Distributed Systems based on Java RMI

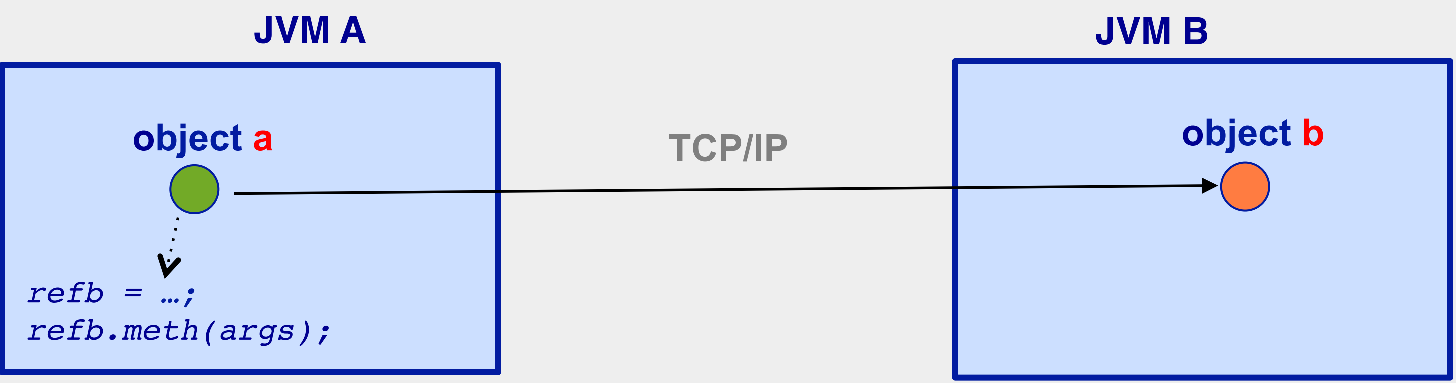
Polytech/INFO 4, 2022-2023
Fabienne Boyer, Olivier Gruber,
UFR IM2AG, LIG, Université Grenoble Alpes
Fabienne.Boyer@imag.fr

RMI Motivations

- **Sockets**
  - Message/Stream based communication
  - The programmer is in charge of
    - defining the communication protocol
    - marshalling parameters & result
    - determining the server's location
- **Remote Method Invocation (RMI)**
  - Object-based communication
  - For the programmer
    - just invoke a method on a remote object
    - location independent (except for naming service)

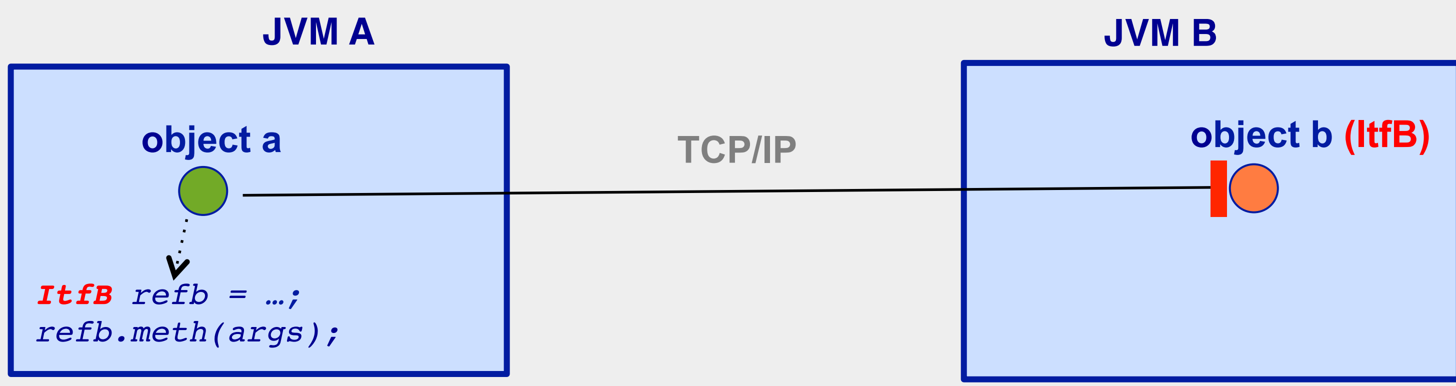
RMI's main principle : Indirection Objects

