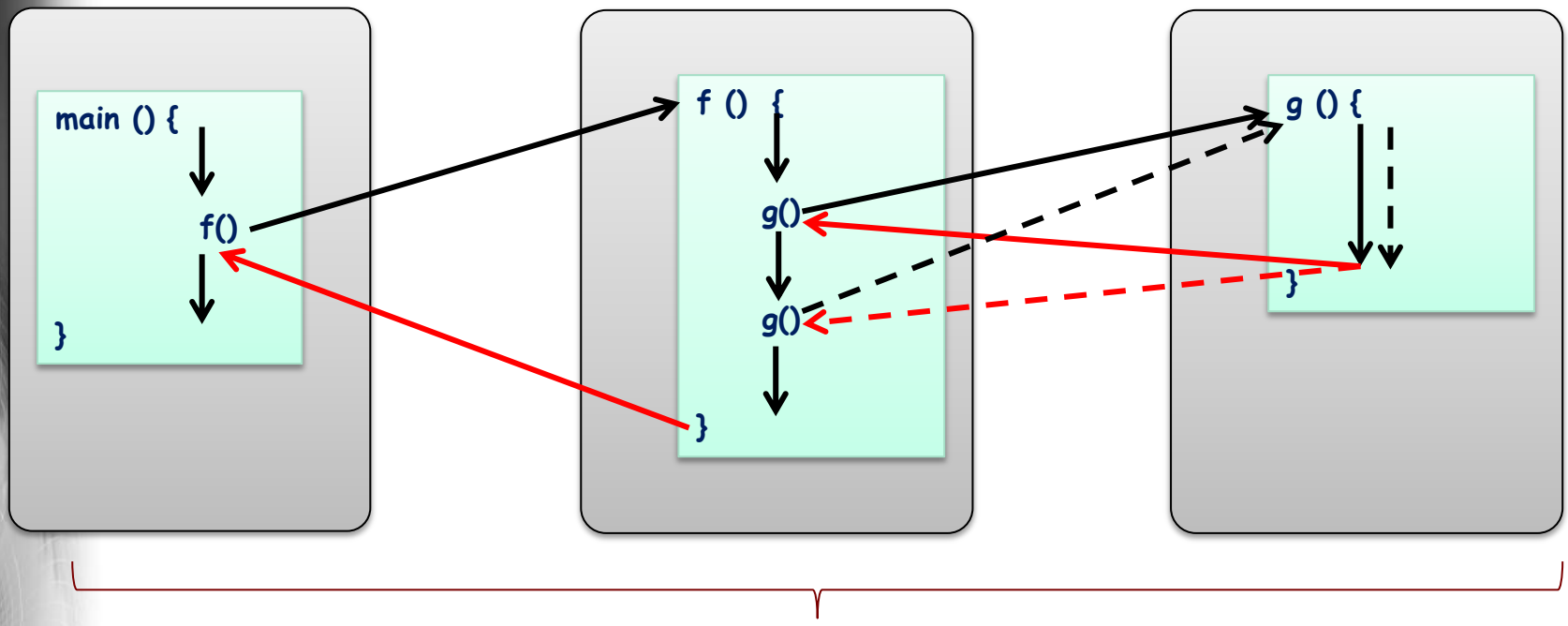
Kurs OMSI
im WiSe 2012/13

Objektorientierte Simulation mit ODEMx

Prof. Dr. Joachim Fischer
Dr. Klaus Ahrens
Dipl.-Inf. Ingmar Eveslage

fischer|ahrens|eveslage@informatik.hu-berlin.de

Routinen (Funktionen)



Koroutinen (Daten + Operationen)

```
class X {…}
class Y {…}
```

```
X *x= new X();
Y *y= new Y();
```

globale Daten

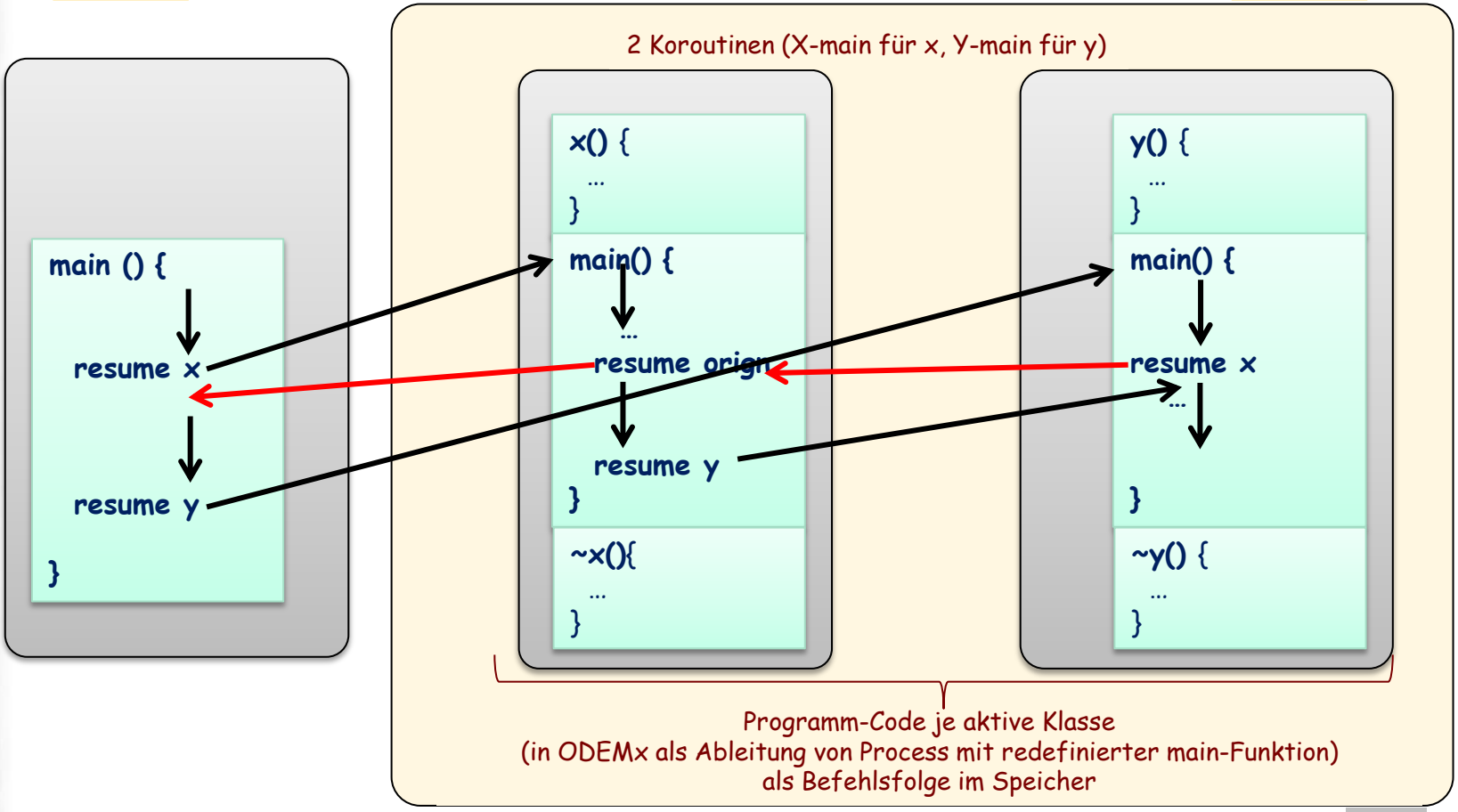
lokale main-Daten

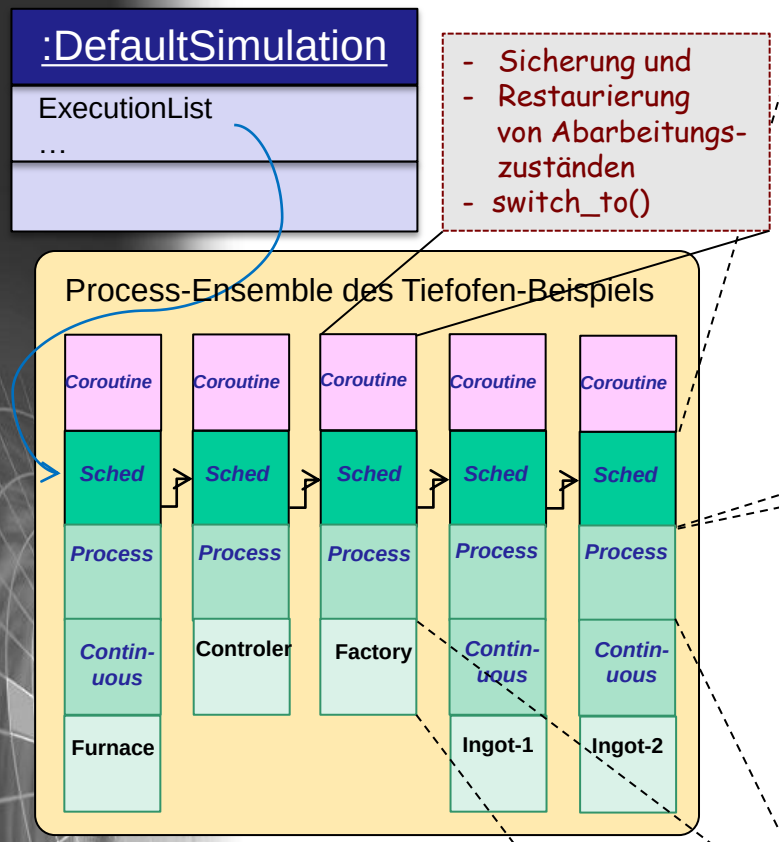
x

lokale x-main()-Daten

y

lokale y-main()-Daten





Sched

#execute()=0
+getExecutionTime() const =0
+getPriority() const =0
+getSchedType() const
+getTime() const
+isScheduled() const
+Sched (Simulation &sim, …)
+setExecutionTime(SimTime time)=0
+setPriority(Priority newPriority)=0
+~Sched ()

abstrakte passive Klasse

Abteilung der **Attribute**
leer: nicht genannt/vorhanden

Abteilung der **Operationen**

Sichtbarkeit: +,-,#,~

Konstruktor sorgt für Zuordnung zu einem Simulationskontext

Process

- ProcessState processState_;
- Priority priority_;
- SimTime executionTime_;

#virtual int main() = 0;
+setExecutionTime(SimTime time);
#void execute();
+void holdFor (SimTime *t*)

abstrakte aktive Klasse

Grundzustände:
CREATED, CURRENT,
RUNNABLE, IDLE, TERMINATED

Priorität:
Gleichzeitigkeitskonfliktbehebung bei der
Sequentialisierung von Ereignissen

Ereigniszeit:
Basis der Sortierung des Terminkalenders (ExecutionList)
SimTime-Typ kann eingestellt werden

Abstrakter Lebenslauf: main
redefinition von execution
(Fortsetzung im Lebenslauf)

Zeitverbrauch (Terminkalender)

Factory

-SimTime dt
-Ingot* ing

#int main() {
holdFor (dt);
ing= new Ingot (…);
}

Konkreter Lebenslauf

Grundidee einer hierarchischen Prozessverwaltung

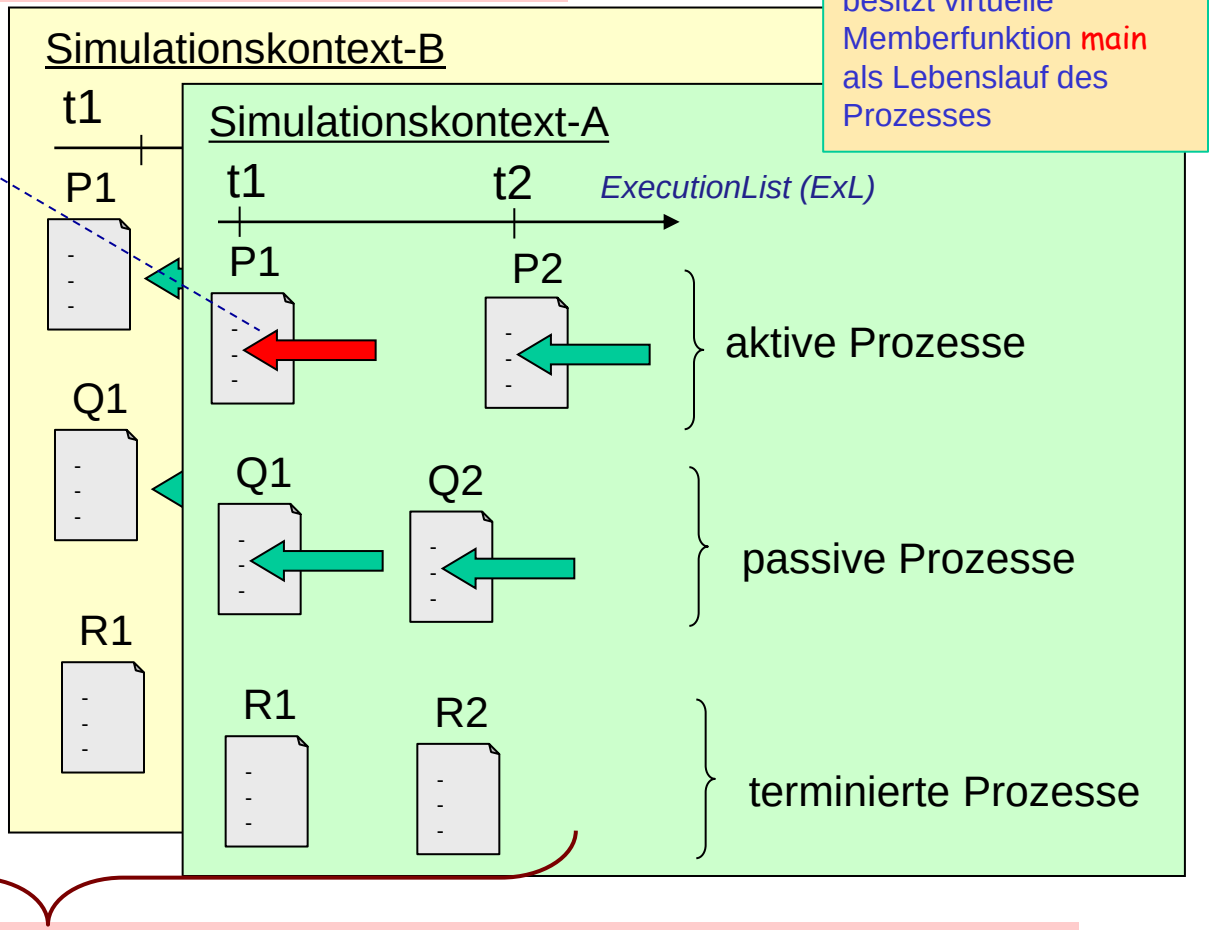
*Zu einem Zeitpunkt kann immer **nur ein Kontext** und in dem **nur ein Prozess** aktiv sein (current)*

