In diesem Kapitel:

Java Spaces - Konzepte und Einfache Beispiele

1.1. JavaSpaces

Java Spaces sind neben Jini Diensten das wesentliche Element der neuen Netzwerk Konzepte in Java. *JavaSpaces*, ist ein Schlüsselkonzept, ein Schlüsseldienst von Jini, mit dem aufeiner sehr hohen Abstraktionsebene kollaborative verteilte Applikationen entwickelt werden können. JavaSpaces werden zum Beispiel eingesetzt, um die Teilnehmer einer Jini Föderation zu koordinieren.

Man kann JavaSpaces aber auch losgelöst von Jini betrachten, als Werkzeug zum Aufbau verteilter Systeme in Java.

In beiden Fällen sollten Sie JavaSpaces genauer anschauen, da es ihr Leben wesentlich vereinfachen kann.

1.1.1. Das Modell

In JavaSpaces Anwendungen kommunizieren Prozesse nicht direkt, sondern sie koordinieren ihre Aktivitäten indem sie Objekte mit Hilfe eines *Spaces,* eines gemeinsamen Speicherbereichs, austauschen

Ein Prozess kann ein neues Objekt in den Space schreiben (write()), ein Objekt aus dem Space entfernen (take()) oder lesen (read(()) . Die folgende Skizze zeigt die Grundfunktionen. Jeder Prozess wird als Duke dargestellt. Zum herausholen oder lesen eines Objekts wird ein Matching Algorithmus verwendet. Falls kein Objekt den Suchkriterien entspricht, kann der lesende Prozess warten, bis ein Objekt eintrifft, welches den Kriterien entspricht. Die Prozesse rufen keinerlei Methoden der Objekte in den Spaces auf. Spaces dienen also lediglich der Speicherung der Objekte. Um ein Objekt zu verändern, muss ein Prozess ein Objekt lesen (=kopieren) oder herausholen, modifizieren und wieder in den Space schreiben.

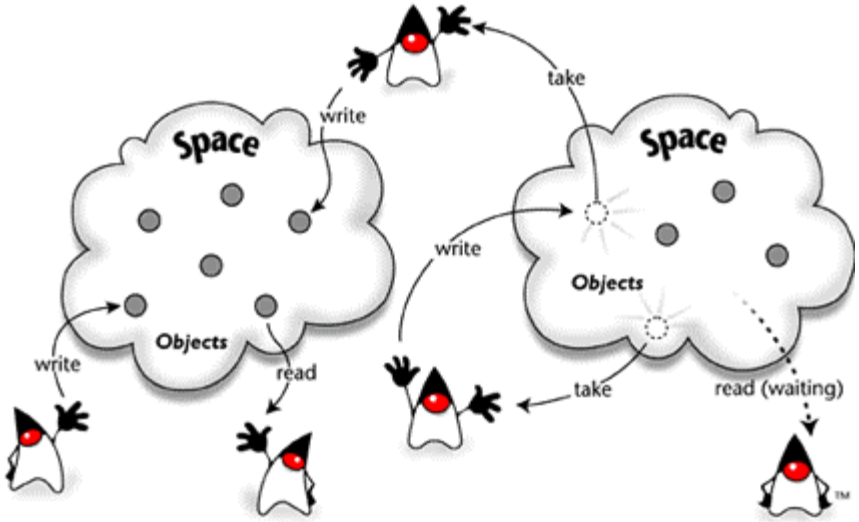


Abbildung 1. Prozesse benutzen Spaces und einfache Operationen, um Aktivitäten zu koordinieren.

Spaces sind Objektspeicher, mit mehreren wichtigen Eigenschaften, welche JavaSpaces zu einem mächtigen Werkzeug machen.

1.1.2. Eigenschaften von JavaSpaces

JavaSpaces besitzen folgende Eigenschaften:

- **Spaces werden gemeinsam genutzt:**
  viele verteilte Prozesse können nebenläufig mit einem Space kommunizieren. Spaces koordinieren die Zugriffe selbst, eine Synchronisation auf Client Ebene entfällt somit.