- An object *a* in JVM A can invoke another object *b* in JVM B



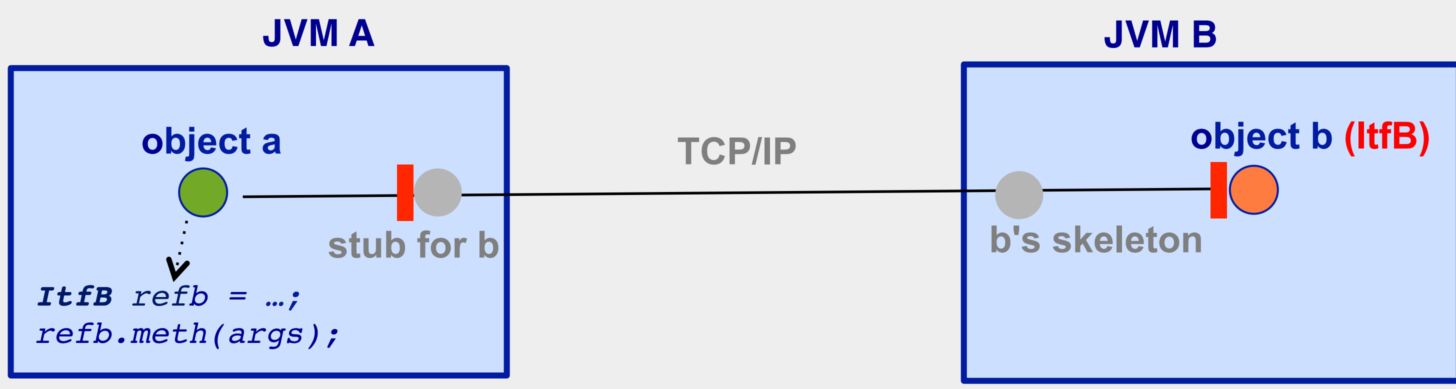
RMI's main principle : Indirection Objects

- An object *a* in JVM A can invoke another object *b* in JVM B through its remote interface



RMI's main principle : Indirection Objects

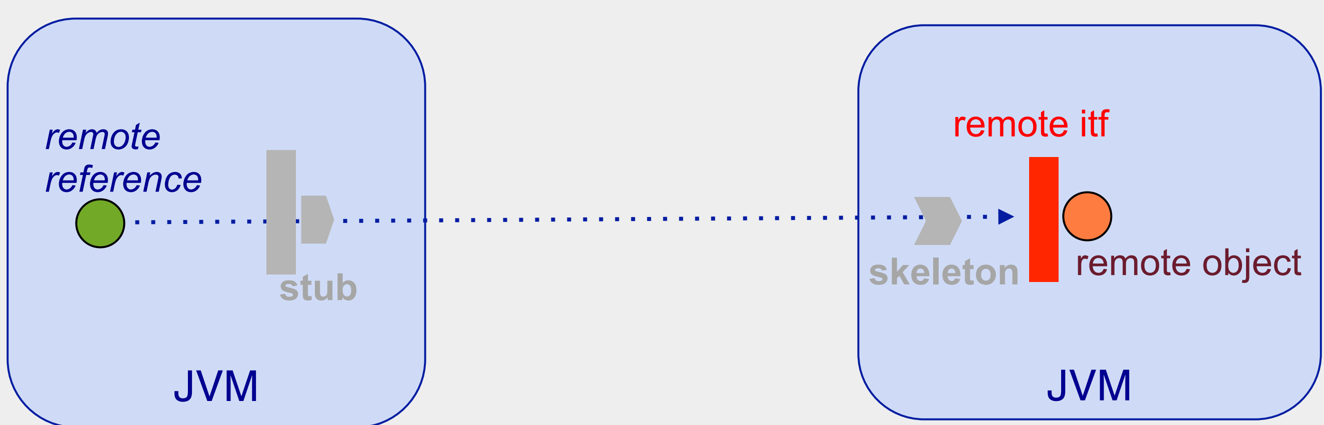
- An object *a* in JVM A can invoke another object *b* in JVM B
- This remote invocation relies on two underlaying objects (called indirection objects, or also stub/skeleton)
- These indirection objects are transparent to the programmer



[b is in fact areference to the stub]

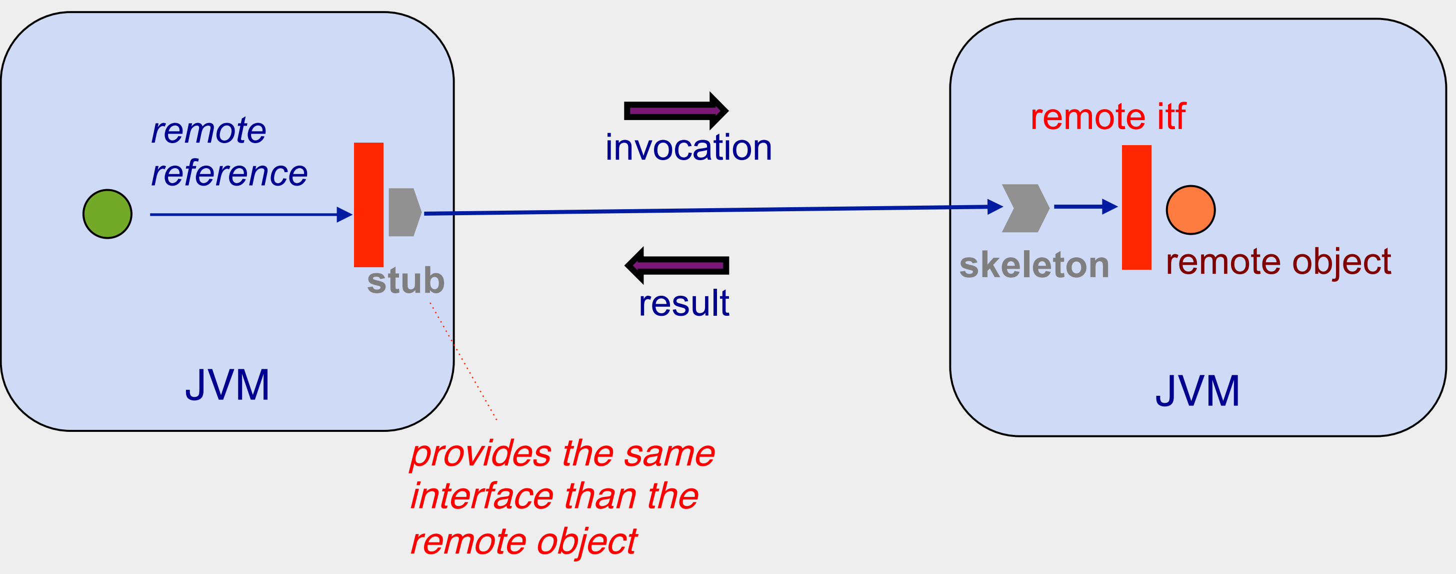
RMI definitions
remote object / remote reference / remote interface

