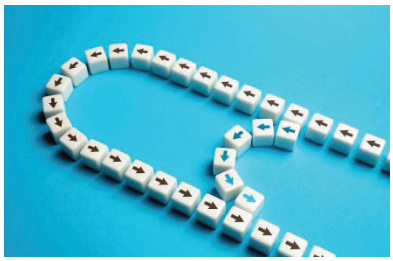
Fach: Anwendungsentwicklung-Java

Name: _____ Datum: _____

Skript zur 12. Klasse

Unsere Aufgabe:
Im letzten Schuljahr wurden die Grundsätze der Java-Programmierung gelernt. Theoretisch wäre es möglich mit diesem Wissen jedes gewünschte Programm zu erstellen. Doch neben Funktionalität sind noch weitere Aspekte in der Anwendungsentwicklung wichtig. In diesem Abschnitt wollen wir unser Programm übersichtlicher gestalten und den Code so gut wie möglich verkürzen.



Wiederholung anonyme Klassen

Eine Möglichkeit den Code zu verkürzen sind anonyme Klassen. Dieses Konzept wurde bereits beim Thema Listener behandelt. Im folgenden Beispiel wird ein Code dargestellt, welcher beim Drücken auf den Button Beenden ein Fenster schließt. Dies wird einmal mithilfe einer überschriebenen Klasse und ein andern Mal mithilfe eines anonymen Listeners realisiert.

```
public class HauptfensterDemo1 extends JFrame {
	public HauptfensterDemo1() {
		this.setSize(300, 100);
		JButton beenden = new JButton("Beenden");
		this.setLayout(new FlowLayout());
		this.add(beenden);
		// ein Objekt der Klasse BeendenListener wird
		// der Methode addActionListener übergeben
		beenden.addActionListener( new BeendenListener() );
	}
}
class BeendenListener implements ActionListener{
	@Override
	public void actionPerformed(ActionEvent e) {
		System.exit(0);
	}
}
```

Dieser Abschnitt wird nur **einmal** benötigt und beinhaltet nur wenige Anweisungen. Deswegen wird er anonym erstellt.

Abbildung 1 überschriebene Klasse

```
public class HauptfensterDemo1 extends JFrame {
	public HauptfensterDemo1() {
		this.setSize(300, 100);
		JButton beenden = new JButton("Beenden");
		this.setLayout(new FlowLayout());
		this.add(beenden);
		// ein Objekt der anonymen Klasse wird
		// der Methode addActionListener übergeben
		beenden.addActionListener( new ActionListener() {
			public void actionPerformed(ActionEvent e){
				System.exit(0);
			}
		});

}
```

Abbildung 2: anonyme Klasse

Begründung des anonymen Listeners:
Da die Klasse BeendenListener nur ein einziges Mal verwendet (instanziiert) wird, ist es einfacher wenn der entsprechende Code direkt dorthin geschrieben wird, wo man ein Objekt dieser Klasse benötigt. Da die Klasse nur einmal verwendet wird, benötigt diese auch keinen Namen → anonyme Klasse.

Dieser bereits verkürzte Code soll nun noch weiter verkürzt werden. Dazu muss man sich folgende Fragen stellen:

- Was erwartet die addActionListener-Methode?
- Welche Methode implementiert die ActionListener-Schnittstelle?
- Welchen Parameter erwartet diese Methode?

Die Antworten auf diese Fragen sind eindeutig. Das wissen wir und auch der Compiler. Noch deutlicher wird es, wenn wir uns überlegen welche Zeilen Code bei einem zweiten anonymen Listener gleichbleiben. Diese Teile wurden in folgendem Code markiert. Die markierten Bereiche sind eindeutig festgelegt.

```
beenden.addActionListener( new ActionListener() {
      public void actionPerformed(ActionEvent e){
            System.exit(0);
      }
});
```

Das ausgelöste (übergebene) Event muss festgelegt werden

Die Aktion ist logischerweise eine andere.

Abbildung 3: zu kürzende Stücke

Wenn man den eindeutigen Code weglässt (da er ja eindeutig und sowieso immer gleich ist), bleibt nur das Event e und die Aktion übrig. Also muss nur gesagt werden, welches Event was auslöst. Das kann so dargestellt werden:

```
beenden.addActionListener(e → { System.exit(0); });
```

Abbildung 4: maximal verkürzt

Bei nur einer Zeile der Aktion können die geschweiften Klammern weggelassen werden. Somit kann unser Programm vom Anfang so dargestellt werden:

```
public class HauptfensterDemo1 extends JFrame {
      public HauptfensterDemo1() {
            this.setSize(300, 100);
            JButton beenden = new JButton("Beenden");
            this.setLayout(new FlowLayout());
            this.add(beenden);
            // ein Objekt der Klasse BeendenListener wird
            // der Methode addActionListener übergeben
            beenden.addActionListener( e →  System.exit(0));
      }
}
```

Abbildung 5: das fertige Programm

Der Ausdruck e → System.exit(0) wird als Lambda-Ausdruck oder nur Lambda bezeichnet.

Lambdas

Hauptzweck der Verwendung von Lambdas ist kürzeren und besser lesbaren Code zu schreiben!

Ein Lambda ist ein Stück Sourcecode, der

- **keinen Namen** besitzt, sondern **nur Funktionalität**, und
- **keine explizite Angabe eines Rückgabetyps** und
- **keine Deklaration von Exceptions** erfordert und erlaubt.