- **Spaces sind persistent:**
  Spaces können Objekte zuverlässig speichern. Ein ein einem Space gespeichertes Objekt kann dort beliebig lange gespeichert bleiben. Sie können auch eine Leasingzeitdauer setzen. Damit bleibt ein Objekt in diesem Space solange gespeichert, bis die Leasingdauer abgelaufen ist oder ein Prozess dieses Objekt entfernt.

- **Spaces sind assoziativ:**
  Objekte in einem Space werden mit Hilfe eines *assoziativen Suchalgorithmus* gesucht, entweder aufgrund der Speicherlokaltion oder eines Namens. Um Objekte zu finden, wird ein *Template* verwendet (ein Objekt mit Datenfeldern, die spezifische Werte annehmen können oder lehrgelassen werden). Ein Objekt in einem Space entspricht den Suchkriterien eines Templates, falls die im Template angegebenen Felder mit jenen des Objekts übereinstimmen.

- **Spaces sind transaktionell sicher:**
  JavaSpaces verwendet den Jini Transaktionsdienst, um garantieren zu können, dass die Operationen auf den Spaces atomar sicher sind. Transaktionen werden sowohl für einfache als auch verteilte Operationen definiert.

- **Spaces erlauben den Austausch ausführbarer Anweisungen,von Objekten:**
  In einem Space selbst sind die Objekte lediglich passiv - die Methoden des Objekts können nicht im Space aufgerufen werden. Beim Lesen oder herausholen eines Objekts aus dem Space, wird eine lokale Kopie angelegt. Sobald das Objekt lokal ist, kann es modifiziert oder erweitert werden. Damit haben wir einen effizienten Mechanismus, bestehende Applikationen zu erweitern, sofern sie auf JavaSpaces basieren.

Die obigen Eigenschaften haben zur Folge, dass JavaSpace Applikationen mächtig sind und dynamisch rekonfiguriert werden können, zudem sind sie relativ leicht fehlertolerant zu realisieren.

1.1.3. Woher stammt JavaSpaces ?

Wir haben JavaSpaces als neues Konzept für verteilte Systeme hingestellt. Aber die Konzepte hinter JacaSpaces gehen zurück auf Arbeiten an der Yale University um 1980s. Dort hat Dr. David Gelernter ein Werkzeug *Linda* entwickelt, um verteilte Applikationen zu entwickeln. Linda besteht aus einer kleinen Menge Operationen kombiniert mit persistenter Speicherung. Die Speicherung wurde *tuple space* genannt. Die Operationen waren "orthogonal" zu den Operationen der üblichen Anweisungen in Programmiersprachen; sie sind Teil der *coordination language*, die zu einer *computation languag* hinzugefügt werden kann.Das Ergebnis des Forschungsprojektes war erstaunlich: mit Hilfe weniger einfacher Operationen und einem Objektspeicher konnten komplexe Probleme gelöst werden, ohne einige der Probleme traditioneller Systeme. Mit andern Worten : einfache Operationen gestatten die Konstruktion komplexer Systeme.

Dr. Gelernter's Arbeit inspirierte Sun's JavaSpaces Service die Designs des Lookup und Discovery Dienstes. JavaSpaces erweitern das Tupel-Modell von Gelernter und sind ausschliesslich Java basiert, sprich Java Objekte, Jini RMI und Objekt Serialisierung.

1.1.4. Mögliche JavaSpaces Anwendungen

Unsere Beschreibung ist bisher recht abstrakt. Daher werden wir jetzt einige Beispiele betrachten.

1.1.4.1. Chat Systeme

Betrachteb wir zum Beispiel ein Multiuser Chat System, in welchem ein Space als Chatraum dient, ein Raum also, der die Meldungen aufbewahrt. Um zu sprechen liefert ein Benutzer eine Nachricht an das Space ab. Alle Chat Teilnehmer warten auf neue Nachrichten, lesen sie und zeigen die Nachrichten in den jeweiligen Frontendsystemen an.Ein Teilnehmer, der sich später in die Diskussion einschalten möchte, kann sich rasch die gespeicherten Meldungen ansehen und anschliessend aktiv an der Diskussion teilnehmen.Auch die Liste der Teilnehmer kann gespeichert werden und im Space aufbewahrt werden.

