Kurs OMSI
im WiSe 2014/15

Objektorientierte Simulation mit ODEMx

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

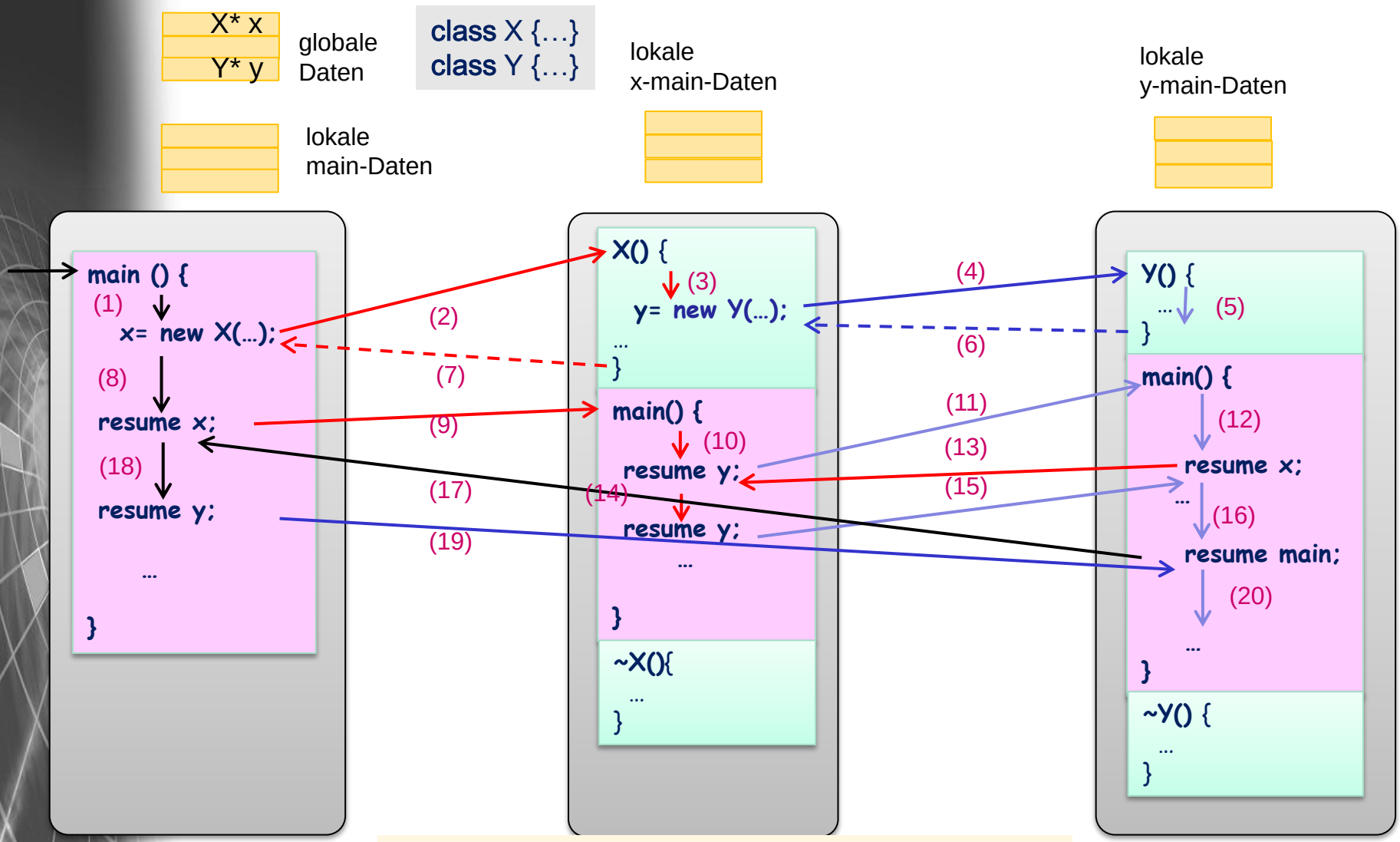
fischer|ahrens|eveslage@informatik.hu-berlin.de

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
   - Vorbild Simula
   - Aufbau von ODEMx (erster Blick)
   - Triviales Clock-Beispiel

1.Simula-Idee: *Koroutinen* als ausgezeichnete Member-Funktion einer Klasse



3 Koroutinen: X::main für x, Y::main für y
plus ausgezeichnete Koroutine ::main

2.Simula-Idee: Process, Event, Scheduling-Operationen

- Die vordefinierte Klasse **Process** ist mit einer Member-Funktion ausgestattet, die als Koroutine eingesetzt werden kann.
  Objekte der Klasse sind mit Objekten der Klasse **Event** verlinkt.
  Diese wiederum nach der Größe ihres Attributs *eventTime* in einem **Ereigniskalender** sortiert.

- Das **Process**-Objekt mit dem ersten Ereigniseintrag ist der Current-Prozess.
  Dessen Koroutine ist die, die im Moment aktiv ist.

- Der **Steuerungswechsel** zwischen den Koroutinen
  (die die Lebenslaufrealisierungen ihrer Objekte darstellen) erfolgt in Abhängigkeit ihres Modellzeitverbrauchs

- Es gibt vordefinierte **Scheduling-Operationen**: hold, activate, passivate, mit der die aktive Koroutine ihre Steuerung an eine andere Kororoutine übergeben kann:
- explizit: **activate** x, **reactivate** y

  implizit: **hold** (dt), **passivate**, **terminate**

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
   - Vorbild Simula
   - Aufbau von ODEMx (erster Blick)
   - Triviales Clock-Beispiel

Umsetzung in C++

- Basis für Realisierung der **Koroutinen-Sprünge** sind die C-Operationen **set_jmp** und **long_jmp**

setjmp.h is a header defined in the C standard library to provide "non-local jumps": control flow that deviates from the usual subroutine call and return sequence.

The complementary functions **setjmp** and **longjmp** provide this functionality.

A typical use of setjmp/longjmp is implementation of an exception mechanism that exploits the ability of longjmp to reestablish program or thread state, even across multiple levels of function calls.

A less common use of setjmp is to create syntax similar to coroutines.

- Die **Speicherung** der Aufrufstacks der aktiven Koroutine (bei Verlassen) und die **Restaurierung** der Aufrufstacks (bei Wiedereintritt)
- **Laufzeitverwaltung** der Prozesse und ihrer Ereignisse zur Implementierung der **Scheduling-Operationen**

Grundidee einer hierarchischen Prozessverwaltung

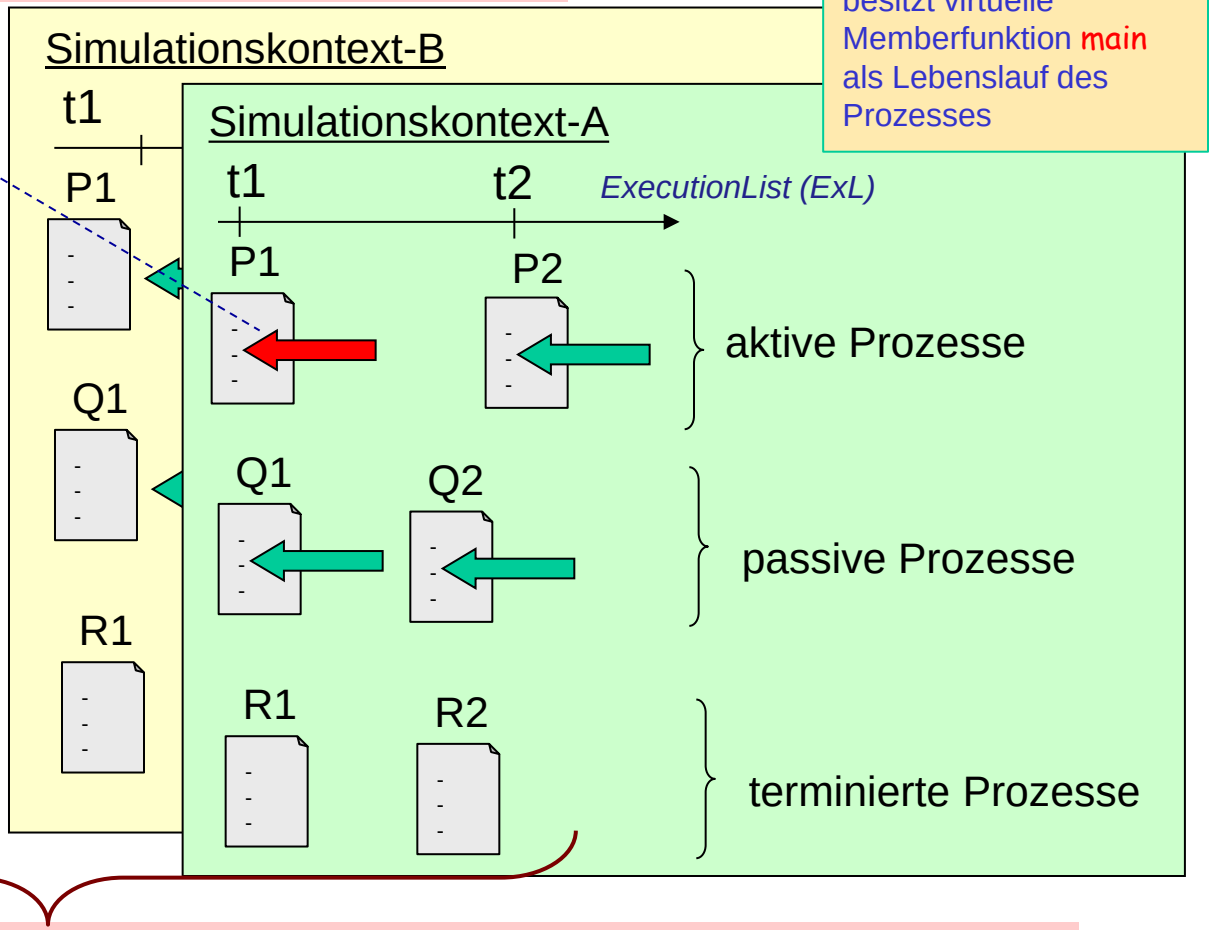
*Zu einem Zeitpunkt kann immer **nur ein Kontext** und in dem **nur ein Prozess** aktiv sein (current)*

Klasse Process
besitzt virtuelle Memberfunktion main als Lebenslauf des Prozesses

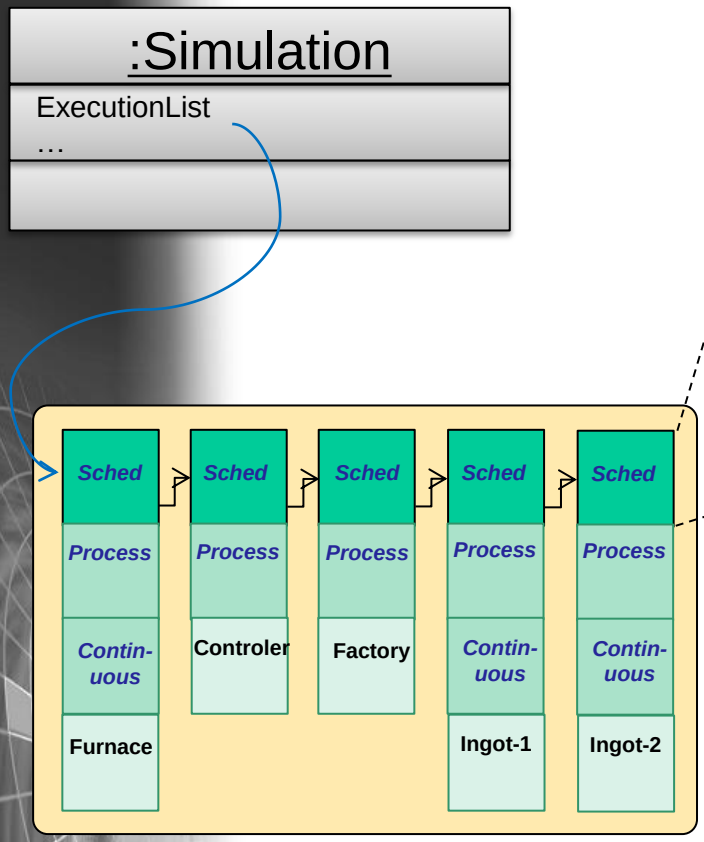
Current- Prozess
- erster Eintrag
- kleinste Zeit
- höchste Priorität

C++ Hauptprogramm

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



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



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,
RUNABLE, IDLE, TERMINATED

Priorität:
Gleichzeitigkeitskonfliktbehebung bei der
Sequentialisierung von Ereignissen

Ereigniszeit:
Basis der Sortierung des Terminkalenders (ExecutionList)
ModelTime-Typ ist variabel

Abstrakter Lebenslauf: main
Redefinition von execute
(Fortsetzung im Lebenslauf)

