Java Stichwortverzeichnis

19. März 2020

Kategorie	Unterkategorie	Beschreibung	Skript
Arrays	Initialisierung	• Datentyp[] myArray = new Datentyp[length];	1D/4
	Indizes	• Zugriff auf Speicherort, startet bei [0]	
	Literal null	• Referenz auf keine Speicheradresse, führend ins 'Nichts'	1D/17
	Längenabfrage	• myArray.length;	1D/24
	Durchlauf	• for(int i = 0; i < myArray.length; i++) {}; • Kurzform: for(Datentyp myNewArray : myArray) {} • statt myArray[i]...; → myNewArray...;	1D/26
	Gemischte Klassen	• Mithilfe von Vererbung möglich (statischer/dynamischer Typ)	1F/89
Arithmetrische Ergebnistypen	Kombinationen	• Kombination immer int, außer einer der Operanden ist long • falls float vorhanden, dann Ergebnis float (selbiges für double) • double > float > long > int	1B/205
Arithmetrische Operationen	Regeln	• Punkt vor Strich	1A/106
	Division bei int	• 2 + 7/3 = 4, falls alle Zahlen vom Datentyp int sind • → Abschneiden der Nachkommastelle	1A/107
	modulo	• Ganzzahlige Division mit Rest (Beispiel: 11 mod 4 = 3)	1A/108
	Kurzformen	• i = i + 5; → i += 5; (-=, *=, /=, %=) • i = i + 1; → i++;	1B/194
	binär und Infix	• Binär: jeder zwei Operanden • Infix: Operator zwischen Operanden • Operationen: + - * / %	1B/195
	unär und präfix	• Unär: nur ein Operand • Präfix: Operator vorangestellt • Vorzeichen: + -	1B/196

	Inkrement Dekrement	• Unär und Postfix (nachgestellt) | i++; i–;
		• Auch als Präfix: ++n; –n;
		  → Zurückliefern des neuen Werts von n
		  • erst n, dann erhöhen | erst erhöhen, dann n
			1B/197

	Zuweisungs-basiert	• = += -= *= /= %=	1B/198
	Vergleiche	• == != < > <= >=	1B/199
	Ternär	• Bedingungsoperator: '?' • z.B.: x = (Bedingung b) ? Falls b true : Falls b false	1B/226
Begriffserklärung	Variablen	• Container, deren Werte veränderbar sind (z.B.: Integer, myVariable,..)	1B/95

	Referenz	• Dadurch wird ein Objekt referenziert (Verweis) (z.B.: Name eines Objekte von Klassen, Arrays,..)	1B/96
	Objekt	• Der Teil, der mit dem Operator new reserviert wird • Objekte != Variablen/ Konstanten und andersrum	
	Konstante	• Syntax: final ...; • Wert kann nicht mehr geändert werden • Referenzen auch nicht änderbar (referenziertes Objekt schon)	1B/100

Bytedaten	Eigenschaften	• byteweises Lesen bzw Schreiben in Dateien • in Praxis jedoch wahrscheinlich irrelevant → Binärdateien einfacher über Bibliotheken zugänglich • hier sinnvoll bei Zugriff auf Bilder oder Videos	8/134
	InputStream	• Abstrakte Klasse • z.B. FileInputStream für Lesezugriff auf Dateien • Subtypen: → java.util.zip.ZipInputStream (für Zip-Dateien) → java.util.jar.JarInputStream (für Jar-Dateien) → javax.sound.sampled.AudioInputStream	8/142
	Einlesen	• Einlesen mithilfe While-Schleife (z.B.: while(true){} • Speichern als int: int n = InputStream.read(); • Erreichen des Dateiendes: if (n == -1) { return; } •-1 als Konvention für unmöglichen Wert	8/135
	Schreiben	• Ziel eines OutputStream: Datensenke • Analog zu InputStream • Methode FileOutputStream.write(int);	8/145
	Buffered-Inputreader	• Kann mehrere Bytes auf einmal einlesen • Erhält intern ein Array von Bytes und gibt immer eins davon aus (Puffer) • Wenn Array leer, wird neues Array eingelesen • Bei Write genau umgekehrt, Byte wird in Puffer zwischengespeichert → Sobald Puffer voll → komplett geschrieben	8/150
	PrintStream	• Dient als Konvertierer in byteweise Repräsentation • System.out: Klassenattribut out ist von Klasse PrintStream • System.in: Klasssenattribut in ist von Klasse InputStream • System.err: Seperates Log für Fehlerdateien → Um z.B. Fehlermeldungen aus anderem Output zu filtern (Log-Datei)	8/157
		•	

Collections	Definition	• Sammlungen von Elementen eines bestimmten Typs • Typ als generischer Typparameter offen • alle im Package java.util	7/2
	Zentrale Elemente	• Collection (Interface): → Alle Collections implementieren dieses Interface • Collections (Klasse): → nützliche Basisalgorithmen (Sortieren) • List (Interface) → Erweitert Collection (z.B.: Reihenfolge auf Elementen) → Elemente in Reihenfolge (definierte Position) • Iterator (Interface): → Durchlauf aller Elemente einer Collection • Beispiele Collection-Klasssen: → Vector, LinkedList, ArrayList, TreeSet, HashSet	7/3
	Syntax	• Collection<Number> c1 = new ArrayList<Number>; → Variable vom Interface Collection instanziiert mit Number → Objekt von Klasse ArrayList (implementiert Collection)	7/10