1.1.4.2. Rechen Server

Ein völlig anderes Beispiel wäre ein Echtzeit Radio Teleskop, welches Daten über extraterrestrisches Leben sammelt und analysiert, wie bei Seti@home. Das Sammeln und analysieren der Daten ist sehr zeitaufwendig und kann am besten verteilt, mit vielen Rechnern gleichzeitig getan werden. Was wir benötigen ist ein Speicherraum, auf den viele Prozesse zugreifen können und die dann parallel an den Daten arbeiten können. Mit JavaSpaces könnte also ein solches System realisiert werden. Auch ein *load balancing,* wäre möglich, da jeder Arbeiter, jeder Prozess aus dem JacaSpace genau soviele Daten liest, wie er auch verarbeiten kann.

1.1.4.3. Broker Systeme

Als drittes Beispiel könnten wir ein Broker System realisieren, welches Ware anbietet und Gebote dafür entgegen nimmt. Das System soll Anbieter und potentielle Käufer zusammen bringen. Die Anwendung ist recht naheliegend und kann natürlich auch mit andern Technologien realisiert werden..

1.2. Kurze Übersicht über das API

Bevor wir über das JavaSpaces Interface sprechen, müssen wir den Begriff Entry klären.

1.2.1. Entries

Ein Objekt, welches in einem Space gespeichert wird, bezeichnet man als *entry*. Um ein Entry zu sein, muss das Objekt das `Entry` Interface implementieren.

Zum Beispiel:

```
import net.jini.core.entry.Entry;
public class Message implements Entry {
    public String content;
    // ein Konstruktor ohne Argumente
    public Message() {
    }
}
```

Die Message Klasse besitzt ein String Datenfeld, welches eine Meldung aufnehmen kann. Wie müssen `net.jini.core.entry.Entry`, aus dem Package `net.jini.core.entry` implementieren. `Entry` ist ein *marker interface;* **mit andern Worten, das Interface enthält keine Konstanten oder Methoden und ist daher einfach zu implementieren: wir benötigen nur den Zusatz `implements Entry` zu unserer Klassendefinition hinzuzufügen**.

1.2.1.1. Das Interface Entry

net.jini.core.entry
Interface Entry

public interface **Entry**
extends java.io.Serializable (deswegen muss ein Default Konstruktor definiert werden)

Diese Klasse ist die Oberklasse, der Supertyp aller Entries, welche in einem Jini Lookup Service gespeichert werden können. Jedes Feld einer Entry muss public sein. Man kann keine Basisdatentypen in einer Entry abspeichern. Eine Entry kann eine beliebige Anzahl Konstruktoren haben. Jedes Datenfeld einer Entry wird separat serialisiert werden. Damit können zwei in einer Entry zusammengehörige Datenfelder nicht als zusammengehörig erkannt werden, beide werden als unabhängige Teile angesehen.

1.2.2. Das JavaSpace interface

Jede Klasse, die das JavaSpace Interface implementiert, muss folgende Methoden umfassen:

- **`write`:**
  ein Objekt wird in ein JacaSpace eingefügt. Falls die Methode mehrfach mit dem selben Objekt aufgerufen wird, werden mehrere Kopien des Objekts eingefügt.

- **`read`:**
  liest einen Eintrag aus dem JavaSpace und erstellt eine Kopie. Der Benutzer kann auch angeben, dass notfalls eine bestimmte Zeitdauer warten muss, um ein eventuell noch eintreffendes Objekt auch noch mitzukriegen.

- **`take`:**
  funktioniert wie `read`, wobei aber das Objekt, weches den Auswahlkriterien entspricht, aus dem Space entfernt wird.