Konkrete aktive Klasse

Factory

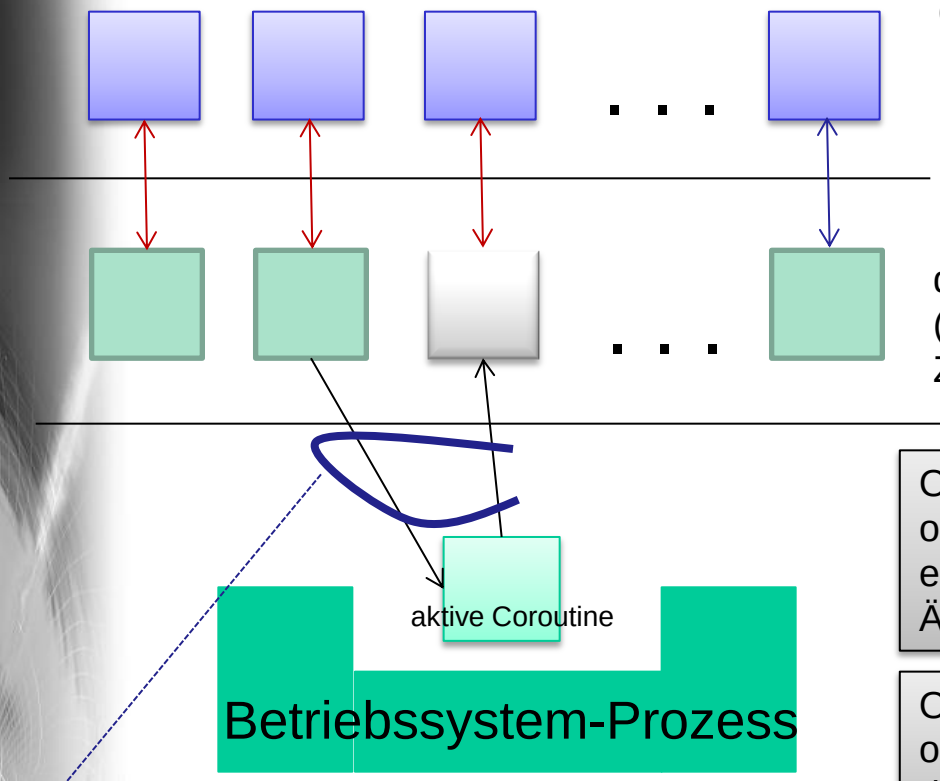
-SimTime dt
-Ingot* ing

```
#int main() {
      holdFor (dt);
      ing= new Ingot (…);
      ing->activate ();
}
```

Konkreter Lebenslauf

Zeitverbrauch (Terminkalender)
Erzeugung und
Aktivierung eines Prozess-Objektes

Universelle ODEMx- Urvariante



dynamisches Ensemble von realen/gedachten Prozessen (zeitdiskret/zeitkontinuierlich) (**Original**)

Struktur-Äquivalenz (1:1)

dynamisches Ensemble von ODEMx-Prozessen (**Simulationsmodell**)
Zustand jedes Prozesses auf der Halde

ODEMx-Laufzeitsystem
organisiert **Scheduling** der ODEMx-Prozesse entsprechend ihrem Modellzeitverbrauch bei Änderung ihrer Zustandsgrößen in ihrem Kontext

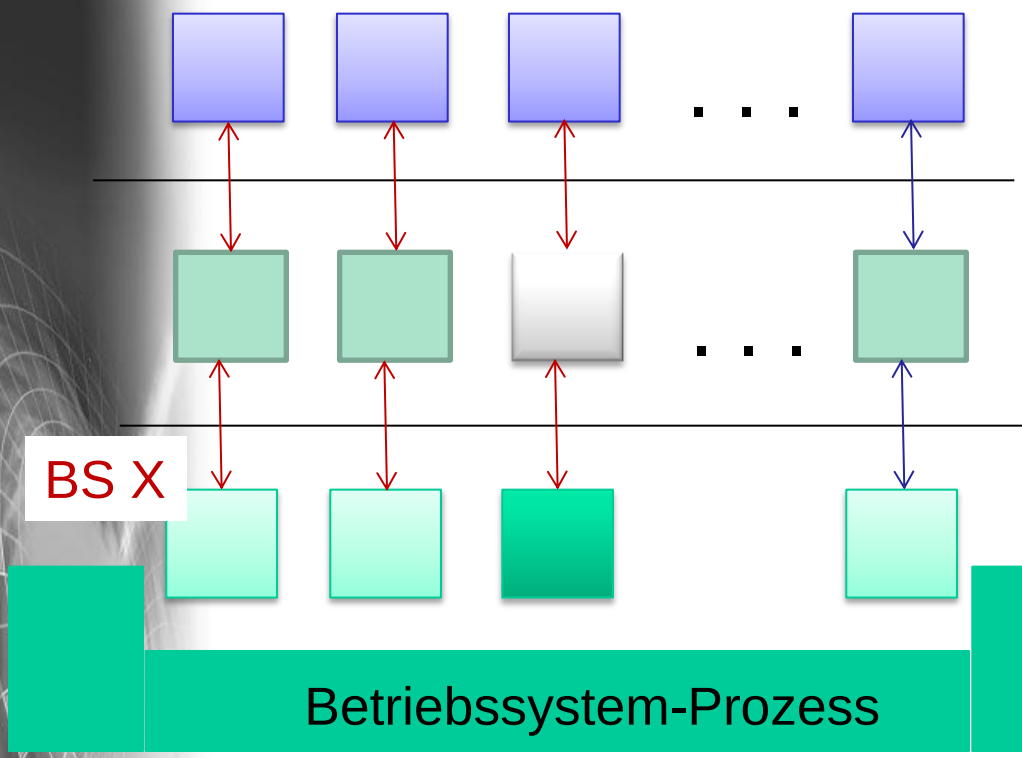
ODEMx-Laufzeitsystem
organisiert Speicherung und Restaurierung der Koroutinen-Zustände (Laufzeitkeller, Register)

rechenzeitaufwändiger

… im Vergleich zu prozeduralen Next-Event-Verfahren

aber hoher Grad an (auf natürliche Weise erreichbarer) Strukturäquivalenz

Laufzeitverbesserte Varianten



dynamisches Ensemble von realen/gedachten Prozessen (zeitdiskret/zeitkontinuierlich) (Original)

Struktur-Äquivalenz (1:1)

dynamisches Ensemble von ODEMx-Prozessen (Simulationsmodell)

Struktur-Äquivalenz (1:1)

ODEMx-Laufzeitsystem
organisiert Scheduling der ODEMx-Prozess- entsprechend ihrem Modellzeitverbrauch für Änderung ihrer Zustandsgrößen

X= Fibres, MS Windows

X= uContext, Linux

ODEMx-Laufzeitsystem
organisiert Umsetzung von Zeigern zu Koroutinen-Zuständen (Laufzeitkellern) und speichert und restauriert lediglich Register

2. Prinzip der Next-Event-Simulation

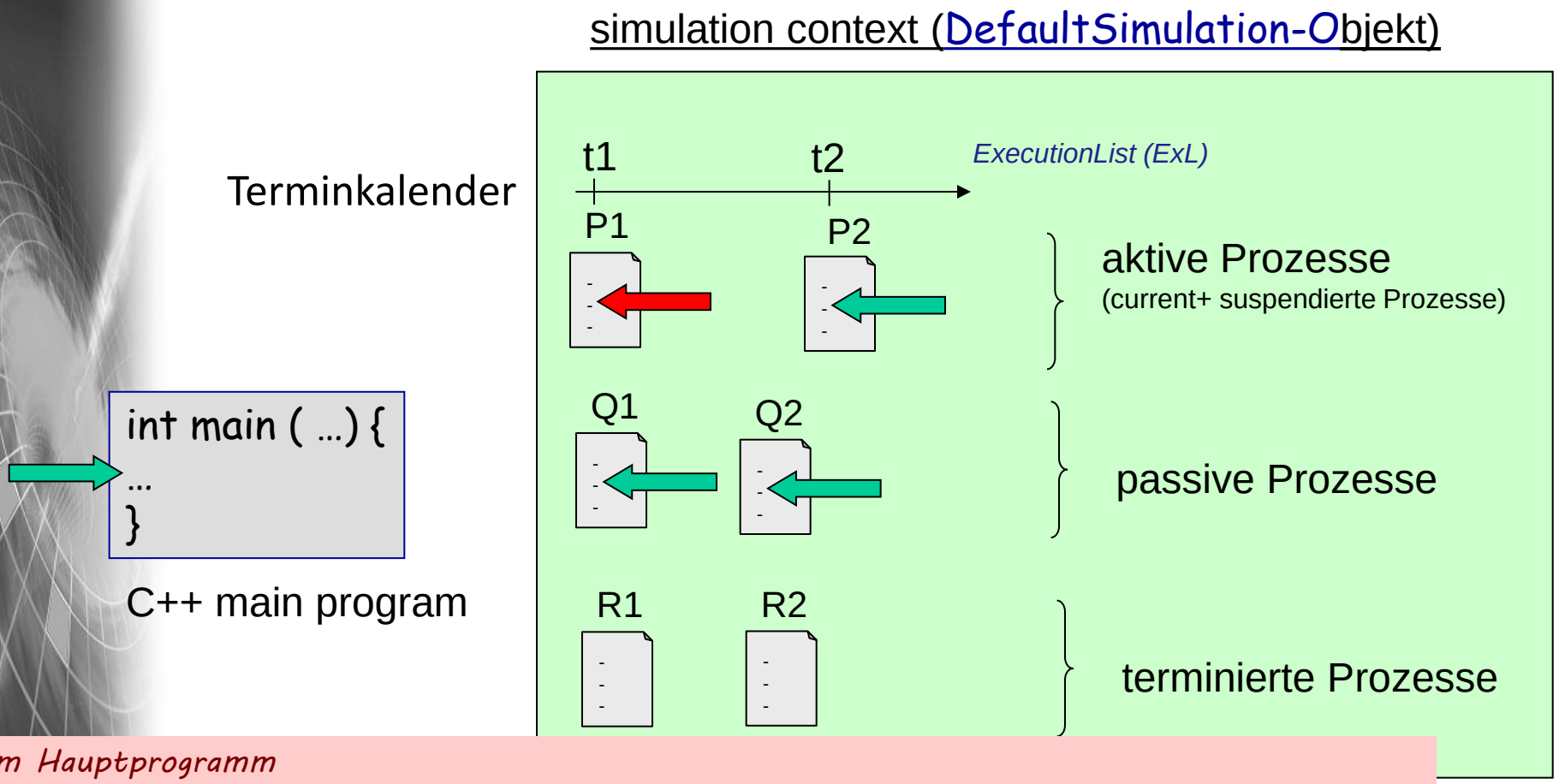
1. Charakterisierung der Next-Event-Simulation
   - Ereignisse, Next-Event-Scheduler
   - Barren-Beispiel
   - Zusammenhang von ereignisbasierter und prozessbasierter Modellbeschreibung

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

Standardfall: nur ein Simulationskontext

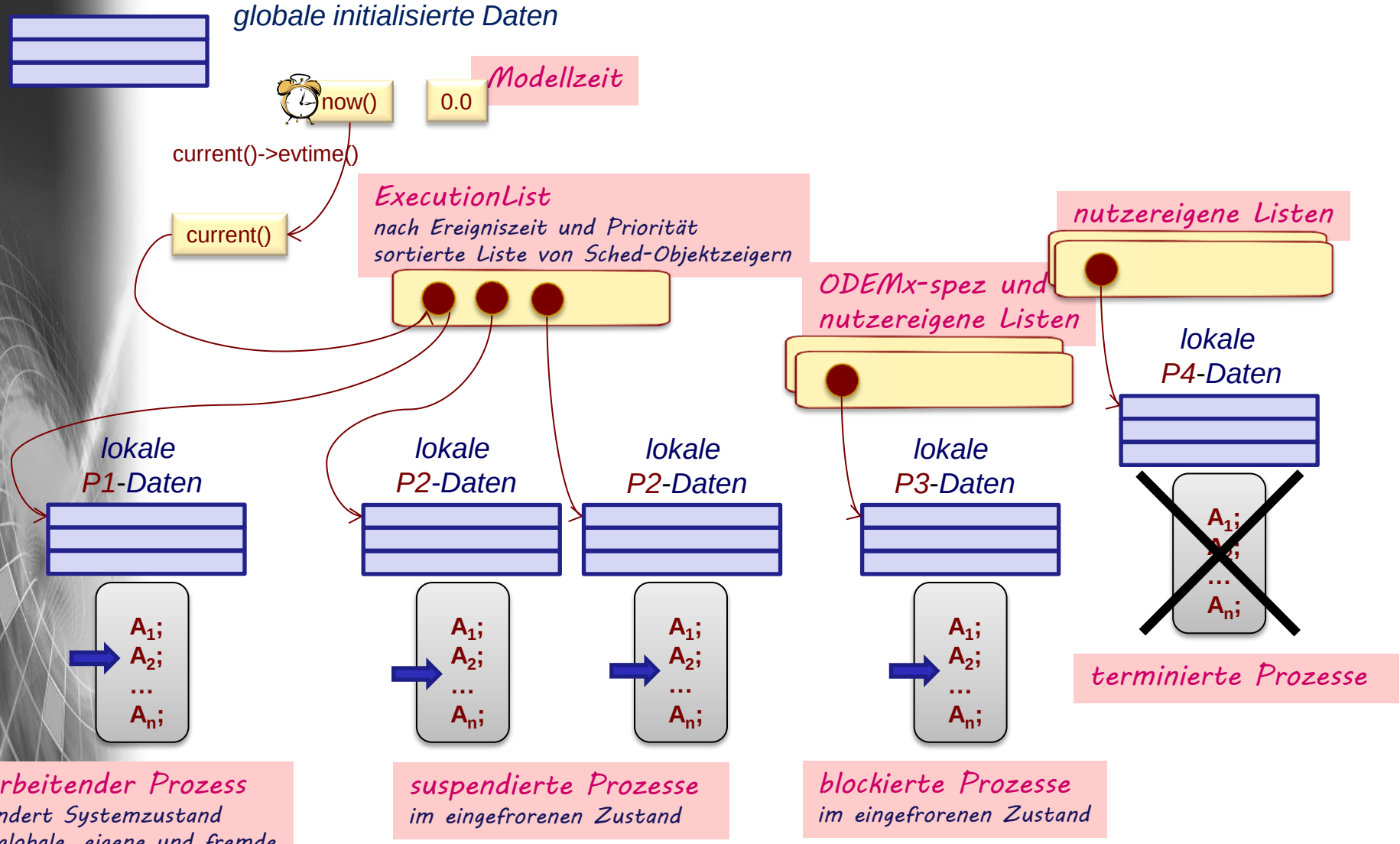
Zu einem Zeitpunkt ist entweder
- *das Hauptprogramm oder*
- *der Current-Prozess des Kontextes*