Collection	• boolean add (T element) throws ... → Hinzufügen von Elementen zur Liste / true falls erfolgreich • addAll, Parameter Typ Collection → Fügt alle Objekte aus übergebener Collection in Aufruf ein • size → Gibt Anzahl der Elemente in Collection aus • isEmpty → Liefert dann true, wenn Collection keine Elemente enthält • contains, Parameter Typ Object → überprüft ob Object in Collection enthalten ist • containsAll, Parameter Typ Collection → true, falls contains für alle Elemente der Collection true • clear → Entfernt alle Elemente aus der aufrufenden Collection • remove, Parameter Typ Object → Liefert true, falls Object vorhanden und entfernt dieses	7/18
List	• indexOf, Parameter Typ Object → Liefert ersten Index zurück, an dem Object zu finden ist → Liefert -1 zurück, falls Object nicht enthalten ist • set, Parameter Typ int und Typ Object → Ersetzt Wert an der Stelle int durch Object → Gibt Element zurück, das vorher an der Position war • add, Parameter Typ int und Typ Object → Fügt Object an Stelle int ein → Schiebt alle Elemente ab int um eins nach hinten → Ohne int: an Ende der Liste (add von Collection) • get, Parameter Typ int → Zugriff auf Element in Liste	7/30
Wildcards	• Funktionsweise synchron zu Generics (siehe Generics	7/52
Lambda in Liste	• Funktionsweise synchron zu Arrays → List<IntPredicate> pred = new ...; → pred.add(n -> n % 2 == 1);	7/65
Sortieren	• mit Comparator (siehe Generics) → Collections.sort(list, new BedingungComparator()); → Sortierlogik durch zweiten Parameter vorgegeben → Comparator Klasse mit selben Typparameter wie list	7/66
Iterator	• Syntax: (Collection<Number> c1 = ArrayList<Number>();) → Iterator<Number> it1 = c1.iterator(); → Gibt Interface Iterator von eigener Iterator Klasse zurück • Verwendung: → while(it1.hasNext()) { it1.next().function(); } • hasNext() → min. 1 Element noch nicht vom Iterator zurückgeliefert • next() → nächstes Element in Collection → bei Collection Reihenfolge unbestimmt → bei Liste Reihenfolge nach Index (ab 0) • Kurzform for-Schleife: → for(String str : coll) {...} → Alle Strings der Collection coll nacheinander	7/75

	Map	• Zwei Parameter: Key und Value → Beispiel Map<String, X> = ... ; → Value wird zum jeweiligen Key gespeichert → Abbildung von den Keys ind die Values • Keys müssen immer unterschiedlich sein, Values nicht • Methode put(key,value); → fügt Element ein • Methode get: X x1 = map.get(key); → Liefert Value zu übergebenen Key zurück → Falls Key nicht vorhanden, return null	7/90

	Eigene LinkedList	• public class ListItem <T> {
		public T key;
		public ListItem<T> next; }
		→ Verkettung der Elemente durch Attribut der eigenen Klasse
		• public class MyLinkedList<T> implements List {
		private ListItem<T> head; }
		→ Verweis auf den Kopf der Liste (erstes Element)
		• Durchlauf durch alle Elemente:
		→ Laufpointer: ListItem<T> p = head; (Laufvariable)
		→ Nächster Schritt: p = p.next;
		→ Vorwärtsschritt = Verweis auf nächstes Element
		→ Abbruchkriterium p gleich null, da Liste zu Ende
		→ for(ListItem<T> p = head; p != null; p = p.next) {}
		• Element hinzufügen:
		• An den Anfang (Index 0):
		→ ListItem<T> tmp = new ListItem<T>();
		→ tmp.key = head;
		→ tmp.next = head;
		→ head = tmp;
		→ Reihenfolge WICHTIG! head ist einziger Verweis auf Liste
		• Beliebige Stelle (Index n):
		→ Arbeiten mithilfe des Laufpointers p
		→ Code erst dann ausführen wenn p an Stelle n-1
		→ tmp.key = key;
		→ tmp.next = p.next;
		→ p.next = tmp;
		• Methode add in java.util.List:
		→ public void add (int pos, T key) {}
		→ Position und einzufügendes Objekt
		• Entfernen eines Elements:
		• Am Anfang (Index 0):
		→ head = head.next;
		→ Überspringen des erstes Elements
		• Beliege Stelle (Index n):
		→ Analog zum Einfügen
		→ p.next = p.next.next;
		• Iterator:
		class MyLinkedListIterator<T> implements Iterator <T> {
		private ListItem<T> p; ... }
		Iterator intern = Verweis auf Listenelement
		Wird durch Konstruktor auf Kopf der Liste gesetzt	

Enums	Syntax	• Modifier enum myEnum {}	1E/6
	Zweck	• Zusammenfassung mehrerer Konstanten zu einer Einheit • vordefinierte Menge an Objekten	1E/7
	Verwendung	• myEnum.ENUM;	1E/9

Exceptions	Laufzeitfehler	• werden nicht vom Compiler entdeckt • Abbruch des Programms • Fehlermeldung mit Call-Stack und Zeilennummer • z.B.: Teilen durch 0	5/2
	Definition	• Klasse java.lang.Exception und alle die davon abgeleitet sind	5/12
	Werfen	• Exception wird in Methodenkopf geworfen: → my-method (...) throws Exception {...} → auch mehrere Exceptions möglich • Wurf: throw new Exception (”Fehlermeldungstext”); → Wirft Exception mit String als Konstruktorparameter • Führt zu Methodenabbruch an der Stelle	5/13
	Fangen	• Falls Methoden potenziell Exceptions werfen könnte: • Einfassen in try-Block: try { .. Methode.. } • Bei Exceptionwurf → Abbruch des try-Blocks → Fangen durch catch-Block: catch (Exception name) {...} • Kein anderer Code zwischen try und catch Block	5/24
	Methoden	• exc.getMessage() Gibt gespeicherte Fehlermeldung aus • exc.printStackTrace() Gibt Call-Stack aus	5/33
	Overwrite	• In überschreibender Methode Ersetzen der Exception durch abgeleitete Exception erlaubt	5/44
	Eigene Klassen	• public myClass extends Exception {} • Konstruktor: .. super(String);..	5/52
	Mehrere Exceptions	• throws Excep1, Excep2 {} → mehrere catch-Blöcke • Jedoch auch allgemein throws Exception möglich, da Supertyp aller Exceptions →selbes Prinzip lässt sich auf catch-Blöcke übertragen → Erster Block in den Exception passt wird verwendet • 2 Exceptions im selben catch-Block: → catch (ExcpA | ExcpB) {...}	5/82
	Weiterreichen	• Methode mit Exception in anderer Methode • Weiterreichen der Exception an aufrufende Methode: → ... throws Exception {} in aufrufender Methode → Dadurch kein try und catch Block für aufgerufene Methode • Spätestens main muss alle weitergereichten Exceptions fangen	5/93
	try-with-ressources	• Dokumentation → Parameter bei try-Block	5/98
	Run-Time-Exceptions	• Ausnahme: Diese müssen nicht gefangen werden • Hauptsächlich zur Übersichtlichkeit, da häufige Verwendung • z.B.: IndexOutOfBoundsException → Sonst jeder Arrayaufruf in try/catch-Block	5/99
	Testphase	• Throwable und Error • Exception und Error von Throwable abgeleitet • Error-Klassen führen immer zum Programmabbruch, werden generell nicht gefangen • Sonderform AssertionError • Funktionstest innerhalb Methode • Kurzform: assert Bedingung: ”String”; → negierte Bedingung in Kurzform! • Warum? Abschaltbar durch Compiler!	5/117

