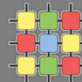
Architecture of software systems

Course 14: Memory performance recommendations, references, Intelligent systems, Multi-agent systems

David Šišlák
david.sislak@fel.cvut.cz

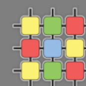
Performance recommendations



- prefer short-lived immutable objects instead of long-lived mutable objects
- avoid needless allocations
  - more frequent allocations will cause more frequent GCs
- large objects:
  - expensive to allocate (not in TLAB, not in young)
  - expensive to initialize (zeroing)
  - can cause performance issues
  - fragmentation for CMS (non-compacting) GC
- avoid frequent array-based re-sizing
  - several allocations
  - a lot of array copying
  - use:

```
ArrayList<String> list = new ArrayList<String>(1024);
```

Performance recommendations

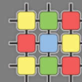


» objects in the wrong scope

```
class Foo {
    private String[] names;
    public void doIt(int length) {
        if (names == null || names.length < length)
            names = new String[length];
        populate(names);
        print(names);
    }
}
```

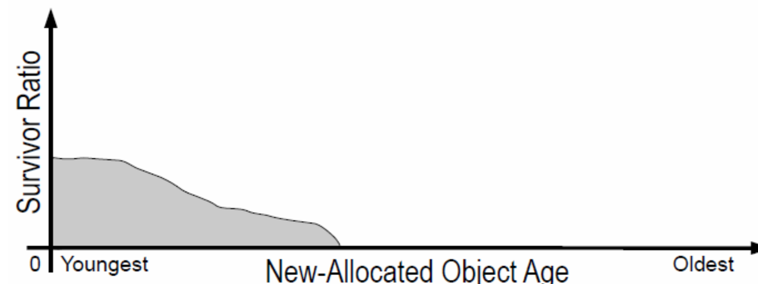
```
class Foo {
    public void doIt(int length) {
        String[] names = new String[length];
        populate(names);
        print(names);
    }
}
```

Performance recommendations



- instances of inner classes have an implicit reference to the outer instance
- larger heap space for both generations -> less frequent GCs, lower GC overhead, objects more likely to become dead (smaller heap -> fast collection)
- tune size of young generation -> implies frequency of minor GCs, maximize the number of objects released in young generation, it is better to copy more than promote more
- tune tenuring distribution (-XX:+PrintTenuringDistribution),

```
Desired survivor size 6684672 bytes, new threshold 8 (max 8)
- age   1:    2315488 bytes,    2315488 total
- age   2:      19528 bytes,    2335016 total
- age   3:         96 bytes,    2335112 total
- age   4:         32 bytes,    2335144 total
```

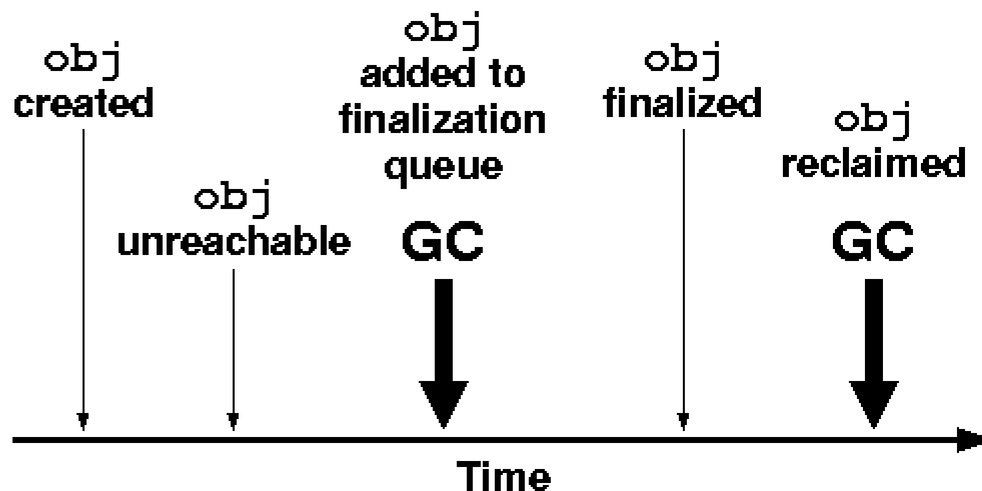


- overall application footprint should not exceed physical memory !
- different Xms and Xmx implies full GC during resizing (consider Xms=Xmx)