Klasse Process
besitzt virtuelle Memberfunktion main als Lebenslauf des Prozesses

Current- Prozess
- erster Eintrag
- kleinste Zeit
- höchste Priorität

C++ Hauptprogramm

```
int main ( ...) {
...
}
```



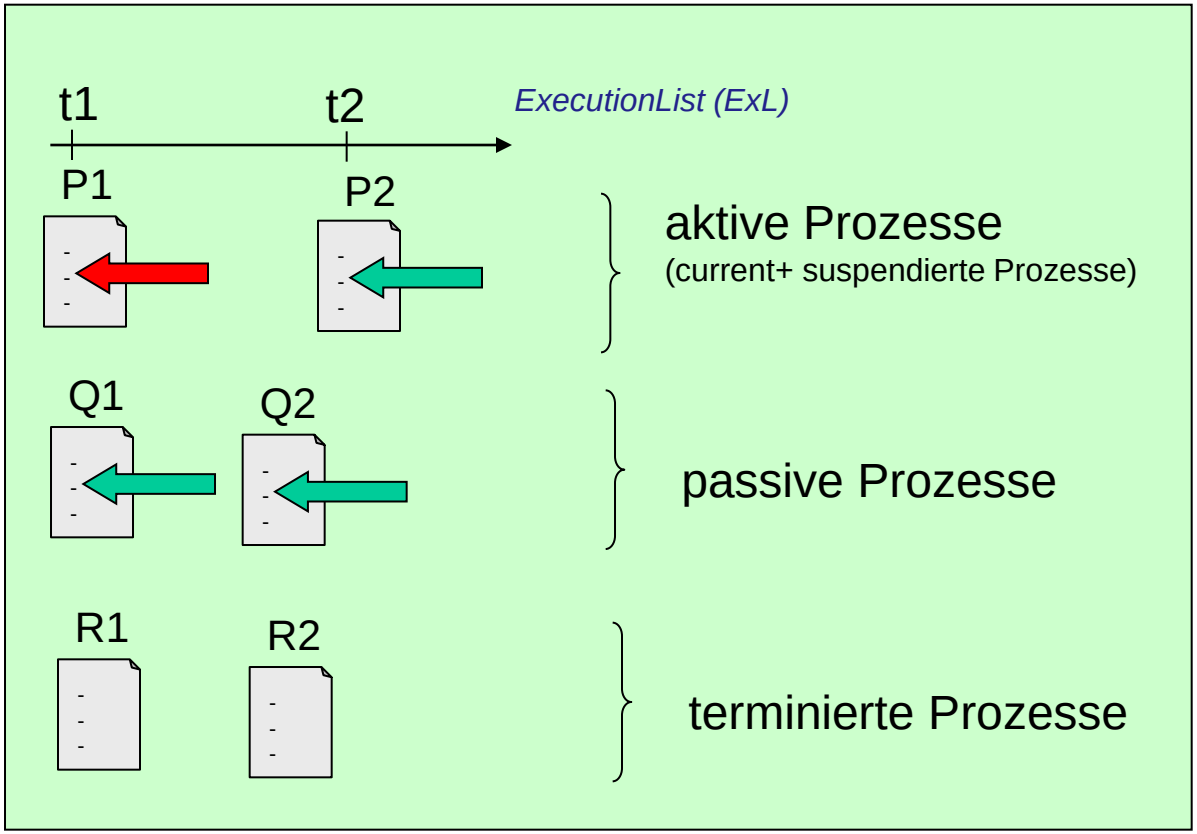
*Hauptprogramm (main-Fkt) und Prozesse (lokale main-Fkt) aller Simulationskontexte bilden ein **hierarchisches Koroutinensystem** auf einer Ein-Prozessor-Maschine*

Standardfall: nur ein Simulationskontext

Zu einem Zeitpunkt ist entweder
- *das Hauptprogramm oder*
- *der Current-Prozess des Kontextes*

aktiv

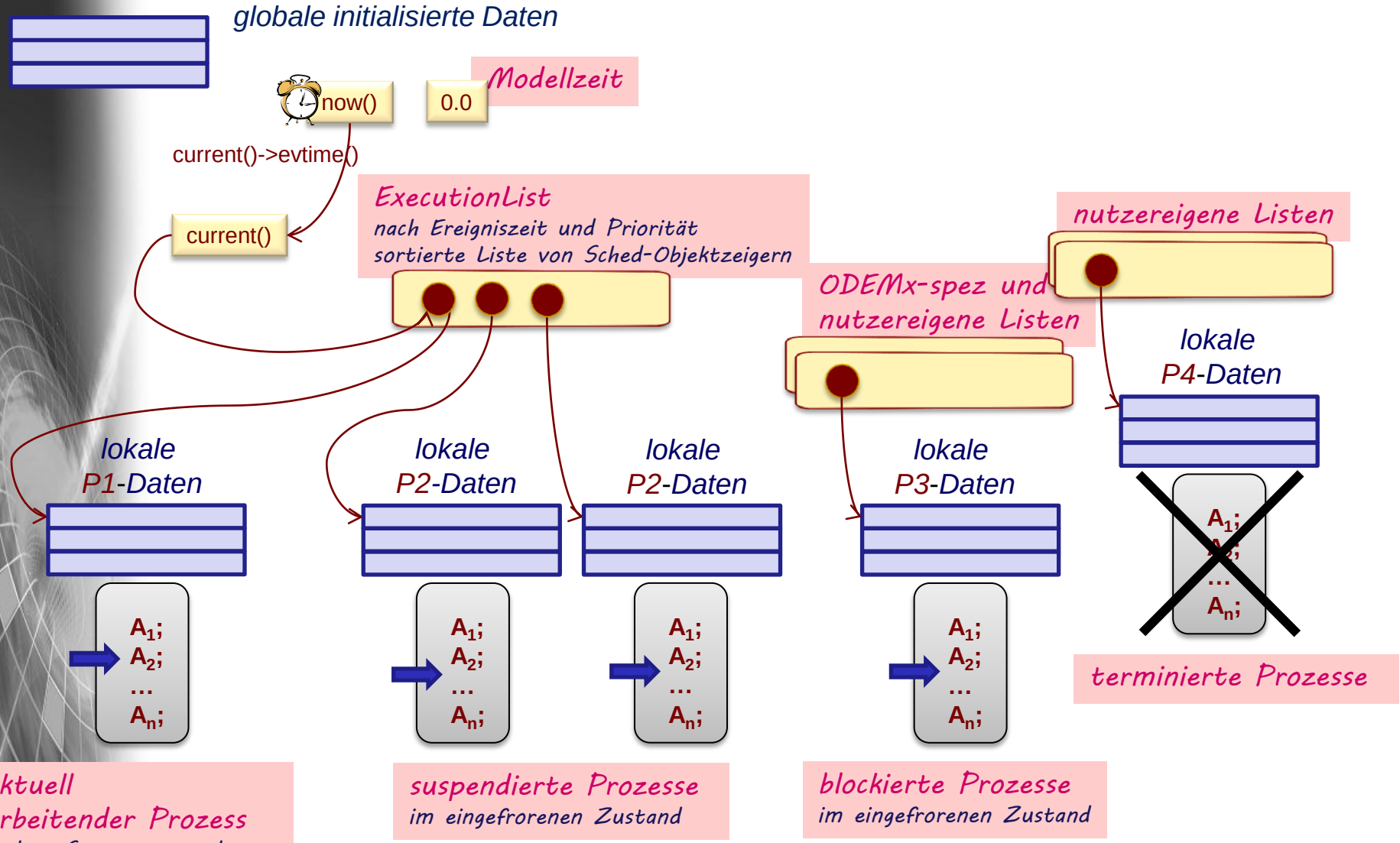
simulation context (DefaultSimulation-Objekt)




```
int main ( ...) {
...
}
```

C++ main program

Schema der Zustandsänderung von ODEMx



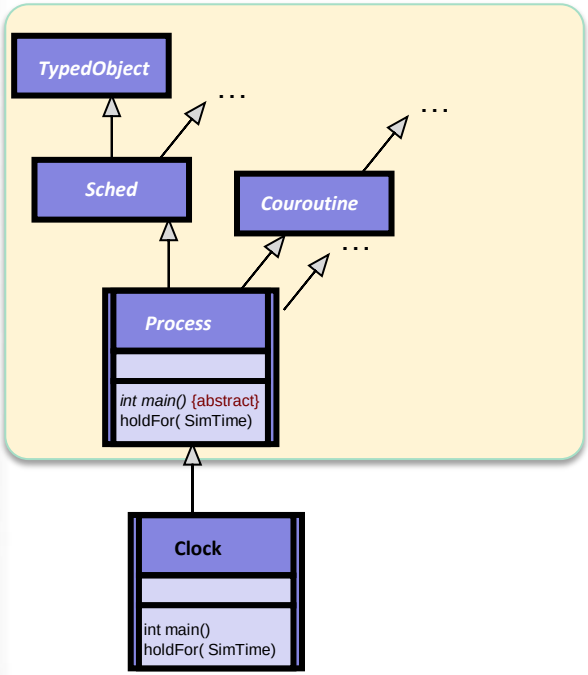
2. Prinzip der Next-Event-Simulation

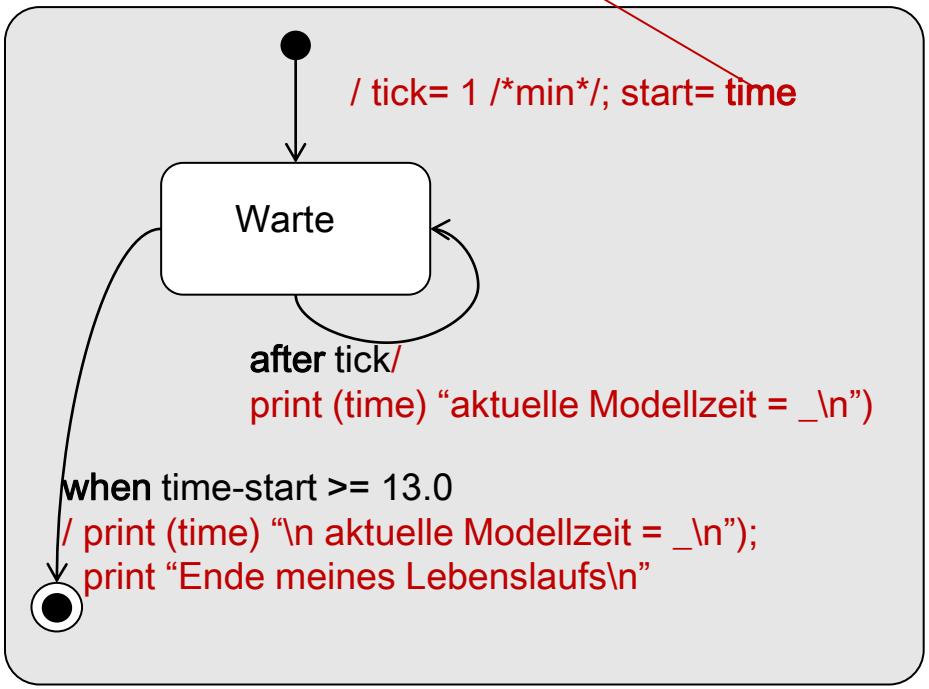
1. Charakterisierung der Next-Event-Simulation
   - Ereignisse, Next-Event-Scheduler
   - Barren-Beispiel
   - Zusammenhang von ereignisbasierter und prozessbasierter Modellbeschreibung

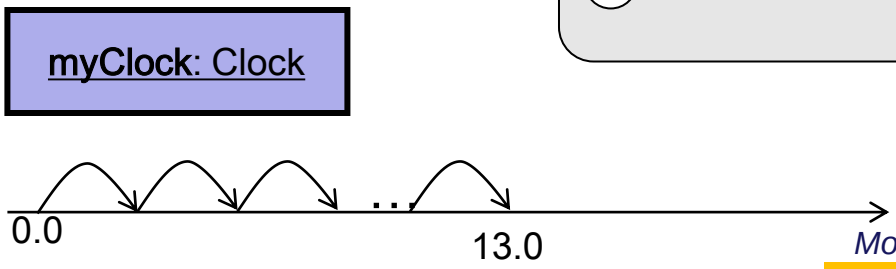
2. Umsetzung des Prinzips in ODEMx
   - Aufbau von ODEMx (erster Blick)
   - Triviales Clock-Beispiel

Nachbildung einer Uhr

aktuelle Modellzeit
zum Generierungs-/Startzeitpunkt
des jeweiligen Clock-Objektes