	JUnit-Tests	• Zweck: Funktionstest der Methode als Ganzes • Imports: import static org.junit.Assert.assertEquals; import static org.junit.Assert.assertTrue; import static org.junit.jupiter.api.Assertions.assertThrows; import org.junit.jupiter.api.Test; import org.junit.jupiter.api.BeforeEach; • Verwendung: • Verwenden eigener Quelldatei und Import der Testdatei • @Test führt zur direkten Ausführung der Methode • @BeforeEach Ausführung dieser Methode vor @Test Methoden • Erstellung von void Methoden zum Testen • Funktionen: • assertEquals(param1, param2); → Test auf Gleichheit von 2 Parametern → Mit 3 Parametern: 3. Parameter = Intervall für Abweichung • assertSame(...); • Objektidentität • assertThrows(...); • Überprüft ob erwarteter Error geworfen wird	5/132

Fehler	Compiler	• werden vom Compiler erkannt • z.B.: falsche Klammerung, fehlendes ;, etc	3A/3
	Runtime	• werden während Laufzeit erkannt • z.B.: Division durch 0, Array out of bounds, etc.	3A/4

Files	Wissenswertes	• Package java.nio.file • Sammlung von Klassenmethoden	8/184
	Methoden	• Path path = Paths.get(...); • if(Files.exists(path){} (True, falls dort etwas existiert) • if(Files.isReadable(path){} (True, falls Leserechte vorhanden) • if(Files.isWritable(path){} (True, falls Schreibrechte vorhanden) • if(Files.isRegularFile(path){} (True, falls Typ der Datei regulär ist) • if(Files.isDirectory(path){} (True, falls Typ Verzeichnis) • long size = File.size(path); (Ruft Länge der Datei ab)	8/186
	Manipulation	• Files.createFile(path) (Erzeugt Datei an Pfadstelle • Files.copy(path1,path2); (Kopiert Datei an Pfad 1 nach Pfad 2) • Files.move(path3, path4); (Bewegt bzw benennt die Dateien um) • Files.delete(path); (Entfernt das Objekt an der Pfadstelle) → NoSuchFileException bei Nichtexistieren des Objekts • Files.deleteIfExists(path); (Falls nicht vorhanden, passiert nichts)	8/194

GUI	Window Manager	• Systemprozess → Wird beim Booten gestartet • Läuft als Service im Hintergrund • Stellt generelle Funktionen zur Verfügung	10/2

Frame	• Window mit Rahmen (Subtyp von Klasse Window) • Teil des Packages java.awt (abstract window toolkit) • Standardmäßig unsichtbar • Erzeugung: → Frame frame = new Frame(new String("Hello World!")); → Ein Konstruktor z.B. mit String → Titel des Rahmens • Umgang: → Erzeugung eines Subtypes von Frame um Sachen hinzuzufügen → ActionListener werden oft in SubFrame geschrieben (private class) • Methoden: → frame.setVisible(true); (Fenster sichtbar) → frame.setVisible(false); (Fenster unsichtbar) → frame.setBackground(new Color(0, 0, 0)); (setzt Farbe) → frame.dispose(); (alle Ressourcen werden freigegeben) → frame.setExtendedState(Frame.ICONIFIED); (ikonifiiert) → frame.setExtendedState(Frame.NORMAL); (deikonifiziert) → frame.setExtendedState(Frame.MAXIMIZED_HORIZ); → Setzte Framebreite auf Bildschirmbreite → frame.add(...); (Hinzufügen von z.B. Buttons)	10/11
Buttons	• Erzeugung: → Button button1 = new Button(”String”); → String = Text auf Button → Größe des Buttons wird durch String determiniert • Werden mithilfe von frame.add(button1); ins Frame eingefügt • Methoden: → button.setFont(new Font(”font”, Font.BOLD, size); → button.addActionListener(actionListenerClass); → button.setLabel(”String”); (ändert Titel)	10/25
Action- Listener	• Klasse die ActionListener (java.awt.event) implementiert benötigt • funktionale Methode: void actionPerformed (ActionEvent event) {} • Dann Hinzufügen zum Button: → button.addActionListener(actionListenerClass); → Verwendung eines Lambda-Ausdrucks hier auch möglich (Folie 69) → Automatische Erzeugung eines Threads, der auf Eingaben wartet → Event Dispatch Thread	10/33
ActionEvent	• Hier Verwendung als Parameter für actionPerformed(ActionEvent event) • Methoden: → event.getWhen(); (Rückgabe des Zeitpunkt als long in ms seit 1970) → Hilfreiche Umrechnung dieser mithilfe der Klasse Timestamp → Methoden von Timestamp: getHour(), getMinute()	10/50
Weitere Listener und Events	• Aufbau: Listener / Event • KeyListener / KeyEvent • MouseListener / MouseEvent • MouseMotionListener / MouseEvent • MouseWheelListener / MouseWheelEvent • WindowFocusListener / WindowEvent • WindowListener / WindowEvent • WindowStateListener / WindowEvent • frame.addKeyListener(new MyKeyListener(..)); → analog für Rest	10/70
KeyListener	• public interface KeyListener { public void keyPressed (KeyEvent event); public void keyReleased (KeyEvent event); public void keyTyped (KeyEvent event); } • Jede dieser Methoden ähnlich wie actionPerformed • Zu jedem nicht functional Listener-Interface gibt es eine zugehörige Adapter-Klasse → Damit nicht alle Methoden implementiert werden müssen → Sondern hier z.B. mithilfe von ”extends KeyAdapter” nur keyPressed	10/77

Mouse Listener	• public interface MouseListener { public void mouseClicked(MouseEvent event)
	public void mousePressed(MouseEvent event);
	public void mouseReleased(MouseEvent event);
	public void mouseEntered(MouseEvent event);
	public void mouseExited(MouseEvent event); }
	• public interface MouseMotionListener {
	public void mouseDragged(MouseEvent event);
	public void mouseMoved(MouseEvent event); }
	• public interface MouseWheelListener {
	public void mouseWheelMoved(MouseWheelEvent event); } • MouseAdapter im
		10/88