Finalizable objects

- » have a non-trivial finalize() method
- » postmortem hook
- » used for clean-up for unreachable object, typically reclaim native resources:
  - GUI components
  - file
  - socket

```
public static class Image1 {
    private int nativeImg;
    // ...

    private native void disposeNative();
    public void dispose() { disposeNative(); }
    protected void finalize() { dispose(); }

    static private Image1 randomImg;
}
```



Finalizable objects

- finalizable object allocation:
  - slower - VM track finalizable objects
- finalizable object reclamation
  - at least two GC cycles:
    - identification and enqueue object on finalization queue
    - reclaim space after finalize()
- not guaranteed when finalize() is called, whether is called (can exit earlier) and the order in which it is called
- finalizable objects occupy memory longer along with ***everything reachable from them*** !!!
- implementation based on references (see Finalizer class)

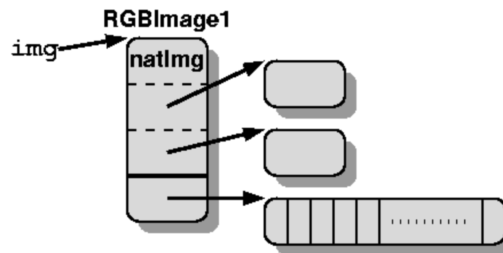
Finalizable objects - example

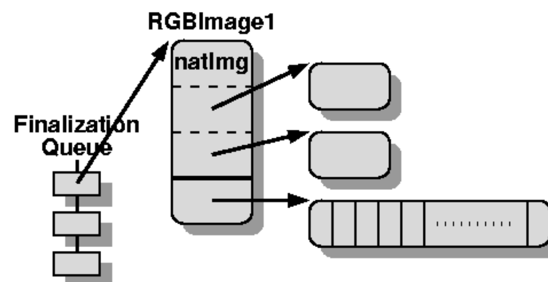
» subclassing issue

- delay reclamation of resources not explicitly used

```
public class RGBImage1 extends Image1 {
    private byte rgbData[];
}
```

- RGBImage1 inherit finalize() method





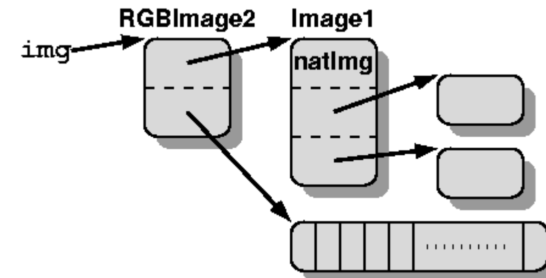
Finalizable objects – example solution 1

» contains reference instead of extends

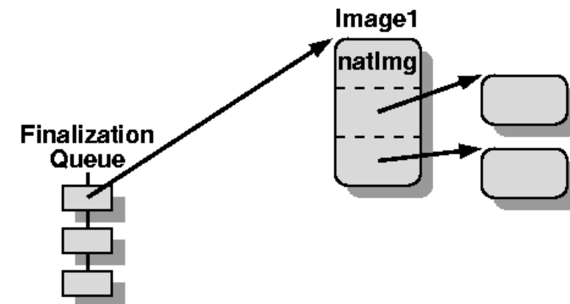
```
public class RGBImage2 {
    private Image1 img;
    private byte rgbData[];

    public void dispose() {
        img.dispose();
    }
}
```

» BUT no access to non-public, non-package members

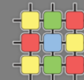
```
img = new RGBImage2();
```



img = null; and after a subsequent GC...



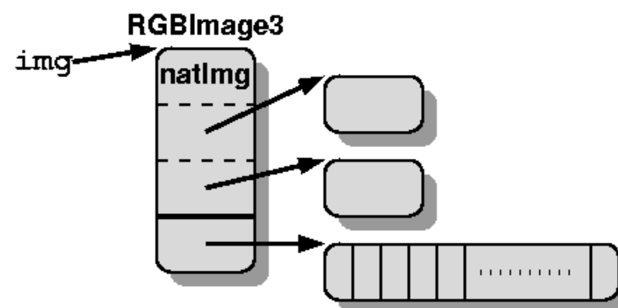
Finalizable objects – example solution 2