Zusätzlich stellt das `JavaSpace` Interface zur Verfügung, die in vielen Anwendungen hilfreich sein können:

- **`notify`:**
  wann immer ein Objekt, welches dem Template entspricht, ins Space eingefügt wird, wird Space den Abonnenten informieren. Dieser Mechanismus ist Teil des Jini Distributed Event Modells.
- **`snapshot`:**
  erlaubt Optimierungen bei der Serialisierung.

Dies ist mehr oder weniger alles, was zum JavaSpace API zu sagen ist.

1.2.2.1. net.jini.space Interface JavaSpace

net.jini.space
Interface JavaSpace

public interface **JavaSpace**

Dieses Interface wird von Servern implementiert, welche JavaSpaces Services anbieten.
Die Operationen / Methoden des Interfaces sind die einzigen, die ein JavaSpace unterstützt.

Datenfelder	
`static long NO_WAIT`	wartet nicht (Timeouts können also eintreten)

Methoden

`EventRegistration notify(Entry tmpl, Transaction txn, RemoteEventListener listener, long lease, java.rmi.MarshalledObject handback)`

Falls eine `Entry` in den Space geschrieben wird, auf welche dieses Template zutrifft soll der Listener mit einem `RemoteEvent`, welches das `handback` Objekt enthält, informiert werden.

`Entry read(Entry tmpl, Transaction txn, long timeout)`

Liest Entries aus dem Space, welche dem Template entsprechen und blockiere bis eines vorliegt (warte also bis eine solche Entry im Space drin ist).

`Entry readIfExists(Entry tmpl, Transaction txn, long timeout)`

Lies Entries aus dem Space, welche dem Template entsprechen, sofern welche vorhanden sind, sonst wird `null` zurückgegeben.

`Entry snapshot(Entry e)`

Liefert einen Snapshot einer Entry. Der Zustand des Snapshots bleibt der selbe wie vom Original, selbst wenn sich dieses verändert.

`Entry take(Entry tmpl, Transaction txn, long timeout)`

Entnimmt dem Space eine Entry und wartet notfalls bis eine vorhanden ist.

`Entry takeIfExists(Entry tmpl, Transaction txn, long timeout)`

Entnimmt dem Space eine Entry, sofern eine vorhanden ist. Sonst wird ein null Element zurück geliefert.

`Lease write(Entry entry, Transaction txn, long lease)`

Schreibt eine neue Entry in den Space.

1.3. Illustration des Konzeptes

1.3.1. Grundlegende Operationen

Die folgenden Beispiele finden Sie auf der CD / dem Server. Sie zeigen Ihnen, wie mit JavaSpaces gearbeitet, programmiert werden kann. JavaSpaces sind Objekt Registries / Depositories mit einigen zusätzlichen Funktionen / Methoden, welche sei recht universell einsetzbar machen.

1.3.1.1. 'Hello World,' JavaSpaces Style

Wir betrachten zwei Versionen:

1) eine Version, welche einfach eine Meldung / Message in den Space schreibt und anschliessend wieder liest
2) eine zweite Variante mit einem Client, welche die Messages aus dem Space liest, die ein anderer Prozess hineinschreibt.

```
package javaspacehelloworld;

import net.jini.core.entry.Entry;

public class Message implements Entry {
    public String content;

    public Message() {
    }
}
```

```
package javaspacehelloworld;

import net.jini.core.lease.Lease;
import net.jini.space.JavaSpace;
import java.util.*;

public class HelloWorld {
    public static void main(String[] args) {
        String msgs[] = {"Wie geht's Dir heute?",
                         "Mir geht's gut! Wie geht's Dir?",
                         "Mir geht's beschissen!",
                         "Ich gehe gleich in die Badi",
                         "Meine Freundin hat mich verlassen!",
                  "Mein Freund hat mich verlassen, er liebt eine andere!"};
        Random r = new Random();// zufällige Auswahl eines Textes
        try {
            JavaSpace space = SpaceAccessor.getSpace();
            System.out.println("[Main]Start");
            Message msg = new Message();
            for (int j=0;j<20; j++) {// 20 Meldungen schreiben+lesen
              msg.content = msgs[r.nextInt(5)];
              space.write(msg, null, Lease.ANY);
              Message template = new Message();
              Message result =
                (Message)space.take(template, null, Long.MAX_VALUE);
              System.out.println("[main]Message Take: "+result.content);
            }
        } catch (Exception e) { e.printStackTrace();     }
    }
}
```