Window Listener	• public interface WindowListener {
	public void windowOpened(WindowEvent event);
	public void windowClosing(WindowEvent event);
	public void windowClosed(WindowEvent event);
	public void windowActivated(WindowEvent event);
	public void windowDeactivated(WindowEvent event);
	public void windowIconified(WindowEvent event);
	public void windowDeiconified(WindowEvent event); }
	• public interface WindowStateListener {
	public void windowStateChanged (WindowEvent event); }
	• public interface WindowFocusListener {
	public void windowGainedFocus(WindowEvent event);
	public void windowLostFocus(WindowEvent event); }
		10/98

Adapter	• Verbindung zwischen Interfaces und der Klasse Adapter: • abstract public class KeyAdapter implements KeyListener • abstract public class MouseAdapter implements MouseListener, MouseMotionListener, MouseWheelListener • abstract public class WindowAdapter implements WindowListener, WindowFocusListener, WindowStateListener	10/101
Canvas	• abgegrenzte Zeichenfläche in einem Fenster • public class MyCanvas extends Canvas { ... } → Eigener Subtyp um Methode paint selbst zu implementieren • Zeichenbeispiel mit Code ab 10/112	10/107
Checkbox	• Besteht aus kleinem Button und kurzem Text • Interface ItemListener mit funktionaler Methode: → public void itemStateChanged(ItemEvent event); • checkbox.isSelected(): True, falls checkbox an ist	10/132
Choice	• Repräsentiert ein Auswahlmenü • Choice choice = new Choice(); (am Anfang leer) • choice.add("Yes"); choice.add(No"); • choice.select(int); (Zur Anfangswahl) • Greift auch auf ItemListener zurück • choice.getSelectedItem() (Wird benötigt für Listener) → Zugriff auf ausgewähltes Item	10/146
Label	• Rechteck mit Text / Nicht vom User interagierbar • Label label = new Label(”String”); • label.setAligment(Label.CENTER); (LEFT,RIGHT) • label.setBackground(Color); • Benötigt anderen Komponenten für Veränderungen, z.B. Button	10/154
List	• Diese List in java.awt, andere List (Generics) in java.util • Menü mit Auswahlpunkten • List list = new List(int, boolean); → int gibt an, wieviele Menüpunkte gleichzeitig angezeigt werden → Mehr Objekte als Int → Scrollbar → boolean gibt an, ob mehrere Menüpunkte auswählbar sind • int[] selectedIndexes = list.getSelectedIndexes; → Speichert Positionen der Punkte, die ausgewählt sind	10/167

Scrollbar	• Scrollbar bar = new Scrollbar(..,..,..,..); • 4 Parameter: → Scrollbar.VERTICAL oder Srollbar.HORIZONTAL → Startwert als int → Minimalwert z.B. 0 (keinen Einfluss auf Größe) → Maximalwert z.B. 255 (keinen Einfluss auf Größe) • Funktionales Interface AdjustmentListener mit Methode: → public void adjustmentValueChanged (AdjustmentEvent event); • Methode: event.getValue(); (Liefert Wert der Scrollbar)	10/172
Text-Component	• zwei Subtypen TextField und TextArea • TextField: → Eingabezeile für den Nutzer (Passworteingabe) → Textfield field = new TextField(maxLength); → field.setEchorChar('*'); (Ersetzen der Zeichen) → Reset mithilfe von field.setEchoChar(0); → nutzt KeyListener bzw KeyAdapter → Relevant: event.getKeyChar() • TextArea: → Bereich mit beliebig vielen Eingabezeilen → TextArea area = new TextArea("String", int1, int2, scrollbars); → int1: Anzahl Zeilen | int2: Länge Zeilen → scrollbars: TextArea.SCROLLBARS_BOTH für beide Scrollbars → VERTICAL_ONLY | HORIZONTAL_ONLY | NONE | BOTH → nutzt FocusListener mit focusGained und focusLost	10/185
Hierarchie	• Von Component abgeleitet: → Button | Canvas | Checkbox | Choice → Label | List | Scrollbar | TextComponent → Container | Von Container abgeleitet: → Window | Von Window abgeleitet: → Frame	10/208
Container	• fasst mehrere Komponenten zu einer zusammen • Wichtig: Kann auch andere Container enthalten • Methoden: → void paint (Graphics graphics) → void add (component) (wie bisher) → void add(component, Object constraints) (dazu gleich mehr) → void setLayout (LayoutManager manager) (Layout, nutzt constraints) → void validate() (Aktualisierung)	10/216
Layout-Manager	• Automatische Erstellung bei Erstellung eines Containers/Subtyps → Abhängig vom Typ | Window und Frame: BorderLayout • BorderLayout: → Findet seine Verwendung in add-Methode mit constraints → z.B. BorderLayout.NORTH (.CENTER,.WEST,.EAST,.SOUTH) → Automatische Unterteilung des Frames in 5 Bereiche → bei normaler add-Methode wird immer .CENTER angewandt • Andere LayoutManager: → BoxLayout: Anlegung der Komponenten in Reihe → GridLayout: Matrixartige Anlegung → FlowLayout: Automatische Größenanpassung in Zeilen → CardLayout: Anzeigung der Komponenten nacheinander • frame.validate(): → MUSS nach Größen-/Anzahländerungen aufgerufen werden → richtet Layout wieder richtig aus	10/226

	Java Swing	• Selbe Funktionen wie java.awt, jedoch mehr Funktionen • Package javax.swing enthält: → JFrame extends java.awt.Frame → JComponent extends Java.awt.container → von JComponent abgeleitet: → JButton | JCheckBox | JLabel | JList<T> → JScrollBar | JTextComponent(JTextArea | JTextField) → JButton z.B. indirekt über AbstractButton von JComponent • Tooltips: (MouseOver-Text) → button.setToolTipText("String"); • Randdarstellungen: → .setBorder(Border border); → Nutzung von Klasse BorderFactory → Bsp: BorderFactory.createLineBorder(Color.RED, thickness); → BorderFactory.createEmptyBorder(); • Look and Feel: Plattformunabhängig gleiches Aussehen → Folien 10/260-267 • Key Bindings: → sehr codelastig, deswegen Folie zum Verständnis → 10/268-276 • Drag&Drop • Assistive Technologies • Separierung Hauptmenü und eigentliches Fenster • Weitere Swing-Klassen: → JFormattedTextField (einzugebender Text vorformattiert) → JPasswordField → JRadioButton (kleiner anklickbarer Kreis) → JToolBar → JSlider (Schieberegler) → Popup → JTable (TAbelle) → JTable table = new JTable (Object[][], Object[]) → Erstes Array Matrix | Zweites Array Namen der Spalten → JScrollPane scrollPane = new JScrollPane(table); → Einkapselung des Objekts | Zeigt nur Ausschnitt mit Scrollbars → table.setFillsViewportHeight(true); → Streckung der Tabelle auf die vertikal benötigte Höhe → int[] selectedRows = table.getSelectedRows(); → Abfragen der momentan angeklickten Zeilen	10/240

