Object-Oriented Middleware

The Java Platform

Professeur Olivier Gruber

Université Joseph Fourier

Projet SARDES (INRIA et IMAG-LSR)

Olivier.Gruber@inria.fr

Outline

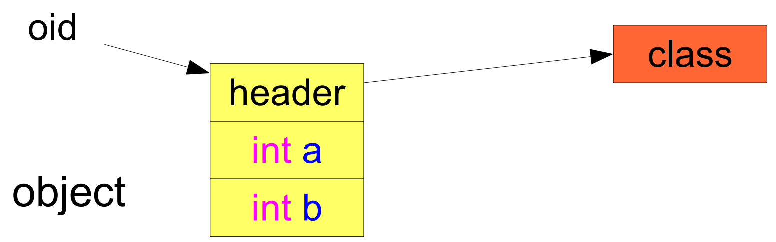
- Today
  - Basics of the Java Platform
    - Design choices of the Java language
      - Java reflection
    - Design choices of the Java platform
      - Garbage collection, finalizers, soft references
      - Class loaders
      - Threads and monitors
  - Java Remote Method Invocation (RMI)
    - Principles and analysis
    - Discussing distributed garbage collection

Java Platform - Basics

- Object-oriented model
  - An object is a triplet
    - An identity, a state, and a behavior
  - An object is an instance of a class
    - A class is a factory for its instances
    - Instances of a class form its extent
  - Classes are types
    - Define a structure (fields)
    - Define a behavior (methods)
    - Define constructors

```
class Line {
  int a;
  int b;
  Line(int a, int b) {
    this.a = a; this.b = b;
  }
  int equation(int x) {
    return a*x+b;
  }
}
```

```
int x,y;
Line line = new Line(2,3);
x = 5;
y = line.equation(x);
```



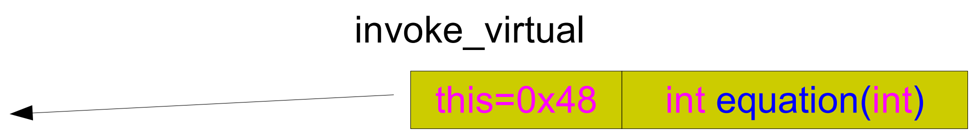
Olivier.Gruber@inria.fr

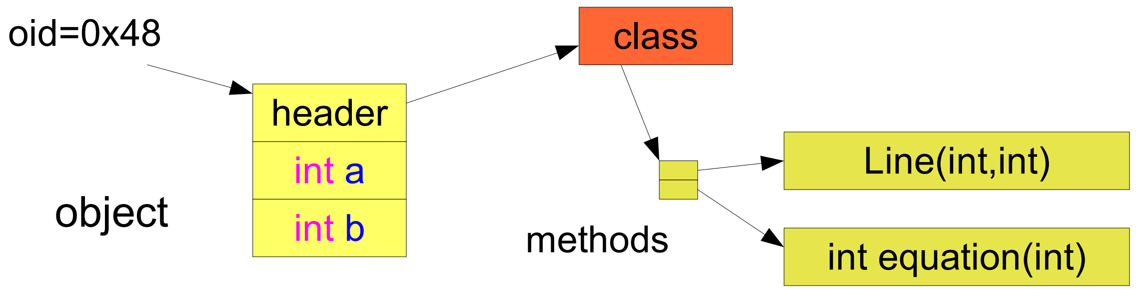
Java Platform - Basics

- Object-oriented model
  - Method invocation
    - Sending a message to an object
    - The object is called the receiver
  - The class dispatches the message
    - This is called late binding (finding the code)
    - Matching the method signature to the method declared in the class

```
class Line {
  int a;
  int b;
  Line(int a, int b) {
    this.a = a; this.b = b;
  }
  int equation(int x) {
    return a*x+b;
  }
}
```

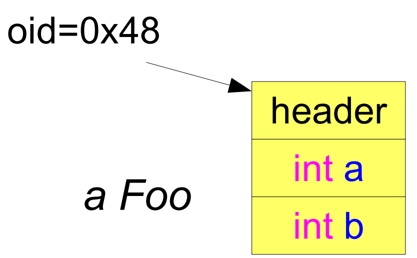
```
int x,y;
Line line = new Line(2,3);
x = 5;
y = line.equation(x);
```





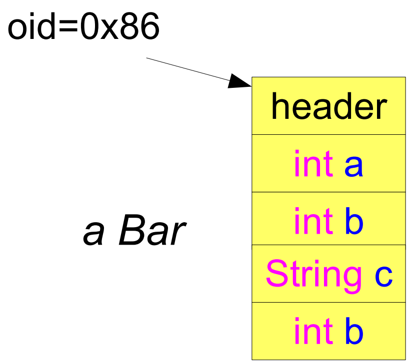
Java Platform - Basics

- Object-oriented model
  - Classes are organized in a sub-typing hierarchy
    - Subtypes inherit both the structure and behavior of super types
    - Do not confuse with aggregation
  - Structural inheritance
    - All fields are inherited
    - No matter the names or types




```
class Foo {
  int a;
  int b;

  Foo(int a, int b) {...}

  int foo(int x) {...}

}
```

```
class Bar extends Foo {
  int b;
  String c;

  Bar(String c, int b) { ... }

  int foo(int x) {... }
  void foo(int x, int y) {... }

  int bar(int x, int y) { ... }
}
```



Olivier.Gruber@inria.fr

Java Platform - Basics

- Object-oriented model
  - Classes are organized in sub-typing hierarchy
    - Subtypes inherit both the structure and behavior of super types
    - Do not confuse with aggregation
  - Method inheritance
    - Method overloading
      - Same name, but different signatures
    - Method overridding
      - Same signature

```
class Foo {
   int a;
   int b;

   Foo(int a, int b) {...}

   int foo(int x) {...}
}
```

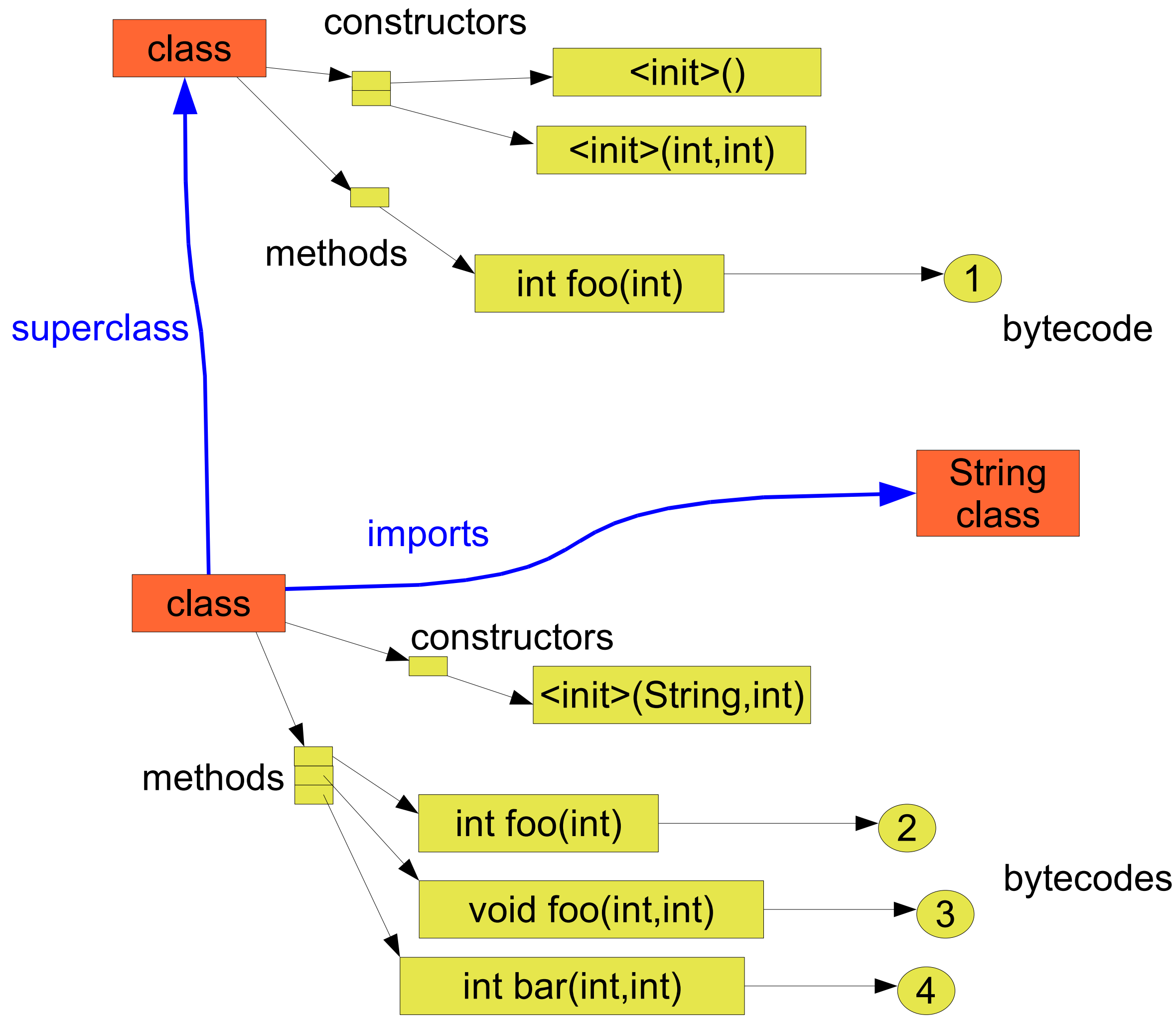
```
class Bar extends Foo {
   int b;
   String c;

   Bar(String c, int b) { ... }

   int foo(int x) {... }        // overriding

   void foo(int x, int y) {... }   // overloading

   int bar(int x, int y) { ... }
}
```

Olivier.Gruber@inria.fr

Java Platform - Basics


```
class Foo {
  int a;
  int b;
  Foo() {...}
  Foo(int a, int b) {...}

  int foo(int x) { 1 }
}
```

```
class Bar extends Foo {
  int b;
  String c;

  Bar(String c, int b) { ... }

  int foo(int x) { 2 }

  void foo(int x, int y) { 3 }

  int bar(int x, int y) { 4 }
}
```



Olivier.Gruber@inria.fr

Java Platform - Basics

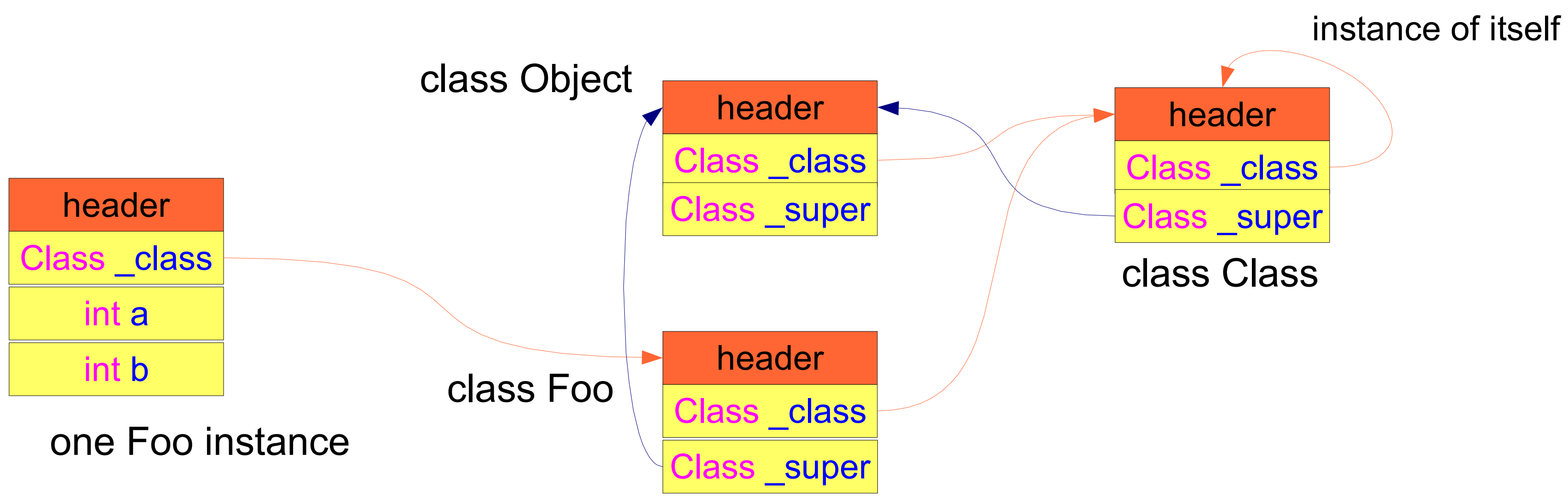
- Java
  - Introduces interface types
    - Interfaces only define behaviors
    - Interfaces support multiple inheritance
    - A class implements one or more interfaces
  - Abstract classes
    - Classes that cannot be instantiated
    - Interfaces are always abstract
- Service-oriented pattern
  - Service contracts are interfaces or abstract classes
  - Service objects are created through a factory pattern