Die zweite Version benötigt zusätzlich einen Client (der Server schreibt und liest). Zudem wird die Message mit einem Zähler versehen, sodass festgestellt werden kann, wie oft sie gelesen wurde.

```
package javaspaceshelloworld;
import net.jini.core.entry.Entry;

public class Message implements Entry {
    public String content;
    public Integer counter;
    public Message() {// Default Konstruktor
    }
    public Message(String content, int initVal) {
        this.content = content;
        counter = new Integer(initVal);
    }
    public String toString() {
        return "Die Message "+content+" wurde "+counter+" Male gelesen.";
    }
    public void increment() {
        counter = new Integer(counter.intValue() + 1);
    }
}
```

Der Message produzent bleibt im Wesentlichen unverändert:

```
package javaspaceshelloworld;

import net.jini.core.lease.Lease;
import net.jini.space.JavaSpace;
import java.util.*;

public class HelloWorld {
    public static void main(String[] args) {
        String msgs[] = {"Wie geht's Dir heute?",
                         "Mir geht's gut! Wie geht's Dir?",
                         "Mir geht's beschissen!",
                         "Ich gehe gleich in die Badi",
                         "Meine Freundin hat mich verlassen!",
                   "Mein Freund hat mich verlassen, er liebt eine andere!"};
        Random r = new Random();
        try {
            Message msg = new Message("Ich bin's....", 0);

            JavaSpace space = SpaceAccessor.getSpace();
            space.write(msg, null, Lease.ANY);

            for (;;) {
              msg.content = msgs[r.nextInt(5)];
              System.out.println("[Produzent]HelloWorld - "+msg.content);
              space.write(msg, null, Lease.ANY);

              Message template = new Message();
              Message result = (Message)
                    space.take(template, null, Long.MAX_VALUE);
              System.out.println(result);
              Thread.sleep(1000);
            }
        } catch (Exception e) {           e.printStackTrace();       }
    }
}
```

Der Client ist denkbar einfach. Sie können problemlos gleich mehrere Clients starten: die Zugriffsmethoden für JavaSpaces sind synchronisiert!

```
package javaspaceshelloworld;

import net.jini.core.lease.Lease;
import net.jini.space.JavaSpace;

public class HelloWorldClient {
    public static void main(String[] args) {
        try {
            JavaSpace space = SpaceAccessor.getSpace();

            Message template = new Message();
            for (;;) {
                Message result = (Message)
                    space.take(template, null, Long.MAX_VALUE);
                result.increment();
                space.write(result, null, Lease.FOREVER);
                Thread.sleep(1000);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

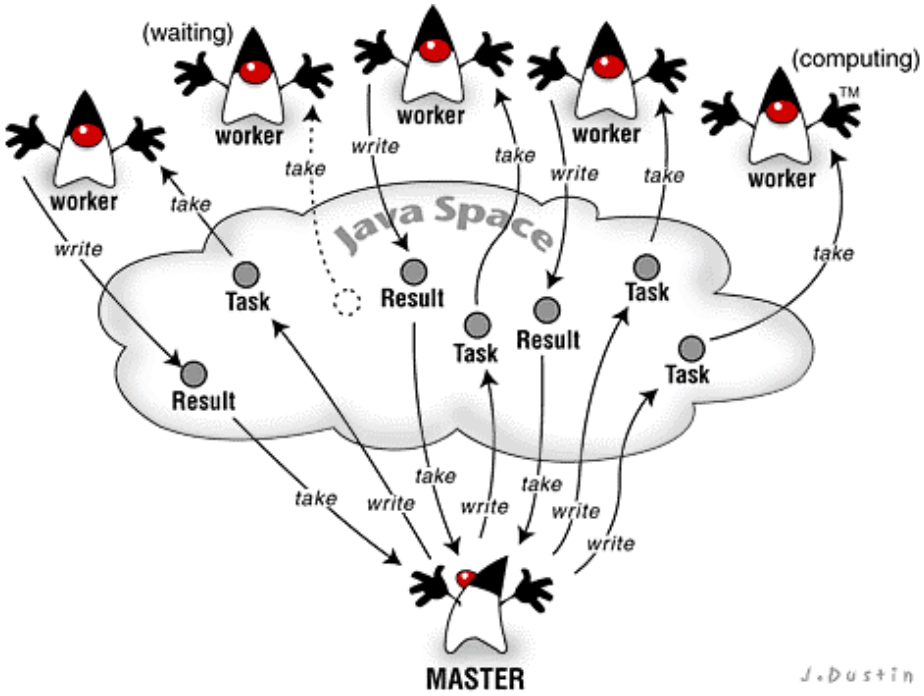
Sie können problemlos Varianten dieser beiden Beispiele konstruieren, welche komplexeres Verhalten zeigen.

Alle Batch Prozeduren zum Starten der Dienste und Programme finden Sie auf dem Server / der CD. Ohne diese werden Sie wahrscheinlich grössere Probleme haben, die Beispiele zum Laufen zu bringen.....

1.3.1.2. Compute Server Design

Ein Master-Prozess generiert eine Anzahl Tasks und schreibt diese in ein JavaSpace. Jedes Objekt verfügt über Methoden Berechnungen duchführen zu können. Worker Prozesse beobachten den Space und entnehmen Aufgaben sobald diese anfallen, also im Space abgelegt werden.

.



Ein Space-basierter Compute Server
(Illustration by James P. Dustin, Dustin Design)

Die Kommunikation zwischen Master und Worker erfolge aufgrund folgender einfachen Befehlsstruktur.

