In Java

- Funktionen sind auch Date und können so behandelt werden
- In **Java** über Interfaces eingebaut, Methode `apply` wendet dann die Funktion auf Parameter an

In Racket

- `(define add +) : number number → number` definiert Konstante `add` vom Typ aller Funktionen, für die dieser Vertrag gilt
- Funktionen-> Konstanten, Struct-Attribute und Listenelemente verwenden

---

Funktionen höhrer Ordung

- Funktionen, die Funktionen aufnehmen(Parameter) und Funktionen zurückgeben (Rückgabe)
- Grundlegendes Konzept des Funktionales Programmieren

in Java

- Ein Interface wird übergeben, eine der Methoden des Interfaces implementiert eigentliche Funktion

Racket

```
(define (add-fct-results fct x fct 2)
(+ (fct1 x) (fct 2 y)))
)
// fct 1 wird auf x definiert und fct 2 auf y und dann zusammen addiert.

bsp→ (add-fct-results sqrt 9 sqrt 6)→ 3+36 = 9
```

Functional Interfaces und Lambda Ausdrücke in Java

```
public interface Int{}
```

- Interfaces können zum beispiel Klassenkonstanten-> implizit `public` und `final` `int N = 1;`
- Nicht implementierte Objektmethoden-> implizit `public`
  - `void m1();`
- Implementierte Klassenmethoden, implizit `public`
  - `static void m2();`
- Implementierte Objektmethoden in Form von Default-> `public` und `default`
  - `default void m3();`

Wenn Klasse (über `default` zwei Implementationen derselben Methode erbt, wird Übersetzung abgebrochen, Fix: Methode in Klasse implementieren

Unterschied Interface / abstrakte Klasse

- **Interfaces** können Mehrfach vererbt werde `implements Intf1,Intf2`
- **Abstrakte Klassen** können von Klassen abgeleitet werden und Attribute sowie Methoden haben, die nicht public sind

Functional Interfaces

- Ein Interface, die nur eine Methode hat die weder default noch static ist->funktionale Methode

```
//funktionales Interface
public interface IntToDoubleFunction{
double applyAsDouble(int n);
}
//beispiel implementation der funktion
public class Mult implements IntToDoubleFunction{
private double x;
public Mult(double factor){
x = factor;
}
public double applyAsDouble(int m){
return m*x;
}
```

```
}

...
//ohne lambda
IntToDoubleFunction fct1 = new Mult(10);
double y = fct1.applyAsDouble(11);

//mit lambda Ausdruck
IntToDoubleFunction fct2 = x->x*10
double z = fct2.applyAsDouble(11)
//Compiler erstellt Objekt, namenloser unsichtbarer Klasse
```

Closure

- Lambda Information aus dem Entstehungskontext werden mitgeispeichert

Prädikate

- geben auf Input boolean-Wert zurpck
- Consumer haben Methode `accept()`, die void ist und konsumiert
- `Predicate` methoden zum Beispiel

Lambda Ausdrücke in Java

- abgekürzte Schreibweise für den Aufruf der funktionalen Methode eines (namenlosen, nicht explizit definierten) Functional Interface

Beispiele:

```
n-> n%2 == 1  // Intpredicate Ausdruck in Kurzform
(int n) ->{return n% 2 == 1;} // gleicher Ausdruck in langform
(int n, double x) -> .... // bei mehr als 1 Parameter ist () notwendig
```

- Langform ermöglicht mehrere Anweisungen auf einmal auszuführen

Lambda Ausdrücke in Racket

- Ähnlich zu javas Kurzform

- `(lambda (x y) (+(* x x) (* y y)` ==> `(x,y) → x*x + y*y`

Verschiedene Funktionen in Racket

- `filter`-> filtert alle elemente einer Liste und fügt diese einer neuen hinzu
- `map` -> wendet eine Funktion auf alle Elemente einer liste
- `foldr` und `foldl` durchlaufen alle Elemente von "rechts nach links" oder "links nach rechts"

Methodennamen als Lambda-Ausdrücke

```
public interface DoubleConsumer{
void accept(double x);
}
public class DoublePrinter implements DoubleConsumer{

public void accept(double x){
System.out.print(x);
}
}
//eigene Klasse mit Objekt
DoubleConsumer cons1 = new DoublerPrinter;

//Lambda-Ausdruck
DoubleConsumer cons2 = x→ {System.out.print(x);
// Methodenreferenz
DoubleConsumer cons3 = System.out::print
```

- Methodenreferenz-> `DoubleConsumer cons3 = System.out::print` `
- Compiler kann aus überladener Methode `print` schließen, welche die Richtige ist
- Solange es Methode mit zur funktionalen Methode passender Signatur gibt, gibt der Compiler keine Fehlermeldung