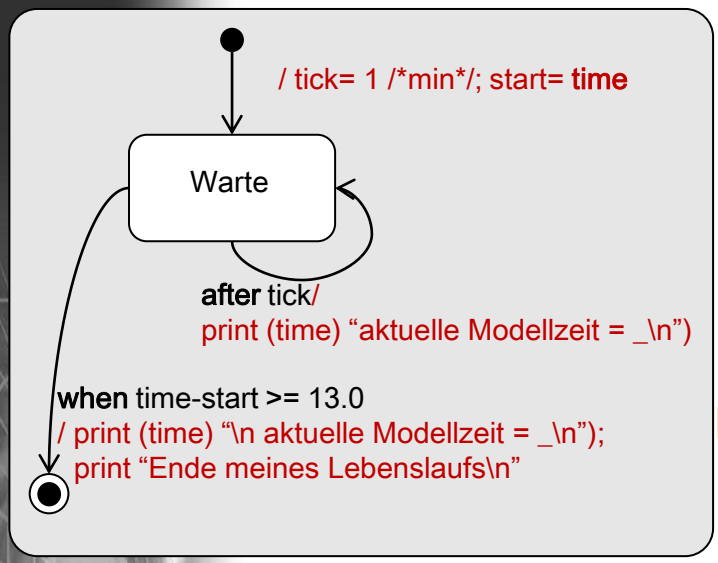


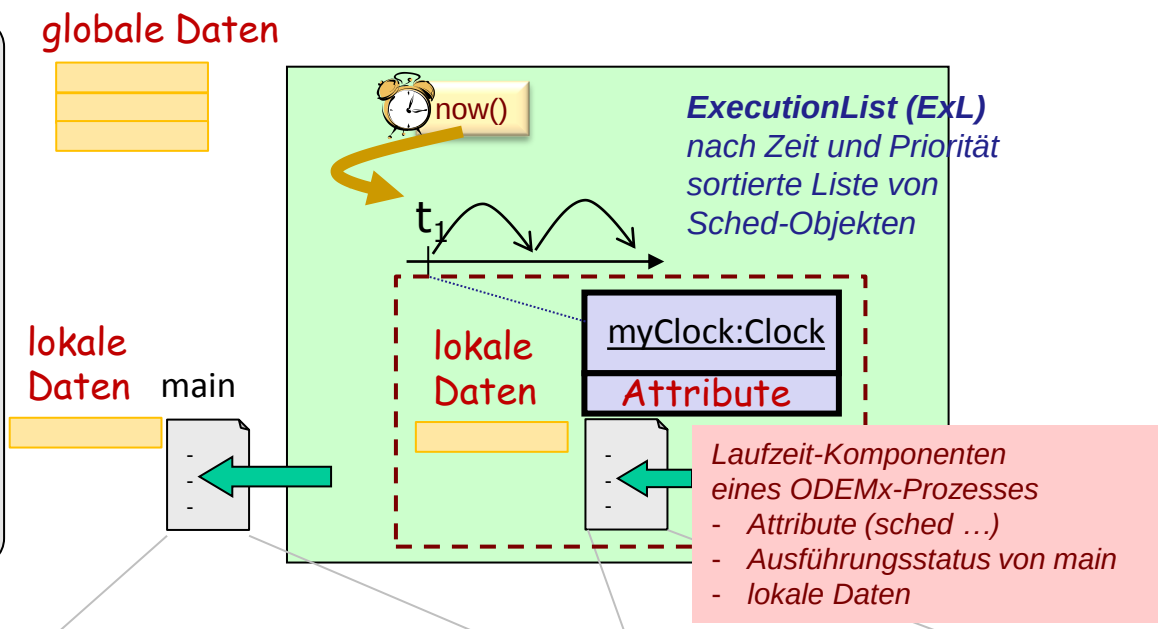


Ausgabe eines Punkt-Zeichens (als Tick)
zu ganzzahligen Modellzeitpunkten
als Prozess (der Ereignisfolge erzeugt)

System-Erweiterung:
zwei Uhren werden nicht synchron gestartet: Ausgabe einer überlagerten farblich markierten Punktfolge

Einfaches Beispiel





Laufzeit-Komponenten eines ODEMx-Prozesses
- *Attribute (sched …)*
- *Ausführungsstatus von main*
- *lokale Daten*

- Einrichten von myClock
- Laden der EL
- AUSGABE
- Aktivieren des Kontextes
- AUSGABE

forever
- Zeitverbrauch
- AUSGABE
  Punkt-Zeichen

Varianten
a) step()
b) runUntil(simTime t)
c) run()

Einfaches Beispiel: Quelltext

```
#include  <odemx.h>
#include <iostream.h>
using namespace std;
```

```
class Clock : public Process {
public:
      Clock (Simulation& sim) :
          Process(sim, "Clock") {}

      virtual int main() {
            while (true) {
                  holdFor(1.0);
                  cout << '.';
            }
            return 0;
      }
};
```

```
int main(int argc, char* argv[]) {
      Clock* myClock= new Clock (getDefaultSimulation());
      myClock->activate();

      cout << "Basic Simulation Example" << endl;
      cout << "========================" << endl;

      for (int i=1; i<5; ++i) {
          getDefaultSimulation().step();
          cout << endl << i << ". time step at =" <<
              getDefaultSimulation().getTime() << endl;
      }
      cout << endl;
      cout << "continue until SimTime 13.0 is reached or passed";
      getDefaultSimulation().runUntil(13.0);
      cout << endl  << "time=" << getDefaultSimulation().getTime() << endl;

      cout << "========================" << endl;
      return 0;
}
```

Einfaches Beispiel: Quelltext

```
class Clock : public Process {
public:
    Clock (Simulation& sim) :
        Process(sim, "Clock") {}

    virtual int main() {
        while (true) {
            holdFor(1.0);
            cout << '.';
        }
        return 0;
    }
};
```

```
int main(int argc, char* argv[]) {
    Clock* myClock= new Clock (getDefaultSimulation());
    myClock->activate();

    cout << "Basic Simulation Example" << endl;
    cout << "========================" << endl;

    for (int i=1; i<5; ++i) {
        getDefaultSimulation().step();
        cout << endl << i << ". time step at =" <<
            getDefaultSimulation().getTime() << endl;
    }
    cout << endl;
    ut << "continue until SimTime 13.0 is reached or passed";
    tDefaultSimulation().runUntil(13.0);
    ut << endl  << "time=" << getDefaultSimulation().getTime() << endl;

    cout << "========================" << endl;
    return 0;
}
```

beim 1.Aufruf von getDefaultSimulation() wird DefaultSimulation-Objekt als statisches Objekt bereitgeste

Eintrag in EL des Kontextes

Kontext-Aktivierung

Modelluhrzeitabfrage

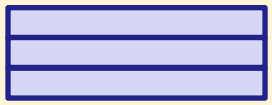
Scheduling-Operation
Wirkt im zugewiesenen Kontext des Prozesses

Kontext-Aktivierung

Modelluhrzeitabfrage

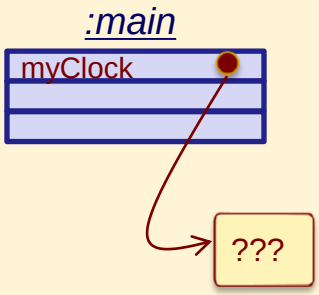
Beendigung der Programms (unabhängig von Koroutinenausführung)

Übergabe der Steuerung durch das Betriebssystem



globale Null-initialisierte Daten,Objekte (inkl. statische Objekte)

Halde

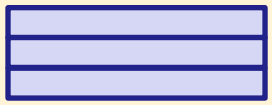


Stack

```
int main(int argc, char* argv[]) {
        Clock* myClock= new Clock (getDefaultSimulation());
        myClock->activate();

        cout << "Basic Simulation Example" << endl;
        cout << "========================" << endl;

        for (int i=1; i<5; ++i) {
            getDefaultSimulation().step();
            cout << endl << i << ". time step at =" <<
                 getDefaultSimulation().getTime() << endl;
        }
        cout << endl;
        cout << "continue until SimTime 13.0 is reached or passed";
        getDefaultSimulation().runUntil(13.0);
        cout << endl  << "time=" << getDefaultSimulation().getTime() << endl;

        cout << "========================" << endl;
        return 0;
}
```

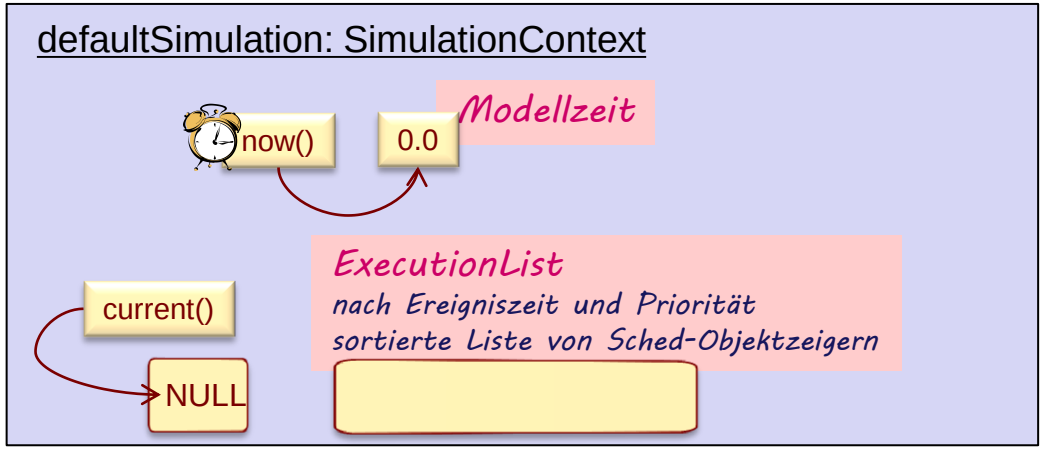
Ausführung von Konstruktoren globaler Objekte des Anwenderprogramms

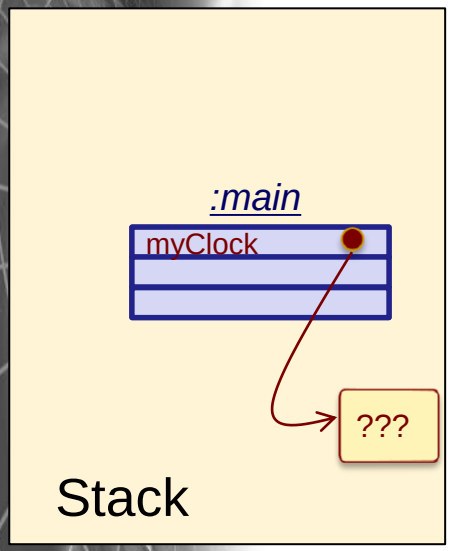
Übergabe der Steuerung durch das Betriebssystem



globale initialisierte Daten, Objekte (inkl. Statische Objekte)

Halde



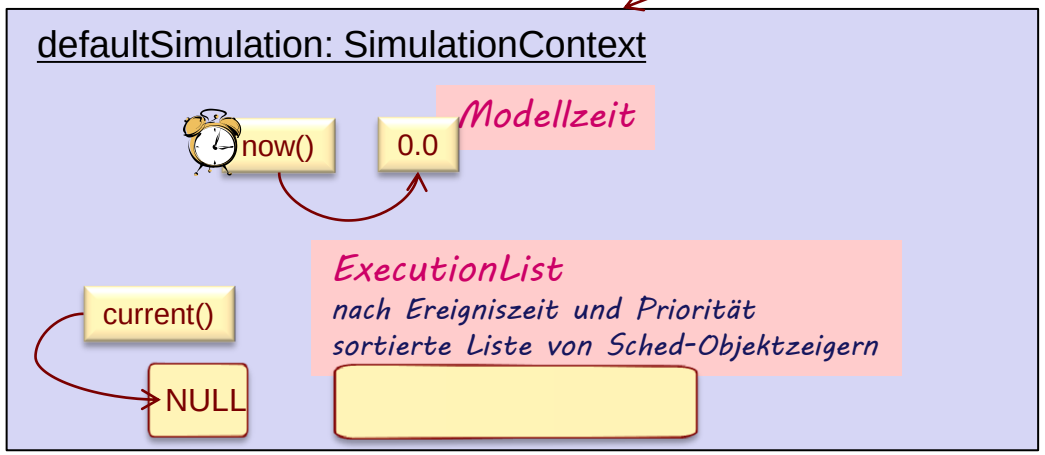



```
int main(int argc, char* argv[]) {
    Clock* myClock= new Clock (getDefaultSimulation());
    myClock->activate();

    cout << "Basic Simulation Example" << endl;
    cout << "========================" << endl;

    for (int i=1; i<5; ++i) {
        getDefaultSimulation().step();
        cout << endl << i << ". time step at =" <<
            getDefaultSimulation().getTime() << endl;
    }
    cout << endl;
    cout << "continue until SimTime 13.0 is reached or passed";
    getDefaultSimulation().runUntil(13.0);
    cout << endl  << "time=" << getDefaultSimulation().getTime() << endl;

    cout << "========================" << endl;
    return 0;
}
```

erster Aufruf ruft den Konstruktor von DefaultSimulation

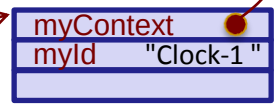
main: Objekt-Generierung

globale initialisierte Daten



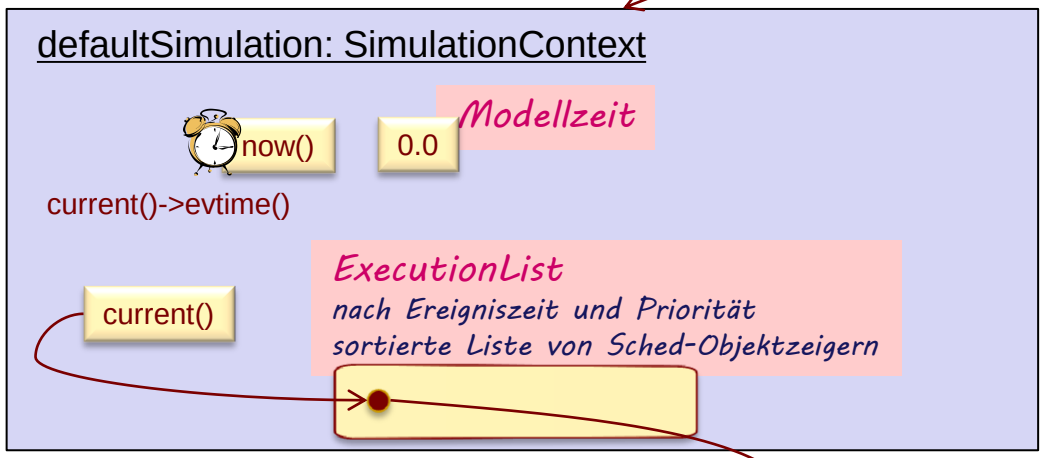

```
int main(int argc, char* argv[]) {
        Clock* myClock= new Clock (getDefaultSimulation());
        myClock->activate();

        cout << "Basic Simulation Ex
        cout << "==============

        for (int i=1; i<5; ++i) {
            getDefaultSimulation().st
            cout << endl << i << ". ste
                 getDefaultSimulation
        }
        cout << endl;
        cout << "continue until SimT
        getDefaultSimulation().runU
        cout << endl  << "time=" << getDefaultSimulation()-.getTime() << endl;

        cout << "======================" << endl;
        return 0;
}
```

```
class Clock : public Process {
public:
    Clock (Simulation& sim) :
        Process(sim, "Clock") {}

    virtual int main() {
            while (true) {
                holdFor(1.0);
                cout << '.';
            }
            return 0;
    }
};
```



