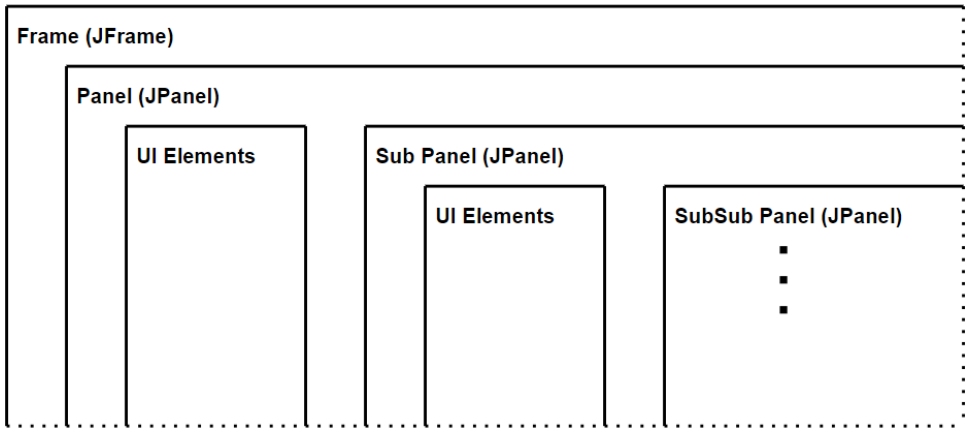
2.12 Combo box

Default action handler used for all events. `String choice = e.getSelectedItem();` is used to check the selected item.

```
JComboBox box = new JComboBox();
box.addItem("one");
box.addItem("two");
box.addItem("three");
```

2.13 Swing Hierarchy



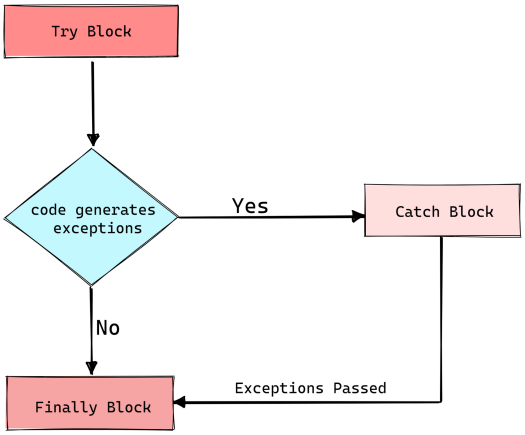


2.14 Exceptions

The basic idea behind exceptions and their handling is to either try to recover from the exceptions or to end the program in a controlled way to not loose any data. Not every exception can be recovered from, so the decision to try to recover or exit needs to be made during development. In the following list some of the possible reactions to exceptions are listed.

- Do nothing, try again (not always possible)
- Exit the program (Possible data lose)
- Message to user (eg. Tell user to correct some input)

2.14.1 Try-Catch-Finally




```
try {
    ... // Code that can cause a exception
} catch (IOException e) {
    ... // Handling of first type of exception
} catch (Exception e) {
    ... // Handling of n'th type of exception
} finally {
    ... // Run after all other code blocks are executed
}
```

2.14.2 Self defined exceptions

`Exception` is a class, that can be extended to create custom exception types. Every custom exception class needs to have a constructor with a String argument, this string needs to be passed to the constructor of the superclass.

```
public class NoGugusException extends Exception {
    public NoGugusException (String message) {
        super(message);
    }
}

String getMessage() // Gets the exception message
String toString() // Classname + exception message
void printStackStrace() // Prints the stack trace to the caused exception
```

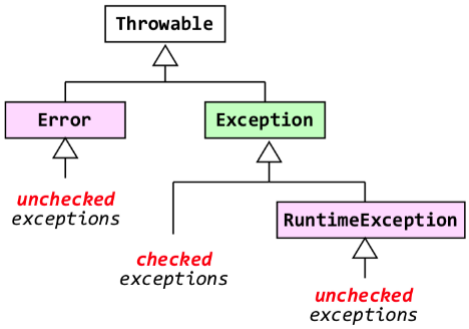
2.14.3 Passing of exceptions

Methods that can cause exceptions need to defined that possible behavior. This is done trough the keyword `throws`.

```
public void getJake() throws NoGugusException, IOException {
    ...
}
```

2.14.4 Trowable class

Errors are major errors that can't be catched and recover from. Exceptions can be catched and can be recover from most of the time.



Checked exceptions need to be handled by the program if not will cause the program to terminate. (eg. `IOException`, `ClassNotFoundException`)

Unchecked exceptions don't have to be handled, but can still cause the program to terminate if not dealt with. If they are not handled, they will be passed onto the JVM. (eg. `NullPointerException`, `NumberFormatException`, `ArrayIndexOutOfBoundsException`)

2.14.5 Things to avoid when dealing with exceptions

- Empty catches (At least print the exception message to stdout)
- Combinatory logic in catches (eg. Catch all exception and use if to differentiate)
- Replacing control logic with exceptions (eg. Indexing of arrays)

2.15 Java shenanigans

- A empty condition trows a compiler error.