```
package javaspacesverteilterrechner;

import net.jini.core.entry.Entry;

/**
 * Title:        Distributed Computing
 * Description:  einfacher Rechner, welcher JavaSpaces als
Kommunikationsplattform, Objektrepository für die Kommunikation zwischen
Clients und dem Master / Server verwendet.
 * Copyright:    Copyright (c) J.M.Joller
 * Company:      Joller-Voss
 * @author J.M.Joller
 * @version 1.0
 */

public interface Command extends Entry {
    public Entry execute();
}
```

1.3.1.2.1. Der Worker

```
package javaspacesverteilterrechner;

import net.jini.core.entry.Entry;
import net.jini.core.lease.Lease;
import net.jini.space.JavaSpace;

/**
 * Title:        Distributed Computing
 * Description:  einfacher Rechner, welcher JavaSpaces als
Kommunikationsplattform, Objektrepository für die Kommunikation zwischen
Clients und dem Master / Server verwendet.
 * Copyright:    Copyright (c) J.M.Joller
 * Company:      Joller-Voss
 * @author J.M.Joller
 * @version 1.0
 */

public class Worker {
    JavaSpace space;

    public static void main(String[] args) {
        System.out.println("[Worker]main() Start");
        Worker worker = new Worker();
        worker.startWork();
    }

    public Worker() {
        space = SpaceAccessor.getSpace();
    }

    public void startWork() {
        System.out.println("[Master]startWork()");
        TaskEntry template = new TaskEntry();

        for (;;) {
            try {
                TaskEntry task = (TaskEntry)
                    space.take(template, null, Long.MAX_VALUE);
                Entry result = task.execute();
                if (result != null) {
                    space.write(result, null, 1000*60*10);
                }
            } catch (Exception e) {
                System.out.println("Task wurde gecancelled");
            }
        }
    }
}
```

1.3.1.2.2. Der Master Hilfsklassen

1.3.1.2.2.1. Tasks