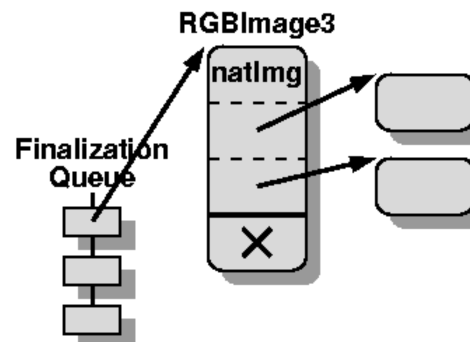


» manual nulling

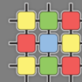
```
public class RGBImage3 extends Image1 {
    private byte rgbData[];

    public void dispose() {
        super.dispose();
        rgbData = null;
    }
}
```

» BUT requires explicit disposal





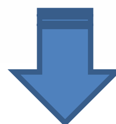
Performance recommendations



» avoid finalizable objects (non-trivial finalize() method)

- slower allocation due to their tracking
- require at least two GC cycles:
  - enqueues object on finalization queue
  - reclaims space after finalize() completes
- beware of extending objects which define finalizers

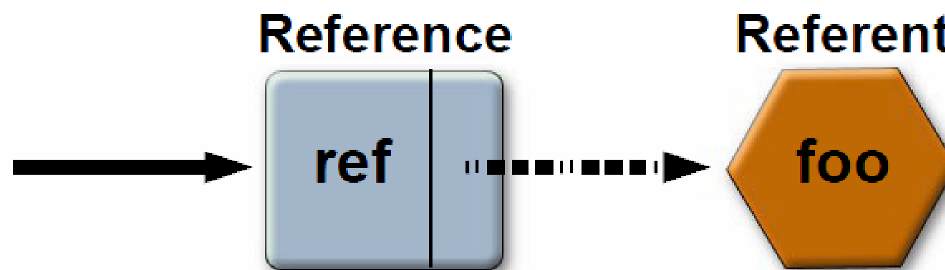
```
class MyFrame extends JFrame {
    private byte[] buffer = new byte[16 * 1024 * 1024];
    ...
}
```



```
class MyFrame {
    private JFrame frame;
    private byte[] buffer = new byte[16 * 1024 * 1024];
    ...
}
```

Reference objects

- » mortem hooks
- » are more flexible than finalization
- » types (ordered from strongest one):
  - {strong reference}
  - soft reference
  - weak reference
  - phantom references
- » can enqueue the reference object on a designated reference queue when GC finds its referent to be unreachable, referent is released
- » references are added only if you have strong reference to REFERENCE !
- » GC has to run !



Weak reference

- pre-mortal hook, pre-finalization processing
- usage:
  - do not retain this object because of this reference
  - canonicalizing map – e.g. ObjectOutputStream
  - don't own target, e.g. listeners
  - implement flexible version of finalization:
    - prioritize
    - decide when to run finalization
- get() returns
  - referent if not reclaimed
  - null, otherwise
- referent is cleared by GC (cleared before enqueued) -> need copy referent to strong reference and check that it is not null !!!
- WeakHashMap<K,V> - uses weak keys