aktiv

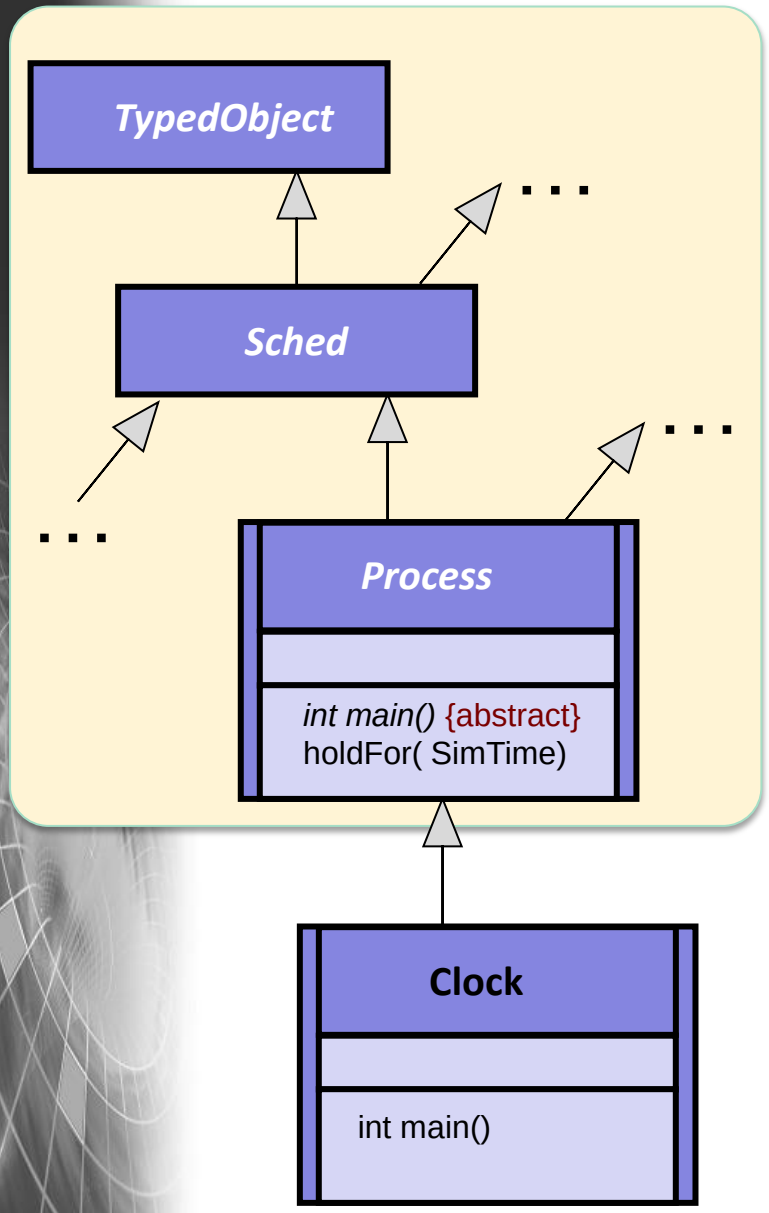


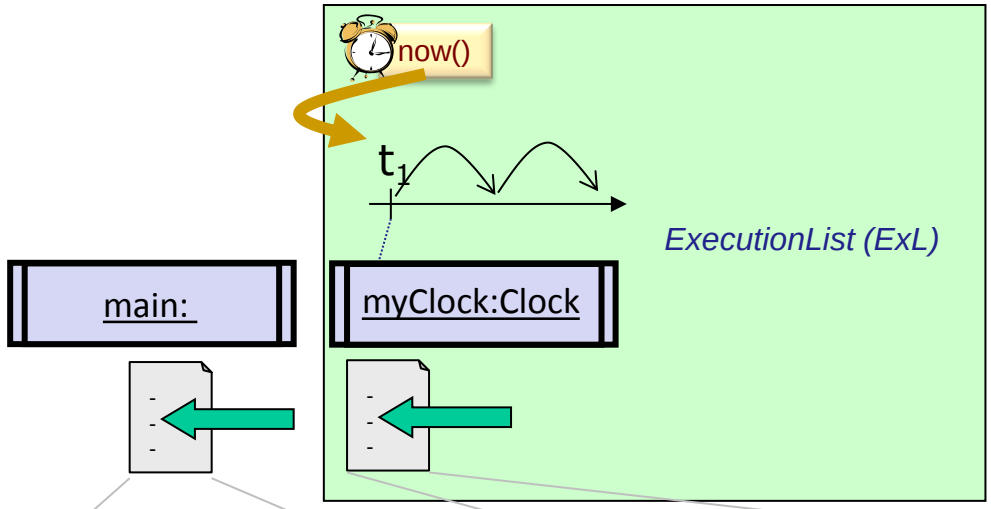
Dem Hauptprogramm
fallen nur übergeordnete Management-Aufgaben zu (Context-Verwaltung),
*es nimmt **nicht** wie in Simula, SLX, oder früheren ODEMx-Versionen am Scheduling **selbst** teil·*

Schema der Zustandsänderung von ODEMx



Einfaches Beispiel





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

forever
- Zeitverbrauch (1 ZE)
- AUSGABE
  Punkt-Zeichen

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

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;
        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;
        cout << "continue until SimTime 13.0 is reached or passed";
        getDefaultSimulation()->runUntil(13.0);
        cout << endl  << "time=" << getDefaultSimulation()->getTime() << endl;

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

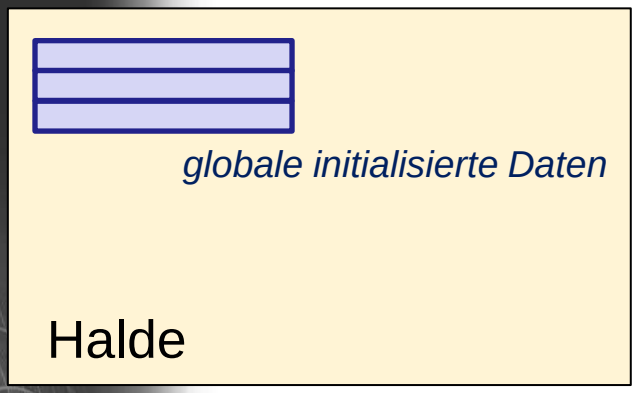
StatusAbfrage-Operationen

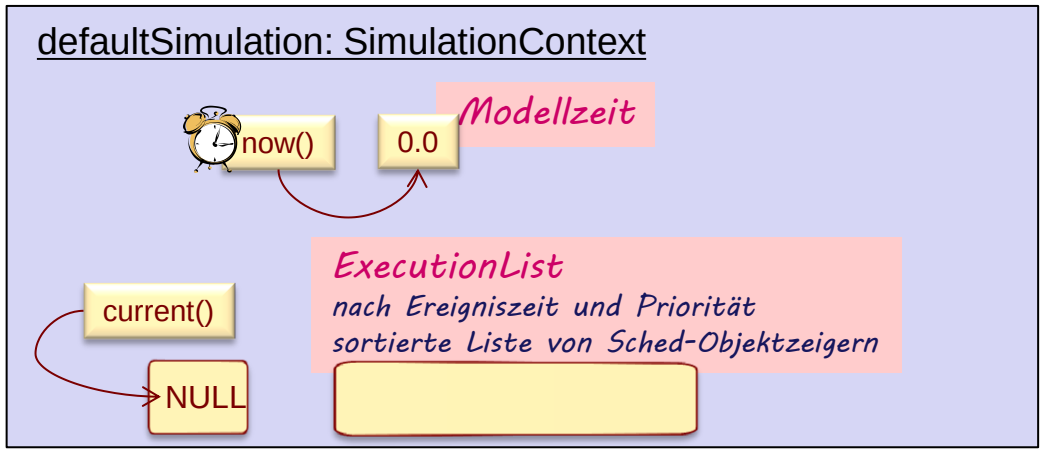
Scheduling-Operationen

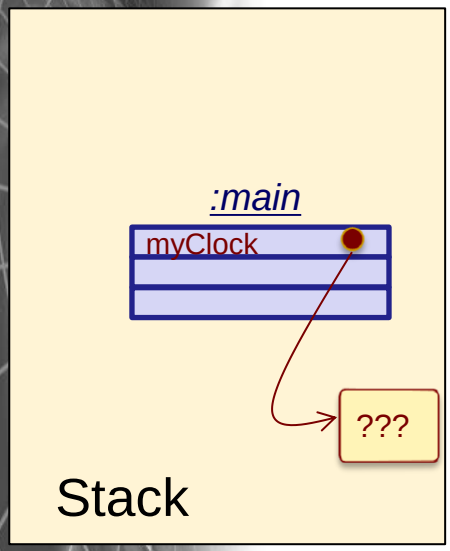
Simulationskontext

Kontext-Aktivierungsoperationen

Übergabe der Steuerung durch das Betriebssystem





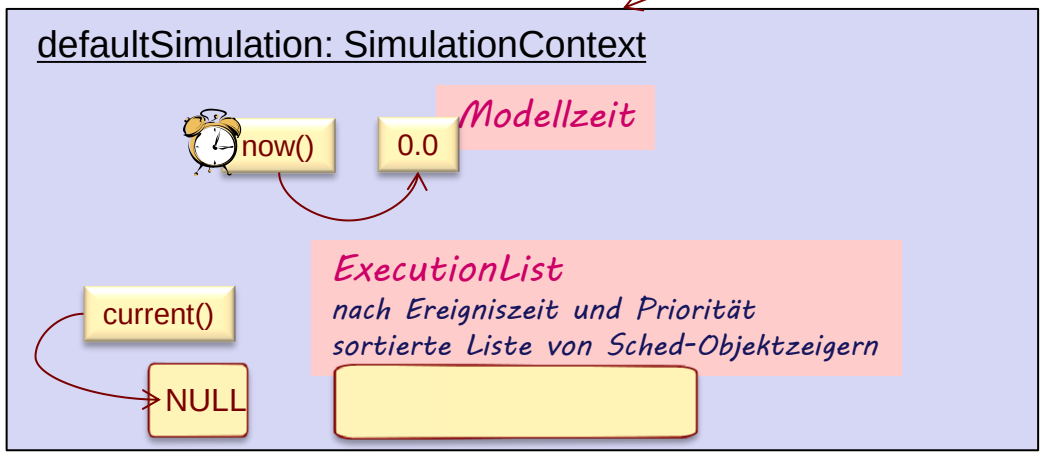



```
int main(int argc, char* argv[]) {
        Clock* myClock= new Clock (getDefaultSimulation());
        myClock->activate();

        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;
}
```

nichtsichtbarer InitialisierungsCode
von globalen Variablen, Objekten
a) C++ Laufzeitsystem
b) ODEMx-Laufzeitsystem
c) Anwenderprogramm

main: Objekt-Generierung

globale initialisierte Daten

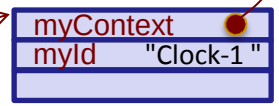



```
int main(int argc, char* argv[]) {
        Clock* myClock= new Clock (getDefaultSimulation());
        myClock->activate();

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

        for (int i=1; i<5; ++i) {
            getDefaultSimulation()->s
            cout << endl << i << ". ste
                getDefaultSimulation
        }
        cout << endl;
        cout << "continue until SimT
        getDefaultSimulation()->run
        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;
        }
};
```

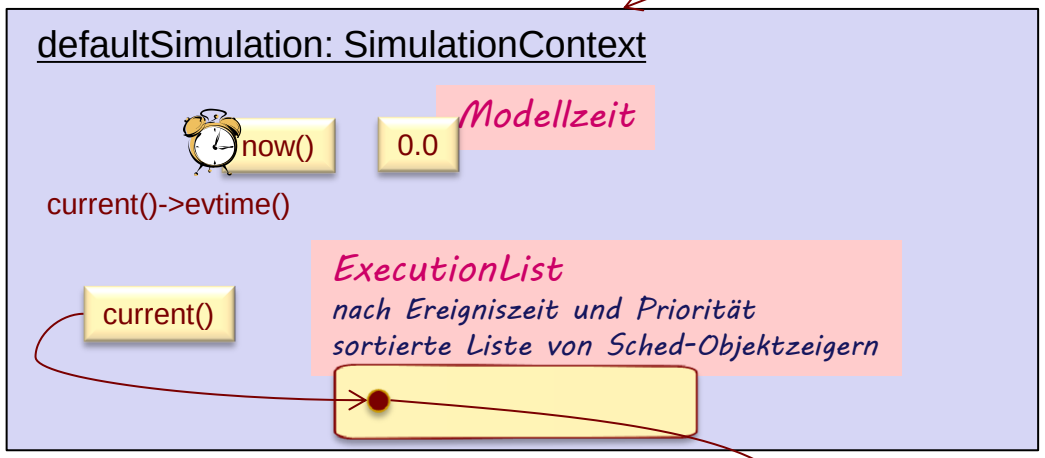
:main

:Clock



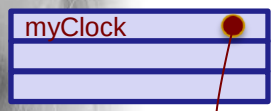
main: Befüllen des Terminkalenders

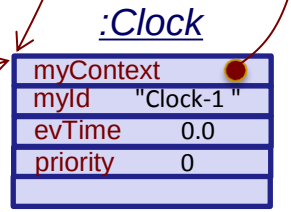
globale initialisierte Daten




```
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;
}
```