Olivier.Gruber@inria.fr

Java Platform - Basics

- Java
  - Implicit class link
    - Instanceof operator
    - The getClass() method
  - Garbage collection
    - Keep classes alive as long as they have an instance
  - Classes are objects
    - So they also have a class
    - The metaclass, called the class lass

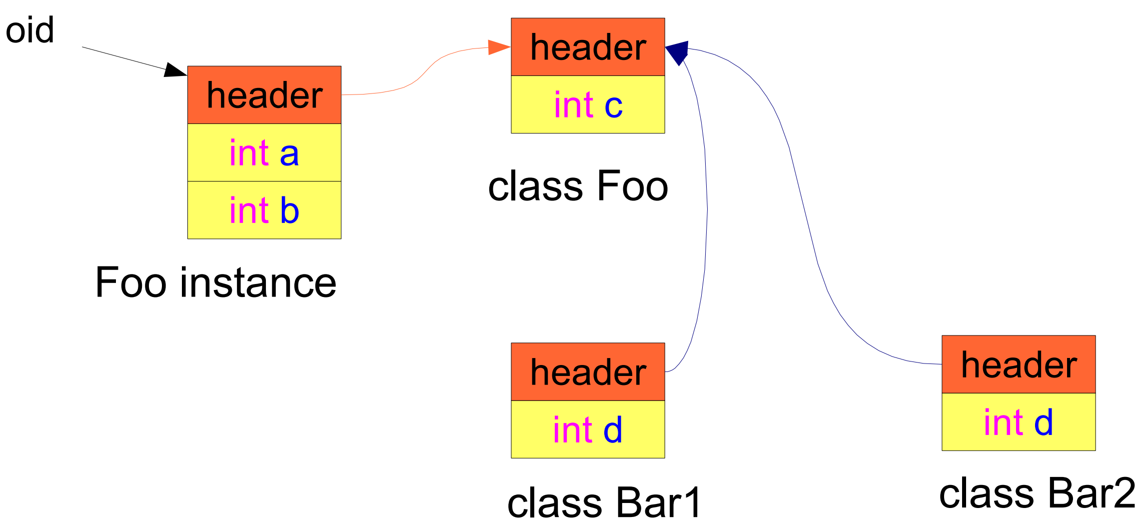
```
class Object {
   Class getClass();
   ...
}

class Foo extends Object {
   int a,b;
   ...
}
```



Java Platform - Basics

- Java
  - Static fields
    - As constants, both in interfaces or classes
    - As non-constant fields, only in classes
  - Statics are named global variables
    - They are not class fields, in the proper sense
    - Indeed, superclass statics are shared

```
class Foo {
   int a,b;
   static int c;
}

class Bar1 extends Foo {
   int e;
   static int d;
}

class Bar2 extends Foo {
   int e;
   static int d;
}
```



Java Platform - Reflection

- Java runtime reflection
  - Reify types at runtime
  - Essentially through
    - java.lang.Class
    - java.lang.reflect.*
- Looking at code
  - Browse the JDK sources
  - Samples to illustrate core functions
    - Walk meta-level description through class objects
    - Instantiate objects
    - Invoke methods
    - Get and set fields

Java Platform - Reflection

- Core reification
  - Classes, interfaces, constructors, methods, fields
- Modifiers
  - Access modifiers (public, private, protected)
  - On fields, methods and classes
- Access and invoke
  - Can get and set fields
  - Can invoke methods
  - Can construct new instances
- But
  - **Cannot create new types**
  - **Add methods or fields, etc.**

Java Platform - Reflection

- Java Arrays
  - **Arrays are objects in Java**
    - **The synthetic field *length***
    - Special operator **[]**
  - Array classes also automatically reified
    - Modifiers
      - Array classes have the access modifiers of their element type
      - An array of private classes is private
    - Arrays are ***cloneable and serializable***
- Can construct new instances
  - Directly:
    - int a[] = new int[3];
  - Through reflection:
    - Person p[] = Array.newInstance(Person.class,3);

Olivier.Gruber@inria.fr

Java Platform – Class Loaders

- Started Simple
  - As a sandbox for applets
  - Wanted a complete isolation of downloaded code
- Essentials
  - Its own copy of classes
    - Avoid sharing statics
    - Avoid name and version conflicts between loaded classes
  - Works hand-in-hand with Java security
    - Controls accesses to resources
- Evolved Poorly – Mixing several concepts
  - A scoping mechanism for types
  - A dynamic and lazy linker for classes
  - A mechanism to define (load) types

Olivier.Gruber@inria.fr

Java Platform – Class Loaders

- Class Loading
  - Only through the class file format
    - This is quite unfortunate
    - Only the JVM can create types programmatically
  - Special native method in the JVM
    - The native method **ClassLoader.define(...)**
    - Passing the byte array of a class file to define the described type
  - The class file is an exchange format
    - Could have been in XML, used a more efficient binary representation
    - **Produced by Java compilers and consumed by class loaders**

Class Loaders – Class File Format

- Meta-data part
  - A Java type description
    - A class name and flags
    - Its superclass and implemented interfaces
    - Its fields and methods
  - **All linking information is through names**
    - Naming types (classes, interfaces)
    - Naming members (fields and methods)
- Constant pool
  - Contains the linking names
  - But also some constant values
    - Primitive types and strings
- Code part
  - Bytecode sequences
  - As attributes on methods

magic number
constant pool size
constant pool
access flags
this class
superclass
interface count
interfaces
field count
fields
method count
methods
attribute count
atrributes

Olivier.Gruber@inria.fr

Classfile Examples


```
public class Line {
   int a;
   int b;
   Line(int a, int b) {
      this.a = a; this.b = b;
   }
   int equation(int x) {
      return a*x+b;
   }
   public String toString() {
      return “a line“;
   }
}
```

magic number
constant pool size
constant pool:
  “a line“
  java.lang.Object

access flags: public
this class: Line
superclass: idx
interface count: 0
interfaces:

field count: 2
  int a;
  int b;
method count: 3
  <init>(int a, int b)
  int equation(int x)
  public String toString()

attribute count: 3
  bytecode arrays

Olivier.Gruber@inria.fr

Classfile Examples

```
package org.xyz;

public class Foo {
   int a;
   int b;

   Foo(int a, int b) {...}

   int foo(int x) {...}

}
```

```
package org.pqr;

import org.xyz.Foo;

public class Bar extends Foo
   implements  IBar {
   int b;
   String c;

   Bar(String c, int b) { ... }

   int foo(int x) {... }
   void foo(int x, int y) {... }

   int bar(int x, int y) { ... }
}
```

```
magic number
constant pool size
constant pool:
  java.lang.String
  org.pqr.IBar
  org.xyz.Foo
access flags: public
this class: Bar
superclass: idx
interface count: 0
interfaces: idx

field count: 2
   int a;
   String c;

method count: 3
   <init>(String c, int b)
   int foo(int x)
   void foo(int x, int y)
   int bar(int x, int y)

attribute count: 4
  bytecode arrays
```

Olivier.Gruber@inria.fr

Java Platform - Reflection

- Class loaders
  - A scope for Java types
    - Two class loaders defining the same type yields two runtime types
    - Even when using the same class file
  - Beware of equivalent names
    - Name equivalence does not mean a thing between class loaders
    - **Same type name** does not mean **the same type**
  - Structural equivalence does not mean the same type
    - Two types are the same only if the two class objects are the same class object

Rule 1: two classes are the same if they are the same class object

Rule 2: one class object belongs to one and only one classloader

Olivier.Gruber@inria.fr

Java Platform - Reflection

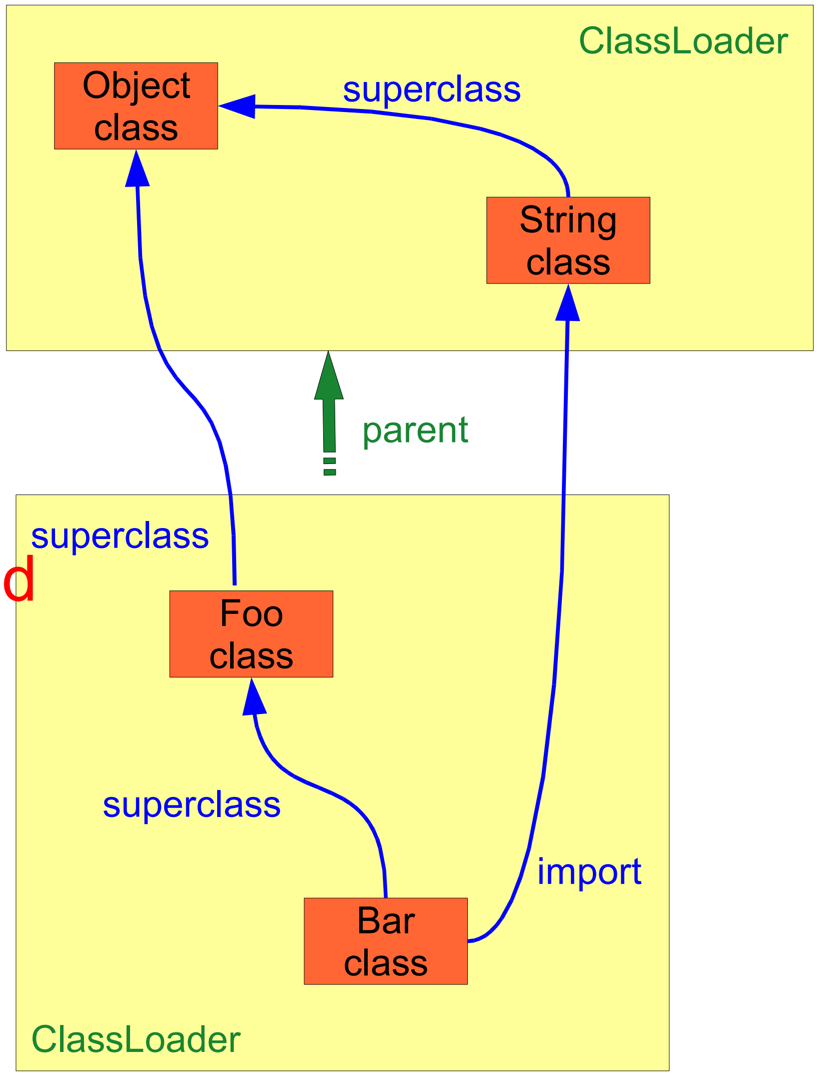
- Hierarchy of scopes
  - A single tree of class loaders per JVM
  - A class loader has a parent class loader
  - Types in the parent class loader are visible
- Bootstrap class loader
  - The root of all class loaders
  - Created at bootstrap by the JVM to load core classes
    - java.lang.Object, java.lang.Class
    - java.lang.String, java.lang.Throwable, java.lang.Exception
    - Etc.

Java Platform - Reflection

- Class loading
  - A tree of class loaders
  - A complex graph of types across all class loaders
- Reminder
  - Could have redundant loading!

the same class file may be loaded in different class loaders...