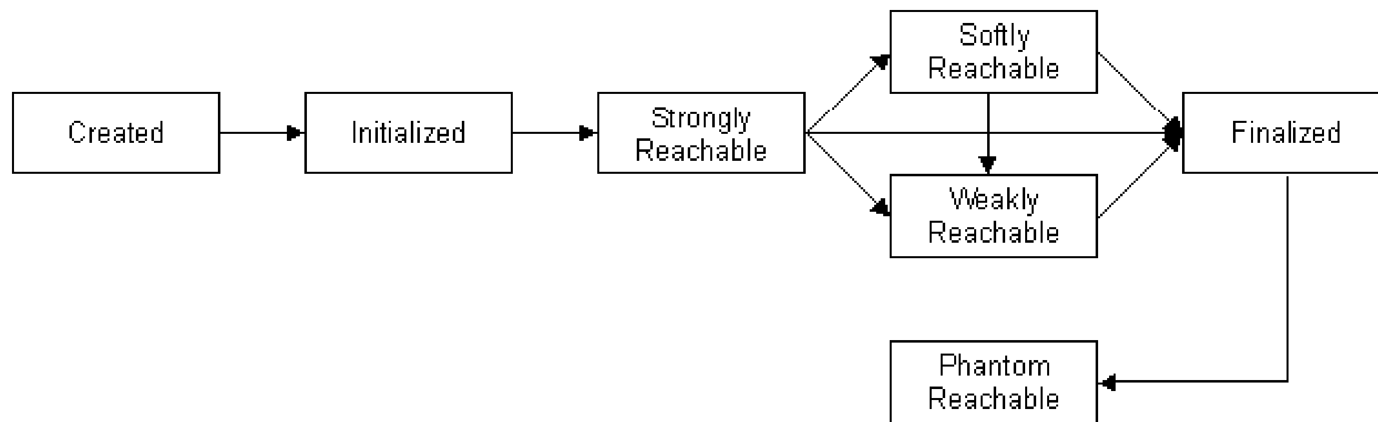
Soft reference

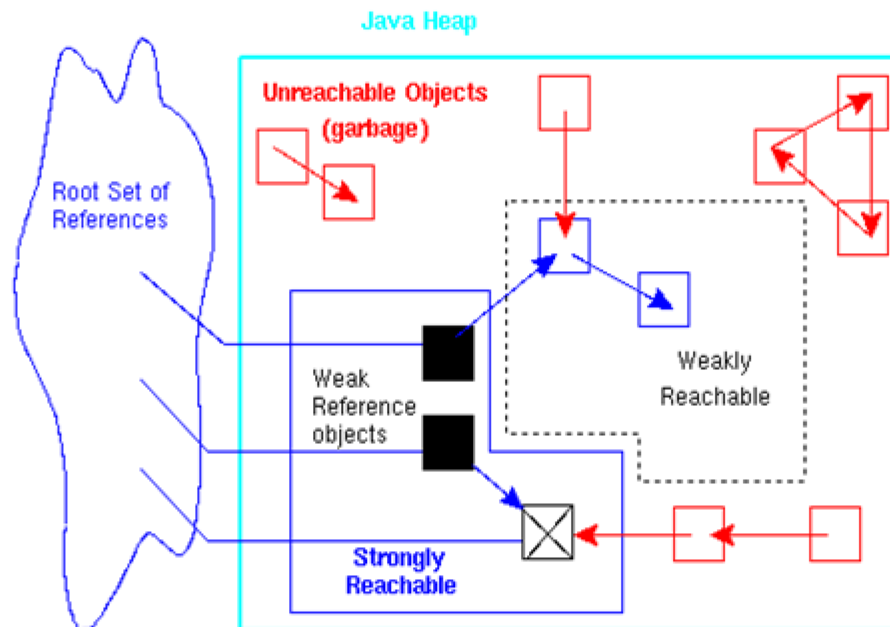
- pre-mortal hook, pre-finalization processing
- usage:
  - reclaim only if there is “memory pressure” based on
  - suitable for caches – create strong reference to data required to keep, best for large objects
  - would like to keep referent, but can loose it
- get() returns:
  - referent if not reclaimed
  - null, otherwise
- referent is cleared by GC (cleared before enqueued)

Phantom reference

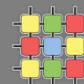
- » post-mortal hook, post-finalization processing
- » designed to be safer than finalizer as the object cannot be resurrected -> ***not true*** !
- » usage:
  - notifies that the object is no longer used
  - keep some data after the object becomes collected
- » get() returns:
  - null always
- » have to specify reference queue for constructor
- » referent is not collected until all references are not become unreachable
- » referent is not cleared automatically, referent can be cleared by method clear()