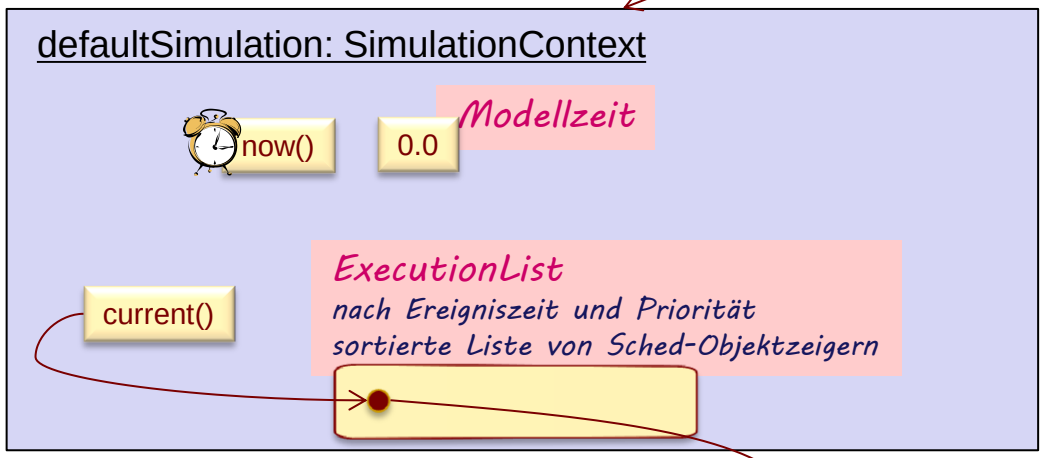
```
class Clock : public Process {
public:
    Clock (Simulation* sim) :
        Process(sim, "Clock") {}

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





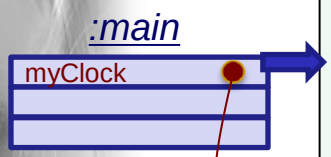
main: Ausgabe

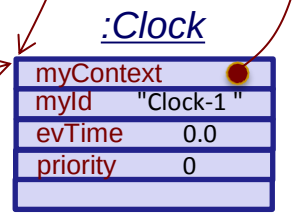
globale initialisierte Daten




```
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;
}
```

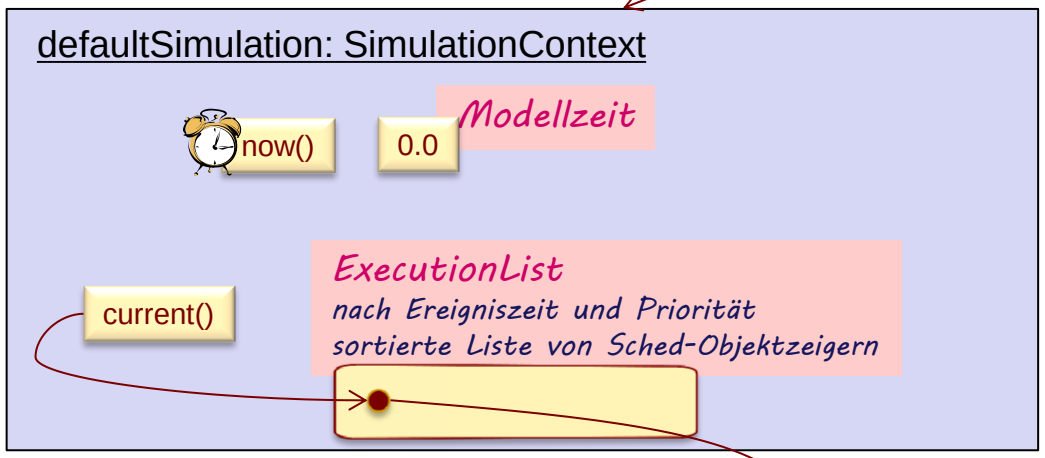
```
class Clock : public Process {
public:
        Clock (Simulation* sim) :
            Process(sim, "Clock") {}

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

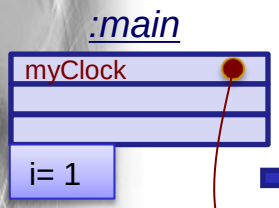




main: Steuerungsübergabe (Schrittmodus)

globale initialisierte Daten



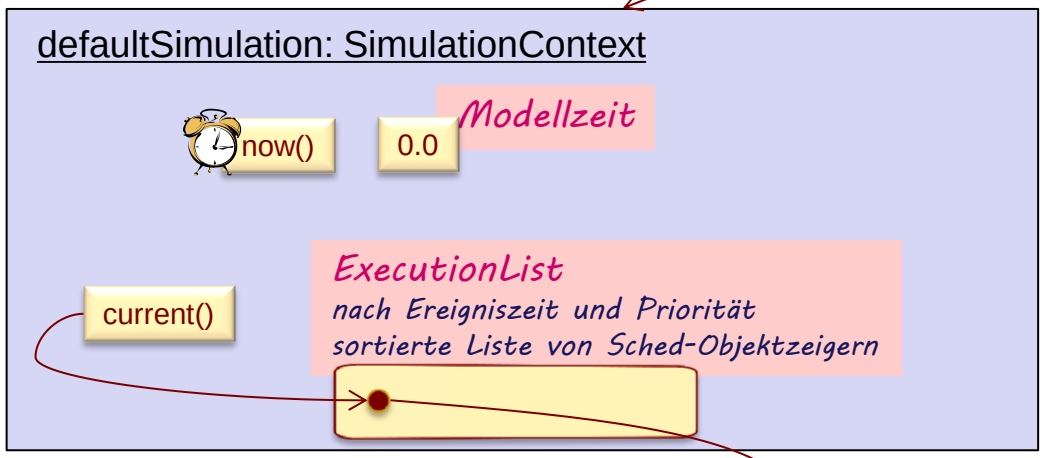


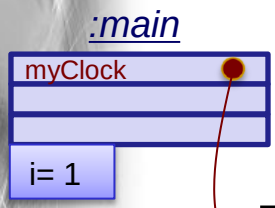

```
int main(int argc, char* argv[]) {
        Clock* myClock= new Clock (getDefaultSimulation());
        myClock->activate();

        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()->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;
        }
};
```

erfolgter Stack-Wechsel u. Sprung

globale initialisierte Daten



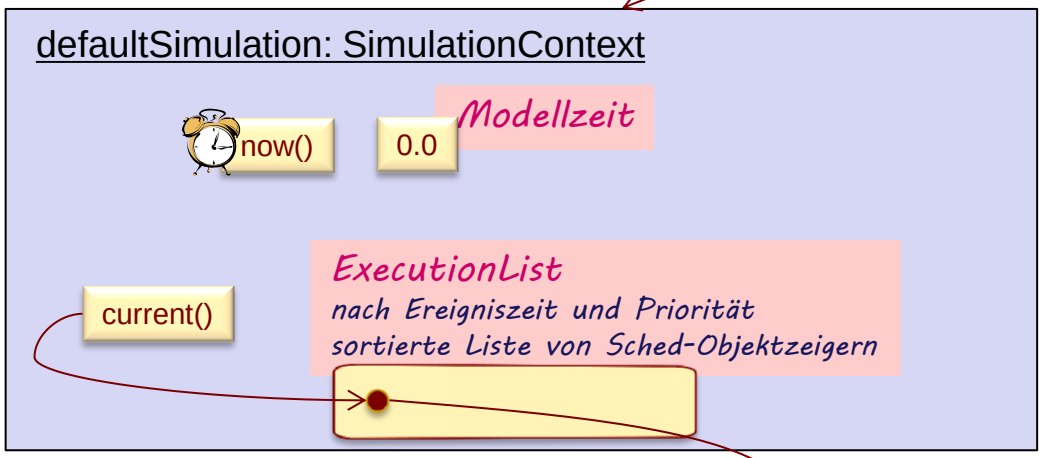



```
int main(int argc, char* argv[]) {
        Clock* myClock= new Clock (getDefaultSimulation());
        myClock->activate();

        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;
}
```

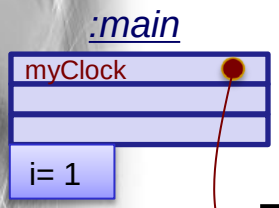

```
class Clock : public Process {
public:
        Clock (Simulation* sim) :
            Process(sim, "Clock") {}

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

Clock-1: Verzögerung um 1 ZE

globale initialisierte Daten



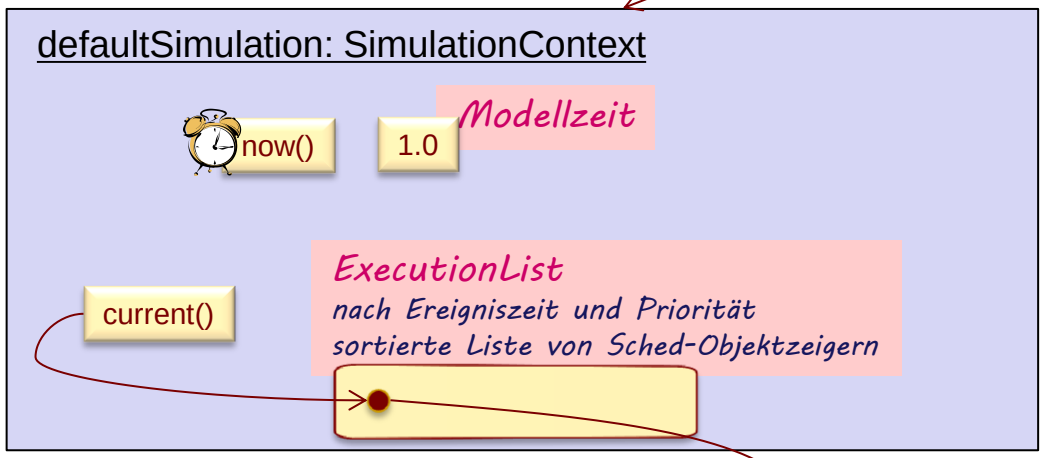

```
int main(int argc, char* argv[]) {
        Clock* myClock= new Clock (getDefaultSimulation());
        myClock->activate();

        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()->get

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



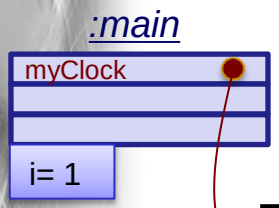

```
class Clock : public Process {
public:
        Clock (Simulation* sim) :
            Process(sim, "Clock") {}

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

Clock-1: vollzogene Verzögerung um 1 ZE

globale initialisierte Daten



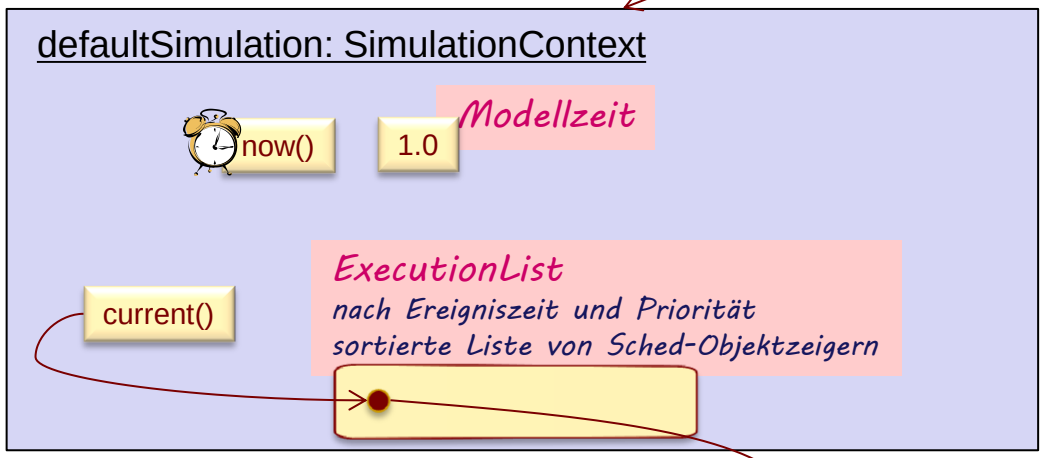

```
int main(int argc, char* argv[]) {
        Clock* myClock= new Clock (getDefaultSimulation());
        myClock->activate();

        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()->get

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



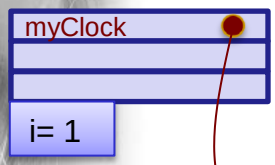

```
class Clock : public Process {
public:
        Clock (Simulation* sim) :
            Process(sim, "Clock") {}

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

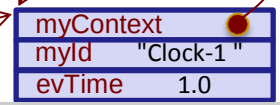
Clock-1: Rücksprung ins Hauptprogramm

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()->get

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



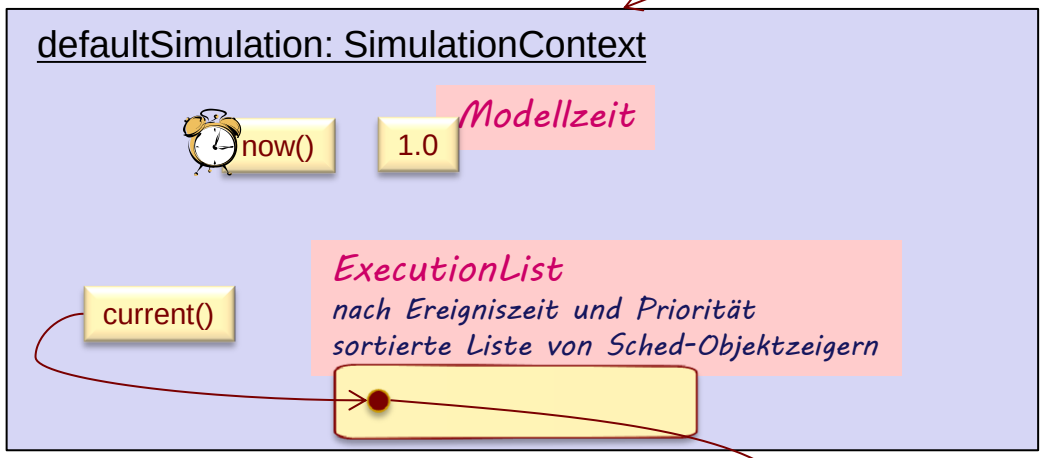