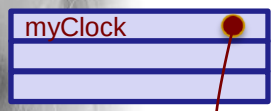
main: Befüllen des Terminkalenders

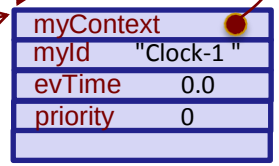
globale initialisierte Daten




```
int main(int argc, char* argv[]) {
    Clock* myClock= new Clock (getDefaultSimulation());
    myClock->activate();

    cout << "Basic Simulation Example" << endl;
    cout << "========================" << endl;

    for (int i=1; i<5; ++i)
        getDefaultSimulat
        cout << endl << i <
            getDefaultSim
    }
    cout << endl;
    cout << "continue un
    getDefaultSimulation
    cout << endl  << "tim

    cout << "==========
    return 0;
}
```

```
class Clock : public Process {
public:
    Clock (Simulation& sim) :
        Process(sim, "Clock") {}

    virtual int main() {
        while (true) {
            holdFor(1.0);
            cout << '.';
        }
        return 0;
    }
};
```



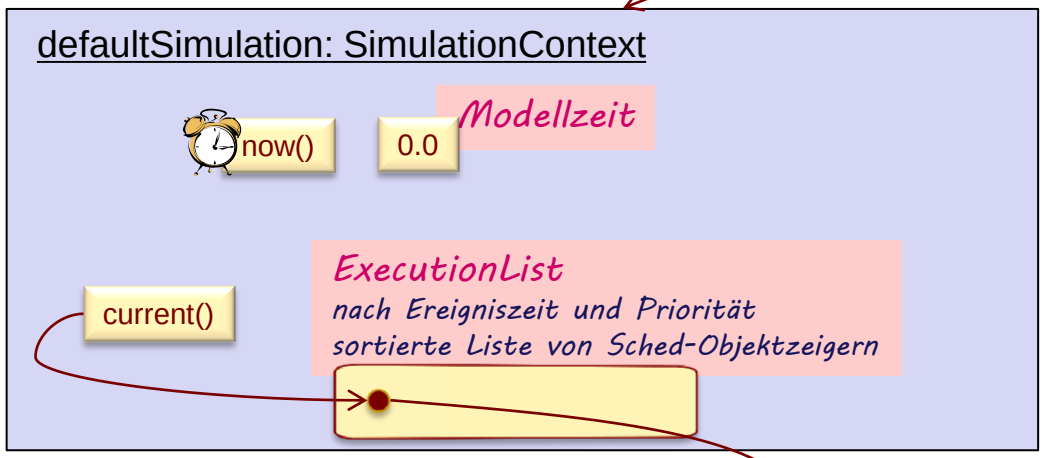


Ausgabe

```
Basic Simulation Example
```

main: Ausgabe

globale initialisierte Daten



:main

myClock

```
int main(int argc, char* argv[]) {
        Clock* myClock= new Clock (getDefaultSimulation());
        myClock->activate();

        cout << "Basic Simulation Example" << endl;
        cout << "========================" << endl;

        for (int i=1; i<5; ++i)
            getDefaultSimulat
            cout << endl << i <
                getDefaultSim
        }
        cout << endl;
        cout << "continue un
        getDefaultSimulation
        cout << endl  << "tim

        cout << "==========
        return 0;
}
```

```
class Clock : public Process {
public:
        Clock (Simulation& sim) :
            Process(sim, "Clock") {}

        virtual int main() {
                while (true) {
                        holdFor(1.0);
                        cout << '.';
                }
                return 0;
        }
};
```

:Clock

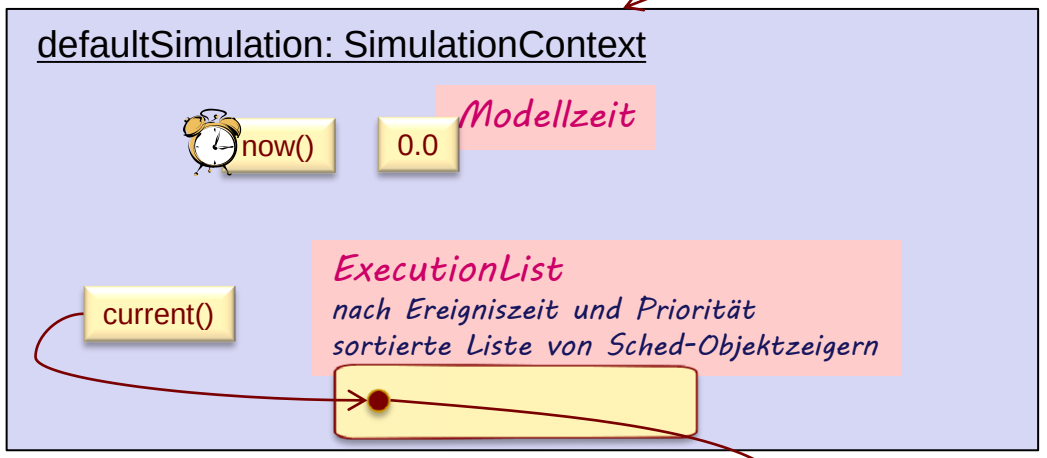
myContext	
myId	"Clock-1 "
evTime	0.0
priority	0

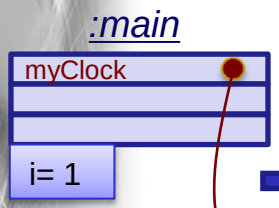
Ausgabe

```
Basic Simulation Example
========================
```

main: Steuerungsübergabe (Schrittmodus)

globale initialisierte Daten



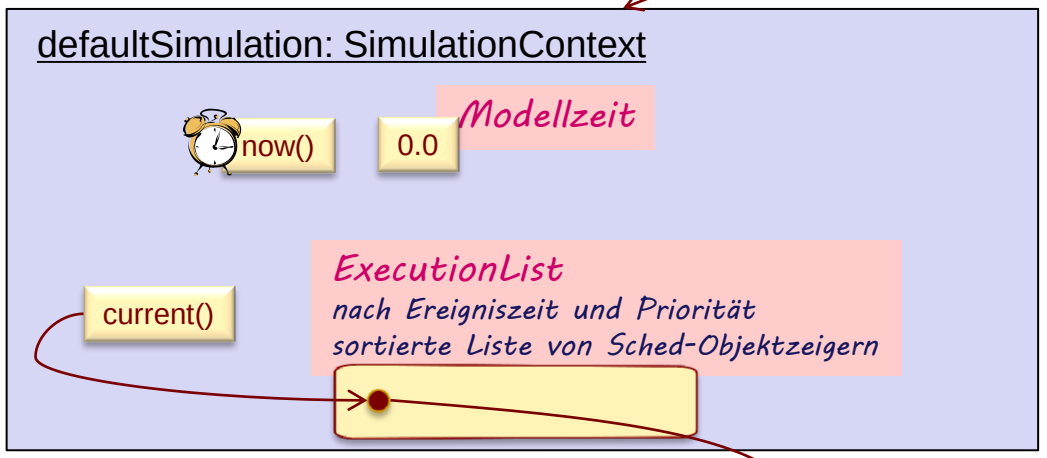


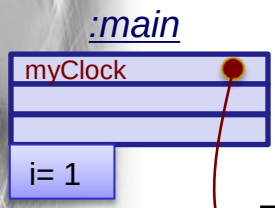

```
int main(int argc, char* argv[]) {
        Clock* myClock= new Clock (getDefaultSimulation());
        myClock->activate();

        cout << "Basic Simulation Example" << endl;
        cout << "=======================" << endl;

        for (int i=1; i<5; ++i) {
            getDefaultSimulation().step();
            cout << endl << i << ". time step at =" <<
                getDefaultSimulation().getTime() << endl;
        }
        cout << endl;
        cout << "continue until SimTime 13.0 is reached or pass
        getDefaultSimulation().runUntil(13.0);
        cout << endl  << "time=" << getDefaultSimulation().getT

        cout << "=======================" << endl;
        return 0;
}
```


```
class Clock : public Process {
public:
        Clock (Simulation& sim) :
            Process(sim, "Clock") {}

        virtual int main() {
                while (true) {
                        holdFor(1.0);
                        cout << '.';
                }
                return 0;
        }
};
```

erfolgter Koroutinen-Wechsel

globale initialisierte Daten



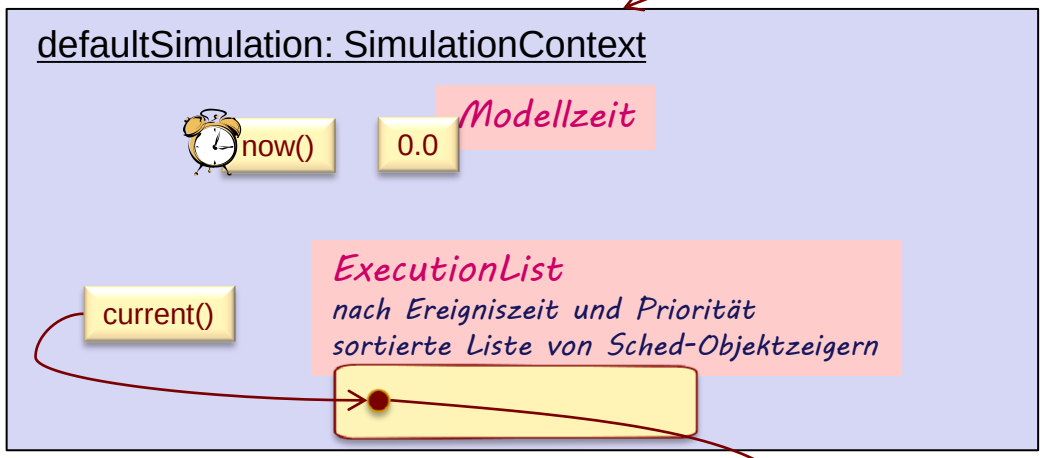



```
int main(int argc, char* argv[]) {
        Clock* myClock= new Clock (getDefaultSimulation());
        myClock->activate();

        cout << "Basic Simulation Example" << endl;
        cout << "========================" << endl;

        for (int i=1; i<5; ++i) {
            getDefaultSimulation().step();
            cout << endl << i << ". time step at =" <<
                getDefaultSimulation().getTime() << endl;
        }
        cout << endl;
        cout << "continue until SimTime 13.0 is reached or pass
        getDefaultSimulation().runUntil(13.0);
        cout << endl  << "time=" << getDefaultSimulation().getT

        cout << "========================" << endl;
        return 0;
}
```

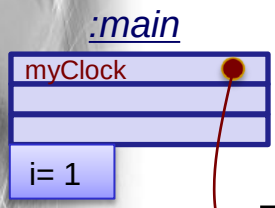

```
class Clock : public Process {
public:
        Clock (Simulation& sim) :
            Process(sim, "Clock") {}

        virtual int main() {
                while (true) {
                        holdFor(1.0);
                        cout << '.';
                }
                return 0;
        }
};
```

Clock-1: „normale“ Anweisung

globale initialisierte Daten



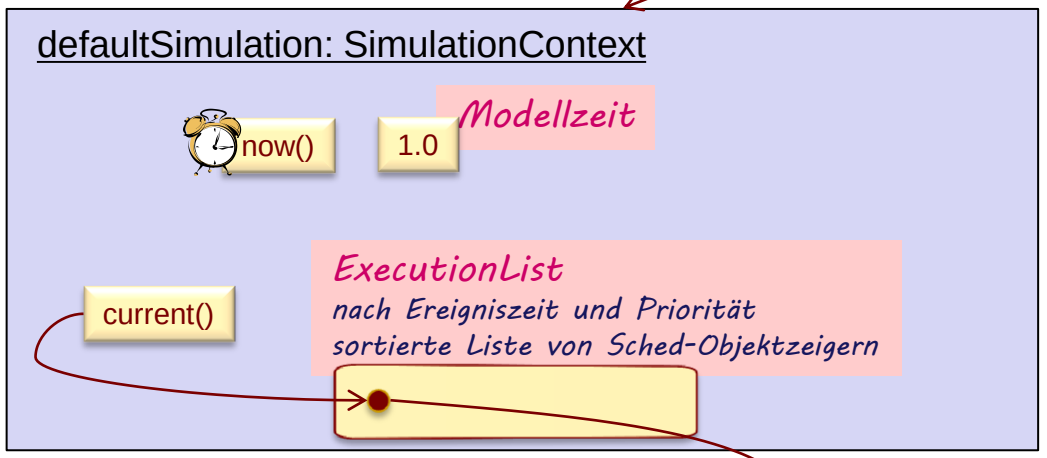

```
int main(int argc, char* argv[]) {
        Clock* myClock= new Clock (getDefaultSimulation());
        myClock->activate();

        cout << "Basic Simulation Example" << endl;
        cout << "========================" << endl;

        for (int i=1; i<5; ++i) {
            getDefaultSimulation().step();
            cout << endl << i << ". time step at =" <<
                getDefaultSimulation().getTime() << endl;
        }
        cout << endl;
        cout << "continue until SimTime 13.0 is reached or pass
        getDefaultSimulation().runUntil(13.0);
        cout << endl  << "time=" << getDefaultSimulation().getT

        cout << "========================" << endl;
        return 0;
}
```



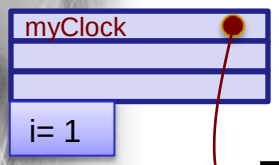

```
class Clock : public Process {
public:
        Clock (Simulation& sim) :
            Process(sim, "Clock") {}

        virtual int main() {
            while (true) {
                holdFor(1.0);
                cout << '.';
            }
            return 0;
        }
};
```

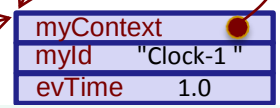
Clock-1: Scheduling-Anweisung (Verzögerung um 1 ZE)

globale initialisierte Daten




```
int main(int argc, char* argv[]) {
        Clock* myClock= new Clock (getDefaultSimulation());
        myClock->activate();

        cout << "Basic Simulation Example" << endl;
        cout << "========================" << endl;

        for (int i=1; i<5; ++i) {
            getDefaultSimulation().step();
            cout << endl << i << ". step time=" <<
                getDefaultSimulation().getTime() << endl;
        }
        cout << endl;
        cout << "continue until SimTime 13.0 is reached or pass
        getDefaultSimulation().runUntil(13.0);
        cout << endl  << "time=" << getDefaultSimulation().getT

        cout << "========================" << endl;
        return 0;
}
```