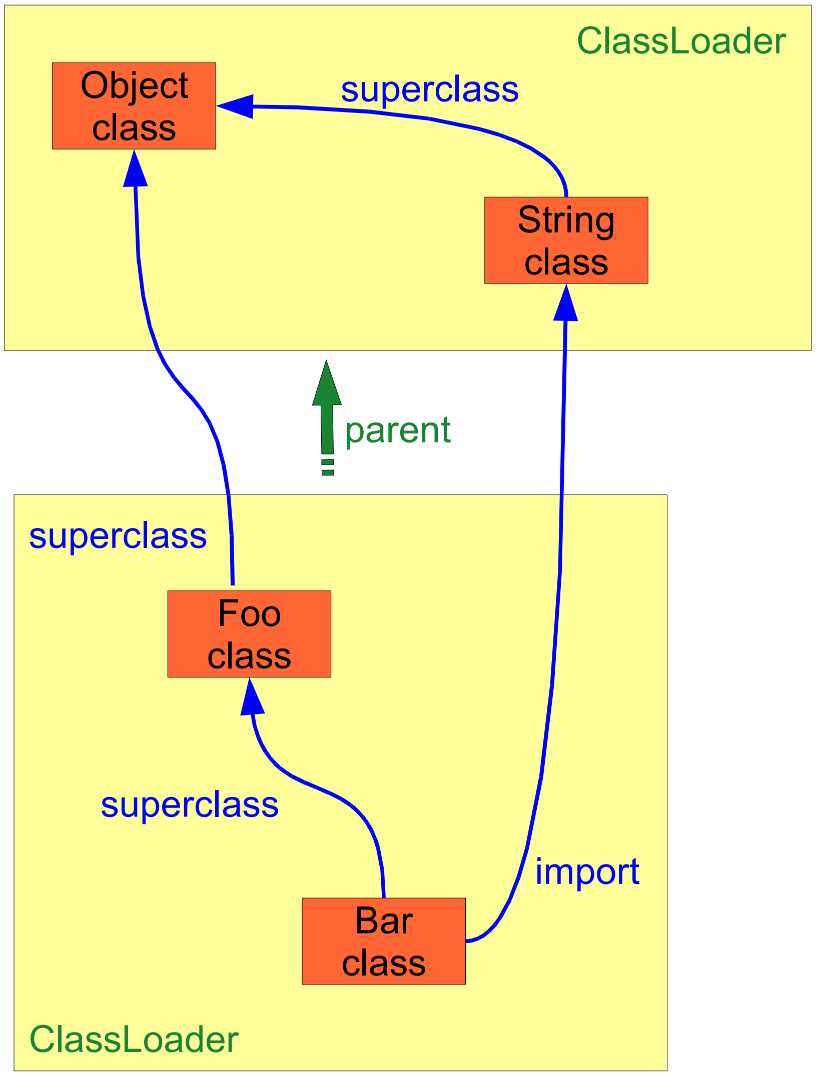
it will be different class objects and therefore different types



Olivier.Gruber@inria.fr

Java Platform - Reflection

- Dynamic and lazy class linker
  - Multi-stage linking
    - Loading
    - Prepared
    - Resolved
    - Initialized (static initializer)
  - Warning
    - Loading may succeed but resolving or initializing may fail much later

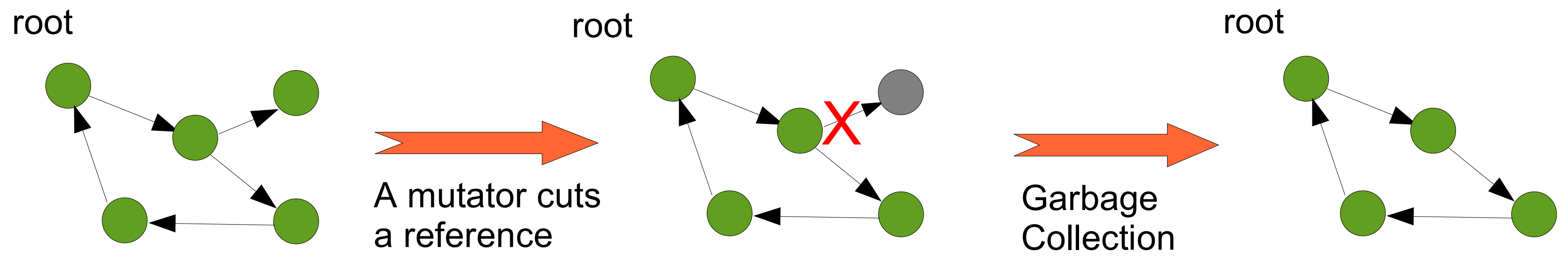


Java Platform - Objects

- Java objects
  - Instances
    - An object is an instance of a class
    - It has an identity and a state (field values)
  - Two way to compare objects
    - Equality of identities (using the operator ==)
    - Equality of states (using the Object.equals(Object) method)
- Hash code
  - Object.hashCode() method
    - Not a real identity, but it is **invariant** per instance
    - Used for collections such as directories or maps
  - WARNING
    - The hash code must work correctly with the value equality
    - **If equals, they must have the same hash code**
    - If you need to override one, override both methods

Java Platform - Objects

- Java is garbage collected
  - **Live objects are kept**
  - Live objects are reachable from roots of persistence
  - Roots are traditionally thread stacks and static fields in loaded classes
- **Being garbage is a stable property**
  - I.e. once an object is garbage, it remains garbage

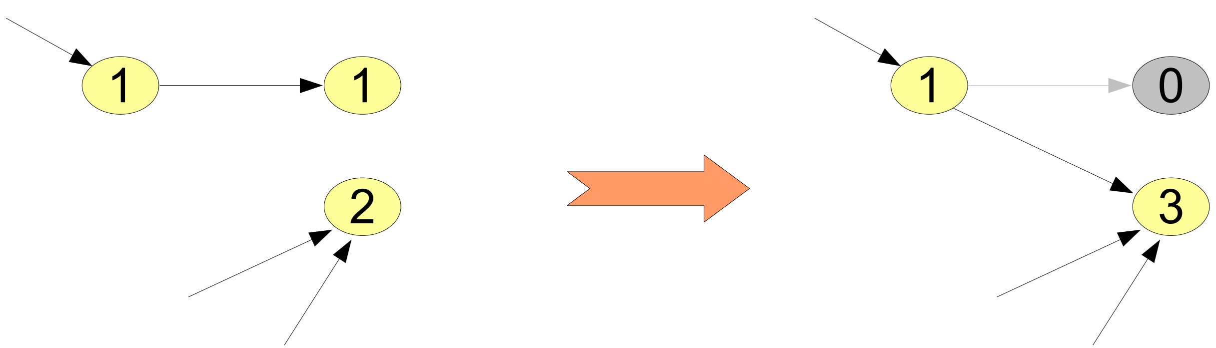


Java Platform - Objects

- Garbage Collector
  - Garbage collection is about detection and reclaimation of garbage objects
  - Different approaches are possible
    - Scavenger, mark&sweep, generational, etc.
- **Performance**
  - Limit the overhead, so run the GC rarely
  - Avoid growing the heap, so run the GC often enough
- **Correctness**
  - **Never detect and reclaim a live object**
- **Liveness**
  - **Detect and reclaim garbage faster than objects are allocated**

Java Platform – Garbage Collection

- Reference Counting
  - Each object is associated a counter
    - Counts the number of references on that object
  - Counter management
    - Happens on assigning reference
      - Decrement the count of the previously referenced object (if any)
      - Increment the counter of the newly referenced object
    - Applies to
      - Reference fields in objects as well as local variables and parameters
    - When a counter reaches zero
      - The object owning that counter is garbage



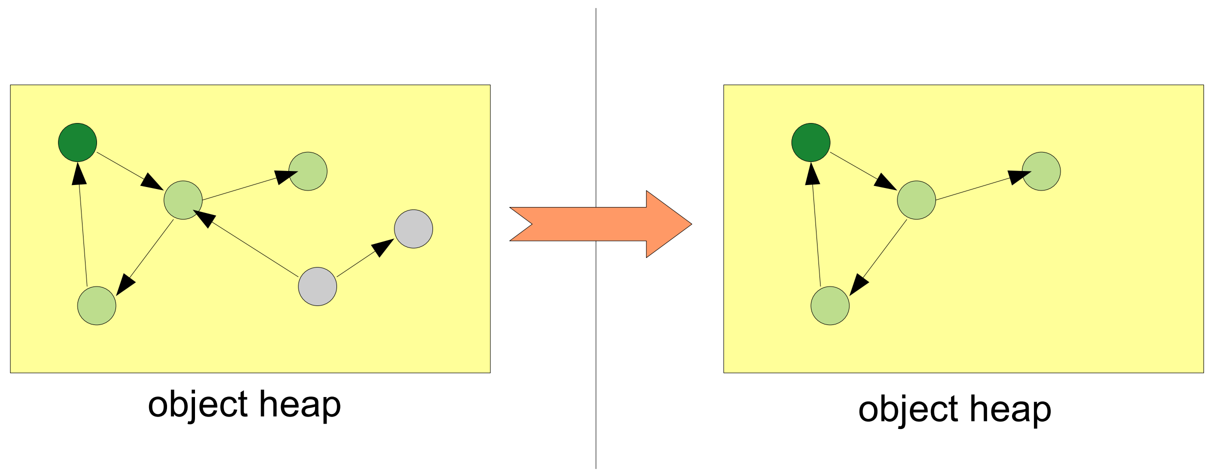
Olivier.Gruber@inria.fr

Java Platform – Garbage Collection

- Discussing Reference Counting
  - Problematic on multi-processors
    - Inherently incremental: impossible to run concurrently
    - Incrementing and decrementing require a critical section
  - Does not require to scan thread stacks
    - But requires to account for local variables and arguments
    - Introduces a high overhead (increment/decrement)
  - Extra paging
    - Accesses objects even if only references are manipulated
    - Dirties memory pages, potentially increasing the overhead of virtual memory paging
  - **Does not reclaim cycles**

Java Platform – Garbage Collection

- Scavenger
  - Copying collector, using two spaces
    - Copy live objects from the old space to the new one
    - Discard the old space



live object

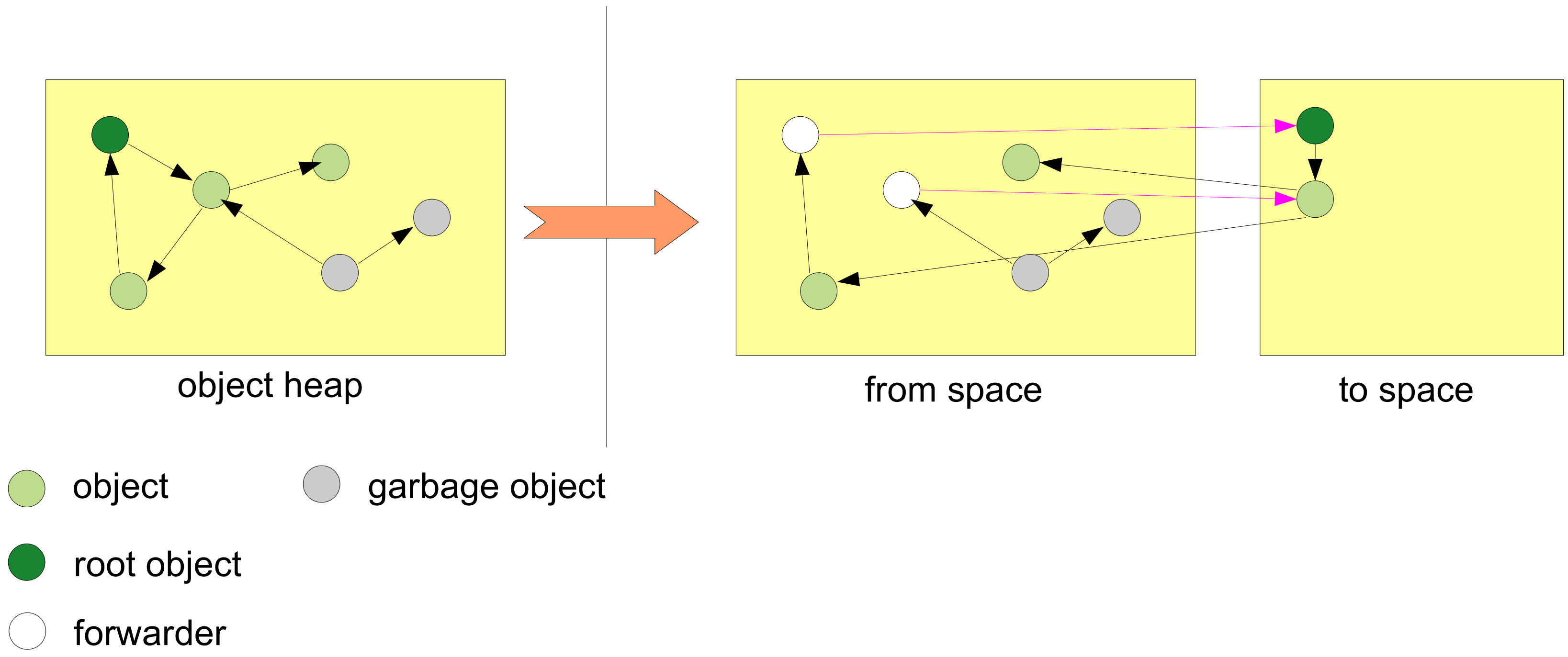
root object

garbage object

Olivier.Gruber@inria.fr

Java Platform – Garbage Collection

- Scavenger details
  - Live objects are reachable from roots (thread stacks and class statics)
  - Leave a forwarder in-place of copied objects
    - Allows to detect cycles (correctness when copying)
    - As well as treat correctly shared objects
  - Use to-space as a recursion stack