```
class Clock : public Process {
public:
        Clock (Simulation* sim) :
            Process(sim, "Clock") {}

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

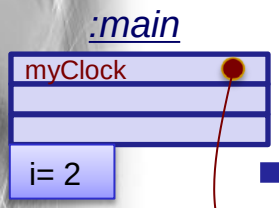
Ausgabe

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

main: Steuerungsübergabe (Schrittmodus)



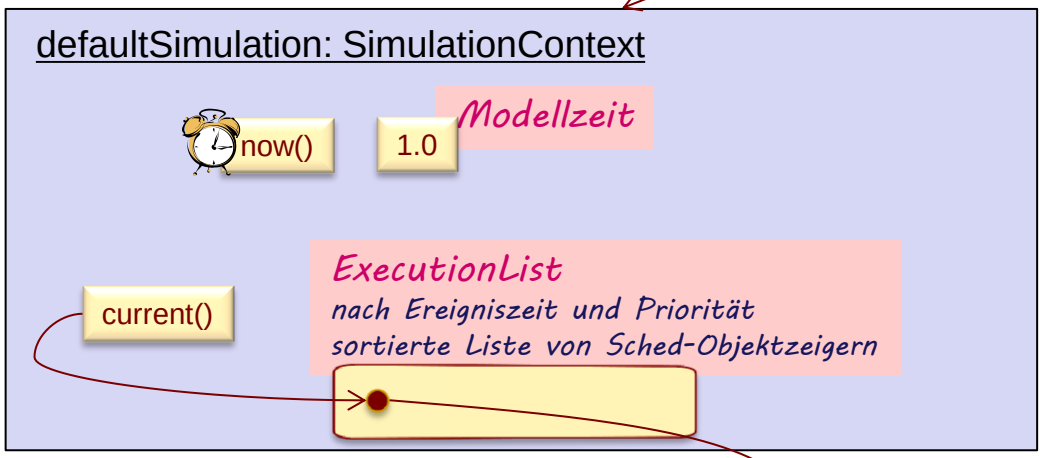

```
int main(int argc, char* argv[]) {
        Clock* myClock= new Clock (getDefaultSimulation());
        myClock->activate();

        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()->get

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



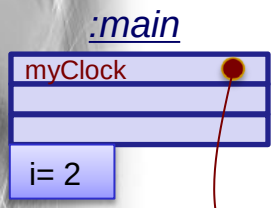

```
class Clock : public Process {
public:
        Clock (Simulation* sim) :
            Process(sim, "Clock") {}

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

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()->get

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



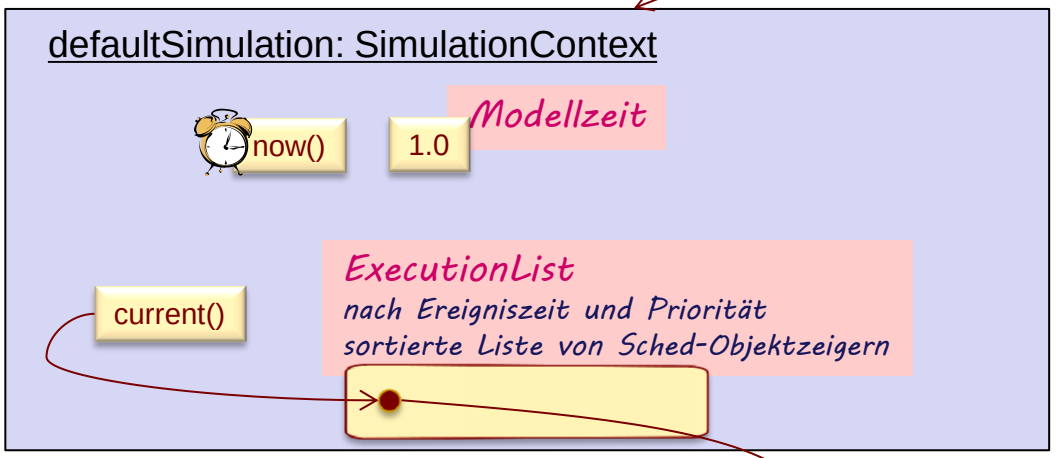

```
class Clock : public Process {
public:
        Clock (Simulation* sim) :
            Process(sim, "Clock") {}

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

Ausgabe

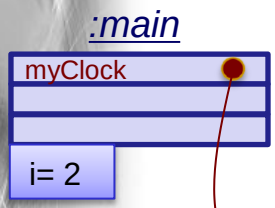
```
Basic Simulation Example
========================
1.  time step at= 1.0
.
```

Clock-1: Verzögerung um 1 weitere ZE

globale initialisierte Daten



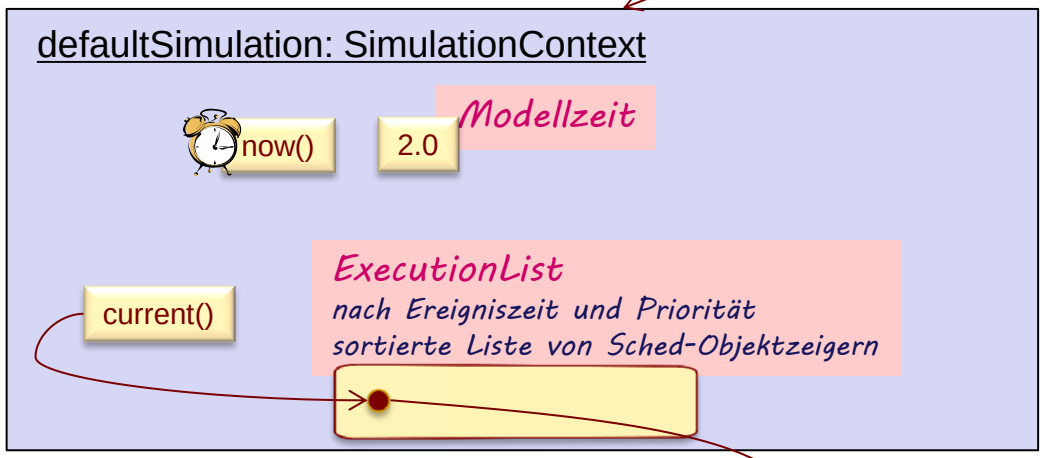

```
int main(int argc, char* argv[]) {
        Clock* myClock= new Clock (getDefaultSimulation());
        myClock->activate();

        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()->get

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



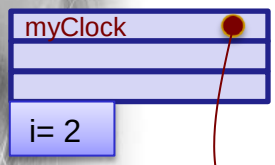

```
class Clock : public Process {
public:
        Clock (Simulation* sim) :
            Process(sim, "Clock") {}

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

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()->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;
        }
};
```

Ausgabe

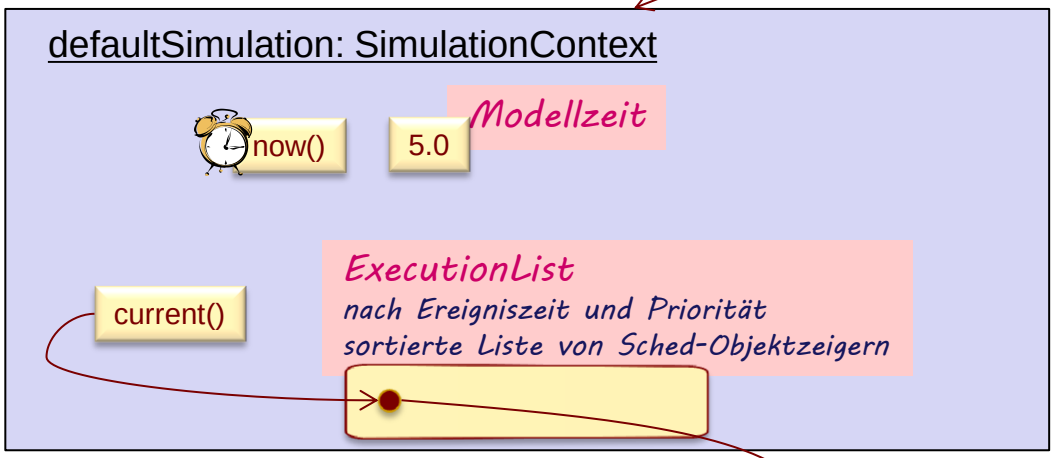
```
Basic Simulation Example
========================
1.  time step at= 1.0
.
2. time step at= 2.0
```

Ausgabe bis zur Beendigung der For-Anw.

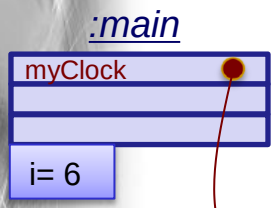
```
Basic Simulation Example
========================
1.  time step at= 1.0
.
2. time step at= 2.0
.
3. time step at= 3.0
.
4. time step at= 4.0
.
5. time step at= 5.0
```

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()->get

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



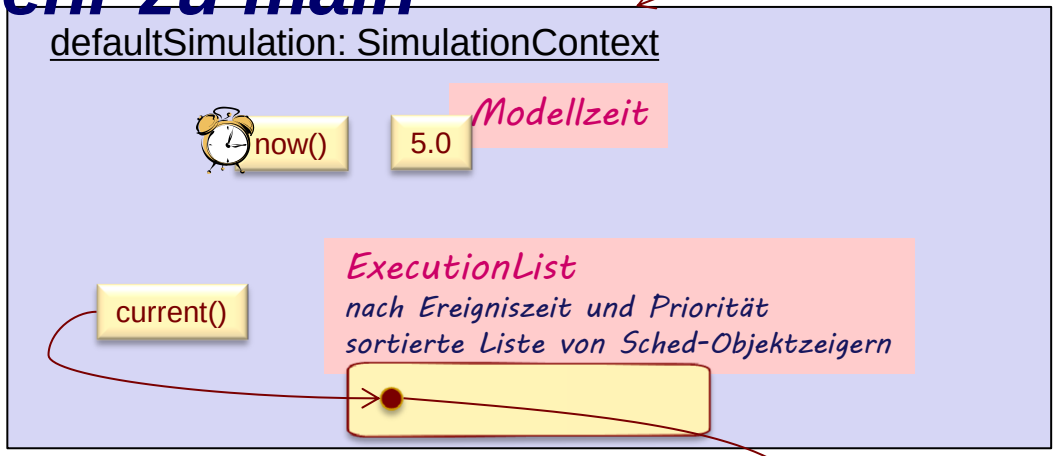

```
class Clock : public Process {
public:
        Clock (Simulation* sim) :
            Process(sim, "Clock") {}

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

Ausgabe bis zur Beendigung der For-Anw.

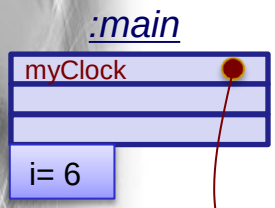
```
Basic Simulation Example
========================
1.  time step at= 1.0
.
2. time step at= 2.0
.
3. time step at= 3.0
.
4. time step at= 4.0
.
5. time step at= 5.0
.
```

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()->get

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



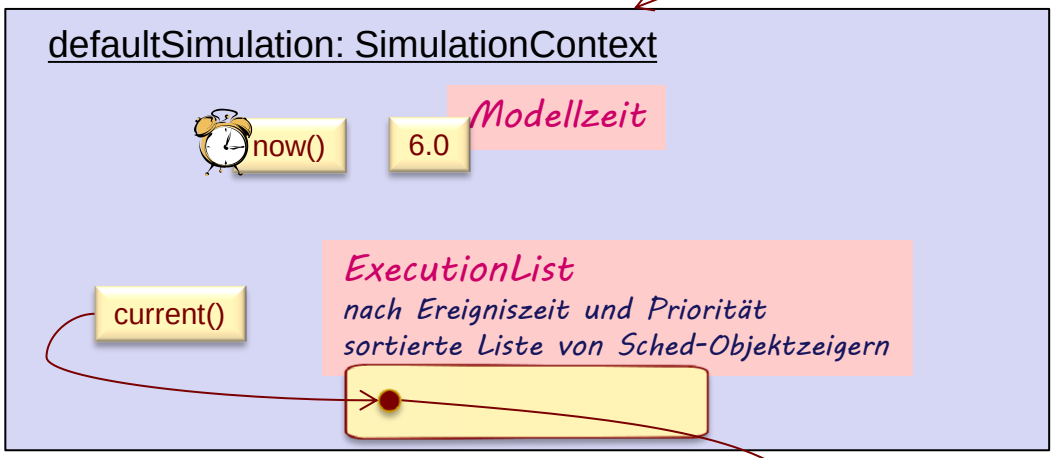

```
class Clock : public Process {
public:
        Clock (Simulation* sim) :
            Process(sim, "Clock") {}

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

Ausgabe nach Beendigung der For-Anw.