if-Anweisung	Syntax	• if (Bedingung) {} ; • bei Einzelner Anweisung auch ohne {}	1B/49
	Funktion	• Verzweigung des Programms an gewissen Stellen	1B/49
	else	• Syntax: else {} • Fängt alle Fälle ab, die nicht bei if auftreten	1B/54
	weitere Verzweigung	• switch (Typ) { case myTyp: ... break; default: ... ; } • myTyp keine Variable, muss zur Laufzeit schon feststehen	3C/218

Generics	Wrapper-Klassen	• Zu jedem primitiven Datentypen eine Wrapper-Klasse • Selber Name mit großem Anfangsbuchstaben • Außer int und char → Integer und Character • Konstruktor mit einzelnem Parameter seines Datentyps • z.B.: in = new Integer; • Zugriff mit z.B.: in.intValue(); • Boxing / Unboxing: • Weitgehend austauschbar → automatische Umwandlung	6/1

Klassen	• Beispiel: public class Pair <T1, T2 > {...} → Klasse Pair ist generisch bzw. mit T1 und T2 parametrisiert → T1 und T2 sind Typparameter der Klasse Pair • Konstruktor: z.B.: → Pair <Integer,Double> p = new Pair<Integer,Double>(i, d); → Instanziierung von Pair mit Integer und Double	6/12
Typparameter	• Verwendung bei Attributen: → z.B.: private T1 attribute1; • Verwendung bei Methoden: → als Rückgabetyp → als aktualer Parameter (auch im Konstruktor) → nur bei Objektmethoden • Mögliche Parameter: → Eigene Klassen → Arrays (Primitiv/Referenz) → Interfaces (eingeschränkt → später) → Parametrisierte Klassen und Interfaces	6/15
Methoden	• In nicht generische Klasse: → Parametrisierung vor Rückgabetyp → z.B.: public <T1,T2> Pair<T1,T2> myMethod() ... • in generischer Klasse: → param. Typparameter müssen nicht mehr angegeben werden	6/38
Vererbung	• z.B.: void m (Pair<X,Y> paar) ... → keine generische Methode, da Parameter fest → X oder Y nicht durch abgeleitete Klassen ersetzbar	6/59
Abkürzung	• Pair<String,Integer> pair • pair = new Pair<> ("Hello", 123); → Weglassen der Typparameter bei Initialisierung	6/65
Eingeschränkte Typparameter	• public class A <T extends X> → beschränkt auf X oder Subtyp • Methoden aus X dann auch in der Klasse A verwendbar • z.B.: <T extends Number> für Zahlen (kein Character) • mehrfach eingeschränkt: (auch Interfaces) → <T extends X & Int1 & Int2> → Klasse muss als Erstes kommen	6/72
Wildcards	• Einschränkungen auch bei nichtgenerischen Methoden • Aktualer Parameter: • Upper-Bounded: → double m (X<? extends Number> n) ... → Typ des Parameters n vom Typ Number oder Subtyp • Lower-Bounded: → double m (X<? super Double> d) ... → Typ des Parameters n vom Typ Double, Basisklasse von Double, allen von Double implementierten Interfaces → Hier als Beispiel: Nur Double, Number und Object • Unterschied: • generische Methode: → public <T extends Number> double m (X<T> n) ... • nichtgenerische Methode: → public double m (X<? extends Number> n) ... • nur (X<?> x) → allgemeinster Fall (extends Object) • Verwendung: → In-Parameter: extends → Out-Parameter: super → In/Out-Parameter: weder noch → Rückgabe: weder noch	6/86

	Comparator	• Methode compare mit 2 Typparametern und Rückgabetyp int → public int compare (T t1, T t2) ... ⇒ in Klasse z.B. <T extends Number> • Rückgabewert int: → -1, falls erster Parameter dem Zweiten vorangehend → +1, falls zweiter Parameter dem Ersten vorangehend → 0, falls beide äquivalent • Funktion: → Erlaubt den Vergleich von Elementen auf verschiedenste Art • Beispiel mit Referenztypen: 6/120	6/105
	Einschränkungen	• Keine primitiven Datentypen als Instanziierungen • Keine Erzeugung von Objekten / Arrays von Typparametern • Keine Klassenattribute von Typparametern • Kein Downcast oder instanceof mit Typparametern • Kein throw-catch mit Typparametern • Keine MethodenÜberladung auf Typparametern	6/140

Graphic Fallbeispiel	Applets	• grafikorientiertes Programm (hier in HTML) • Syntax: public class myApplet extends Applet {}	2/11
	Graphics	•	2/14
	GeomShape2D	•	2/38
	Beachte	• Folien nachlesen, zu spezifisch	2

Interfaces	Syntax	• public interface myInterface {}	1G/68
	Methoden	• Rückgabetyp myMethod(); (nicht implementiert, nur definiert) → dürfen aber auch implementiert werden 1G/69 → automatisch public • Objektmethoden: default myMethod();	
	Attribute	• Möglich, allerdings nur als Klassenkonstanten (public final)	4C/38
	Implementierung	• public myClass implements MyInterface1,MyInterface2 {}	1G/72
	Zweck	• Auslagerung von häufigem Code in Interface	1G/65
	Unterschied Abstrakte Klassen	• Interfaces können Mehrfachvererbung • Abstrakte Klassen können: • von Klassen abgeleitet werden • Methoden und Attribute, die nicht public sind	4C/47
	Funktionale Interfaces	• Interfaces mit genau einer funktionalen Methode → Methode, die weder default oder static ist	4C/52

Keywords	Verwendung	• Kann nur an bestimmten Stellen stehen (for, int,while,..)	1A/147

Klassen	Kopf	• Modifier class myClass {}	1E/21
	Attribute	• Eigenschaften der Objekte von myClass	1E/24
	Methoden	• TBD (siehe Methoden)	
	Konstruktor	• Einrichtung von neuen Objekten von myClass → Initialisierung der Attribute • Syntax: Modifier myClass (Parameter) {};	1E/115
	String	• Aufrufe: (.length, .charAt, .indexOf, .matches)	3B/75