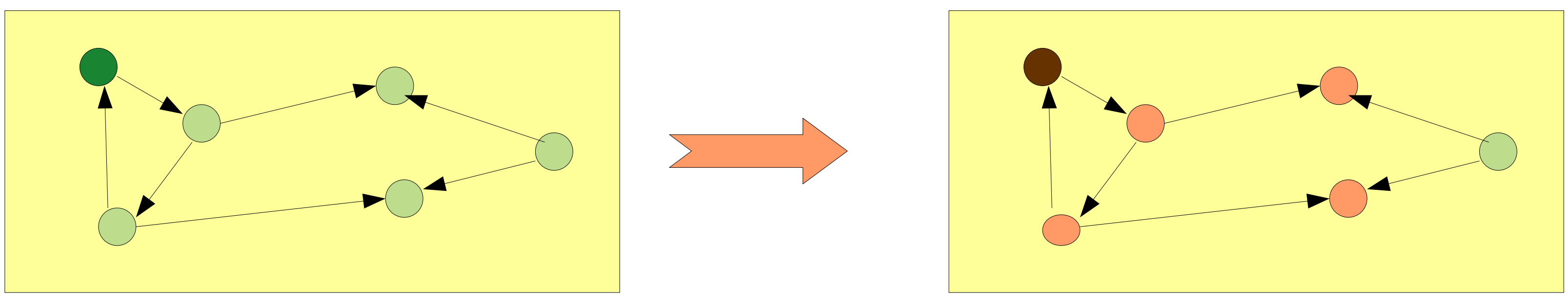
Olivier.Gruber@inria.fr

Java Platform – Garbage Collection

- Discussing Scavenger
  - Simple when designed as stop the world
    - A simple depth-first recursive walk of an object graph
    - Cycles are easily detected through forwarders
    - Require to scan thread stacks
  - Clustering objects
    - Depth-first scavenging produces efficient in-memory clustering of objects
  - Efficiency
    - Depends on the ratio of live versus garbage objects
    - Also depends on the cumulative size of live objects
    - The fewer live objects, the more effective
    - May lead to allocate twice the heap size

Java Platform – Garbage Collection

- Mark & Sweep
  - A two-phase garbage collection
    - A marking phase, coloring live objects
    - A sweeping phase reclaiming garbage objects (not colored)
  - Marking phase
    - Walks the refer-to graph from roots (thread stacks and class statics)
    - Carry the current color



object heap

object heap

object

root object

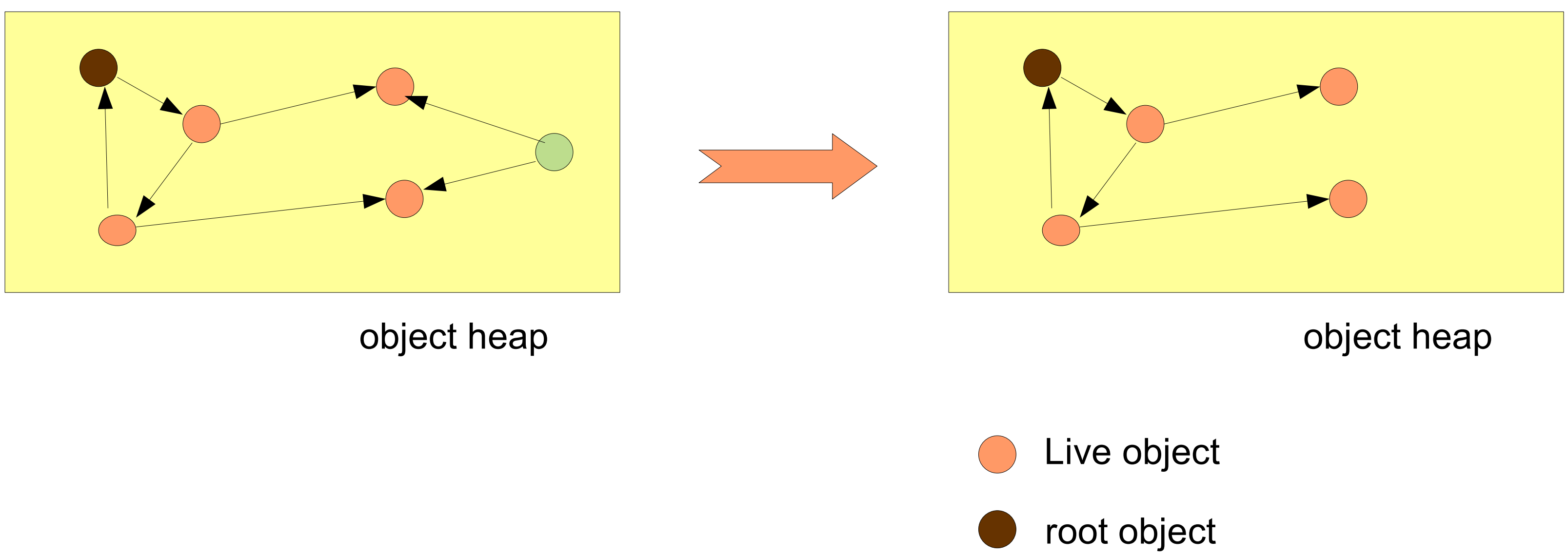
Live object

root object

Olivier.Gruber@inria.fr

Java Platform – Garbage Collection

- Mark & Sweep
  - Sweep phase
    - Sweeps sequentially the object heap to discover garbage objects
  - Reclaiming garbage
    - Using free lists (non-compacting sweeping)
    - Compact as sweeping (challenging to maintain references)



Olivier.Gruber@inria.fr

Java Platform – Garbage Collection

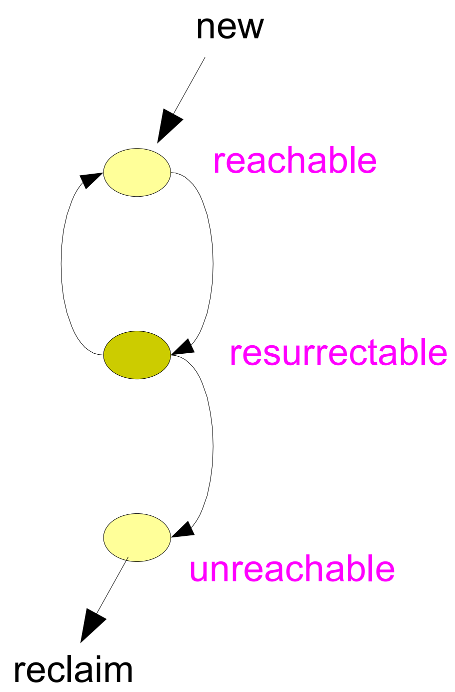
- Discussing Mark & Sweep
  - Not too sensitive to the live/garbage ratio
  - Requires to scan thread stacks
  - Caveats of free-list memory management
    - Can lead to traditional fragmentation
    - Costly allocation (different algorightms such as first-fit, best-fit, etc.)
  - Two scans of the object heap
    - One through references and the other sequentially
    - May lead to heavy paging activity if heap larger than main memory
    - It defeats the LRU policy of most virtual memory systems
- Compacting Mark&Sweep
  - Some mark&sweep do compact the heap during the sweep phase
  - Usually done by slidding objects, does not improve locality

Java Platform - Finalizers

- The problem
  - Java depends on a lot of native resources represented by objects
  - How does one free those resources?
- The finalize method
  - The object class defines a method *finalize()*
    - Any class may redefine this finalize method
    - A class that redefines its finalize method is said to have a ***finalizer***
  - When is it called?
    - The finalize method is called when the object is detected as being garbage
    - If the finalize method is not redefined, it is not called
    - However, the finalize method is called only once
  - Threads?
    - There is no guarantee about which thread is used to call finalize methods
    - But that thread does not hold any user-level Java monitor

Java Platform - Finalizers

- Finalizers introduces resurection
  - **It is legal for a finalize method to make a garbage object live again**
  - Reminder: finalizers are called only once per object
  - Require to detect twice that an object is garbage
- Impacts garbage collection
  - Introduce a new state:
    - Reachable (live)
      - There is a path from roots to the object
    - Resurrectable
      - The object is not reachable
      - The object may be resurrected
      - All objects go through that state
    - Unreachable (garbage)
      - The object is not reachable
      - The object cannot be resurrected



Java Platform - Finalizers

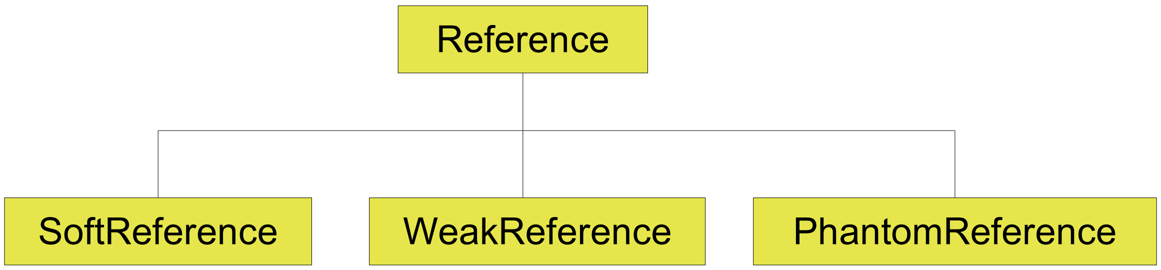
- Compatibility with GC algorithms
  - Compatible with reference counting
    - Easy to call the finalizer when the counts drop to zero
    - Easy to know that the object remained garbage
      - Counter still at zero after the finalizer run
    - But reference counting is rarely used in practice
  - Incompatible with scavenging
    - Reintroduces a sweep to find garbage objects with a finalizer
    - Never know when to free the from-space because of resurection
  - Mark&Sweep is well-suited
    - Easy to extend the sweeping phase to find objects with finalizers
      - But delays the actual reclaimation of garbage objects
    - Still requires two marking phase to really know if an object is garbage

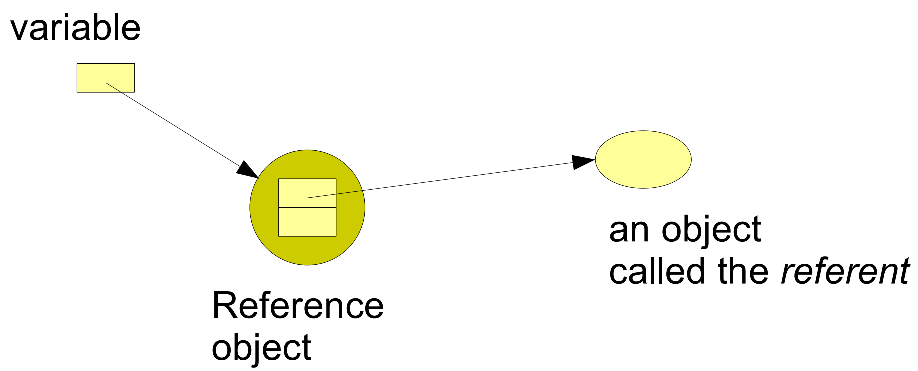
Java Platform - Objects

- Java Finalizers – complex and not enough
  - Native resources are often really scarce
  - Garbage collection is too asynchronous
  - So native resources are not freed fast enough
- Raising the GC frequency is difficult
  - Because it is most often stop-the-world
  - Because it represents an overhead
    - Marking the object graph
    - Sweeping the object heap
- Introduce explicit close/dispose operations
  - On Sockets, files
  - On Widget toolkits
  - Etc.