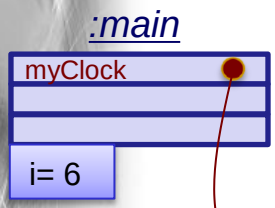
```
Basic Simulation Example
========================
1. time step at= 1.0
.
2. time step at= 2.0
.
3. time step at= 3.0
.
4. time step at= 4.0
.
5. time step at= 5.0
.
continue until SimTime 13.0 is reached or passed
```

main: Steuerungsübergabe (Intervallmodus)

globale initialisierte Daten



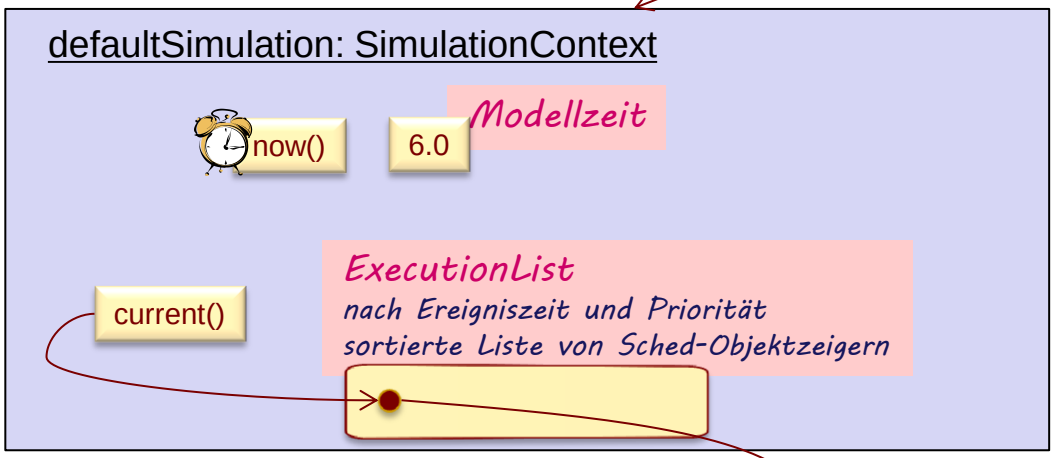

```
int main(int argc, char* argv[]) {
        Clock* myClock= new Clock (getDefaultSimulation());
        myClock->activate();

        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()->get

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



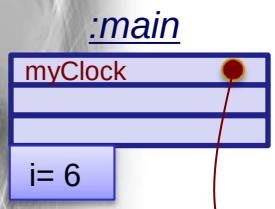

```
class Clock : public Process {
public:
        Clock (Simulation* sim) :
            Process(sim, "Clock") {}

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

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;
        }
};
```

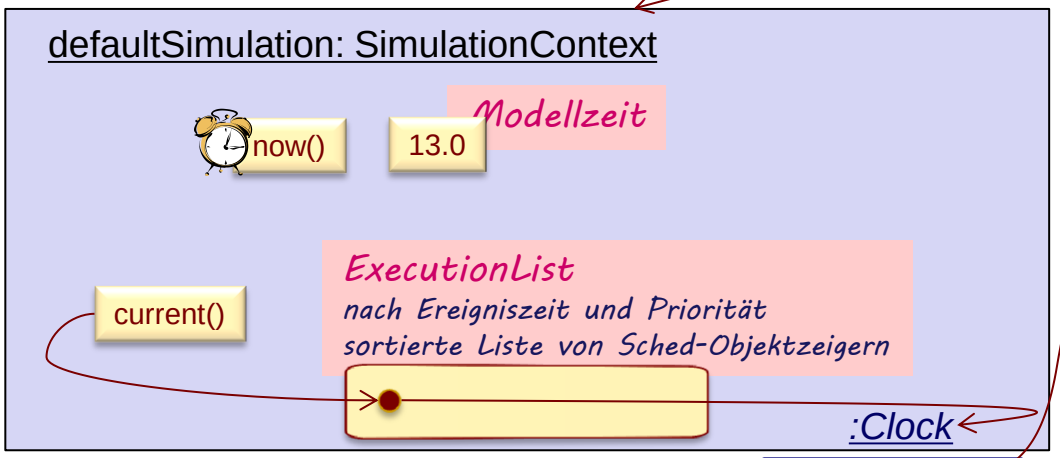
Ausgabe bis zum Erreichen von 13.0

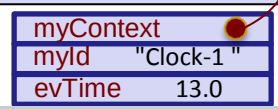
```
Basic Simulation Example
========================
1. time step at= 1.0
.
2. time step at= 2.0
.
3. time step at= 3.0
.
4. time step at= 4.0
.
5. time step at= 5.0
.
continue until SimTime 13.0 is reached or passed
........
```

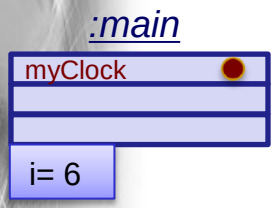
insgesamt 13 Punkte

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;
}
```

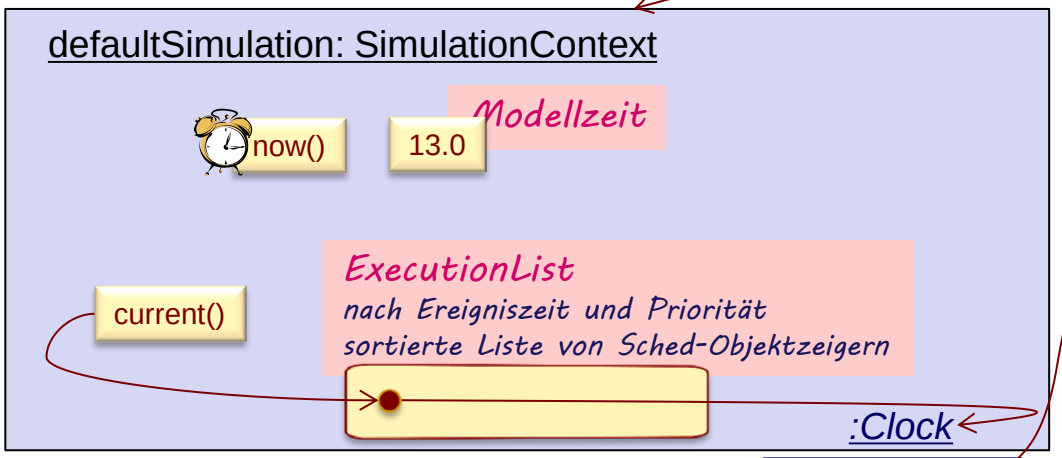
```
class Clock : public Process {
public:
        Clock (Simulation* sim) :
            Process(sim, "Clock") {}

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

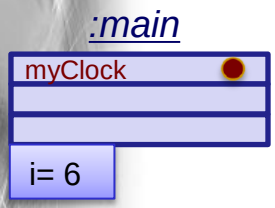
Ausgabe bis zum Erreichen von 13.0

```
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. Prozess-Scheduling

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()

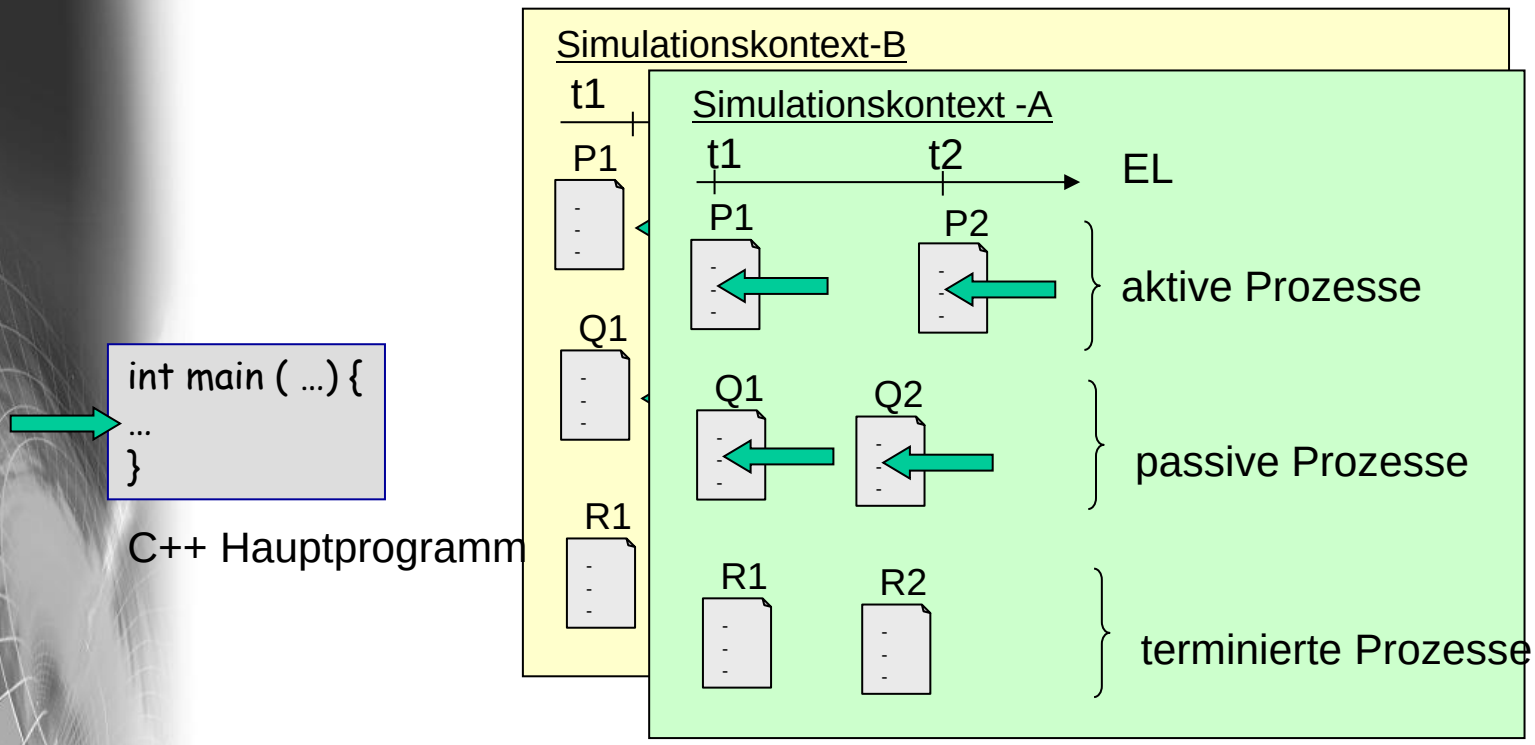
   Rückkehr ins C++ Hauptprogramm:

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

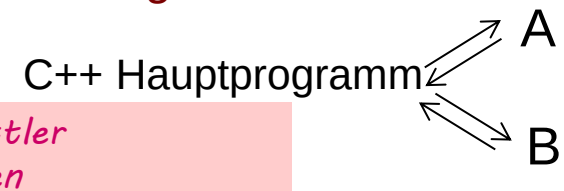
   – ***explizit***: die Simulation wurde mit exitSimulation() durch einen Prozesses des Simulationskontextes beendet

typisch für Arbeit DefaultSimulation-Kontext

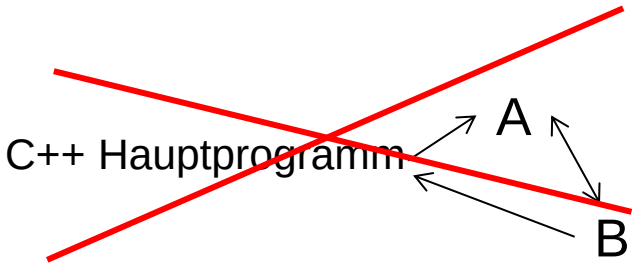
Verwaltung mehrerer Simulationskontexte



Steuerungszenarien:



als Mittler zwischen den Prozess-Systemen



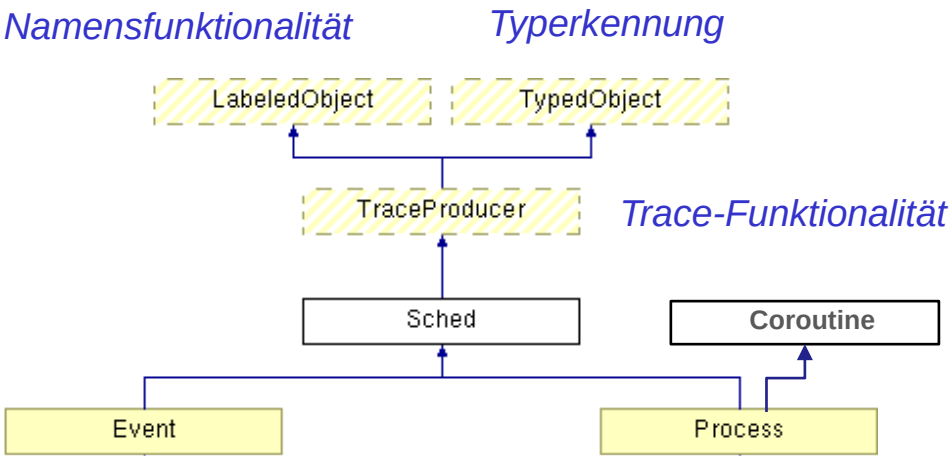
Klasse Sched, Event, Process

Sched

- abstrakte Klasse
- Sched-Objektzeiger werden im Kalender in chronologischer Reihenfolge erfasst

Simulationslauf

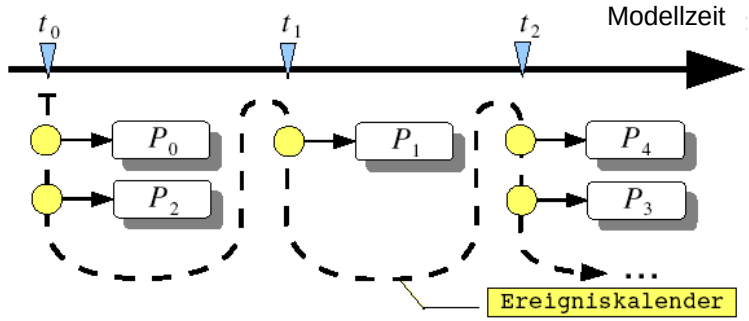
- ist die Ausführung (execute) von Sched-Objekten
- in Abhängigkeit von
  - der jeweiligen Kalenderkonstellation und
  - der Typen der Sched-Objekte