```
package javaspacesverteilterrechner;
import net.jini.core.entry.Entry;

public class TaskEntry implements Command {
    public Entry execute() {
        throw new RuntimeException(
            "TaskEntry.execute() ist nicht implementiert.");
    }
}

package javaspacesverteilterrechner;
import net.jini.core.entry.Entry;

public class AddTask extends TaskEntry { // Addition
    public Integer a;
    public Integer b;

    public AddTask() {}

    public AddTask(Integer a, Integer b) {
        this.a = a;
        this.b = b;
    }

    public Entry execute() {
        Integer answer = new Integer(a.intValue() +
            b.intValue());
        AddResult result = new AddResult(a, b, answer);
        System.out.println("[TaskEntry<-AddTask]execute()");
        return result;
    }
}

package javaspacesverteilterrechner;
import net.jini.core.entry.Entry;

public class MultTask extends TaskEntry {// Multiplikation
    public Integer a;
    public Integer b;

    public MultTask() {}

    public MultTask(Integer a, Integer b) {
        this.a = a;
        this.b = b;
    }

    public Entry execute() {
        Integer answer = new Integer(a.intValue() *
                b.intValue());
        MultResult result = new MultResult(a, b, answer);
        System.out.println("[TaskEntry<-MultTask]execute()");
        return result;
    }
}
```

1.3.1.2.2.2. Results

```
package javaspacesverteilterrechner;
import net.jini.core.entry.Entry;

public class ResultEntry implements Entry {
    public Integer a;
    public Integer b;
    public Integer answer;

    public ResultEntry() {}
}


package javaspacesverteilterrechner;
public class AddResult extends ResultEntry {

    public AddResult() {}

    public AddResult(Integer a, Integer b, Integer answer) {
        System.out.println("[ResultEntry<-AddResult]AddResult: "+a+" +
                                              " +b+" Antwort: "+answer);
        this.a = a;
        this.b = b;
        this.answer = answer;
    }

    public String toString() {
        return a + " plus " + b + " = " + answer;
    }
}

package javaspacesverteilterrechner;
public class MultResult extends ResultEntry {

    public MultResult() {}

    public MultResult(Integer a, Integer b, Integer answer) {
        this.a = a;
        this.b = b;
        this.answer = answer;
        System.out.println("[ResultEntry<-MultiResult]MultiResult: "+a+
                                             " * " +b+" Antwort: "+answer);
    }

    public String toString() {
        return a + " mal " + b + " = " + answer;
    }
}
```

1.3.1.2.3. Master

```
package javaspacesverteilterrechner;

import net.jini.core.entry.*;
import net.jini.core.lease.Lease;
import net.jini.space.JavaSpace;

public class Master {
    private JavaSpace space;

    public static void main(String[] args) {
        System.out.println("[Master]main() Start");
        Master master = new Master();
        master.startComputing();
    }

    private void startComputing() {
        System.out.println("[Master]startComputing()");
        space = SpaceAccessor.getSpace();

        generateTasks();
        collectResults();
    }

    private void generateTasks() {
        System.out.println("[Master]generateTask()");
        for (int i = 0; i < 10; i++) {
            writeTask(new AddTask(new Integer(i), new Integer(i)));
        }
        for (int i = 0; i < 10; i++) {
            writeTask(new MultTask(new Integer(i), new Integer(i)));
        }
    }

    private void collectResults() {
        System.out.println("[Master]collectResults()");
        for (int i = 0; i < 20; i++) {
            ResultEntry result = takeResult();
            if (result != null) {
                System.out.println(result);
            }
        }
    }

    private void writeTask(Command task) {
        System.out.println("[Master]writeTask");
        try {
            space.write(task, null, Lease.FOREVER);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

```
    protected ResultEntry takeResult() {
        System.out.println("[Master]takeResult()");
        ResultEntry template = new ResultEntry();
        try {
            ResultEntry result = (ResultEntry)
                space.take(template, null, Long.MAX_VALUE);
            return result;
        } catch (Exception e) { e.printStackTrace();  return null;
        }
    }
}
```