Darstellungsbeispiele

1. Addition und Rückgabe von zwei int-Zahlen.
2. Multiplikation einer long-Zahl mit 2.
3. Eine parameterlose Funktion zur Ausgabe eines Textes auf der Konsole.

```
1. (int x, int y) -> { return x + y; }
2. (long x) -> return x * 2;
3. ( ) -> {String msg = "Lambda"; System.out.println("Hallo " + msg);}
```

Einsatz:

Ein Funktional-Interface ist ein Interface mit genau einer abstrakten Methode. Ein funktionales-Interfacewird auch als SAM-Typ (Single Abstract Method) bezeichnet. Beispiele für funktionale-Interfaces sind ActionListener, EventHandler, Comparator, Comparable usw. Lambdas können anstelle einer anonymen inneren Klasse zur Realisierung eines SAM_Typs eingesetzt werden:

```
// SAM-Typ als anonyme innere Klasse
new SAMTypAnonymeKlasse() {
      public void samTypMethode(METHOD-PARAMETERS){
            METHOD-BODY
      }
}
// SAM-Typ als Lambda
(METHOD-PARAMETERS) -> { METHOD-BODY }
```

Abbildung 6: SAM-allgemein

```
// SAM-Typ als anonyme innere Klasse
button.addActionListener( new ActionListener() {
      public void actionPerformed(ActionEvent e) {
            System.out.println("Button " + e.getActionCommand() + " gedrückt!");
      }
});
// SAM-Typ als Lambda
button.addActionListener( e -> System.out.println("Button " + e.getActionCommand() +
" gedrückt!") );
```

Abbildung 7: SAM- Beispiel

Beispiel 2: Sortieren von Strings nach deren Länge

Arbeitsauftrag 1: Erforschen Sie die Methode compare der (funktionalen) Schnittstelle Comparator!
Arbeitsauftrag 2: Erforschen Sie die Methode sort der Klasse Arrays!

Gegeben sei ein **String-Array mit Namen**. Dieses String-Array soll **nach der Länge der Namen sortiert** werden.
Die (statische) Methode sort der Klasse Arrays verwendet die Schnittstelle Comparator um die Strings zu vergleichen.

Arbeitsauftrag 3: Implementieren Sie die Methoden demo2() & demo3() wie im Code beschrieben!

```
public class Vergleicher implements Comparator<String> {
	@Override
	public int compare(String str1, String str2) {
		return Integer.compare(str1.length(), str2.length());
	}
}
```

```
public class VergleicherAnwendung {
	public static void main(String[] args) {
		demo1();
		demo2();
		demo3();
	}
	/**
	* demo1 verwendet als Comparator eine eigene Klasse,
	* die das Interface Comparator<T> implementiert
	*/
	private static void demo1() {
		System.out.println("Demo 1:");
		Comparator<String> compareByLength = new Vergleicher();
		String[] names = new String[] { "Andreas", "Björn","Johannes","Felix"};
		printNames(names); //diese Methode soll alle Namen drucken
		Arrays.sort(names, compareByLength);
		printNames(names);
	}
	/**
	* demo2 verwendet als Comparator eine anonyme innere Klasse,
	* die das Interface Comparator<T> implementiert
	*/

	/**
	* demo3 verwendet als Comparator ein Lambda
	* (es wird als Parameter übergeben)
	*/
```

Funktional-Interfaces

Der Begriff Interface ist bereits bekannt. Es handelt sich hierbei um **Klassen, die nur abstrakte Methoden** beinhalten. Eine Erweiterung davon sind die Functional-Interfaces in Java. Das sind Interfaces, die nur **eine Methode** besitzen. Die folgende Klasse ist ein Beispiel davon.

```
public interface EinFunktionalesInterface {
	public void ausfuehren();
}
```

Abbildung 8: Interface

Will man dieses Interface benutzen muss es logischerweise in einer Hilfsklasse implementiert werden. Dazu muss eigentlich nur, die Methode ausfuehren() ausprogrammiert werden (Abbildung 9). Anstatt eine neue Klasse zu erstellen, kann man diesen Schritt wie in Abbildung 10 mit einem Lambda erledigen (Es wird ja wie bei einem Listener nur eine Methode implementiert).

```
public class KlassefuerInterface implements EinFunktionalesInterface {
	@Override
	public void ausfuehren() {
		System.out.println("Dafür hast du ein Interface benötig?");
	}
}
```

Abbildung 9: Hilfsklasse, die das Interface implementiert

```
EinFunktionalesInterface besser = () ->
                    System.out.println("Dafür hast du ein Interface benötig?");
```

Abbildung 10: Interface mit Lambda

In diesem Fall wird deutlich, dass auch hier ein Lambda den Code deutlich verkürzt und übersichtlicher macht. Der grundsätzliche Sinn von funktionalen Interfaces wird allerdings erst im nächsten Schritt deutlich:

Eingebaute Fuctional-Interfaces

Java stellt sein Java 8 im Paket java.util.function einige Schnittstellen zur Verfügung, die wie im Beispiel oben nur eine abstrakte Methode beinhalten und deswegen mit Lambdas verwendet werden können. Diese Schnittstellen sind in Abbildung 11 abgebildet und werden von anderen Klassen/Modellen verwendet. Z.B:

- Collection.forEach(Consumer)
- Collection.removeIf(Predicate)
- List.replaceAll(Function)
- Stream.anyMatch(Predicate)