	Klassenobjekte	• Syntax: Modifier static ... • Klassenattribut → Zugriff auf immer selbe Speicherstelle • auch über myClass.myStaticVariable ansprechbar	3B/169
	java.lang.object	• kein extends → Object als Basisklasse	3B/198
	Verborgene Informationen	• anonymes Objekt: •Informationen zur Verwendung der Klasse • Methodentabelle: Erstellung zur Laufzeit (3C/35)	3C/18
	Klassen-methoden	• Zugriff auf Klassenmethoden/attribute → jedoch nicht auf Objektmethoden/attribiute • Implementation vom statischen Typ abhängig (3C/113)	3C/52

Kommentare	Einzeilig	• //	1A/186
	Bereich	• /* - */	1A/187
	Javadoc	• /** - */ (nach /** Enter) (3C/90) • Tags @ und neue Zeilen mit * @param für Parameter @throws für Fehlermeldungen @return Für Rückgabewerte	1A/188

Konventionen	Identifier	• Methoden/Variablen: myMethod/myVariable • Klassen: MyClass • Konstanten: MY_CONSTANT (_ statt Leerzeichen) • Package: mypackage → Firmen: umgedrehter Domain Name mit _ • Erstes Zeichen darf keine Ziffer sein, keine Keywords	1A/150

Konversionen	implizite	• Von größerem Datentyp in kleineren → ohne Probleme	1B/211
	explizite	• Zieltyp in Klammern angeben (z.B.: i = (int)myLong)	1B/214

Lambda-Ausdrücke	Syntax	• myFunctionalInterface myVariable = ... -> ... • Einrichten unsichtbarer Klasse, die das Interface implementiert	4C/59
	Funktion	• Abgekürzte Schreibweise für den Aufruf einer Hauptmethode eines funktionalen Interfaces • Bilden der funktionalen Methode: → Parameter -> Methode • Allgemeine Form: → (int n, double d)-> {return ...} → Kurzform nur in einfachen Fällen möglich	4C/63
	Closure	• Information aus Entstehungskontext mitgespeichert (Variable) → Bei Verwendung des Lambda-Ausdrucks mitverwendet	4C/67
	Definition	• Literale von Funktionstypen	4C/70
	Fallbeispiel Prädikate	• Beispiel zur Verdeutlichung	4C/71

Methoden	Syntax	• Modifier Rückgabetyp myMethod (Parameterliste) {}; • Signatur(unique): Name + Parameterliste	3C/75
	Methoden-aufruf	• myObject.myMethod(Parameterliste);	1A/33
	Rückgabetypen	• void → Liefert keinen Wert zurück • !void → zwingend return ...;	1E/32

	Parameter	• in Parameterliste: (Datentyp myVariable,..)
		• mit this.myVariable wird das Attribut angesprochen
		• formale Parameter:
		  • alle Parameter bei der Methodendefinition
		• aktuale Parameter:
		  • alle Parameter beim Methodenaufruf
		  • Nicht selber Datentyp benötigt, solange implizit konvertierbar
			1E/107

	Abstrakt	• Syntax: abstract Modifier myMethod() • Erzwingt abstrakte Klasse • 'Rahmen' für andere Klassen	
	Variable Parameterzahl	• ... myMethod (double... d){} • Entweder Übergabe eines Arrays oder viele double Werte → Umwandlung in Array vom Typ Double	3C/65

	Überschreiben	• Eigenschaften:
		  • Signatur zwingend gleich
		  • private → nichts → proctected → public
		  • Zugriff von Basis zu Subtyp erweitert (möglich)
		  • Bei Rückgabe von Referenz → durch Subtyp ersetzbar
		  → auch bei Exceptions
			3C/95

	Überladen	• Selber Name, andere Parameterliste (→ unterschiedl. Signatur) • Modifier und Rückgabetyp können variieren	3C/115

Modifier	Packages	• package myPackage; (oberster Befehl | klein) • import myPackage.*; (* → keine Subpackages) • Namenskonflikte → mypackage.myClass ... (Qualifizierung)	1F/93
	Zugriff	• private: nur in Klasse selbst • nichts: private + package • protected: nichts + Vererbung • public: protected + alle Imports • Nur eine public myClass oder myEnum pro Quelldatei	1F/100

Nested classes	Wissenswertes	• Verschaltelte Klassen → eine Klasse in der anderen eingebettet
		• Äußere und innere Klasse
		  → public oder package Klasse = Äußere Klasse
		  → public, private oder protected = innere Klasse
		  → Nur eine Klasse kann public sein! → Name der Quelldatei
			9/2

	Zugriff	• OuterClass.InnerClass
		  → nur möglich, falls innere Klasse public
		• Objekt von Y benötigt ein Objekt von X zum Bezug
		  → Attribut der äußeren Klasse vom Typ der inneren Klasse benötigt
			9/11

Objekte	Erzeugung	• Klasse myClass = new Klasse (Parameterliste);	1A/147

Optional	Eigenschaften	• Hilfsklasse aus dem Package java.lang • Erzeugtes Objekt kapselt ein Objekt seines Typparameters ein • bequeme Möglichkeit um mit Referenz null umzugehen	8/3

	Methoden	• • Erzeugung: → Optional<Number> opt1 = Optional.OfNullable(new Integer(123)); → Optional<Number> opt1 = Optional.OfNullable(null); (leer") • Abruf: → OptionalObject.get(); → Bei null: NoSuchElementException • orElseGet: → Wenn Object leer, Zurücklieferung eines anderes Wertes → opt1.orElseGet(() -> 0); → Gibt 0 zurück, falls Object == null • isPresent: → Wenn Object nicht null, Ausführung des Lambdas • map: → Optional<Number> opt3 = opt1.map(x -> x * x); → opt3 verweist auf Optional-Object, das ein Number-Objekt einkapselt mit dem Quadrat von opt1 → Falls Object null auch Container null • filter: → Liefert Optional-Object vom selben generischen Typ zurück → Wenn Lambda-Ausdruck true, dann speicher des Verweises → Wenn Lambda-Ausdruck false, dann speicher von null → Optional<Number> opt4 = opt3.filter(x -> x%2 == 1);	8/4