Java Platform - References

- Introducing different semantics for Java references
  - Strong references
    - The usual object references in the Java language
  - Weaker references in *java.lang.ref*
    - SoftReference and WeakReference
    - PhantomReference

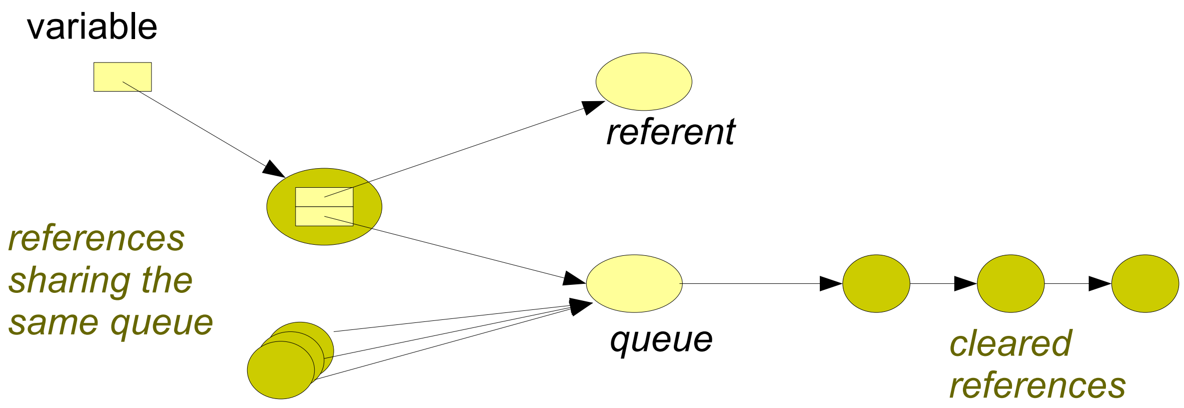




Olivier.Gruber@inria.fr

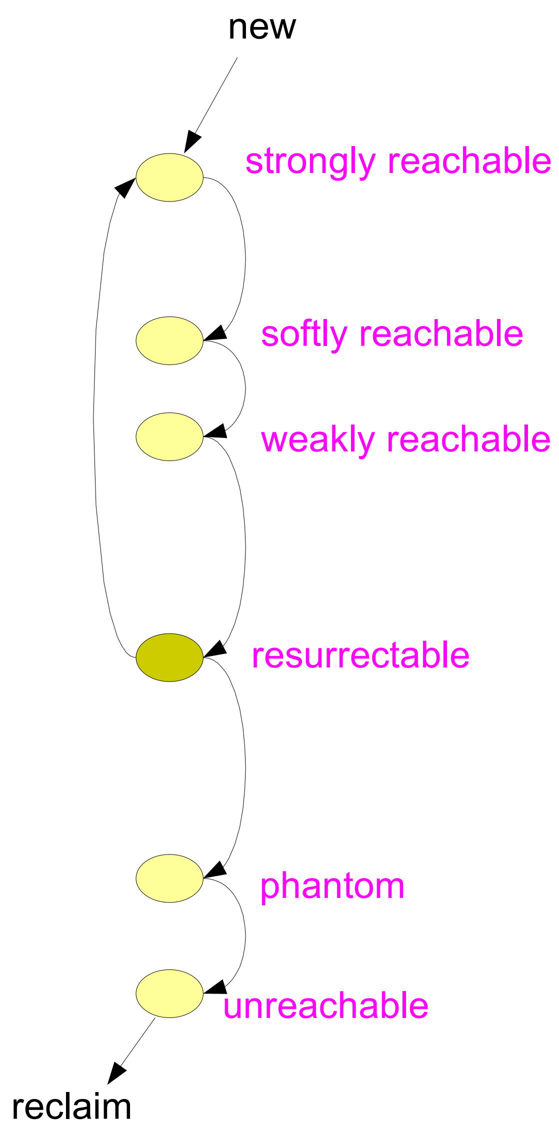
Java Platform - References

- Java References
  - Normal semantics for objects that are strongly reachable
    - If you do not use weaker references, nothing is different than usual Java
  - Weaker references are managed by the GC
    - When an object is no longer strongly reachable
    - The GC may **clear** weaker references to that object at any time
  - Notification
    - A reference may be associated to a reference queue (*ReferenceQueue* class)
    - Once the GC cuts a reference, it push that reference on its associated queue



Java Platform - References

- State changes
  - Reachable is detailled into
    - Strongly reachable
      - Reachable through strong references
    - Softly reachable
      - Not strongly reachable
      - Reachable through soft references
    - Weakly reachable
      - Neither strongly nor softly reachable
      - Reachable through weak references
  - Unreachable
    - Phantom reachable
      - Not reachable but through phantom references
      - Such objects are not resurrectable
    - Unreachable
      - Entirely unreachable
      - Ready to be reclaimed



Java Platform - References

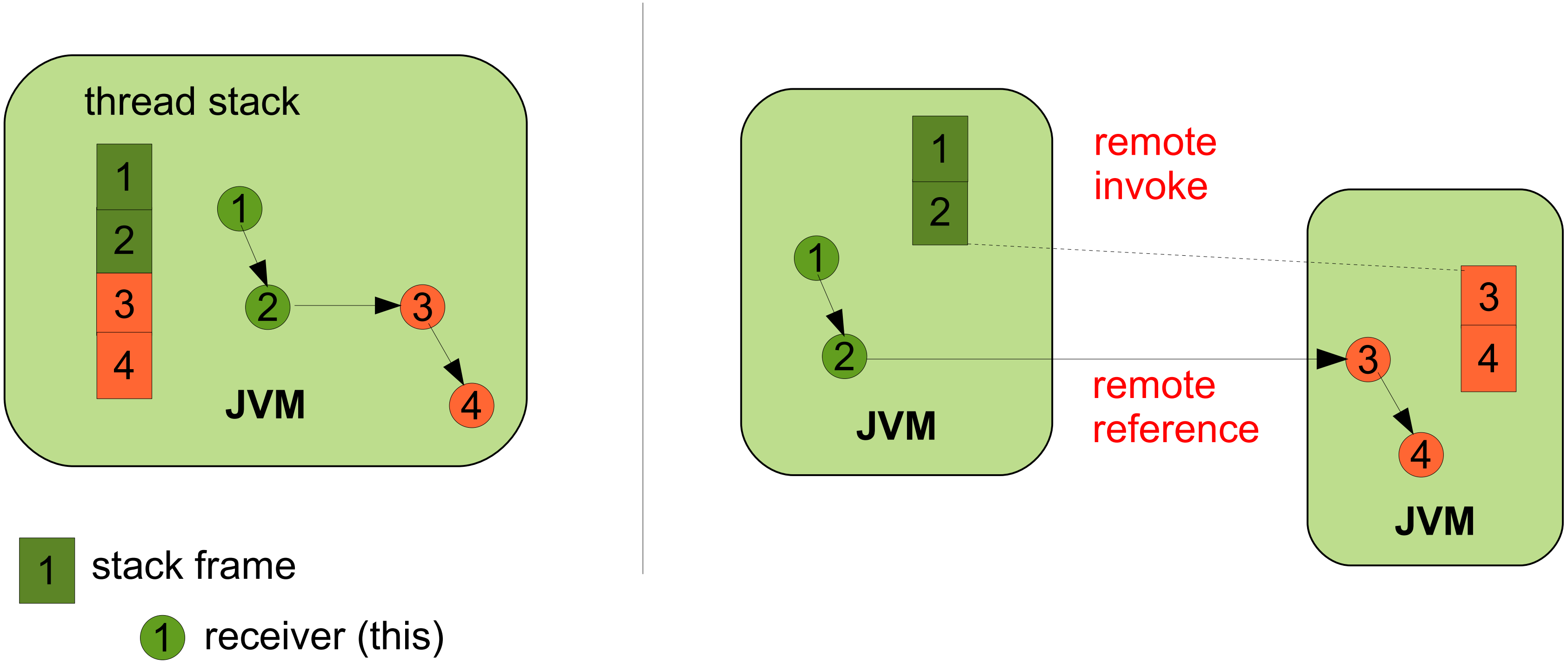
- Discussing soft versus weak references
  - Weak references
    - Weak references must be cleared by the GC as soon as the referenced object is weakly reachable (neither strongly or softly reachable)
    - Used for canonical mappings
      - Keep a mapping key to value
      - Clean the mapping as soon as the key is no longer used (reachable)
  - Soft references
    - Soft references must only be cleared by the GC before it raises an out-of-memory exception, but it may sooner
    - It is suggested that clearing soft references follows the policy:
      - Keep recently created and recently used soft references
    - Used for caching objects
      - A service provides an object
      - Clients keep a reference as long as they need to use the object
      - The GC only reclaims the object and cuts your soft reference if it needs memory

Java Platform - References

- Discussing phantom references
  - More powerful than just finalizers
    - Finalizers are called only once
    - So if objects are resurrected, finalizers can no longer be used for cleanups
  - Phantom references introduce post-mortem resource management
    - An object that is phantom-reachable can no longer be resurrected
    - It is therefore the absolute last moment to do some cleanup

Java Platform – Remote Method Invocations

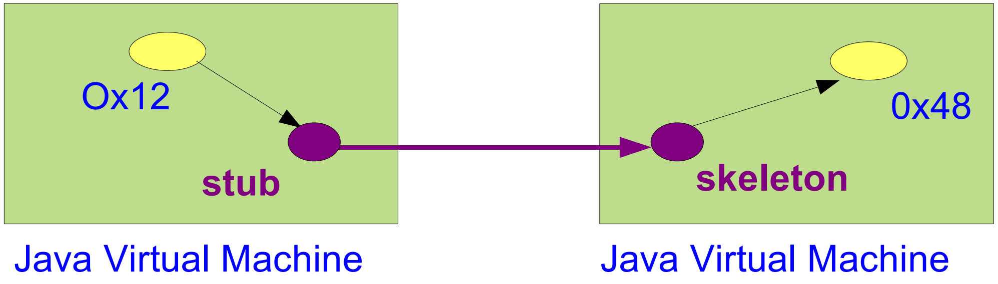
- Requires extensions to the Java Platform
  - Remote references... Java references are local to a JVM
  - Remote method invocations... Java method invocations is local to a JVM
- Available in the JDK 1.4
  - Also as free software such as NinjaRMI (Berkeley) or Jeremie (ObjectWeb)



Olivier.Gruber@inria.fr

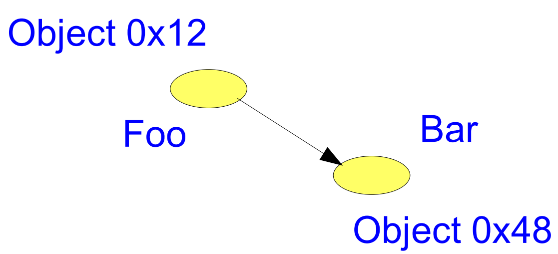
Java Platform - RMI

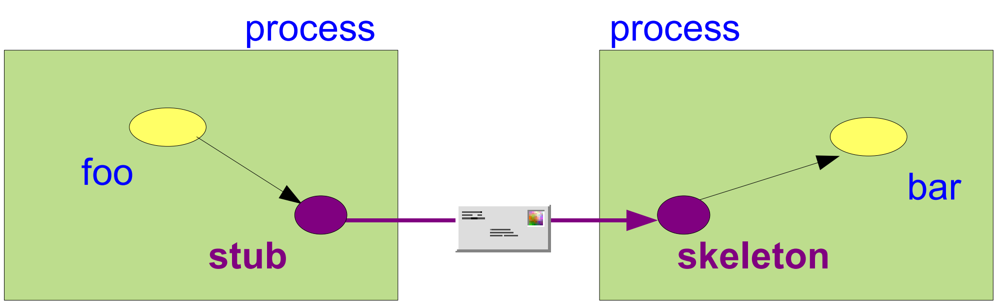
- Remote references
  - Relies on stubs and skeletons
    - Stubs and skeletons are regular Java objects
  - A stub identifies its remote skeleton
    - Using a machine IP, a port number, and a skeleton identifier
  - A skeleton identifies its local object
    - Using a local Java reference

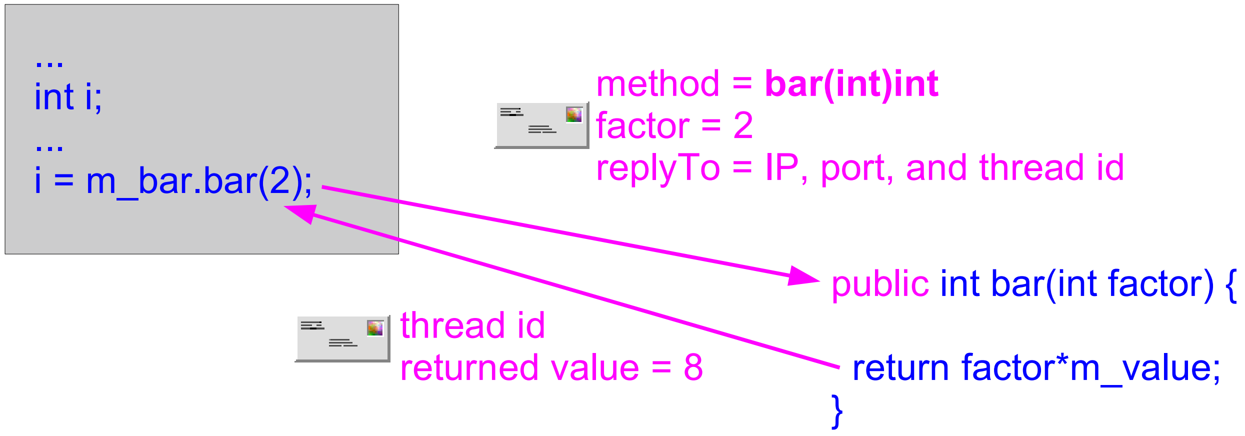


Remoting Method Invocation

- Remote invocations
  - Over remote references, through stubs and skeletons
  - Mashalling and unmarshalling of parameters