1.3.1.2.4. Beispielausgabe

Serverseitig:

```
[Master]main() Start
[Master]startComputing()
found JavaSpaces = com.sun.jini.outrigger.SpaceProxy@0
[Master]generateTask()
[Master]writeTask
[Master]writeTask
[Master]writeTask
[Master]writeTask
[Master]writeTask
[Master]writeTask
[Master]writeTask
[Master]writeTask
[Master]writeTask
[Master]writeTask
[Master]writeTask
[Master]writeTask
[Master]writeTask
[Master]writeTask
[Master]writeTask
[Master]writeTask
[Master]writeTask
[Master]writeTask
[Master]writeTask
[Master]writeTask
[Master]collectResults()
[Master]takeResult()
0 mal 0 = 0
[Master]takeResult()
1 mal 1 = 1
[Master]takeResult()
2 mal 2 = 4
[Master]takeResult()
3 mal 3 = 9
[Master]takeResult()
4 mal 4 = 16
[Master]takeResult()
5 mal 5 = 25
[Master]takeResult()
6 mal 6 = 36
[Master]takeResult()
7 mal 7 = 49
[Master]takeResult()
8 mal 8 = 64
[Master]takeResult()
1 plus 1 = 2
```

```
[Master]takeResult()
2 plus 2 = 4
[Master]takeResult()
3 plus 3 = 6
[Master]takeResult()
4 plus 4 = 8
[Master]takeResult()
5 plus 5 = 10
[Master]takeResult()
9 mal 9 = 81
[Master]takeResult()
6 plus 6 = 12
[Master]takeResult()
7 plus 7 = 14
[Master]takeResult()
8 plus 8 = 16
[Master]takeResult()
9 plus 9 = 18
[Master]takeResult()
```

Clientseitig, je nachdem wieviele Clients gestartet werden:

```
[ResultEntry<-MultiResult]MultiResult: 0 * 0 Antwort: 0
[TaskEntry<-MultTask]execute()
[ResultEntry<-MultiResult]MultiResult: 1 * 1 Antwort: 1
[TaskEntry<-MultTask]execute()
[ResultEntry<-MultiResult]MultiResult: 2 * 2 Antwort: 4
[TaskEntry<-MultTask]execute()
[ResultEntry<-MultiResult]MultiResult: 3 * 3 Antwort: 9
[TaskEntry<-MultTask]execute()
[ResultEntry<-MultiResult]MultiResult: 4 * 4 Antwort: 16
[TaskEntry<-MultTask]execute()
[ResultEntry<-MultiResult]MultiResult: 5 * 5 Antwort: 25
[TaskEntry<-MultTask]execute()
[ResultEntry<-MultiResult]MultiResult: 6 * 6 Antwort: 36
[TaskEntry<-MultTask]execute()
[ResultEntry<-MultiResult]MultiResult: 7 * 7 Antwort: 49
[TaskEntry<-MultTask]execute()
[ResultEntry<-MultiResult]MultiResult: 8 * 8 Antwort: 64
[TaskEntry<-MultTask]execute()
[ResultEntry<-MultiResult]MultiResult: 9 * 9 Antwort: 81
[TaskEntry<-MultTask]execute()
[ResultEntry<-AddResult]AddResult: 1 + 1 Antwort: 2
[TaskEntry<-AddTask]execute()
[ResultEntry<-AddResult]AddResult: 2 + 2 Antwort: 4
[TaskEntry<-AddTask]execute()
[ResultEntry<-AddResult]AddResult: 3 + 3 Antwort: 6
[TaskEntry<-AddTask]execute()
[ResultEntry<-AddResult]AddResult: 4 + 4 Antwort: 8
[TaskEntry<-AddTask]execute()
[ResultEntry<-AddResult]AddResult: 5 + 5 Antwort: 10
[TaskEntry<-AddTask]execute()
[ResultEntry<-AddResult]AddResult: 6 + 6 Antwort: 12
[TaskEntry<-AddTask]execute()
[ResultEntry<-AddResult]AddResult: 7 + 7 Antwort: 14
[TaskEntry<-AddTask]execute()
[ResultEntry<-AddResult]AddResult: 8 + 8 Antwort: 16
[TaskEntry<-AddTask]execute()
[ResultEntry<-AddResult]AddResult: 9 + 9 Antwort: 18
```