Primitive Datentypen	Ganzzahlig	• byte: 8 Bits • short: 16 Bits • int: 32 Bits • long: 64 Bits • Integer.MAX_VALUE / Integer.MIN_VALUE etc.	1B/120
	Gebrochen	• float: 32 Bits • double: 64 Bits (Genauigkeit 500 Millionen mal höher) • Vergleich von gebrochenen Zahlen mit maximalen Fehler • Double.MAX_VALUE / Double.POSITIVE_INFINITY	1B/134
	Literale	• wörtlich hingeschriebene Werte, automatisch int • Typ long: 123L • Typ byte / short: automatisch falls klein genug • gebroche Literale automatisch double • Typ float: 12.34F • Exponenten 1.2E34 ($1{,}2 * 10^{34}$)	1B/125
	Logiktyp	• boolean: binär (true oder false → booleschen Literale) • Operationen: • Negation: !a • Und: a&&b • Oder: a||b	1B/153
	Zeichen	• char: Schriftzeichen in Einzelhochkommas ('a') • Unicode: Kodierung als Zahl mit 16 Bit → ASCII: Zeichen 0-127 | ISO-Latin-1: Zeichen 127-255 • Unicode-Nummer auch hexadezimal '\u....'	
	Overflow	• Falls Wert zu groß für Datentyp • Keine Fehlermeldung, Informationsverlust!	1B/129
	Nullwert	• Falls nicht initialisiert, nur definiert: • Zahlentypen: 0 • boolean: false • Referenzen: null	3C/154

Programm-ablauf	Programm	• Sequenz von Informationen (Quelltext und Java-Bytecode)	1A/61
	Prozesse	• Werden nacheinander von CPU abgearbeitet (Warteschlange) • → Illusion von Multitasking	1A/61
	Anweisung	• kleinste Einheit / der Reihe nach ausgeführt	1A/72

Referenztypen	Definition	• !(Primitive Datentypen)	3B/2
	Aufzählung	• Klassen, Arrays, Interfaces, Enum	3B/3
	Enum	• Begründung: Sind auch Klassen, deswegen Referenzen	3B/14
	Gleichheit	• Test auf Gleichheit problematisch, da Referenzen → Objektidentität vs. Wertgleichheit → deep vs shallow copy	3B/49
	statisch vs dynamisch	• Syntax: static myClass = new dynamic(); • statisch: • unveränderlich mit Referenz verknüpft (darüber definiert) • entscheidet auf welche Methoden zugegriffen werden darf • dynamisch: Typ des Objekts der Referenz • muss gleich oder Subtyp des statischen Typs sein • entscheidet welche Implementation der Methode	3B/152

Schleifen	for-Schleife	• for (Ausführung davor; Bedingung; Ausführung danach) {}; • Beispiel: for(int i = 0; i < x; i++ {}; • bei Einzelner Anweisung auch ohne {} • auch for(;;) möglich → Endlosschleife	1A/130
	while-Schleife	• while (Bedingung) {}; (kopfgesteuert) • do {} while(Bedingung); (fußgesteuert) • bei Einzelner Anweisung auch ohne {}	1A/123
	break;	• Verlassen der nächsthöheren (inneren) Schleife • Fortfahren mit nächster Anweisung nach Schleife	1B/50
	continue;	• Beendet momentanen Schleifendurchlauf • Als Nächstes wieder Prüfen der Bedingung	1B/51

Scope	Definition	• Gültigkeitsbereich von Identifiern	3A/47
	Modifier	• Klassen, Methoden, Variablen etc. → siehe oben	3A/49
	lokale Variablen	• Innerhalb von Methodenrümpfen, Schleifen, etc.	3A/52
	this	• Bei gleichen Namen von lokalen Variablen und Attribute → this	3A/56

Semikolon	Verwendung	• Semikolon nach jeder abgeschlossenen Anweisung (;)	1A/22

Speicher-modell	Abstraktion	• großes Feld von Maschinenwörtern (Länge immer gleich, aber abhängig von der Hardware (32bit / 64bit)	1A/24
	Speicher-nutzung	• Name eines Objekts/Arrays wird als Referenz auf Speicherort abgelegt • → deswegen Operator new"	1A/28
	Primitive Datentypen	• Name verweist tatsächlich auf konkrete Speicherstelle	1A/139
	Program Counter	• enthält Adresse der nächst auszuführenden Anweisung	1A/143
	Methoden-aufruf	• Einrichten des Stackpointers | Callstack | Frame	1E/51
	Referenzen vs Objekte	• Unterschiede im Speicher, siehe Folien	3B/20

	Garbage Collector	• Teil des Laufzeitsystems, löscht unreferenzierte Objekte	3B/210

Streams	Eigenschaften	• Klasse Optional fürs Verständnis relevant! (weiter oben) • generisches Interface im Package java.util.stream • einheitliche Schnittstelle für Listen, Arrays, Dateien,... • von potentiell unendlicher Länge	8/19
	Stream aus Listen	• Stream<Number> stream1 = list.stream(); • Stream<Number> stream2 = stream1.filter(myPredicate); → Neuer Stream mit gefilteren Objekten abh. vom Prädikat • Stream<Number> stream3 = stream1.map(myFunction); → Neuer Stream mit angepassten Objekten abh. von der Funktion • Optional<Number> opt = stream3.max(new Comparator()); → Gibt abhängig vom übergebenen Comparator ein Element zurück → Da der Stream auch leer sein kann, Objekt vom Typ Optional • Funktioniert genauso mit Arrays allerdings Parameter bei Arrays.stream mit Arraytyp	8/21
	Erzeugung	• Stream<Number> stream1 = Stream<Number>.of (new Integer(1), new Integer(2));	8/41
	Iteration	• Iteration iter = stream.iterator(); • while (iter.hasNext()) { Number n = iter.next(); ..code..}	8/48
	Liste aus Stream	• List<String> list = stream.collect(Collectors.toList()); → Collectors.toList() liefert generisches Interface Collector zurück • Number[] a = stream.toArray(Number[]::new); → Methodennamen als Lambda-Ausdrücke (Number[]::new): • Fachbegriff method reference • Folien 8/55 - 8/83 lesen | Dokumentation	8/49
	Streams und Dateien	• Erstellen eines Pfades: Path path = Paths.get(..,..,..); → Path und Paths in der Package java.nio.file → Paths.get() erzeugt einen Pfad anhand der Parameter • Stream<String> stream = Files.lines(path); → Öffnet die Datei und gibt sie als Stream von String zurück → Bei Fehlern IOException aus java.io (importieren) • String fileContentAsString = stream.reduce(String::concat); → Erstellt aus allen Elementen des Streams einen einzelnen String	8/95
	IntStreams	• Auch LongStream und DoubleStream • Handhabung analog zu normalen Streams • IntStream stream1 = IntStream.of(1,2,3);	8/102
	Random Zahlen	• Klasse Random in java.util • Random random = new Random(); (Erzeugung eines Objekts) • random.nextDouble; (oder nextInt,..) • Bei Double/Float zwischen 0.0 und 0.1 • Bei Int/Long im Wertebereich • Bei Streams: IntStream stream1 = new Random().ints(); → Füllt Stream mit int stream	8/107