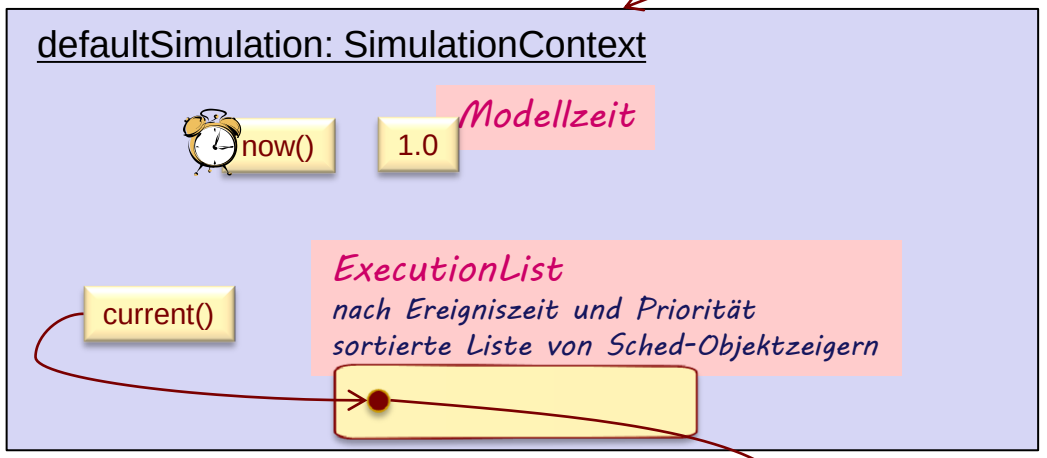




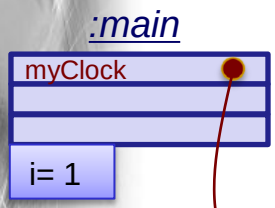

```
class Clock : public Process {
public:
        Clock (Simulation& sim) :
            Process(sim, "Clock") {}

        virtual int main() {
                while (true) {
                        holdFor(1.0);
                        cout << '.';
                }
                return 0;
        }
};
```

Clock-1: *impliziter Rücksprung ins Hauptprogramm*

globale initialisierte Daten




```
int main(int argc, char* argv[]) {
        Clock* myClock= new Clock (getDefaultSimulation());
        myClock->activate();

        cout << "Basic Simulation Example" << endl;
        cout << "======================" << endl;

        for (int i=1; i<5; ++i) {
            getDefaultSimulation().step();
            cout << endl << i << ". time step at =" <<
                 getDefaultSimulation().getTime() << endl;
        }
        cout << endl;
        cout << "continue until SimTime 13.0 is reached or pass
        getDefaultSimulation().runUntil(13.0);
        cout << endl  << "time=" << getDefaultSimulation().getT

        cout << "======================" << endl;
        return 0;
}
```

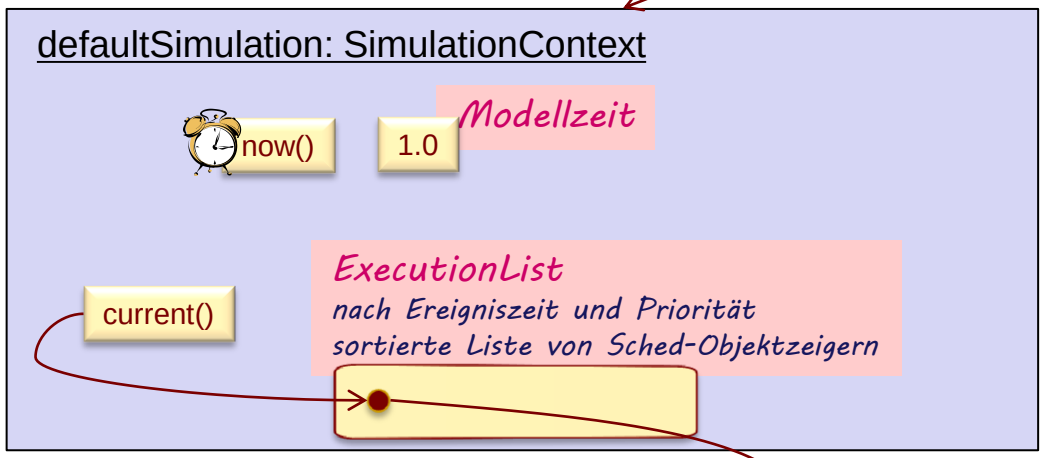



```
class Clock : public Process {
public:
        Clock (Simulation& sim) :
            Process(sim, "Clock") {}

        virtual int main() {
                while (true) {
                        holdFor(1.0);
                        cout << '.';
                }
                return 0;
        }
};
```

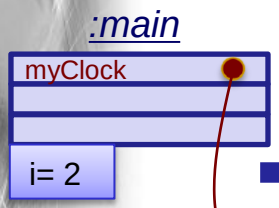
Ausgabe

```
Basic Simulation Example
========================
1. time step at= 1.0
```

main: Steuerungsübergabe (Schrittmodus)



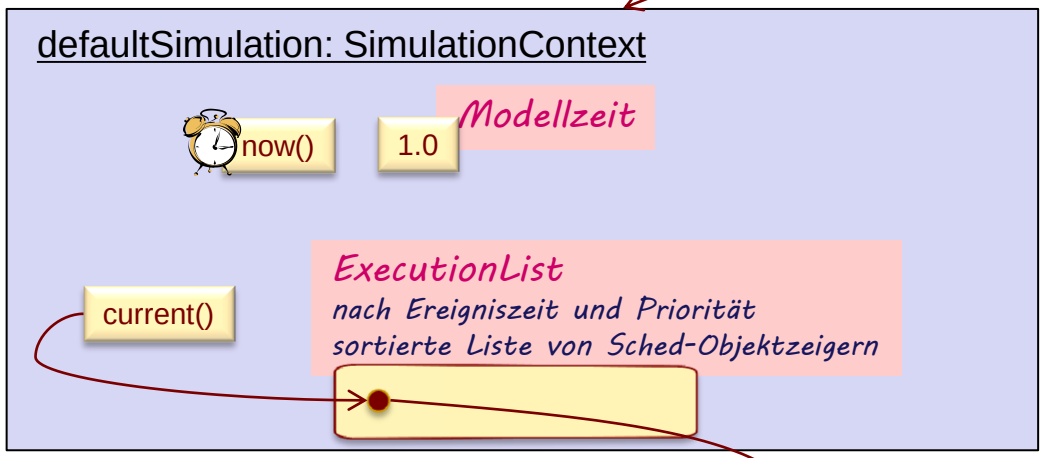

```
int main(int argc, char* argv[]) {
        Clock* myClock= new Clock (getDefaultSimulation());
        myClock->activate();

        cout << "Basic Simulation Example" << endl;
        cout << "======================" << endl;

        for (int i=1; i<5; ++i) {
            getDefaultSimulation().step();
            cout << endl << i << ". time step at =" <<
                getDefaultSimulation().getTime() << endl;
        }
        cout << endl;
        cout << "continue until SimTime 13.0 is reached or pass
        getDefaultSimulation().runUntil(13.0);
        cout << endl  << "time=" << getDefaultSimulation().getT

        cout << "======================" << endl;
        return 0;
}
```



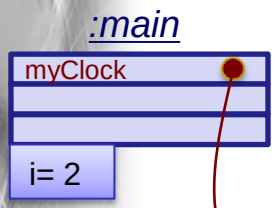

```
class Clock : public Process {
public:
        Clock (Simulation& sim) :
            Process(sim, "Clock") {}

        virtual int main() {
                while (true) {
                        holdFor(1.0);
                        cout << '.';
                }
                return 0;
        }
};
```

Clock-1: 1.Punkt

globale initialisierte Daten




```
int main(int argc, char* argv[]) {
        Clock* myClock= new Clock (getDefaultSimulation());
        myClock->activate();

        cout << "Basic Simulation Example" << endl;
        cout << "======================" << endl;

        for (int i=1; i<5; ++i) {
            getDefaultSimulation().step();
            cout << endl << i << ". time step at =" <<
                 getDefaultSimulation().getTime() << endl;
        }
        cout << endl;
        cout << "continue until SimTime 13.0 is reached or pass
        getDefaultSimulation()-.runUntil(13.0);
        cout << endl  << "time=" << getDefaultSimulation().getT

        cout << "======================" << endl;
        return 0;
}
```



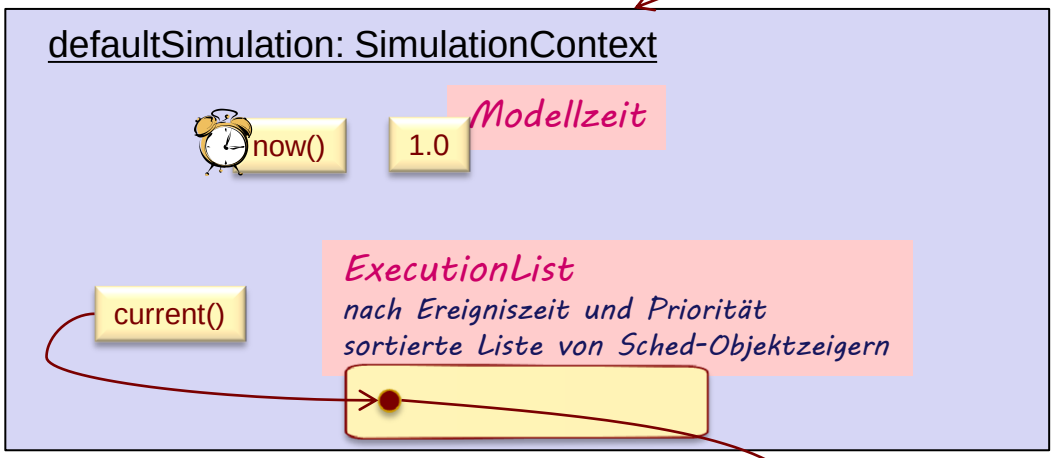

```
class Clock : public Process {
public:
        Clock (Simulation& sim) :
            Process(sim, "Clock") {}

        virtual int main() {
                while (true) {
                        holdFor(1.0);
                        cout << '.';
                }
                return 0;
        }
};
```

Ausgabe

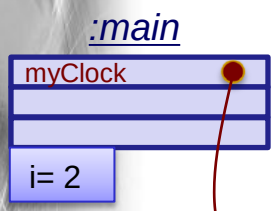
```
Basic Simulation Example
========================
1.  time step at= 1.0
.
```

Clock-1: Verzögerung um 1 weitere ZE

globale initialisierte Daten



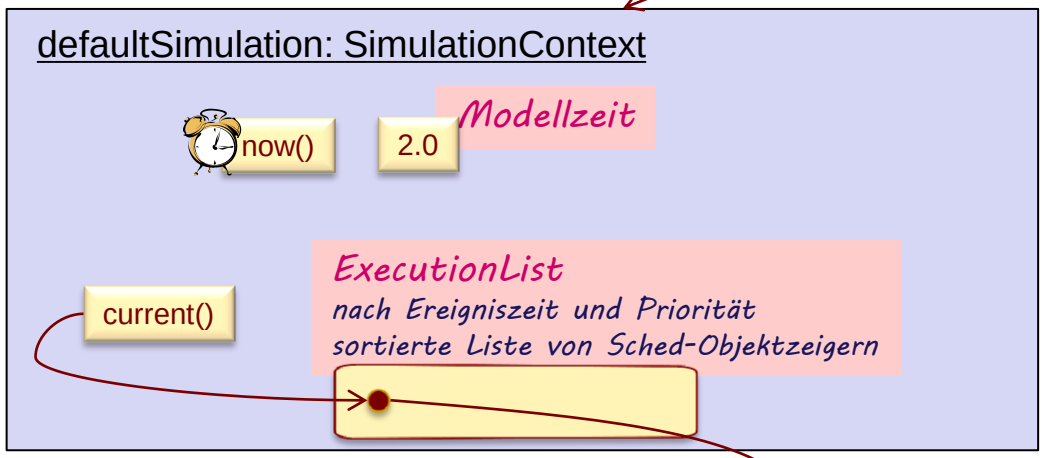

```
int main(int argc, char* argv[]) {
        Clock* myClock= new Clock (getDefaultSimulation());
        myClock->activate();

        cout << "Basic Simulation Example" << endl;
        cout << "======================" << endl;

        for (int i=1; i<5; ++i) {
            getDefaultSimulation().step();
            cout << endl << i << ". time step at =" <<
                 getDefaultSimulation().getTime() << endl;
        }
        cout << endl;
        cout << "continue until SimTime 13.0 is reached or pass
        getDefaultSimulation().runUntil(13.0);
        cout << endl  << "time=" << getDefaultSimulation().getT

        cout << "======================" << endl;
        return 0;
}
```



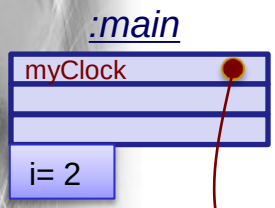

```
class Clock : public Process {
public:
        Clock (Simulation& sim) :
            Process(sim, "Clock") {}

        virtual int main() {
                while (true) {
                        holdFor(1.0);
                        cout << '.';
                }
                return 0;
        }
};
```

Clock-1: Rücksprung ins Hauptprogramm

globale initialisierte Daten




```
int main(int argc, char* argv[]) {
        Clock* myClock= new Clock (getDefaultSimulation());
        myClock->activate();

        cout << "Basic Simulation Example" << endl;
        cout << "======================" << endl;

        for (int i=1; i<5; ++i) {
            getDefaultSimulation().step();
            cout << endl << i << ". time step at =" <<
                 getDefaultSimulation().getTime() << endl;
        }
        cout << endl;
        cout << "continue until SimTime 13.0 is reached or pass
        getDefaultSimulation().runUntil(13.0);
        cout << endl  << "time=" << getDefaultSimulation().getT

        cout << "========================" << endl;
        return 0;
}
```




```
class Clock : public Process {
public:
        Clock (Simulation& sim) :
            Process(sim, "Clock") {}

        virtual int main() {
                while (true) {
                        holdFor(1.0);
                        cout << '.';
                }
                return 0;
        }
};
```

Ausgabe

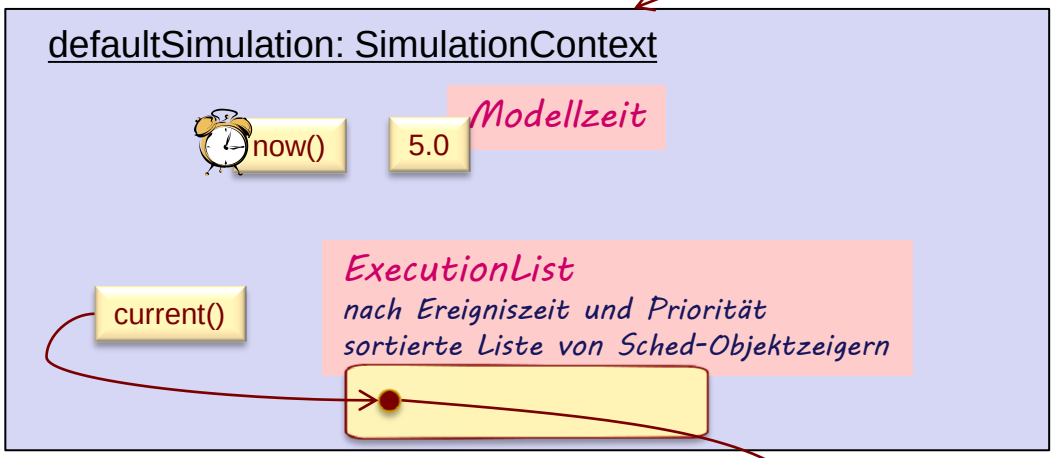
```
Basic Simulation Example
========================
1.  time step at= 1.0
.
2. time step at= 2.0
```

Ausgabe bis zur Beendigung der For-Anw.