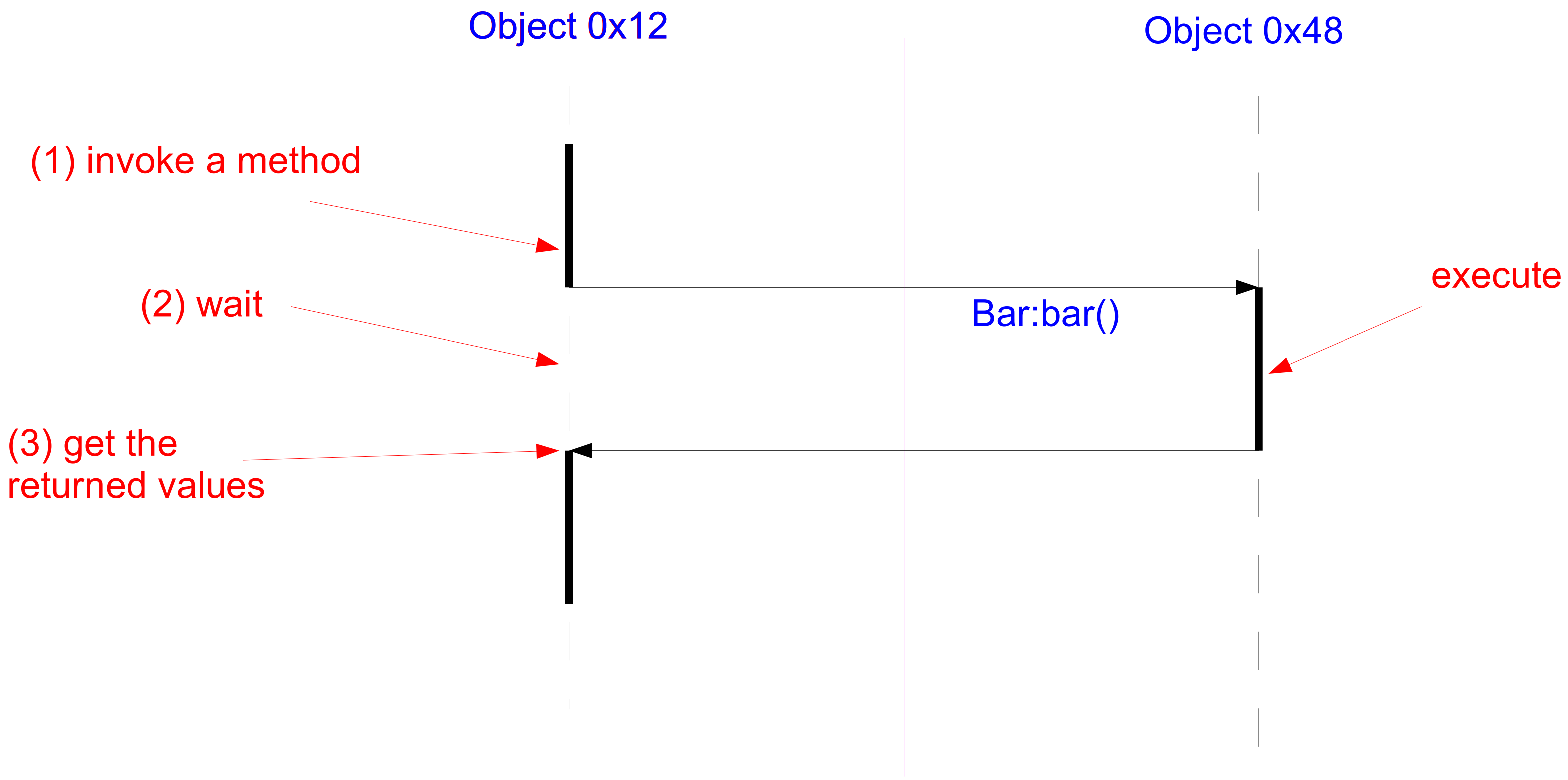






Olivier.Gruber@inria.fr

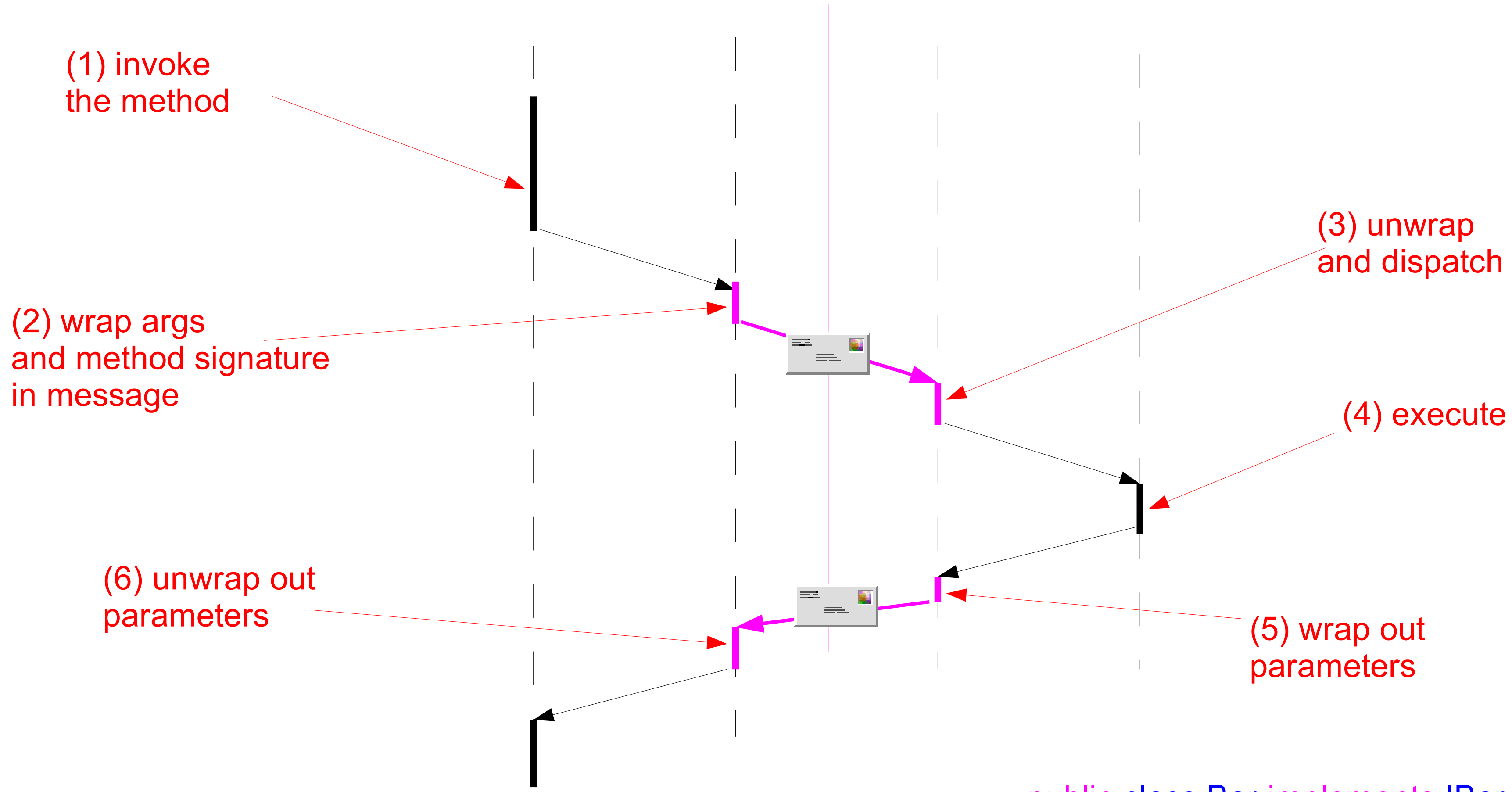
Remote Method Invocation




```
public class Foo {
  IBar m_bar;
  public int foo() {
    return m_bar.bar(2);
  }
}
```

```
public class Bar implements IBar {
  int m_value;
  public int bar(int factor) {
    return factor*m_value;
  }
}
```

Discussing Programming Models

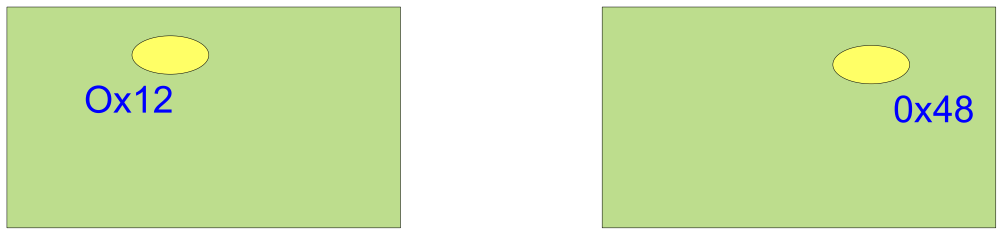


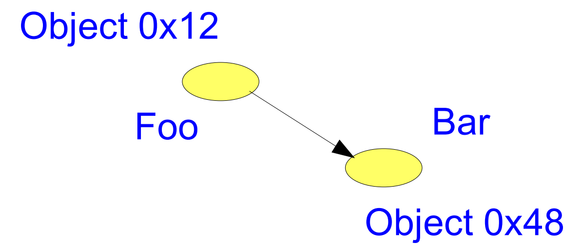

```
public class Foo {
  IBar m_bar;
  public int foo() {
    return m_bar.bar(2);
  }
}
```

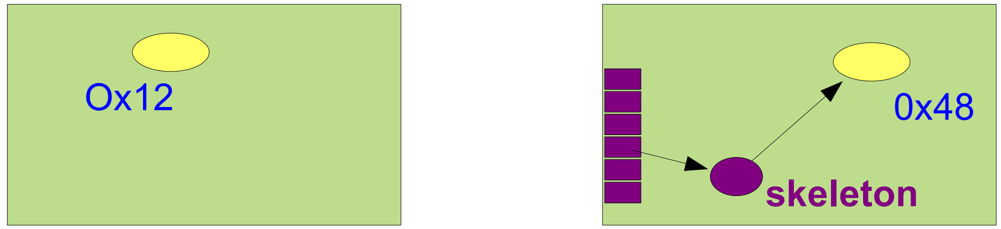
```
public class Bar implements IBar  {
  int m_value;
  public int bar(int factor) {
    return factor*m_value;
  }
}
```

Java Platform - RMI

- Creating a remote reference
  - Passed as arguments or returned



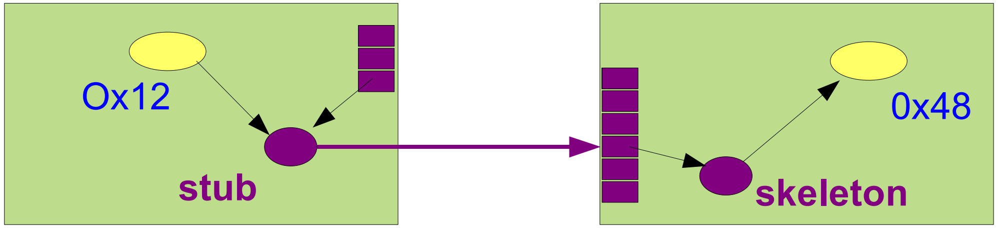




Create skeleton
Allocate skeleton id (skelid)

Skelid is a key to
a local table of skeletons

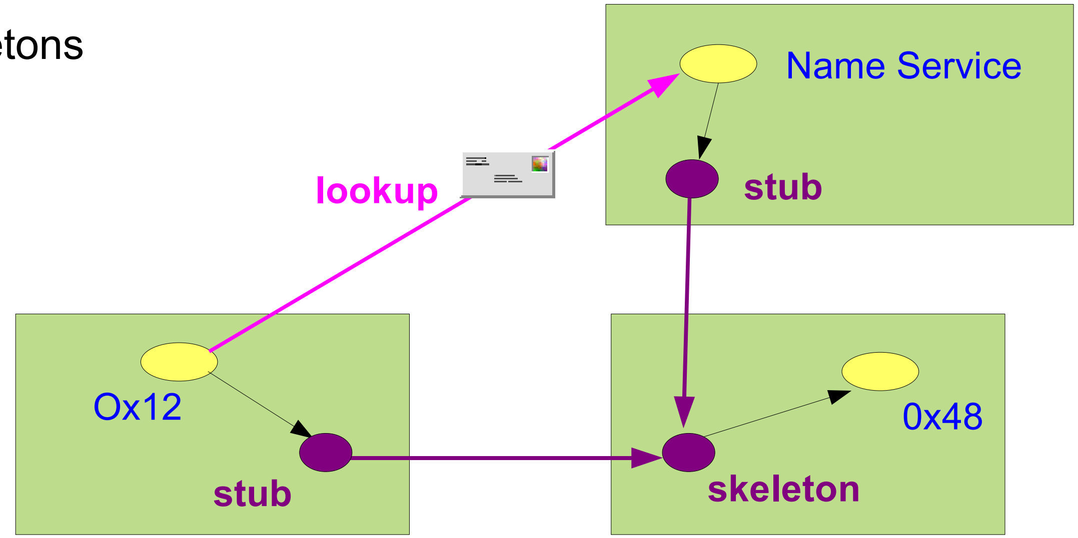
Create stub
Knows the IP, port, and skelid
Indexed in a local table of stubs



Olivier.Gruber@inria.fr

Java Platform - RMI

- Why stub and skeleton tables?
  - Avoids re-creating stubs or skeletons for an object if they exist already
  - Avoids overloading the DGC and the local tables
  - Faster lookups to find a skeleton or a stub
- Use naming service to bootstrap
  - How do we get the first remote reference?
    - Use a name service binding names to remote objects
    - Already uses stubs and skeletons
  - So, how do we bootstrap?