System Properties	Eigenschaften	• java.lang.System • Attribute der Umgebung, in der das Java-Programm läuft • Werden als String gespeichert, kriegen String übergeben	8/84
	Methode	• String homeDirectory = System.getProperty (”user.home”); • Liefert String zurück, übernimmt String als Parameter	8/86
	Mögliche Strings	• ”user.home”: Name des Heimatverzeichnisses • ”user.dir”: Arbeitsverzeichnis des Prozesses • ”user.name”: Name des Nutzers • ”file.seperator”: Zeichen zur Trennung von Pfadnamen • ”line.seperator”: Zeichen zur Trennung von Zeilen	8/87

Textdateien	Wissenswertes	• Textdatei besteht aus Zeichen (chars) • jeder Char ist zwei Byte groß • Suche nach bequemeren Zugriff für Textdateien (nicht byteweise)	8/170
	Reader	• Klasse zum Einlesen von Textdateien • viele reader.read Methoden, Bsp. reader.read(char[] c); → Liest soviele Zeichen ein bis Array voll oder Datenquelle erschöpft → Gibt Anzahl der eingelesenen Chars aus • BufferedReader: reader.readLine(); → Richtet StringObjekt ein und liest ganze Zeile ein • Byteweises Einlesen in Zeichenweises Einlesen: → InputStream in = ...; → Reader reader = new InputStreamReader(in);	8/172
	Writer	• Klasse zum Schreiben von Textdateien • writer.write(char); (schreibt einzelnen Char) • writer.write(String); (schreibt ganzen String) • Umwandlung von Byteweise in Zeichenweise analog zu Reader → OutputStreamReader	8/181

Threads	Wissenswertes	• Verweis auf Kapitel Nested Classes weiter oben • Organisation von parallel laufenden Prozessen • Beschleunigung eines Vorgangs aufgrund der Aufteilung in mehrere Prozesse • Aber nicht immer unbedingt schneller, Wissen notwendig • Starten und Vergessen (fire and forget")	9/21
	Implement-ierung	• Inhalt eines Threads: → Klasse, die das Interface Runnable implementiert → Funktionale Methode public void run() {...} • Einrichtung der Runnable Klasse (9/29-44) • new Thread(runnable).start(); → Kein späterer Zugriff mehr nötig, Thread läuft ab sofort → Programmausführung besteht nun aus zwei Threads → Hier: Main Methode und gestarteter Thread	9/23
	Methoden	• static currentThread → Keine Parameter, liefert ThreadObjekt zurück → Repräsentiert Thread, in dem die Methode aufgerufen wurde • dumpStack → Schreiben des Call-Stacks auf System.err • getAllStackTraces → Liefert Call-Stacks von allen aktiven Threads im Programm → Rückgabe als Map mit Threads als Keys • getId → Gibt ID vom Typ long zurück, bleibt gleich solange Thread aktiv • getName → Gibt Namen zurück • getPriority / setPriority → Prioritätswert vom Typ int • static sleep → Setzt Pause in Millisekunden	9/51
	Streams	• Verknüpfung von Output in Input → Pipe = Verbindung zwischen Lesen und Schreiben → PipedOutputStream out = new PipedOutputStream(); → PipedInputStream in = new PipedInputStream(out); → Runnable runnable = new WriteRunner(out); → new Thread(runnable).start(); → Neuer Thread, der Output ausgibt, der in 2. Stream erzeugt wird → Hier müsste WriteRunner implements Runnable noch erstellt werden	9/60
	Interferierende Threads	• Mehrere Threads Zugriff auf selbe Ressource → Reihenfolge unklar → z.B. printen beide auf Standard Output (System.out) → Vermischung komple	9/79

	Parallelisierung	• Aufspaltung von Prozessen zur Lautzeitoptimierung • Beispiel auf Folien nachzulesen → Beispiel aber schlecht auf andere Situationen übertragbar • (Functional Interfaces, Runnable-Klasse,..) • Abfragen der Prozessorenanzahl: → Runtime.getRunTime().avaiableProcessors();	9/82
	Terminierung	• Hier Verwendung eines booleans toBeTerminated • Falls true, Beenden der While-Schleife in run() → Führt zur Beendigung des Threads • Überprüfung allerdings nur an gewissen Stellen, nicht überall	9/126
	Parallele Streams	• Threads schon in Streams eingebaut • stream.parallelStream().otherStuff()... Verteilt Arbeit an Stream selbstständig auf mehrere Threads	9/151

Vererbung	Syntax	• mySubClass extends myClass	1F/3
	Allgemein	• mySubClass erbt alle Funktionalitäten von myClass	1F/12
	Konstruktor	• Aufruf des Konstruktors von myClass mit super(...); → muss die erste Anweisung im Konstruktor sein • Default Constructor, falls nicht definiert • Konstuktoren werden nicht vererbt	1F/26
	Overwrite	• Neue Methoden Implementierung in mySubClass → Nützlich: super.myMethod(); → Letzte Implementation wird vererbt	1F/45
	Overload	• Methode: Selber Name → unterschiedliche Parameterliste • auch bei Konstruktoren möglich → Aufruf in anderem Konstruktor in selber Klasse mit this()	1E/70
	Subtypen	• alles per ëxtendsäbgeleitete • ëxtends ïmplements"+ Subklassen • Arrays deren Komponententyp ein Subtyp ist • Referenztyp (Supertyp ⟷ Subtyp)	3B/135
	Static Initializer	• Syntax: static {} • Initialisierung von Klassenkonstaten falls sonst nicht möglich • z.B.: Beim Setzen vom Prozessstartdatum	3C/138

		•	
		•	
		•	
		•	
		•	
		•	
		•	