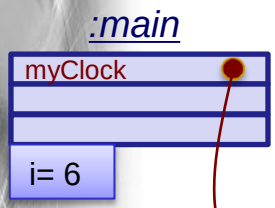
```
Basic Simulation Example
========================
1.  time step at= 1.0
.
2. time step at= 2.0
.
3. time step at= 3.0
.
4. time step at= 4.0
.
5. time step at= 5.0
```

Clock-1: Punktausgabe

globale initialisierte Daten




```
int main(int argc, char* argv[]) {
        Clock* myClock= new Clock (getDefaultSimulation());
        myClock->activate();

        cout << "Basic Simulation Example" << endl;
        cout << "======================" << endl;

        for (int i=1; i<5; ++i) {
            getDefaultSimulation().step();
            cout << endl << i << ". time step at =" <<
                getDefaultSimulation().getTime() << endl;
        }
        cout << endl;
        cout << "continue until SimTime 13.0 is reached or pass
        getDefaultSimulation().runUntil(13.0);
        cout << endl  << "time=" << getDefaultSimulation().getT

        cout << "======================" << endl;
        return 0;
}
```



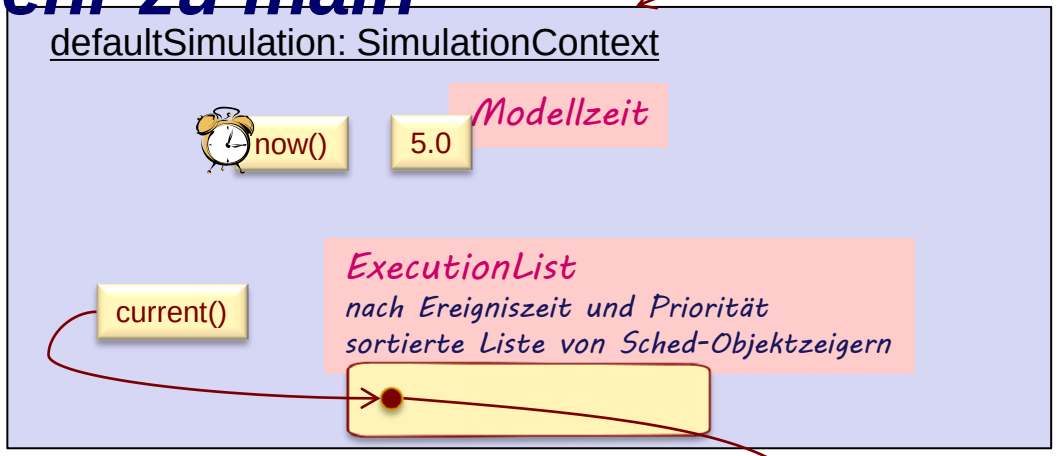

```
class Clock : public Process {
public:
        Clock (Simulation& sim) :
            Process(sim, "Clock") {}

        virtual int main() {
                while (true) {
                        holdFor(1.0);
                        cout << '.';
                }
                return 0;
        }
};
```

Ausgabe bis zur Beendigung der For-Anw.

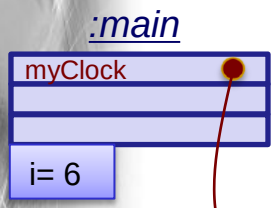
```
Basic Simulation Example
========================
1.  time step at= 1.0
.
2. time step at= 2.0
.
3. time step at= 3.0
.
4. time step at= 4.0
.
5. time step at= 5.0
.
```

Clock-1: Verzögerung um 1 weitere ZE/ Rückkehr zu main

globale initialisierte Daten




```
int main(int argc, char* argv[]) {
        Clock* myClock= new Clock (getDefaultSimulation());
        myClock->activate();

        cout << "Basic Simulation Example" << endl;
        cout << "======================" << endl;

        for (int i=1; i<5; ++i) {
            getDefaultSimulation().step();
            cout << endl << i << ". time step at =" <<
                getDefaultSimulation().getTime() << endl;
        }
        cout << endl;
        cout << "continue until SimTime 13.0 is reached or pass
        getDefaultSimulation().runUntil(13.0);
        cout << endl  << "time=" << getDefaultSimulation().getT

        cout << "======================" << endl;
        return 0;
}
```



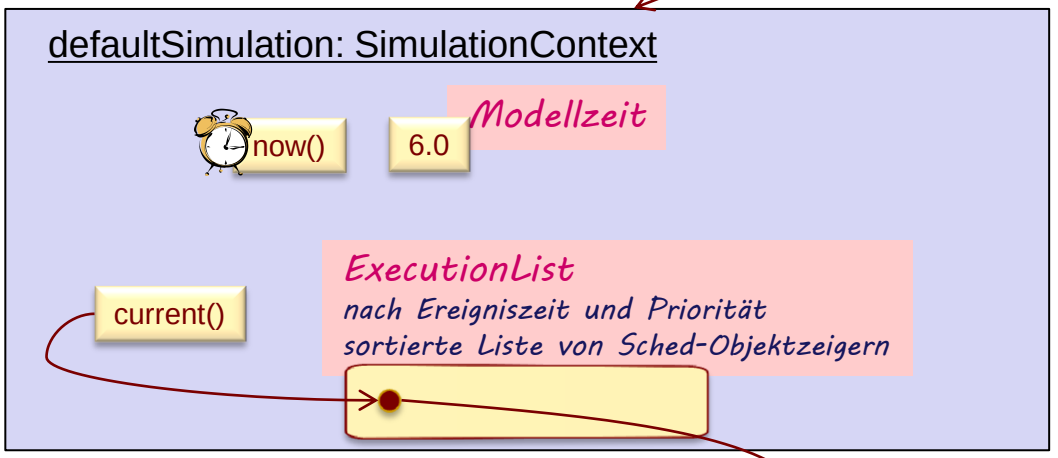

```
class Clock : public Process {
public:
        Clock (Simulation& sim) :
            Process(sim, "Clock") {}

        virtual int main() {
                while (true) {
                        holdFor(1.0);
                        cout << '.';
                }
                return 0;
        }
};
```

Ausgabe nach Beendigung der For-Anw.

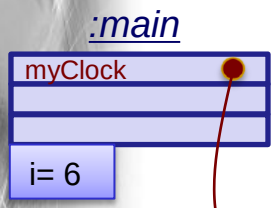
```
Basic Simulation Example
========================
1.   time step at= 1.0
.
2. time step at= 2.0
.
3. time step at= 3.0
.
4. time step at= 4.0
.
5. time step at= 5.0
.
continue until SimTime 13.0 is reached or passed
```

main: Steuerungsübergabe (Intervallmodus)

globale initialisierte Daten



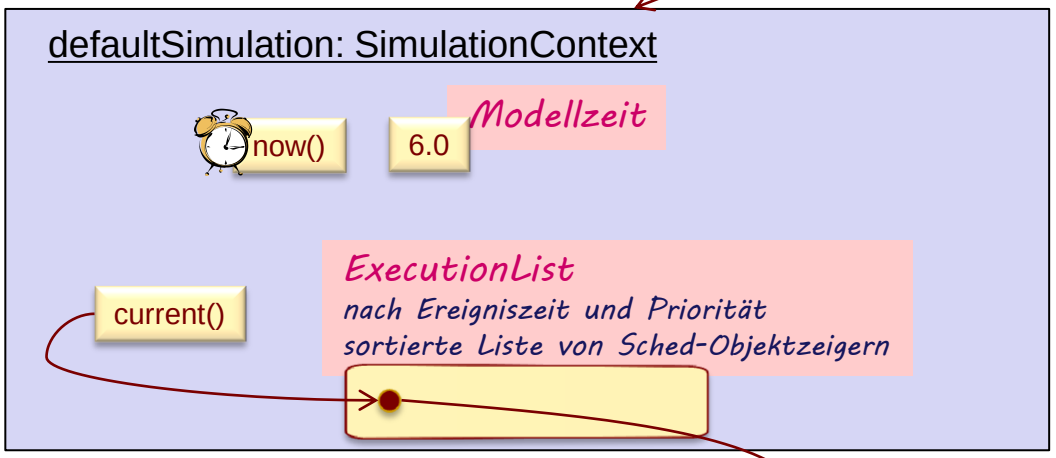

```
int main(int argc, char* argv[]) {
        Clock* myClock= new Clock (getDefaultSimulation());
        myClock->activate();

        cout << "Basic Simulation Example" << endl;
        cout << "======================" << endl;

        for (int i=1; i<5; ++i) {
            getDefaultSimulation().step();
            cout << endl << i << ". time step at =" <<
                 getDefaultSimulation().getTime() << endl;
        }
        cout << endl;
        cout << "continue until SimTime 13.0 is reached or pass
        getDefaultSimulation().runUntil(13.0);
        cout << endl  << "time=" << getDefaultSimulation().getT

        cout << "========================" << endl;
        return 0;
}
```



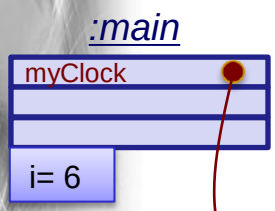

```
class Clock : public Process {
public:
        Clock (Simulation& sim) :
            Process(sim, "Clock") {}

        virtual int main() {
                while (true) {
                        holdFor(1.0);
                        cout << '.';
                }
                return 0;
        }
};
```

main: Steuerungsübergabe (Intervallmodus)

globale initialisierte Daten




```
int main(int argc, char* argv[]) {
        Clock* myClock= new Clock (getDefaultSimulation());
        myClock->activate();

        cout << "Basic Simulation Example" << endl;
        cout << "======================" << endl;

        for (int i=1; i<5; ++i) {
            getDefaultSimulation().step();
            cout << endl << i << ". time step at =" <<
                 getDefaultSimulation().getTime() << endl;
        }
        cout << endl;
        cout << "continue until SimTime 13.0 is reached or pass
        getDefaultSimulation()-.runUntil(13.0);
        cout << endl  << "time=" << getDefaultSimulation()->get

        cout << "======================" << endl;
        return 0;
}
```




```
class Clock : public Process {
public:
        Clock (Simulation& sim) :
            Process(sim, "Clock") {}

        virtual int main() {
                while (true) {
                        holdFor(1.0);
                        cout << '.';
                }
                return 0;
        }
};
```

Kein Rücksprung

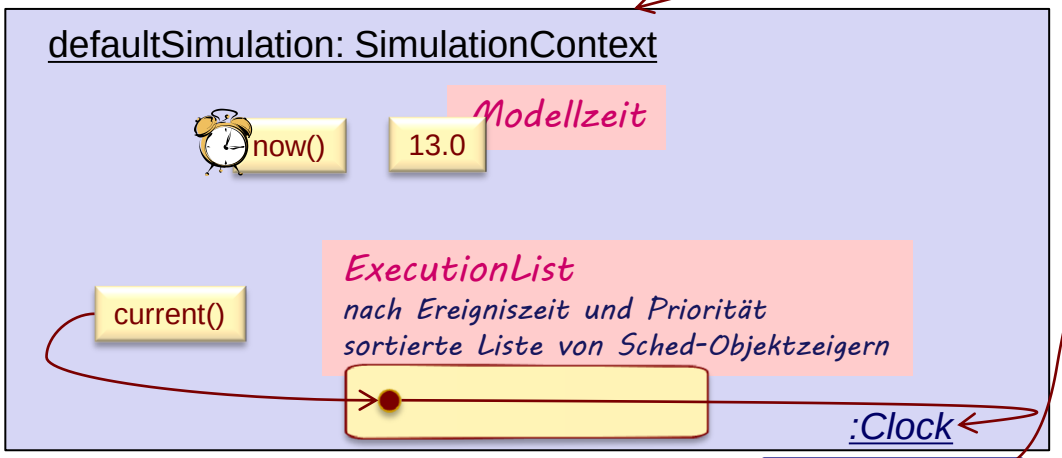
Ausgabe bis zum Erreichen von 13.0

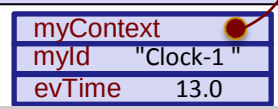
```
Basic Simulation Example
========================
1. time step at= 1.0
.
2. time step at= 2.0
.
3. time step at= 3.0
.
4. time step at= 4.0
.
5. time step at= 5.0
.
continue until SimTime 13.0 is reached or passed
........
```

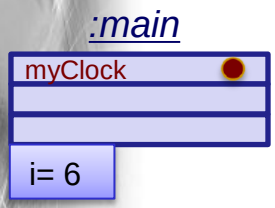
insgesamt 13 Punkte

main: Steuerungsübergabe (Intervallmodus)

globale initialisierte Daten








```
int main(int argc, char* argv[]) {
        Clock* myClock= new Clock (getDefaultSimulation());
        myClock->activate();

        cout << "Basic Simulation Example" << endl;
        cout << "======================" << endl;

        for (int i=1; i<5; ++i) {
            getDefaultSimulation().step();
            cout << endl << i << ". time step at =" <<
                 getDefaultSimulation().getTime() << endl;
        }
        cout << endl;
        cout << "continue until SimTime 13.0 is reached or pass
        getDefaultSimulation().runUntil(13.0);
        cout << endl  << "time=" << getDefaultSimulation().getTime() << endl;

        cout << "========================" << endl;
        return 0;
}
```