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

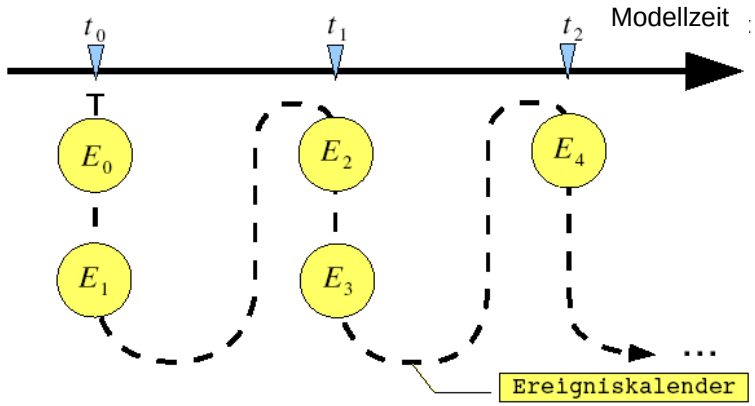
```
virtual SimTime getExecutionTime () const =0          // Get model time.
virtual SimTime setExecutionTime (SimTime time)=0  // Set model time.
virtual Priority getPriority () const =0
virtual Priority setPriority (Priority newPriority)=0    // Set new priority.
bool isScheduled () const                                 // 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]

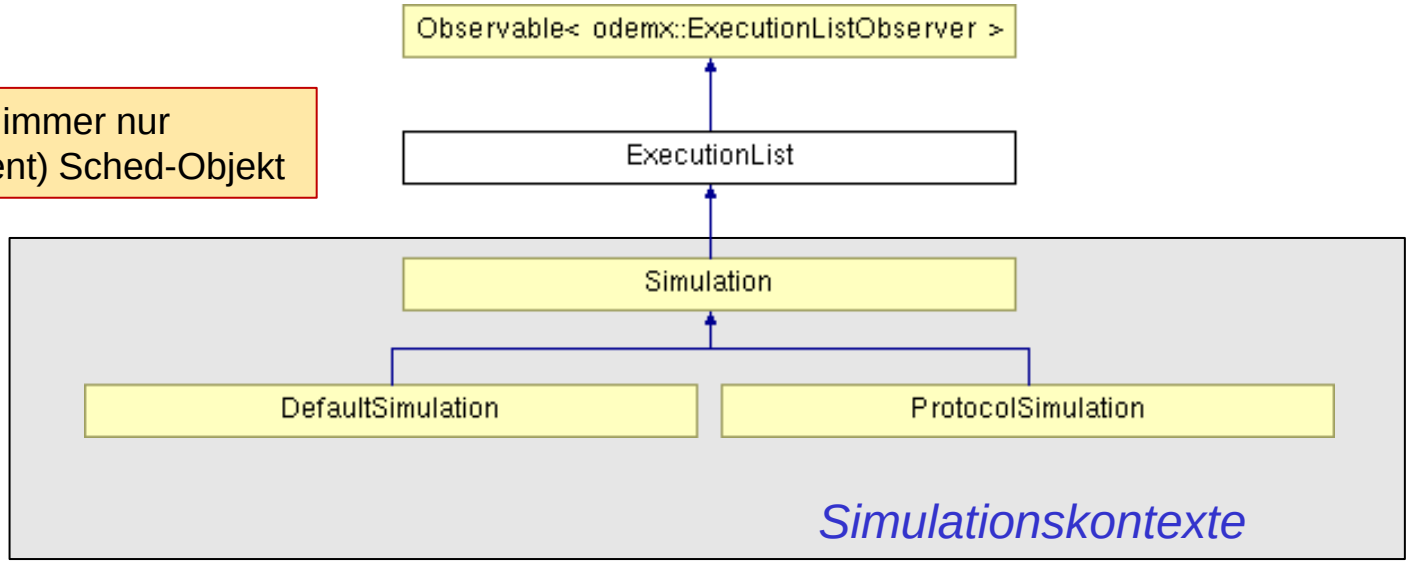
Übungsaufgabe:
Clock-Beispiel als Event-Variante

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
```

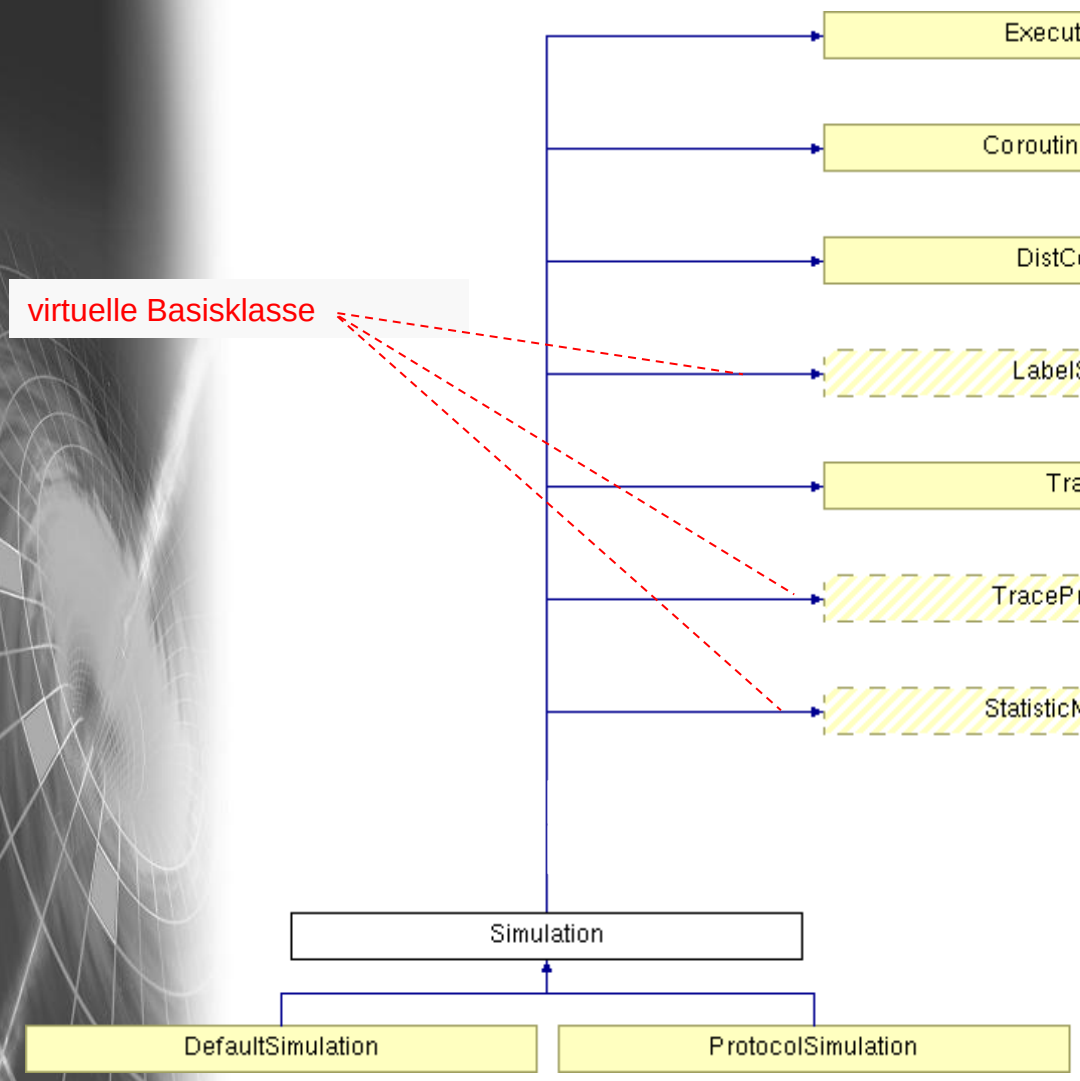
Vergangenheit wird nicht konserviert

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



jeder **Simulationskontext** (Objekt von Simulation bzw. Ableitung von Simulation) 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

3. Prozess-Scheduling

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 **seinem initialen Simulationskontext** zugeordnet
- wird während seines Lebenslaufes (in Abhängigkeit seines Grundzustandes) in vier unterschiedlichen **Listen** seines Simulationskontextes erfasst.

  Grundzustände
  - Created
  - Runable, dann auch in ExL
  - Idle
  - Terminated

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

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

Zugriffsfunktionen für Process-Listen

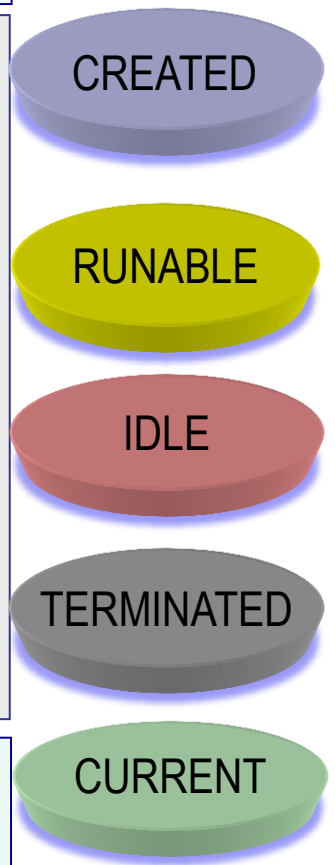
generell
Jeder Prozess wird in seinem gesamten Lebenslauf von seinem Simulationskontext verwaltet *(Zustandslisten eigentlich überflüssig)*

```
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

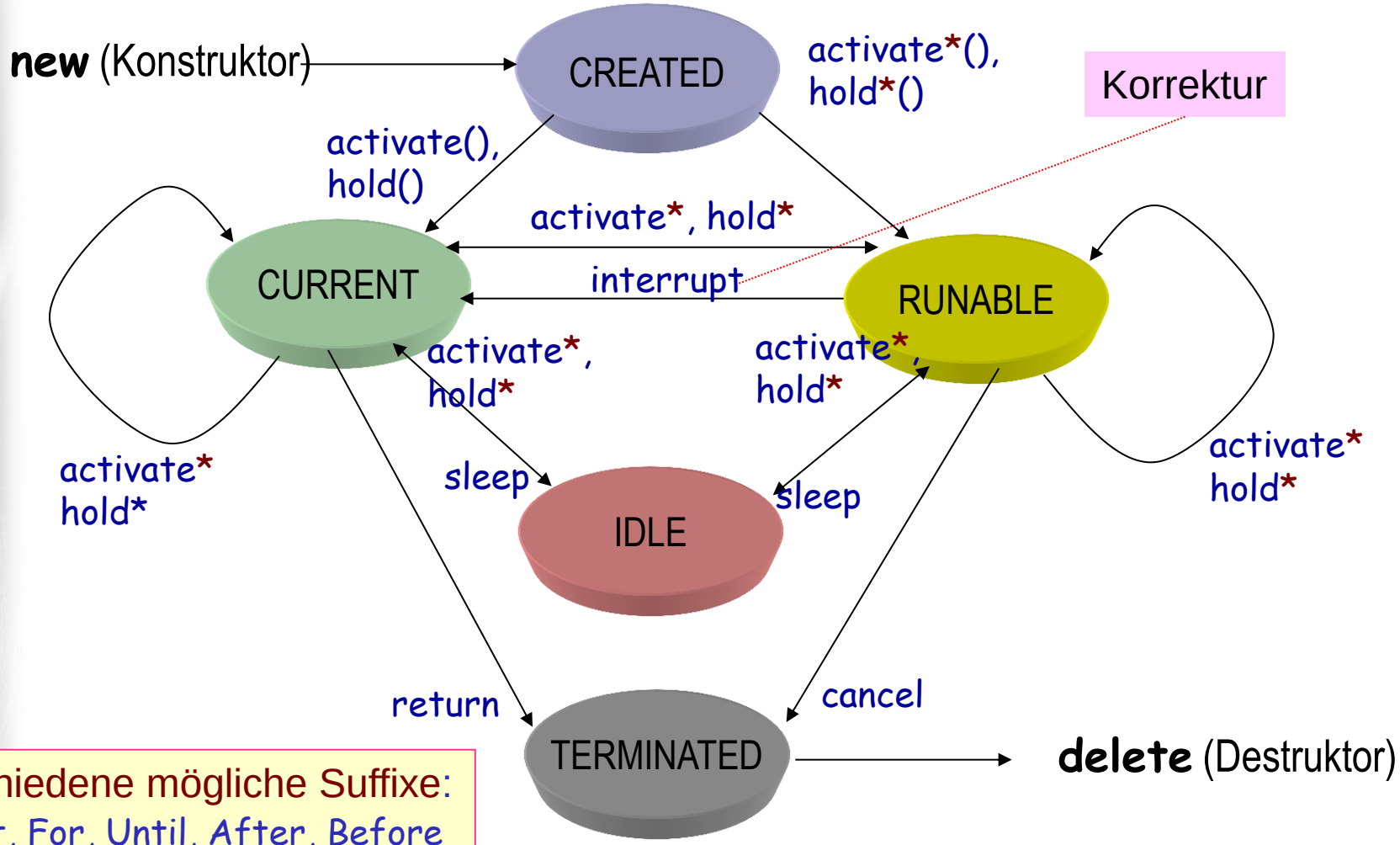
```
Process* Simulation::getCurrentProcess();
        // liefert Zeiger zum aktuellen Prozess der ExL

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

globale Funktion

3. Prozess-Scheduling

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



Process: Scheduling-Operationen (1)