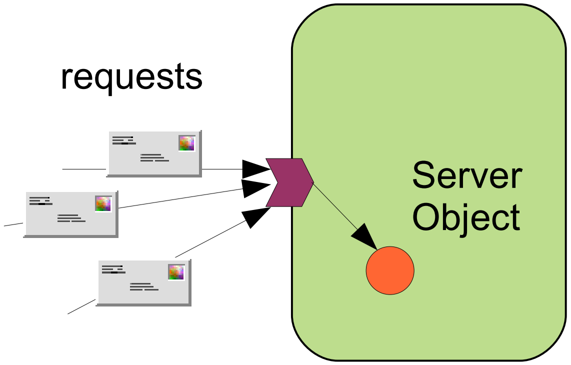
Java Platform - RMI

- Looking at a simple example
  - A simple bank is a remote object, that manages account objects
    - Can either extend the UnicastRemoteObject
  - Account objects are also remote objects, allowing simple operations
    - See the balance of the account, credit or debut money
- Discussing the example
  - About threads
    - Which threads are used to carry remote invocations?
    - Why didn't the server or client process stop at the end of main?
  - Service contracts
    - Why use an interface for typing the remote reference to the bank?
  - The account remained a remote object...
    - What about strings? Aren't they objects?

RMI – Execution Model

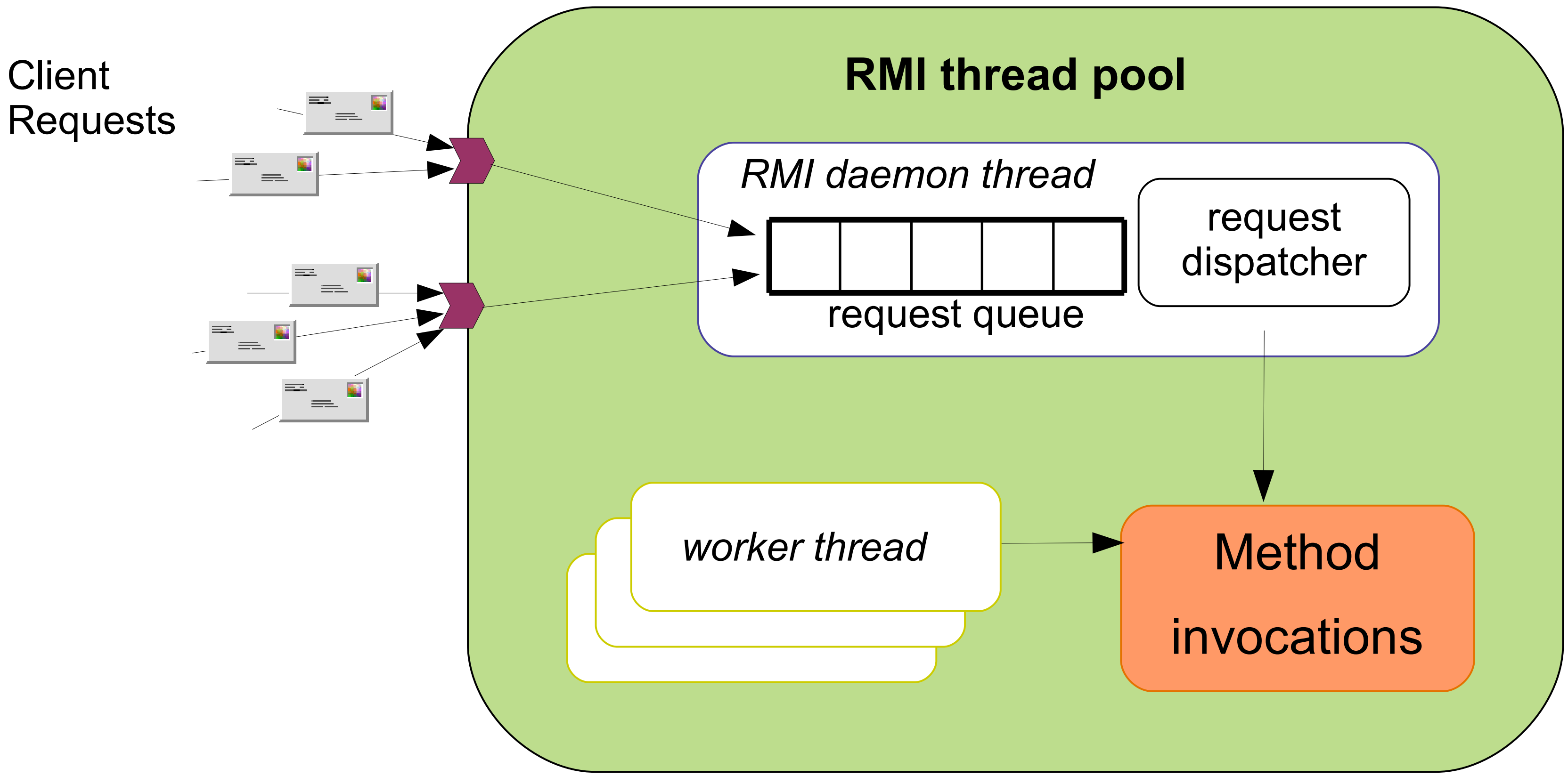
- Multi-threaded execution model
  - Server objects may be invoked from several clients
    - Method invocations happen in parallel
  - Server objects must be developed assuming multiple threads
    - Use synchronized methods
    - Use synchronized blocks

- RMI thread pool
  - Manages a pool of threads
  - Pick one thread to carry one invocation



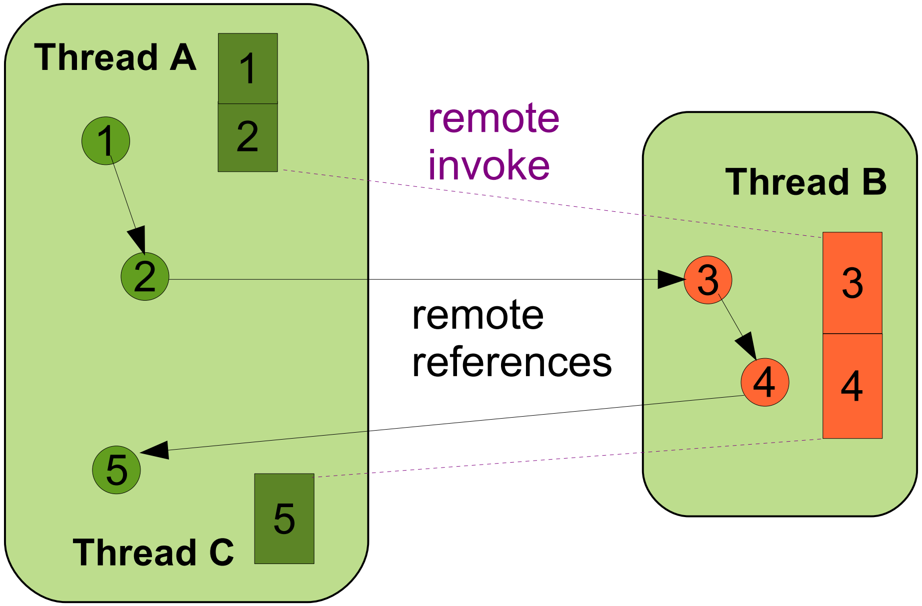
RMI – Execution Model

- Thread pool details



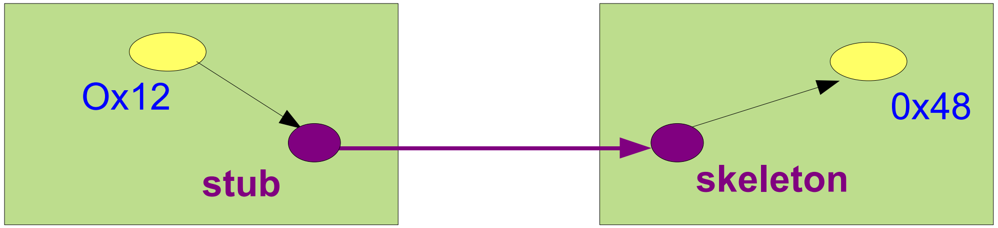
RMI – Execution Model

- The loopback problem...
  - Typical of a callback pattern (like in our example) or a distributed cycle
  - RMI uses no concept of distributed thread
    - Thread A and Thread C are different Java threads
    - Wastes resources and yields a high probability of deadlocks



Java Platform – RMI Programming Model

- Remote objects are used through remote interfaces
  - All remote interfaces extend the java.rmi.Remote interface
  - All remote methods throw at least the java.rmi.RemoteException
- Remote objects
  - All remote objects extend the java.rmi.UnicastRemoteObject
- Why only remote interfaces?
  - Stubs are regular Java, no special support in the Java Virtual Machine
  - Stub classes implement the remote interfaces of their target object




```
public class Foo {
  IBar m_bar;
  public int foo() {
    return m_bar.bar(2);
  }
}
```

```
public class Bar implements IBar  {
  int m_value;
  public int bar(int factor) {
    return factor*m_value;
  }
}
```

Olivier.Gruber@inria.fr

Java Platform – RMI Programming Model

- Arguments and returned values or objects
  - Two semantics: by-value or by-reference
- Primitive types are always passed by-value
  - Primitive types are boolean, byte, char, short, int, float, double
- What about objects?
  - Can be either by-value or by-reference...

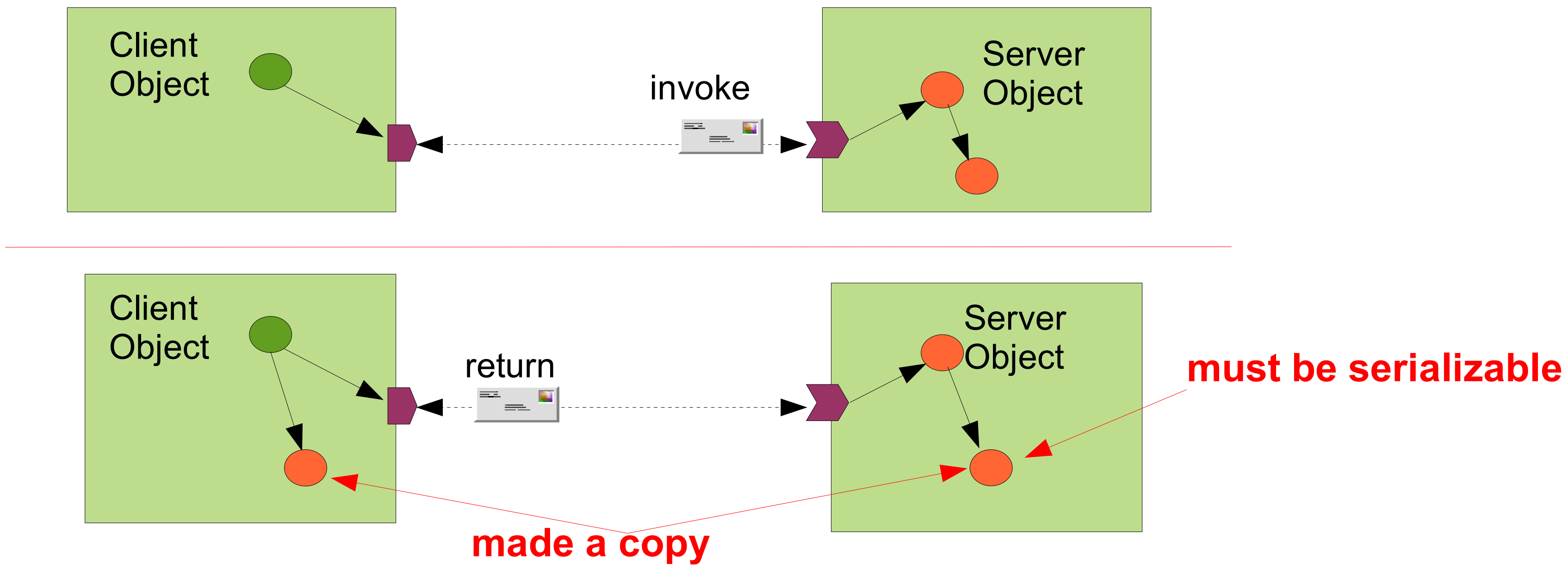
Olivier.Gruber@inria.fr

Java Platform – RMI Programming Model

- Objects by-value
  - Any object which is "serializable"
    - The class of the object implements java.io.Serializable
  - Copy semantics
    - Deep copy... yields two objects: both on server and client sides