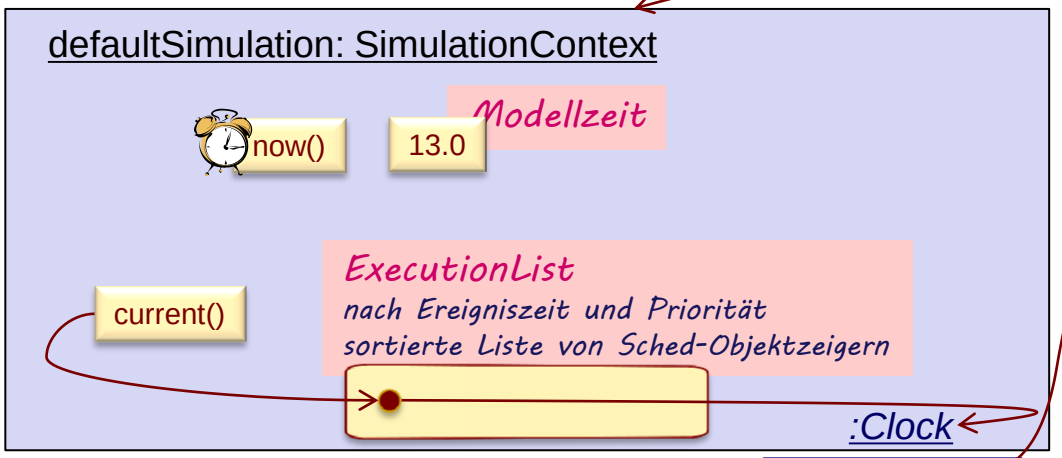
```
class Clock : public Process {
public:
        Clock (Simulation& sim) :
            Process(sim, "Clock") {}

        virtual int main() {
                while (true) {
                        holdFor(1.0);
                        cout << '.';
                }
                return 0;
        }
};
```

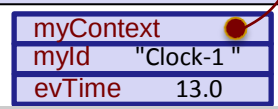
Ausgabe bis zum Erreichen von 13.0

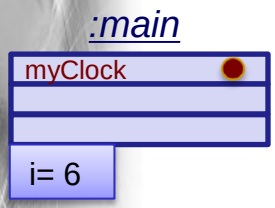
```
Basic Simulation Example
========================
1. time step at= 1.0
.
2. time step at= 2.0
.
3. time step at= 3.0
.
4. time step at= 4.0
.
5. time step at= 5.0
.
continue until SimTime 13.0 is reached or passed
........
time= 13.0
========================
```

main: Beendigung

globale initialisierte Daten








```
int main(int argc, char* argv[]) {
        Clock* myClock= new Clock (getDefaultSimulation());
        myClock->activate();

        cout << "Basic Simulation Example" << endl;
        cout << "======================" << endl;

        for (int i=1; i<5; ++i) {
            getDefaultSimulation().step();
            cout << endl << i << ". time step at =" <<
                 getDefaultSimulation().getTime() << endl;
        }
        cout << endl;
        cout << "continue until SimTime 13.0 is reached or pass
        getDefaultSimulation().runUntil(13.0);
        cout << endl  << "time=" << getDefaultSimulation().getTime() << endl;

        cout << "========================" << endl;
        return 0;
}
```

```
class Clock : public Process {
public:
        Clock (Simulation& sim) :
            Process(sim, "Clock") {}

        virtual int main() {
                while (true) {
                        holdFor(1.0);
                        cout << '.';
                }
                return 0;
        }
};
```

3. Prozessverwaltung

1. Aufgaben von Klasse Simulation
2. Process-Listen eines Simulationskontextes
3. Allgemeines Process-Scheduling
4. Weitere Process-Funktionaliät
5. Prozesswarteschlangen: ProcessQueue, Port
6. Spezielles Process-Scheduling (Memory)

Varianten der Kontextaktivierung

Methoden der Klasse Simulation (Simulationskontext)
(aufgerufen vom C++ Hauptprogramm)

bisher besprochen

1. Einzelschrittausführung: step()

2. Lauf bis zum Erreichen/Überschreiten einer vorgegebenen Modellzeit (SimTime): runUntil(…)

3. Lauf bis zum Ende der Simulation: run()

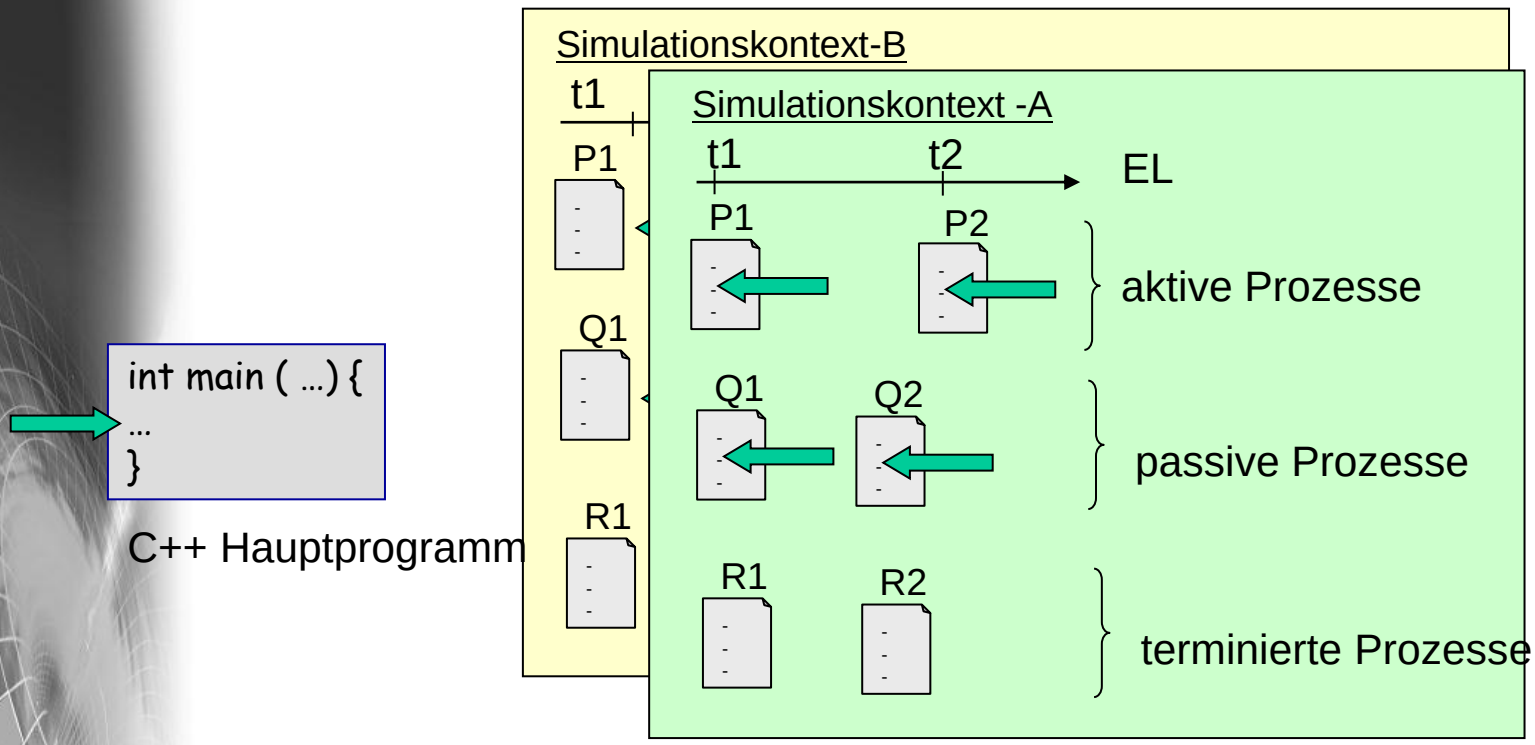
   Rückkehr ins C++ Hauptprogramm:

   – ***implizit***:
     es gibt keinen **aktiven** Prozess mehr im zugehörigen Simulationskontext (Kalender ist leer)

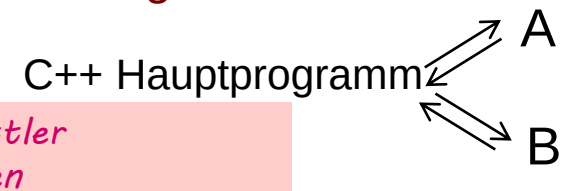
   – ***explizit***: die Simulation wurde mit exitSimulation() durch einen Prozesses des Simulationskontextes beendet

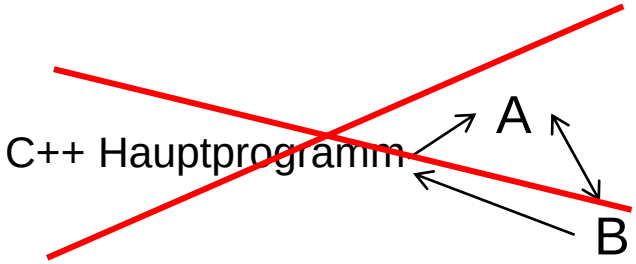
typisch für Arbeit DefaultSimulation-Kontext

Verwaltung mehrerer Simulationskontexte



Steuerungszenarien:





als Mittler zwischen den Prozess-Systemen

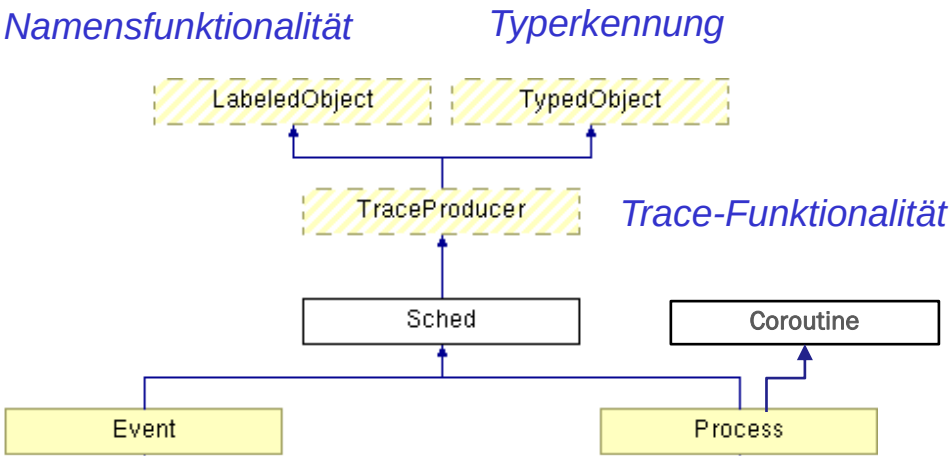
Klasse Sched, Event, Process

Sched

- abstrakte Klasse
- Objekte werden im Kalender in chronologischer Reihenfolge erfasst

Simulationslauf

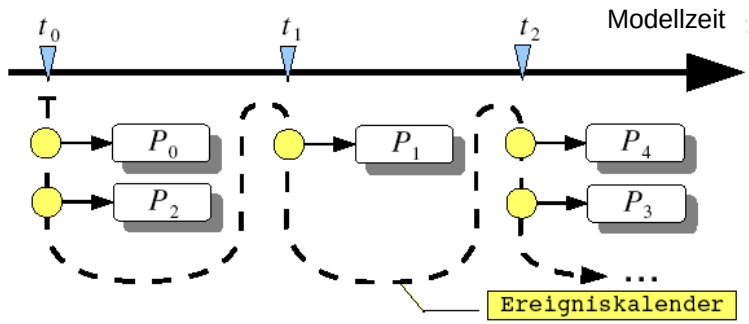
- ist die Ausführung (execute) von Sched-Objekten
- in Abhängigkeit von
  - der jeweiligen Kalenderkonstellation und
  - der Typen der Sched-Objekte



dabei können neben Zustandsänderungen auch
- Eintragungen,
- Verschiebungen und Streichungen von Sched-Objekten vorgenommen

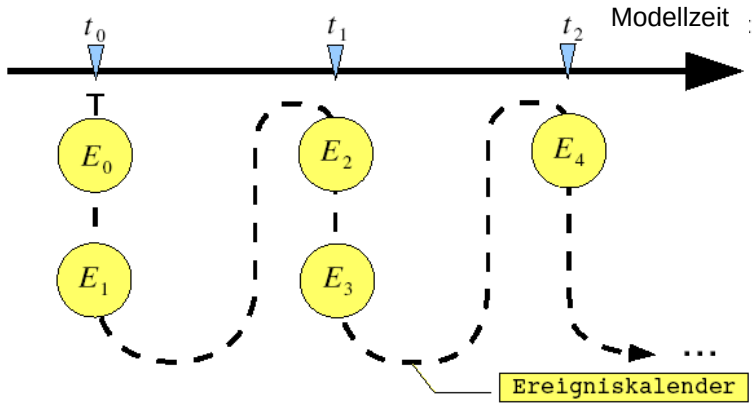
```
virtual SimTime getExecutionTime () const =0          // Get model time.
virtual SimTime setExecutionTime (SimTime time)=0  // Set model time.
virtual Priority getPriority () const =0
virtual Priority setPriority (Priority newPriority)=0    // Set new priority.
bool isScheduled () const                                 // Check if Sched object is in schedule.
SchedType getSchedType () const                           // Determine the Sched object's type.
virtual void execute ()=0                                 // Execution of Sched object.
```

Realisierungen der Next-Event-Simulation



Prozess-Scheduling

(Prozess als Folge von Ereignissen)



Ereignis-Scheduling

(Prozess als Folge von Ereignissen)

ODEMx erlaubt beide Varianten (auch im Mix)

[Sched als abstrakte Basisklasse von Process und Event]

