Built-in Functional Interfaces

Java functional interfaces and C# delegates serve similar purposes in that they both allow you to define methods or functions as first-class objects, enabling you to pass methods as arguments, return them from other methods, and store them in variables.

Schnittstelle	Aufgabe
`Predicate<T>`	Überprüft, ob ein Objekt vom Typ T ein Kriterium erfüllt.
`Supplier<T>`	Liefert Objekte vom Typ T (z.B. für get-Methoden)
`Consumer<T>`	Verarbeitet ein Objekt T, gibt kein Ergebnis zurück.
`Consumer<T,U>`	Verarbeitet zwei Objekte vom Typ T und U, gibt kein Ergebnis zurück.
`Function<T,R>`	Verarbeitet ein Objekt vom Typ T und liefert als Ergebnis ein Objekt vom Typ R zurück.
`BiFunction<T,U,R>`	Verarbeitet zwei Objekte vom Typ T und U und liefert als Ergebnis ein Objekt vom Typ R zurück.
`UnaryOperator<T>`	Entspricht Function<T,T>
`BinaryOperator<T>`	Entspricht BiFunction<T,T,T>

Abbildung 11: Übersicht über Java-Interfaces

show this chat to understand evrything here!!!!!!!!!!!!!!!!!!!!!!!!
https://chatgpt.com/c/136bdfad-baf8-4310-9f99-1c54df2882ee

Wir wollen (warum auch immer) eine Funktion erstellen, die einen Integer übergeben bekommt, diesen mit drei multipliziert und als Integer **zurückgibt**.

In Abbildung 13 wurde eine solche Klasse unter Verwendung des Interfaces Function erstellt. Dieses Interface gibt eine Methode apply() vor.

https://www.youtube.com/watch?v=JEQ2MojhJg0

Datentyp des Übergabeparameters

Rückgabetyp der Methode

```
public class Zahlverarbeiter implements Function<Integer, Integer>{
	@Override
	public Integer apply(Integer t) {
		return t*3;
	}
}
```

Abbildung 12: Klasse mit Interface Function

```
Integer zahl = 5;
Function <Integer, Integer> verarbeiter= new Zahlverarbeiter();
Integer neueZahl = verarbeiter.apply(zahl);
```

Abbildung 13: Verwendung der Klasse

Wesentlich effizienter ist es unter Verwendung eines Lambda-Ausdruckes. Hier muss keine Hilfsklasse erstellt werden.

```
Integer zahl = 5;
Function <Integer, Integer> verarbeiter= (value)-> value*3;
Integer neueZahl = verarbeiter.apply(zahl);
```

Abbildung 14: Lambda-Interface

Weiteres Beispiel: Predicate

Ein Predicate testet, ob ein Objekt ein bestimmtes Kriterium erfüllt. Die Schnittstelle beinhaltet die abstrakte Methode test(T t), welche beim Aufruf einen vorher definierten Test durchführt und entweder true oder false zurückliefert. Angewendet wird diese Schnittstelle beispielsweise von der Methode Collection.removeIf(), welche Elemente der Collection entfernt, wenn der Test true zurückliefert.

Im folgenden Beispiel soll eine Arraylist mit Namen durchlaufen und welche, die mit B beginnen entfernt werden. Mit einem Iterator wird es folgendermaßen gelöst:

```
ArrayList <String> besucher = new ArrayList<>();
besucher.add("Müller");
besucher.add("Boateng");
besucher.add("Kahn");
besucher.add("Neuer");
besucher.add("Hummels");
besucher.add("Lahm");
Iterator<String> i = besucher.iterator();
while(i.hasNext()) {
	String e = i.next();
	if (e.startsWith("B")) {
		i.remove();
	}
}
```

Abbildung 15: Filtern mit Iterator

Statt Iterator und Schleife kann hier removeIf() verwendet werden:

```
public class Rausschmeisser implements Predicate<String> {
	@Override
	public boolean test(String e) {
		return e.startsWith("B");
	}
}
```

```
ArrayList <String> besucher = new ArrayList<>();
besucher.add("Müller");
besucher.add("Boateng");
besucher.add("Kahn");
besucher.add("Neuer");
besucher.add("Hummels");
besucher.add("Lahm");
Rausschmeisser r = new Rausschmeisser();
besucher.removeIf(r);
```

Abbildung 16: Filtern mit removeIf()

Noch schneller geht es mit einem Lambda-Ausdruck. Dann kann die Hilfsklasse gekürzt werden:

```
besucher.removeIf(e -> e.startsWith("B"))
```

Abbildung 17: Filtern mit Lambda-Ausdruck

Bei der Function wird ein Wert zurückgegeben. Das Objekt welches in die Funktion gegeben wurde bleibt dabei gleich. Alternativ dazu kann auch ein Consumer verwendet werden. Dieser verarbeitet das Objekt und gibt nichts zurück.

In diesem Beispiel wird ein Consumer erstellt welcher einen übergebenen Integer mit 3 multipliziert.

```
public class Zahlverarbeiter implements Consumer<Integer>{
	@Override
	public void accept(Integer e) {
		e = e *3;
	}
}
```

Abbildung 18: Hilfsklasse

```
List<Integer> zahlen = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9);
//Mit Verwendung der Hilfsklasse

Zahlverarbeiter zv = new Zahlverarbeiter();
zahlen.forEach(zv);

//Mit Lambda-Ausdruck
zahlen.forEach(e -> e = e*3);

//Ausgabe
zahlen.forEach(e -> System.out.println(e));
```

Abbildung 19: Main-Methode forEach

Auch hier wird bemerkbar, dass eine Hilfsklasse für einen solchen Fall unnötig kompliziert ist, da die Klasse erstellt und zudem noch eine Instanz erzeugt werden muss. Auch die Ausgabe auf der Konsole kann mit einem Lambda-Ausdruck realisiert werden.

Achtung:
Die Ausgabe am Schluss des Programms liefert folgenden Text:
1
2
3
…
Die Werte in der Liste werden also nicht verändert oder ersetzt. Es wird lediglich eine Aktion damit ausgeführt.

Arbeitsauftrag:

a. Erstellen Sie eine Methode printTermine(), die alle Termine ausgibt, auf die ein Kriterium zutrifft.
b. Übergeben Sie an printTermine() ein Array mit Terminen sowie einen Lambda-Ausdruck, der alle Termine ausgibt, die für heute geplant sind.