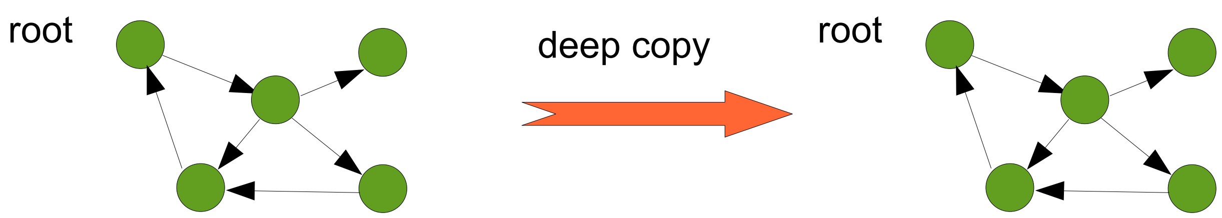
- An example – a simple method returning a reference

```
public Object getObject();
```



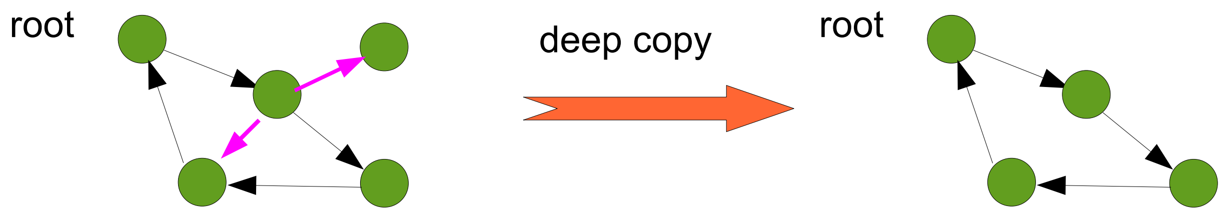
Java Serialization

- Deep copy
  - Recursive depth-first copy of an object graph from a root
  - If any object encountered is not serializable, an exception is thrown
  - Notice that cycles are properly handled



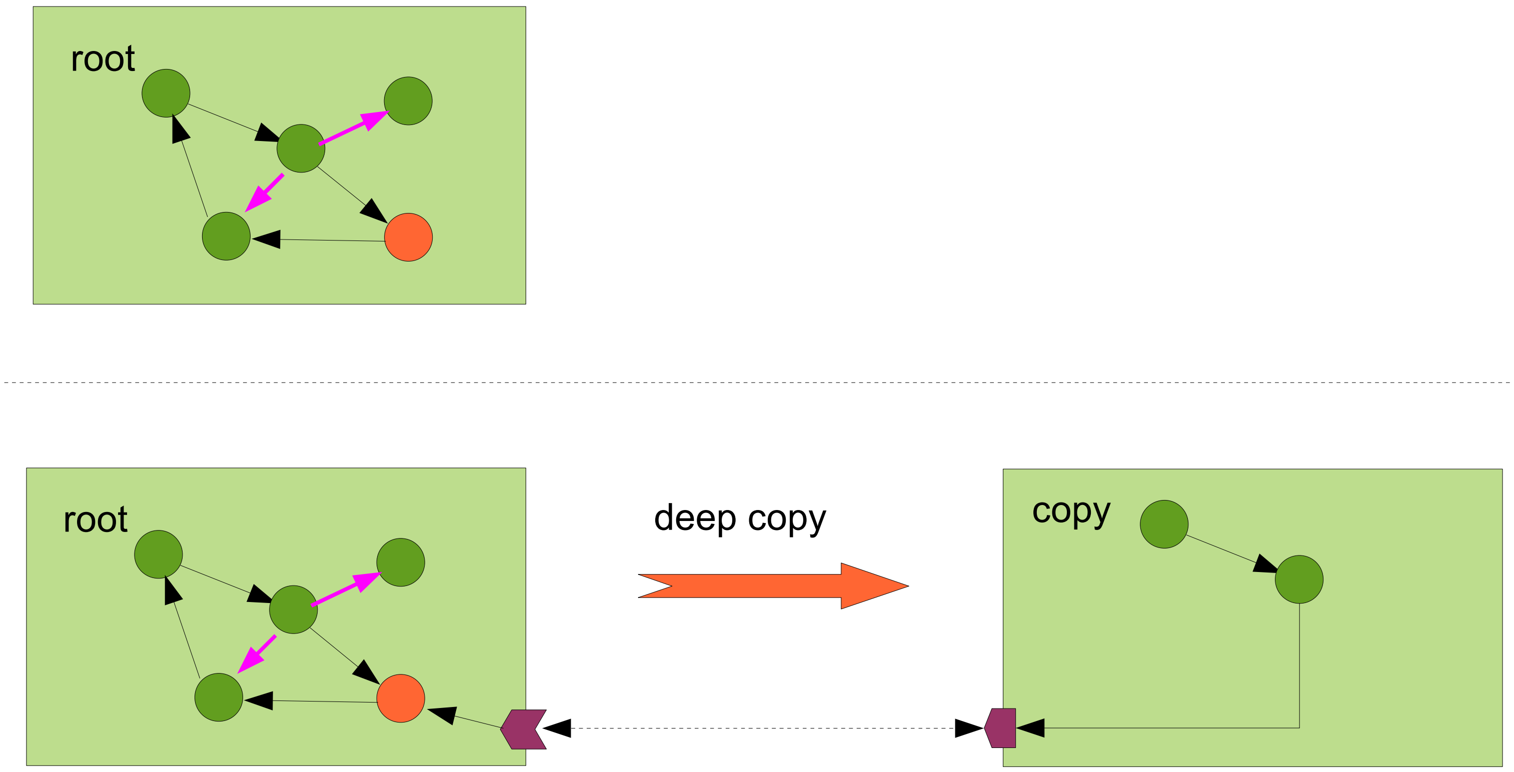
Java Serialization

- Individual object copy
  - By default, all instance fields are copied
  - **Transient modifier** on declared fields transient
    - Set to null when copied
    - Be mindful of sharing across transient and non-transient references
  - Attention
    - Static fields are part of the class
    - Not part of the instances of that class
    - **Therefore, static fields are not serialized**



Java Serialization

- Individual object copy
  - Reference to instances of RemoteObject are handler properly
    - Will create a stub-skeleton pair

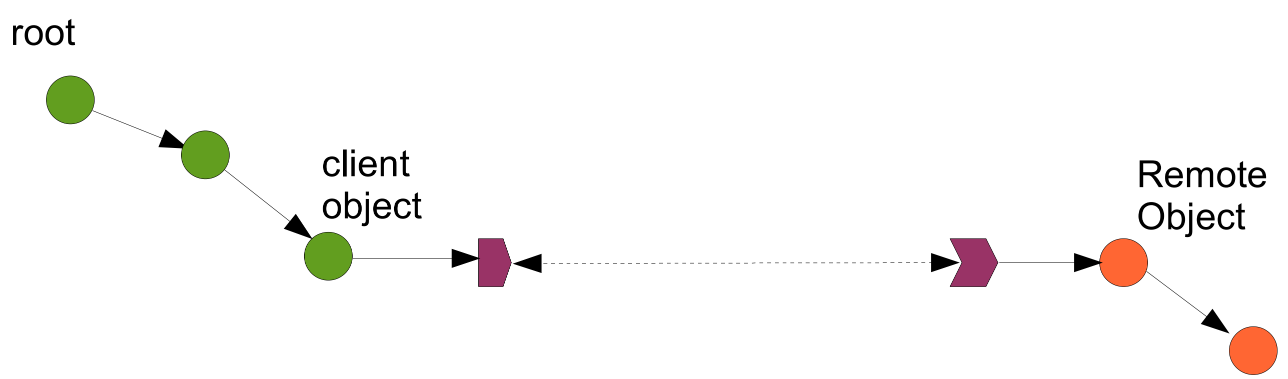


Java Serialization

- Java Runtime Environment
  - Most JRE classes are serializable
  - Their instances will be passed by value
- Examples
  - Java collections such as hash tables or vectors
  - String objects
  - Arrays are serializable objects
- Some classes are not serializable
  - Only make sense locally, such as files, sockets, threads, etc.

Distributed Garbage Collection

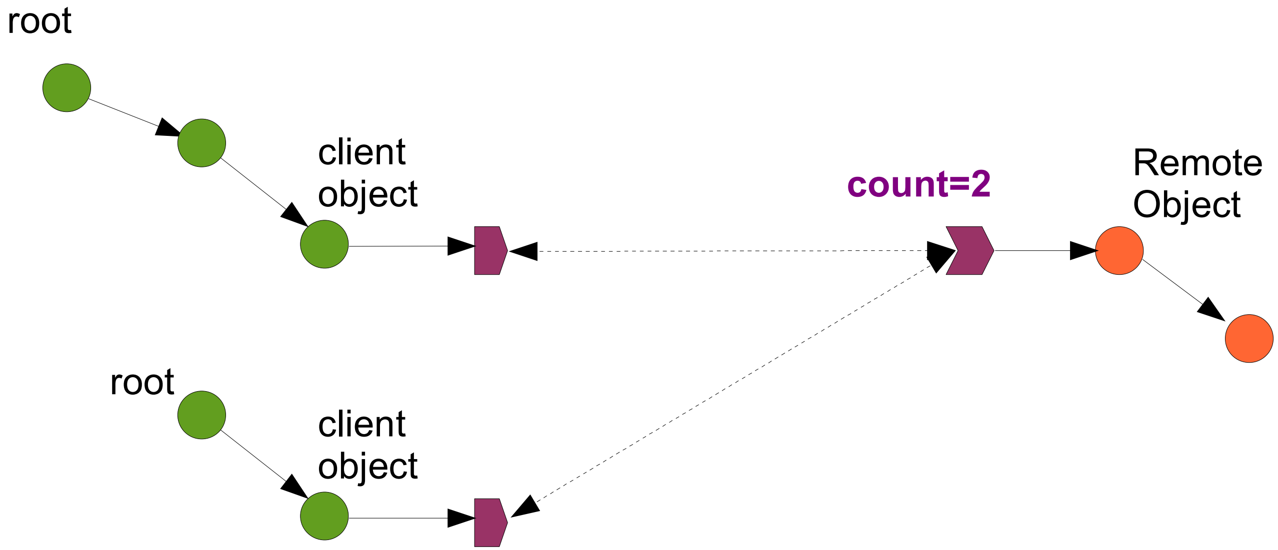
- Live objects
  - Locally or remotely reachable from roots
  - Natural extension to the local case
    - If a stub is reachable, so is the skeleton
    - If the stub is reachable, so is the remote object



Olivier.Gruber@inria.fr

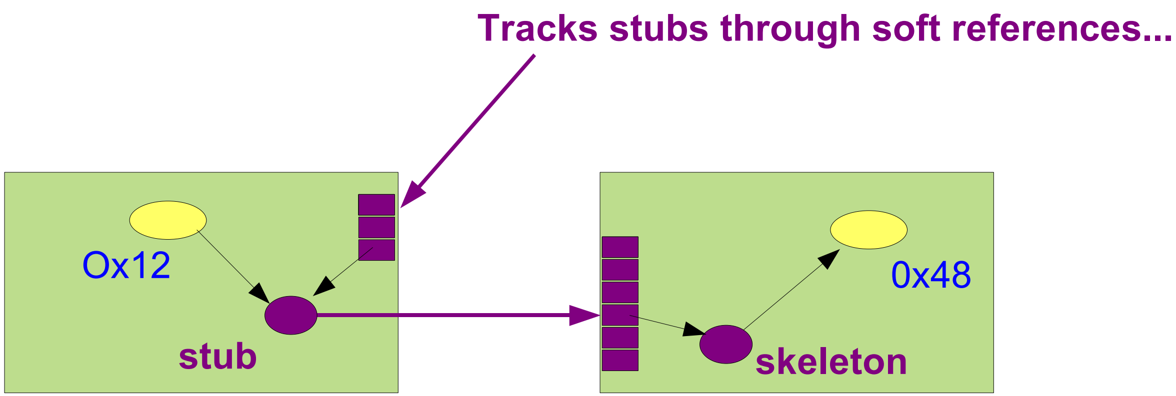
Distributed Garbage Collection

- A simple solution
  - Mixing reference counting and leases
  - Local garbage collectors are left unchanged
  - Counts stub references on skeletons
  - Uses lease to resist failures
    - Default lease is 10 minutes
    - RMI middleware renews leases at half-life



Distributed Garbage Collection

- Design
  - Tracks remote references through soft references
    - Renew leases as long as the soft references are not cleared
    - RMI middleware renews leases at half-life
  - Only increment/decrement remote counters when stubs are collected



Distributed Garbage Collection

- Watch for distributed cycles...
  - RMI DGC neither detects nor collects distributed cycles...