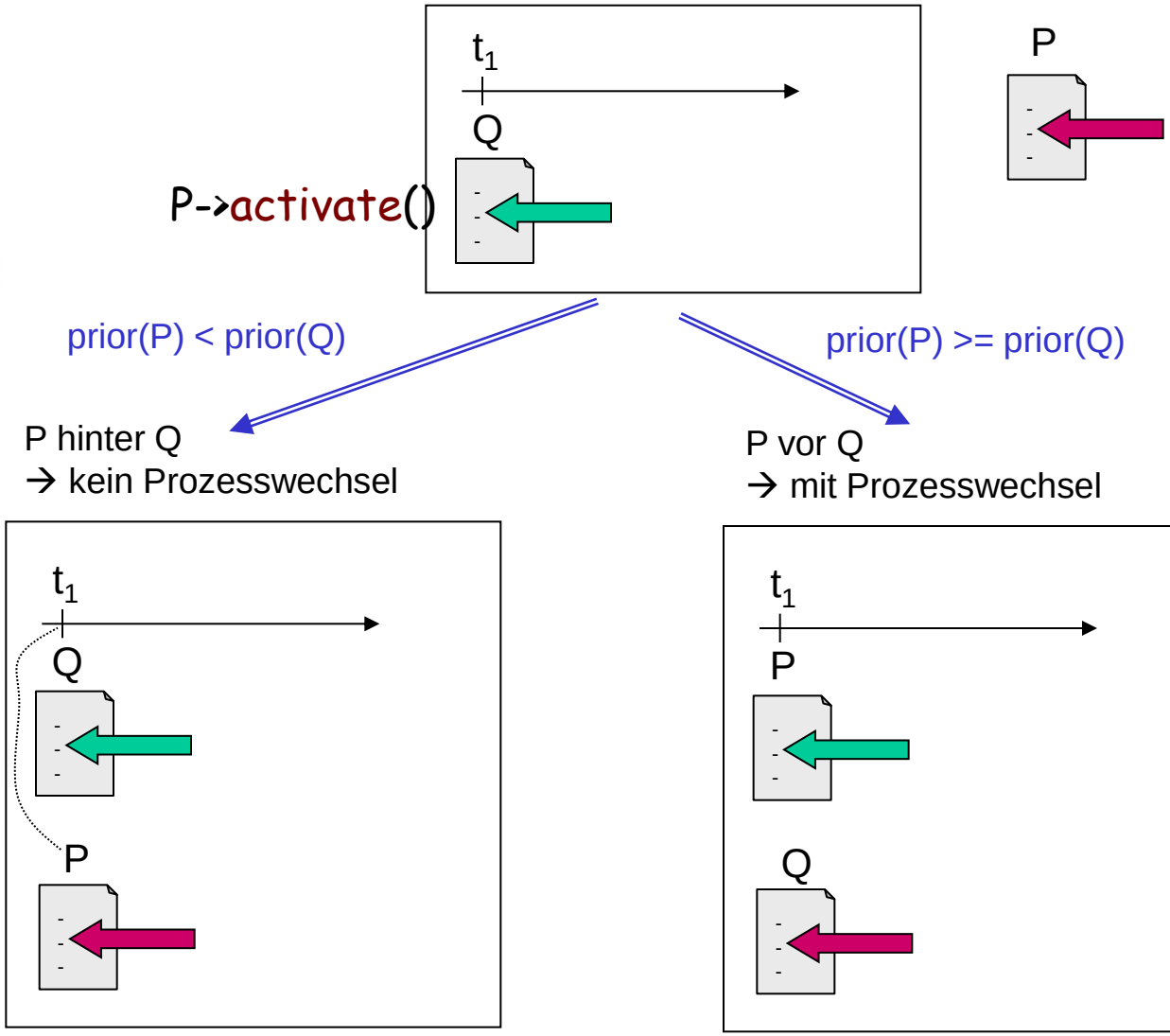
Prozessaktivierungen nach dem LIFO-Prinzip

```
void activate();      // Eintrag in ExL zur aktuellen Ereigniszeit now
                      // nach dem LIFO-Prinzip
                      // Prozesswechsel (falls kein Prioritätskonflikt)

void activateIn (SimTime t);
                      // Eintrag in ExL zur Ereigniszeit now + t
                      // nach dem LIFO-Prinzip bei Gleichzeitigkeit und
                      // Prioritätsgleichheit
                      // falls t<0.0, dann t= 0.0

void activateAt (SimTime t);
                      // Eintrag in ExL zur absoluten Ereigniszeit t
                      // nach dem LIFO-Prinzip bei Gleichzeitigkeit und
                      // Prioritätsgleichheit
                      // falls t<now, dann t= now
```

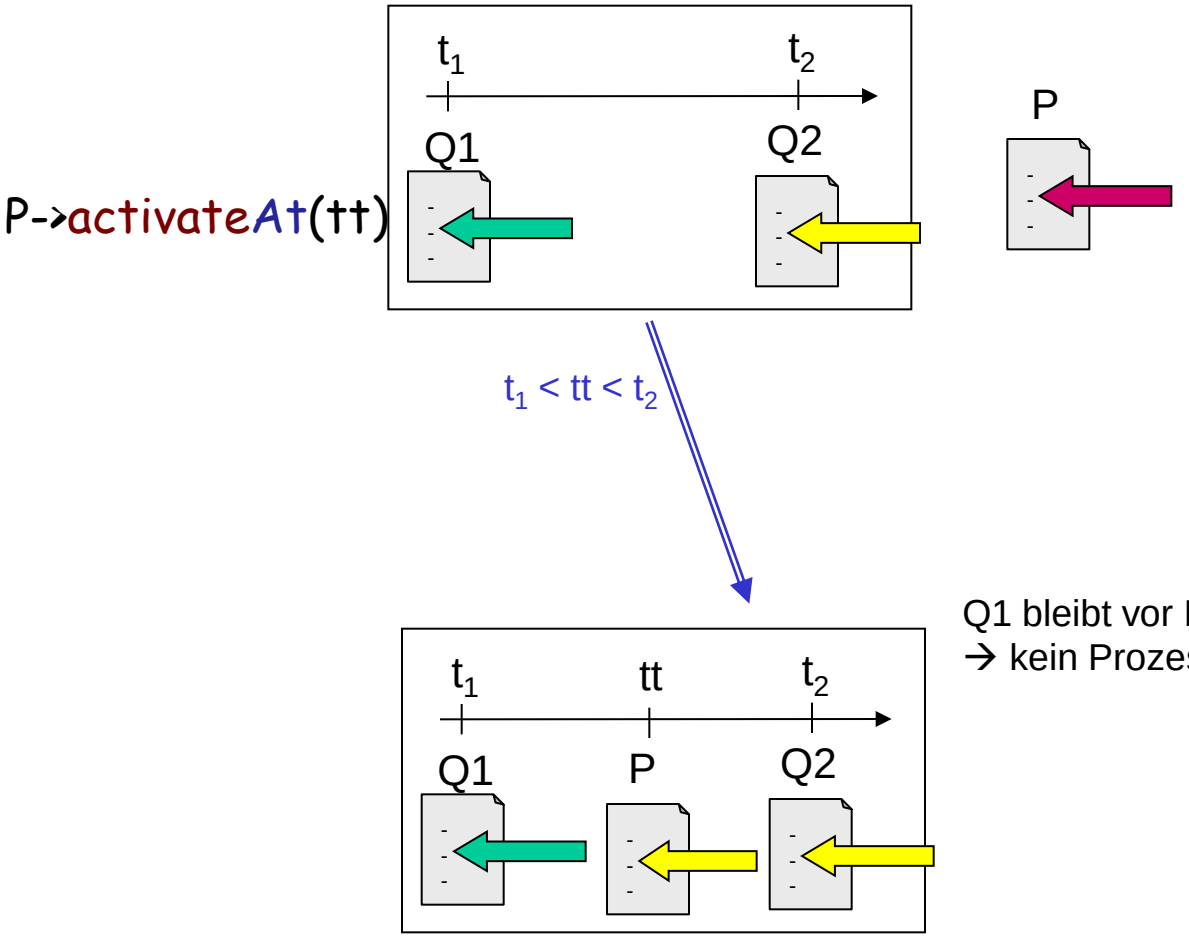
Achtung:
nur aus dem Simulationskontext heraus,
nicht bei Aktivierung aus dem Hauptprogramm

semantisch äquivalent:
P->activate() == P->activateIn(0.0) == P->activateAt(now)

Activate innerhalb eines Simulationskontextes



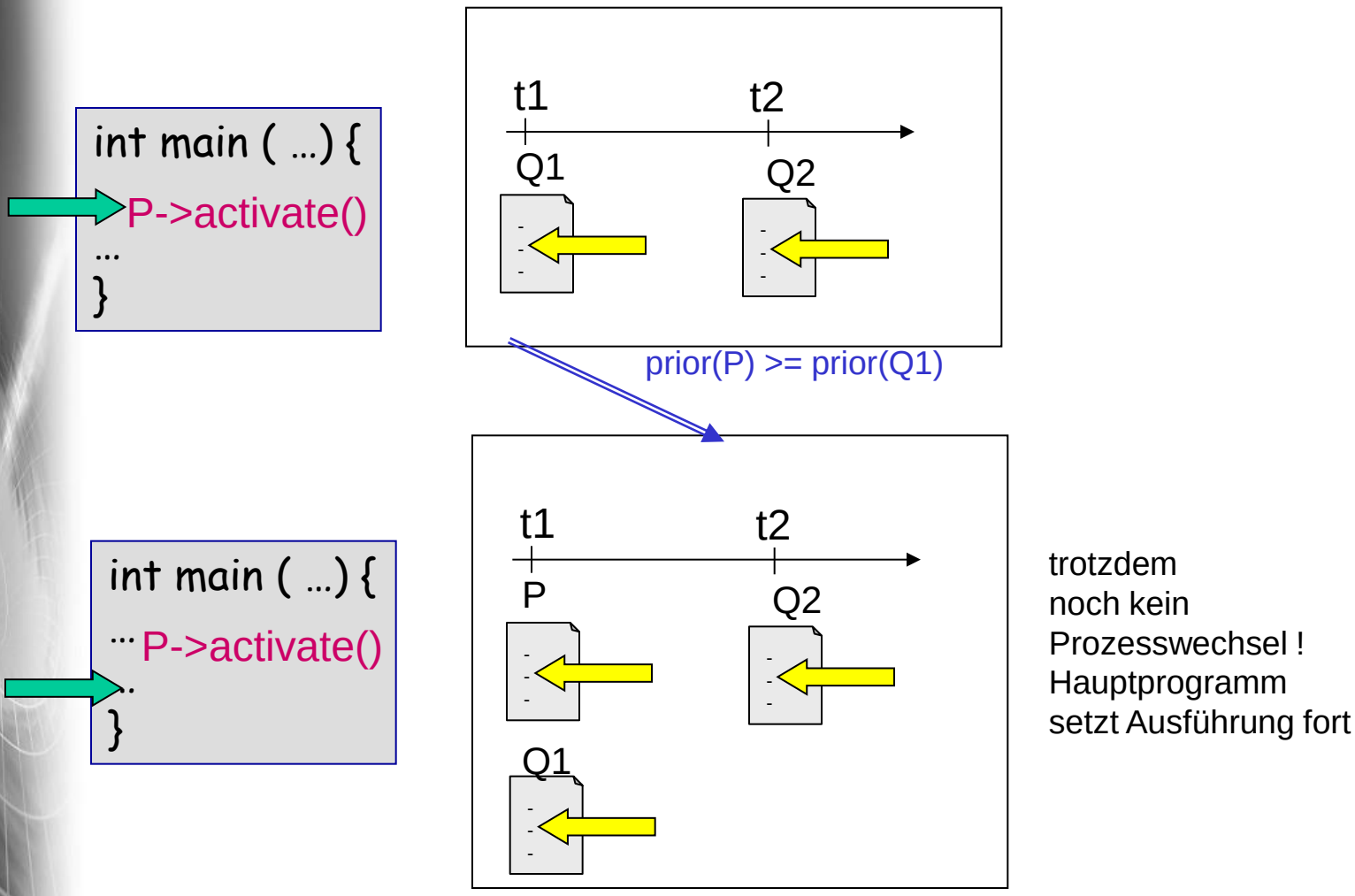
Activate innerhalb eines Simulationskontextes



Q1 bleibt vor P
→ kein Prozesswechsel

Activate außerhalb eines Simulationskontextes

Simulationskontext (DefaultSimulation-Objekt)



trotzdem
noch kein
Prozesswechsel !
Hauptprogramm
setzt Ausführung fort

Process: Scheduling-Operationen (2)

Prozessaktivierungen nach dem Vorher/- Nachherprinzip

```
void activateBefore (Process* p);
                    // unmittelbarer Eintrag vor p mit Ereigniszeit von p,
                    // ggf. Übernahme der Priorität von p
                    // falls p == this: leere Anweisung
                    // falls p nicht in der ExL: Fehlermeldung

void activateAfter (Process* p);
                    // unmittelbarer Eintrag nach p mit Ereigniszeit von
                    // p, ggf. Übernahme der Priorität von p
                    // falls p == this: leere Anweisung
                    // falls p nicht in der ExL: Fehlermeldung
```

Process: Scheduling-Operationen (3)

Prozessverzögerungen nach dem FIFO-Prinzip

```
void hold();
                    // Eintrag zur aktuellen Ereigniszeit
                    // als letzter bei gleicher oder niedrigerer Priorität
    (FIFO)

void holdFor (SimTime t);
                    // Eintrag zur Ereigniszeit now + t
                    // als letzter bei gleicher oder niedrigerer Priorität
                    // falls t<0.0, dann t= 0.0

void holdUntil (SimTime t);
                    // Eintrag zur absoluten Ereigniszeit t
                    // als letzter bei gleicher oder niedrigerer Priorität
                    // falls t<now, dann t= now
```

semantisch äquivalent:

```
p->hold() == p->holdFor(0.0) == p->holdUntil(now)
p->holdUntil(t) == p->holdFor(t-now)
```