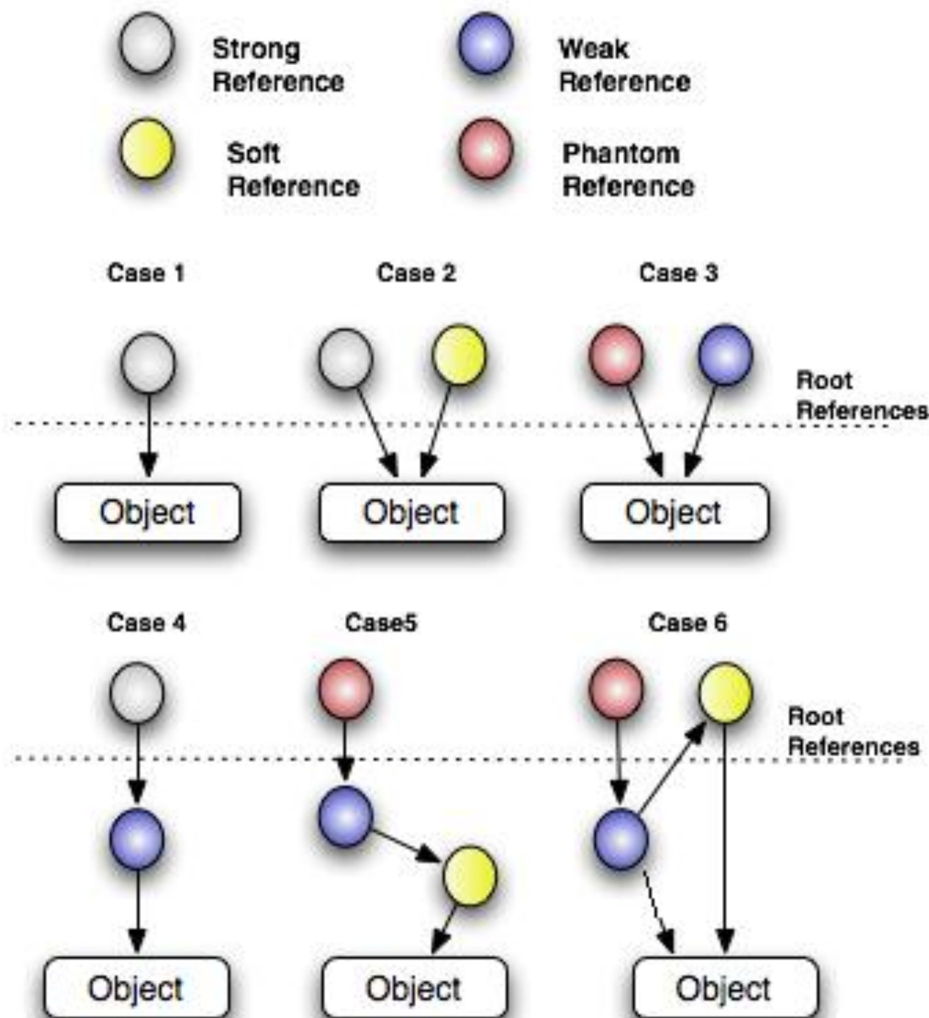
Reachability of an object





Reachability of an object





Intelligent systems

» intelligent system integrates concepts of **artificial intelligence** such as:

- knowledge reprepresentation and expert systems
- advanced search methods
- mathematical reasoning methods
- **nature inspired computing**: artificial neural networks and genetic algorithms
- **agent-based computing**: distributed artificial intelligence and multi-agent systems

Introduction to agents #1

» **autonomous agents and multi-agent systems** (also referred to as agent-based computing):
  - a specific sub-field of computer science and artificial intelligence,
  - investigates the concepts of autonomous decision making, communication and coordination, distributed planning and distributed learning but also game-theoretic aspects of competitive behavior or logical formalization of higher level knowledge structures representing interaction attitude of actors in multi-actor environment.

A multi-agent system is a decentralized computational (software) system, often distributed (or at least open to distribution across hardware platforms) whose behavior is defined and implemented by means of complex, peer-to-peer interaction among autonomous, rational and deliberative units – agents.

Introduction to agents #2

An **agent** is an encapsulated computational (or physical, even human) system, that is situated in some environment, and that is capable of flexible, autonomous behavior in order to meet its design objective (Wooldridge, 2000). The agent can exists on its own but often is a component of a multi-agent system.

Agent technology provides a set of tools, algorithms and methodologies for development of distributed, asynchronous intelligent software applications that leverage the above listed theories.

Key properties of an autonomous intelligent agent

» **Autonomy** – the agent is accountable for execution of its own actions and is not controlled from outside. Often the agent's reasoning mechanism that selects the action to be executed is unknown from outside of the agent (unlike e.g. objects).

» **Reactivity** – the agent is able react qucikly to the events in the environment and to the requests from other agents, it is able to reconsider its activity according to the change of the environment in timely fashion. Often the longest reasoning cycle of an agent needs to perform faster than the fastest change in the environment (calculative rationality).

» **Intentionality** – the agents is able to maintain its long term intention encoded by the agent's designer and is capable to consider both the long term intentions and immediate reactive inputs when selecting the next action.

» **Social capability** – the agent is able to interact, collaborate, form teams but also to perform different levels of reasoning about the other agents.

Agent concepts

Agents design levels:

» **organization-level**: related to the agent communities as a whole (organizational structure, trust, norms, obligations, self-organization, etc.);

» **interaction-level**: concern communication among agents (languages, interaction protocols, negotiations, resource allocation mechanisms);

» **agent-level**: concern individual agents (agent architecture, reasoning, learning, local processing of social knowledge).