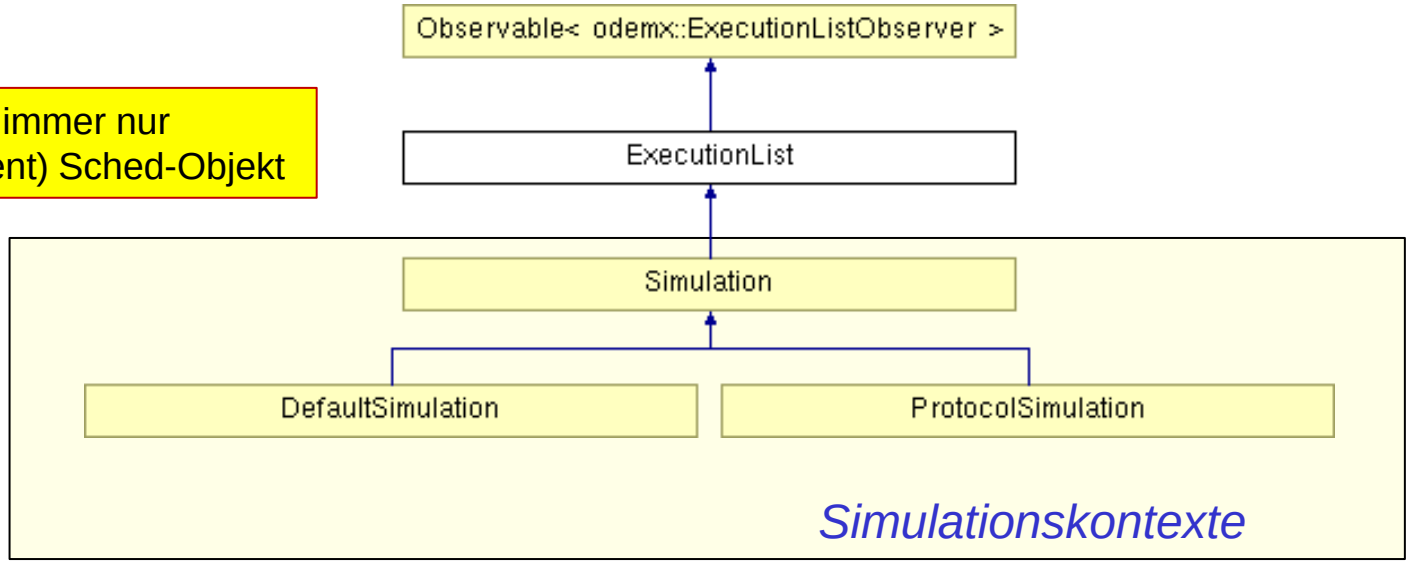
Schöne Übungsaufgabe: Clock-Beispiel als Event-Variante

Die Klasse ExecutionList *(Ereignisliste, Kalender)*

```
Sched * getNextSched ()   // top most Sched in ExecutionList
bool isEmpty ()           // check if ExecutionList is empty
virtual SimTime getTime () // const =0  get model time
```

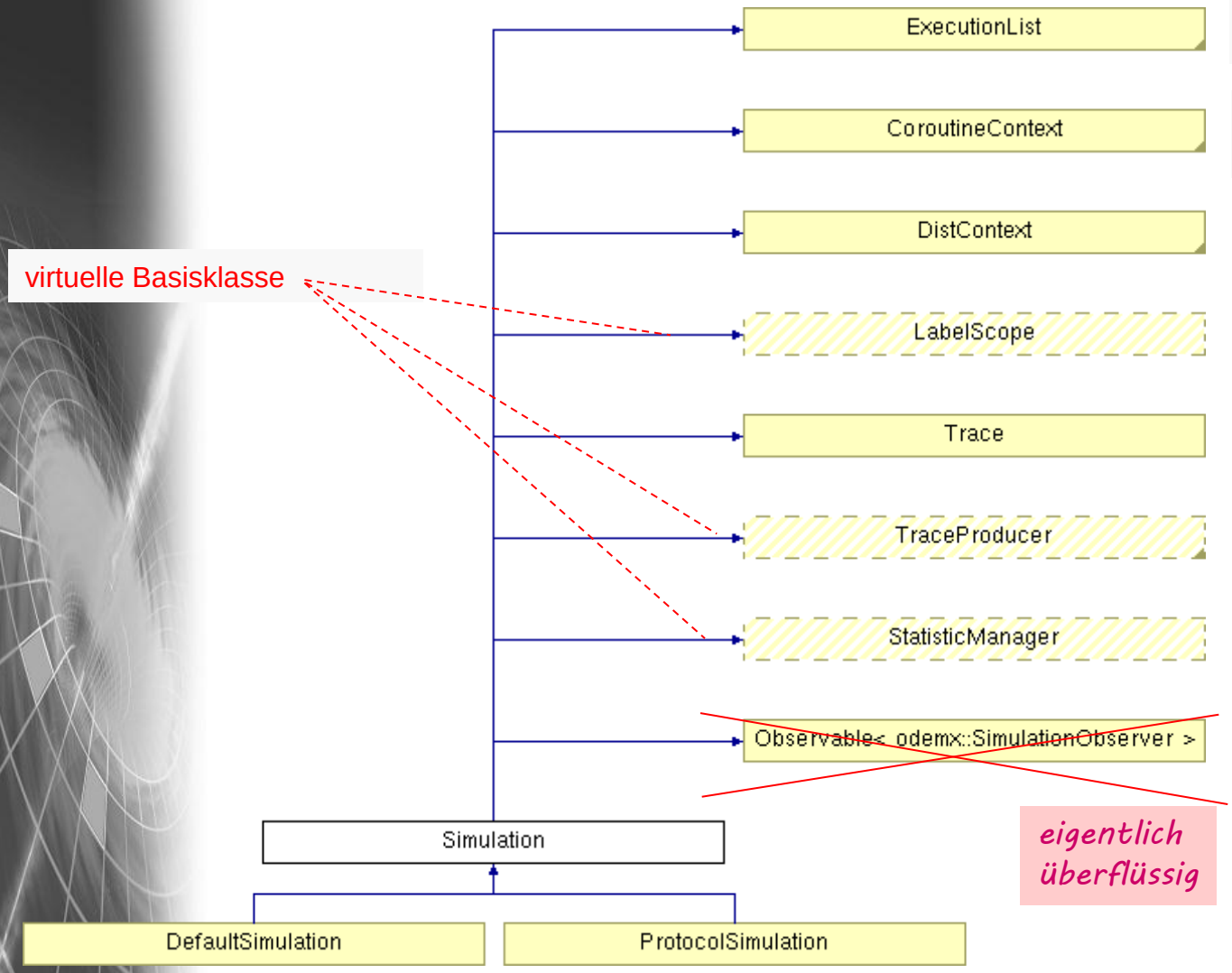
Vergangenheit wird nicht konserviert

ausgeführt wird immer nur das **erste** (current) Sched-Objekt



jeder Simulationskontext (Objekt von Simulation bzw. Ableitung) verfügt über eine eigene ExecutionList-Funktionalität

Simulationskontext



Prozess-Scheduling nach Zeit und Priorität
std::list<Sched*>

ExL

Koroutinen-Ensemble, inkl. C++ Hauptprogramm

Verwaltung aller erzeugten Zufallszahlengeneratoren

Objektnamenverwaltung

Trace-Manager um Ereignisse/Zustandsänderungen von Objekten zu registrieren

Erzeuger von Markierungen (marks) für Trace

Manager von Statistik-Objekten

Beobachtung registrierter Objekte

eigentlich überflüssig

3. Prozessverwaltung

1. Aufgaben von Klasse Simulation
2. Process-Listen eines Simulationskontextes
3. Allgemeines Process-Scheduling
4. Weitere Process-Funktionaliät
5. Prozesswarteschlangen: ProcessQueue, Port
6. Spezielles Process-Scheduling (Memory)

Grundstrategie

ein Prozess (d.h. Pointer zum Process-Objekt)

- bleibt in seinem gesamten Lebenslauf **einem einzigen Simulationskontext** zugeordnet

- ist während seines Lebenslaufes (in Abhängigkeit seines Grundzustandes) in vier unterschiedlichen **Listen** seines Simulationskontextes erfasst.

  Grundzustände
  - Created
  - Runable, dann auch in ExL
  - Idle, dann meist auch in dezentralen Synchronisationslisten
  - Terminated

Process-Member-Funktion
`State getState() const;`

zeitgleich kann ein blockierter Prozess (Idle) in weiteren Warteschlangen erfasst sein.

Zugriffsfunktionen für Process-Listen

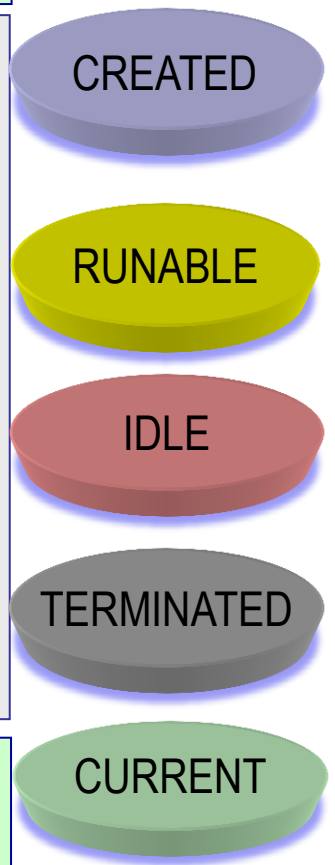
generell
Jeder Prozess wird in seinem gesamten Lebenslauf von seinem Simulationskontext verwaltet *(Zustandslisten eigentlich überflüssig)*

```
std::list<Process*>& Simulation::getCreatedProcesses() {
    return created;
}
std::list<Process*>& Simulation::getRunnableProcesses() {
    return runnable;
}
std::list<Process*>& Simulation::getIdleProcesses() {
    return idle;
}
std::list<Process*>& Simulation::getTerminatedProcesses() {
    return terminated;
}
```

ExL

```
Process* Simulation::getCurrentProcess()
                    Get currently executed process.
Sched * getCurrentSched ()
                    Get currently executed Sched object.
```

Prozess-grundzustand



Zeitbezug

SimTime

Modellzeit: Datentyp bestimmt Varianten von ODEMx: int, double

- now - aktuelle Modellzeit (private Simulation Member-Variable)

Zugriff (nur lesend)

- getCurrentTime()
- getSimulation()->getTime()

geplante Aktivierungszeit eines beliebigen Prozesses p in der ExL

- p->getExecutionTime()

semantisch äquivalent:
now==
getCurrentTime()==
getCurrentProcess()->getExecutionTime() ==
getSimulation()->getTime()

Funktionssignaturen

Process-Member-Funktion

```
SimTime Process::getExecutionTime() const;
        // aktuelle Ereigniszeit
        //      0.0, falls Prozess nicht in ExL eingetragen ist
        // (Vorsicht: 0.0 legt allein noch nicht den Grundzustand fest)
```

Simulation-Member-Funktion

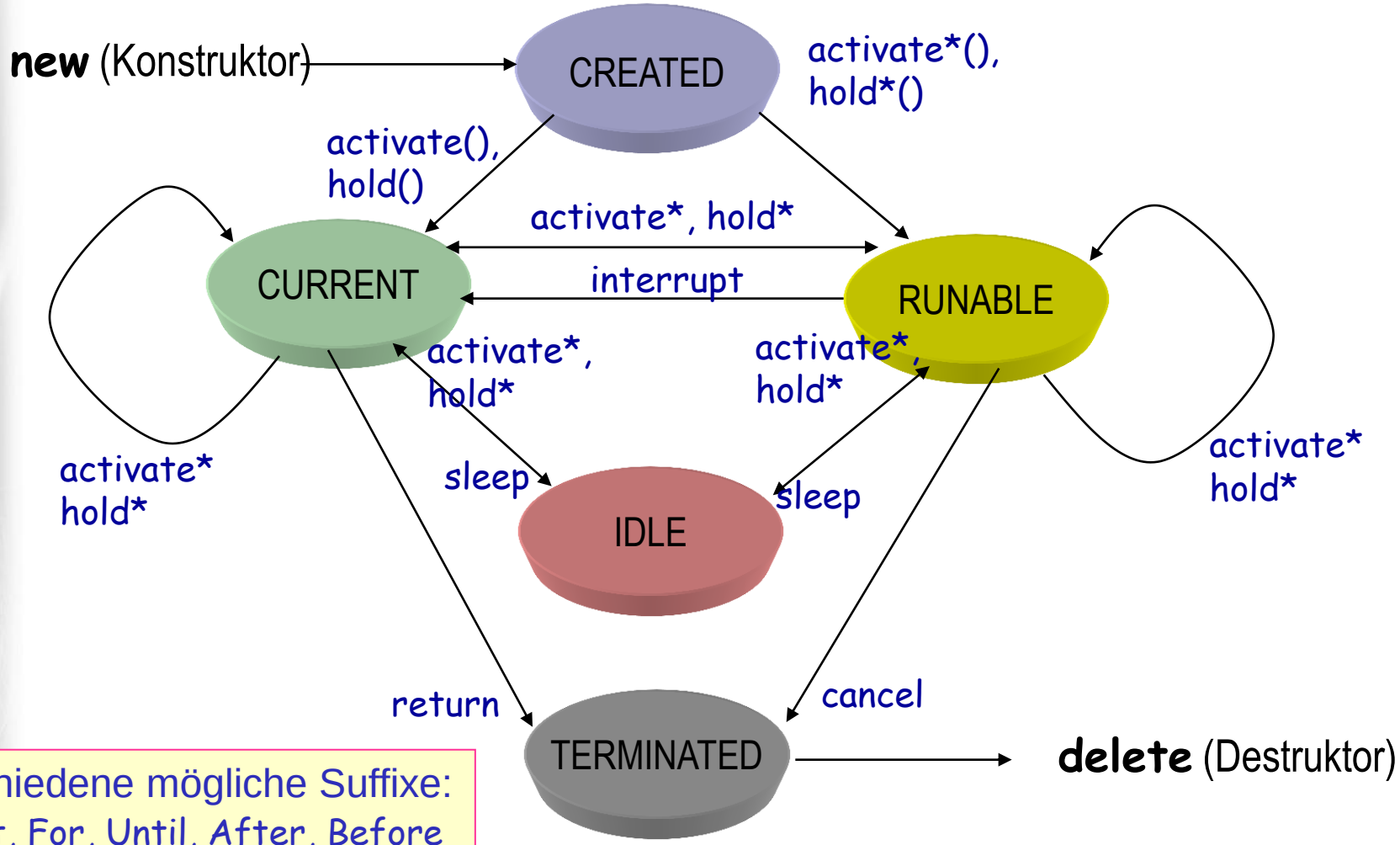
```
Process* Simulation::getCurrentProcess();
        // liefert Zeiger zum aktuellen Prozess der ExL

Simulation* getSimulation();
        // liefert Zeiger zum aktuellen Simulationskontext
```

globale Funktion

3. Prozessverwaltung

1. Aufgaben von Klasse Simulation (Wdh.)
2. Process-Listen eines Simulationskontextes
3. Allgemeines Process-Scheduling
4. Weitere Process-Funktionaliät
5. Prozesswarteschlangen: ProcessQueue, Port
6. Spezielles Process-Scheduling (Memory)

Überblick: Zustände und Scheduling-Operationen



* verschiedene mögliche Suffixe:
In, At, For, Until, After, Before