Similar concepts

objects - computational entity with its encapsulated state, ability to perform methods on the state and communicating with the other objects via message passing

» lesser **degree of autonomy** - possibility to have a public method

» joint goal is set-up at the **design-time**

» multi-agent systems are inherently **multi-threaded**

expert systems - the most important technology of the 1980's

» expert systems are **disembodied** from the environment

» expert systems are not capable of **reactive** and **proactive behavior**

» expert systems are not equipped with the **social ability**

Agent domains

- **manufacturing**: planning highly complex production, control of dynamic, unpredictable, unstable processes, diagnostics, repair, reconfiguration/replanning.
- **virtual enterprises**: forming business alliances, forming long-term/short-term deals, managing supply chains.
- **internet agents**: mainly for intelligent shopping and auctioning, information retrieval and searching, remote access to information and remote system control.
- **transport**: intelligent car, public transport, logistic and material handling, but also peace-keeping missions, military maneuvers, etc.
- **collective robotics operations**: cooperation and autonomy in the group of robotic entities (UAS, ground vehicles, unattended sensors), replacement of teleoperation with autonomous decision making
- **utility networks**: energy distribution networks, mobile operators networks, cable provider networks - simulation and predication of alarm situations, prevention to black-out and overload, intrusion detection.

Agent architectures #1

» **Reactive agents** are agents that contain no symbolic knowledge representation (ie: no state, no representation of the environment, no representation of the other agents, ...). Their behaviour is defined by a set of perception-action rules.

$$\text{rules} \times \text{percept} \rightarrow \text{action}$$

Agent architectures #2

The classical approach to building agents is to view them as a particular type of knowledge-based system, and bring all the associated methodologies of such systems to bear. We define a **deliberative agent** or SR agent architecture to be one that:

- contains an explicitly represented, **symbolic model of the world**;
- makes decisions (e.g. about what actions to perform) via **symbolic reasoning**.

Agent architectures #3

Belief-desire-intention (**BDI**) model is framework for reasoning about formal abstract models of mental states (based on Theory of Practical Reasoning).

» contains representations (as objects, data structures, or whatever) of:
  - **beliefs**, which constitute its knowledge of the state of its environment (and perhaps also some internal state),
  - **desires**, which determine its motivation what it is trying to bring about, maintain, find out, etc.,
  - **intentions**, which capture its decisions about how to act in order to fulfill its desires (committed desires)

» **intention** is something between the agents' **state of mind** (belief) and the immediate action to be performed

» unlike **desire/goal** an intention may be seen as agents immediate **commitment** to implementing an action.

Interaction schemes

From the point of view of interaction among agents we distinguish:

» Open systems
- Interaction among various types of agents from different developers
- Common understanding of messages is necessary – specification of message structure
- May act as self-interested and try to harm others/whole system
  – security issues
- Strong emphasis on interoperability

» Closed systems
- Interact with a predefined set of agents known to the developer in advance
- Proprietary data formats can be used
- Interoperability can be sacrificed for the sake of performance optimizations

FIPA specifications #1

Foundation for Intelligent Physical Agents - FIPA

» Founded to create specification that will ensure interoperability among agents

» Complete set of specifications from different categories:

- agent communication
- agent transport
- agent management
- abstract architecture and applications

» Most significant for agent interoperability is agent communication and transport

Agent communication language

FIPA-ContractNet-Protocol

Initiator

Participant

cfp

m

i^2 n

refuse

n

dead-
line

j=n-i

propose

reject-proposal

k^2j

accept-proposal

l=j-k

failure

inform-done : inform

Inform-result : inform

Mobile agents #1

Mobile agents are characterized by code mobility

» Mobile agents travel to places, where they perform tasks on behalf of a user

» Reason why to travel - insufficient computational power, unreliable communication, remote data source

» Security issues

» Mobility vs. cloning, stand-in agents

Types of agent mobility

» **Strong mobility** - mobility of code, data and execution state, transparent for the computational process, requires support from the OS and execution environment

» **Weak mobility** - only the code and data are transferred, intentional mobility

Mobile agents #2

Problems with agent mobility

» Various operating systems, programming languages, agent platforms

» Security features - permissions, authorities, access control

» Necessity to transfer all required data, libraries

» Interaction with message transport system - new communication address, delivery of messages received by old message transport

Security

Important aspect especially in the case of open systems

- » Thread-safe agent execution model which holds rest of the system harmless against agent failure
- » Communication security
  - message encryption/signing
  - security certificates with public/private keys
- » Trust and reputation models to create a set of trustful collaborators in open systems
- » Protect private knowledge of individual agents