- A remote object is an object that can be accessed remotely, meaning that its class implements at least one remote interface
- A remote interface is an interface that extends java.rmi.remote
- A remote reference is a reference toward a remote object



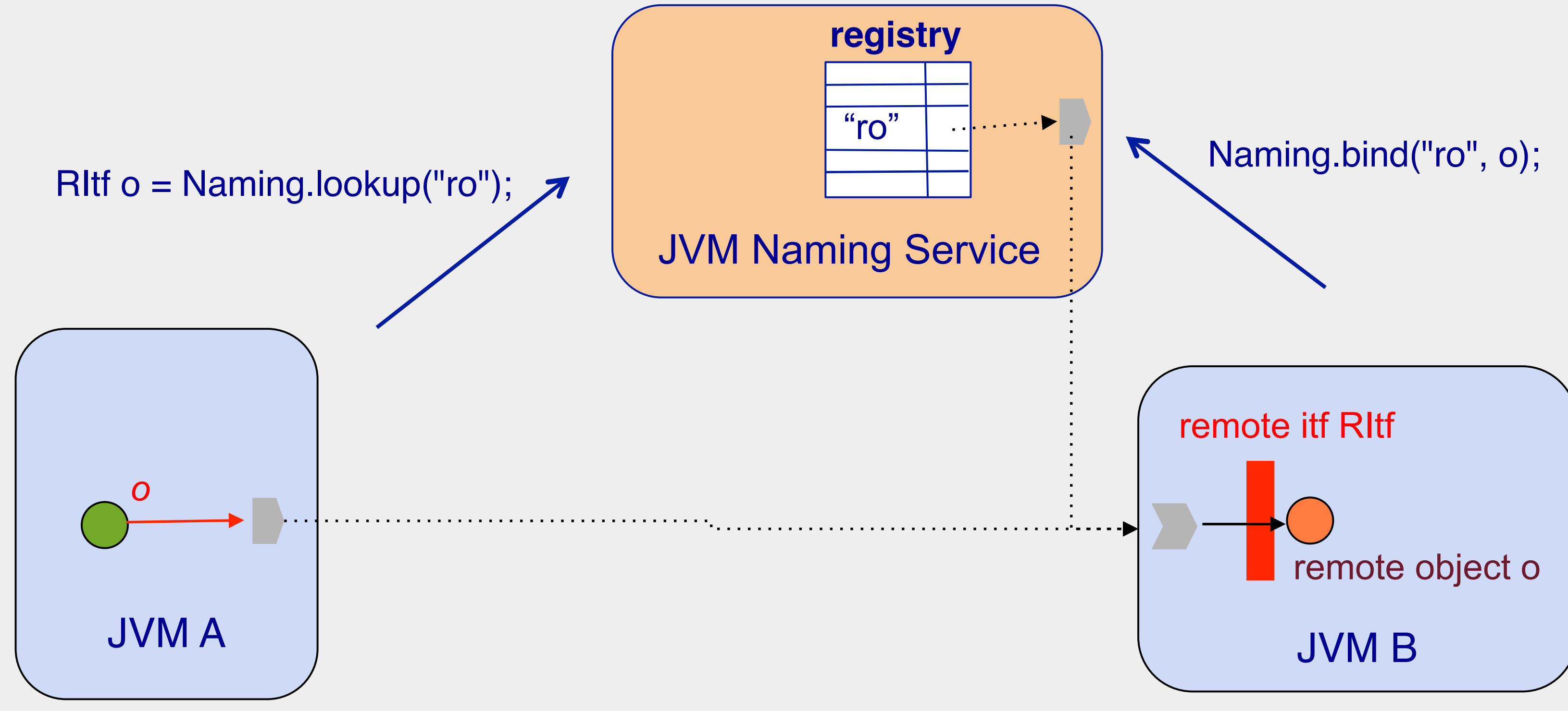
RMI invocations

- Invoking a method on a remote reference is in fact invoking a method on a stub
- The stub forwards the invocation to the skeleton
- The skeleton invokes the remote object and returns the result to the stub



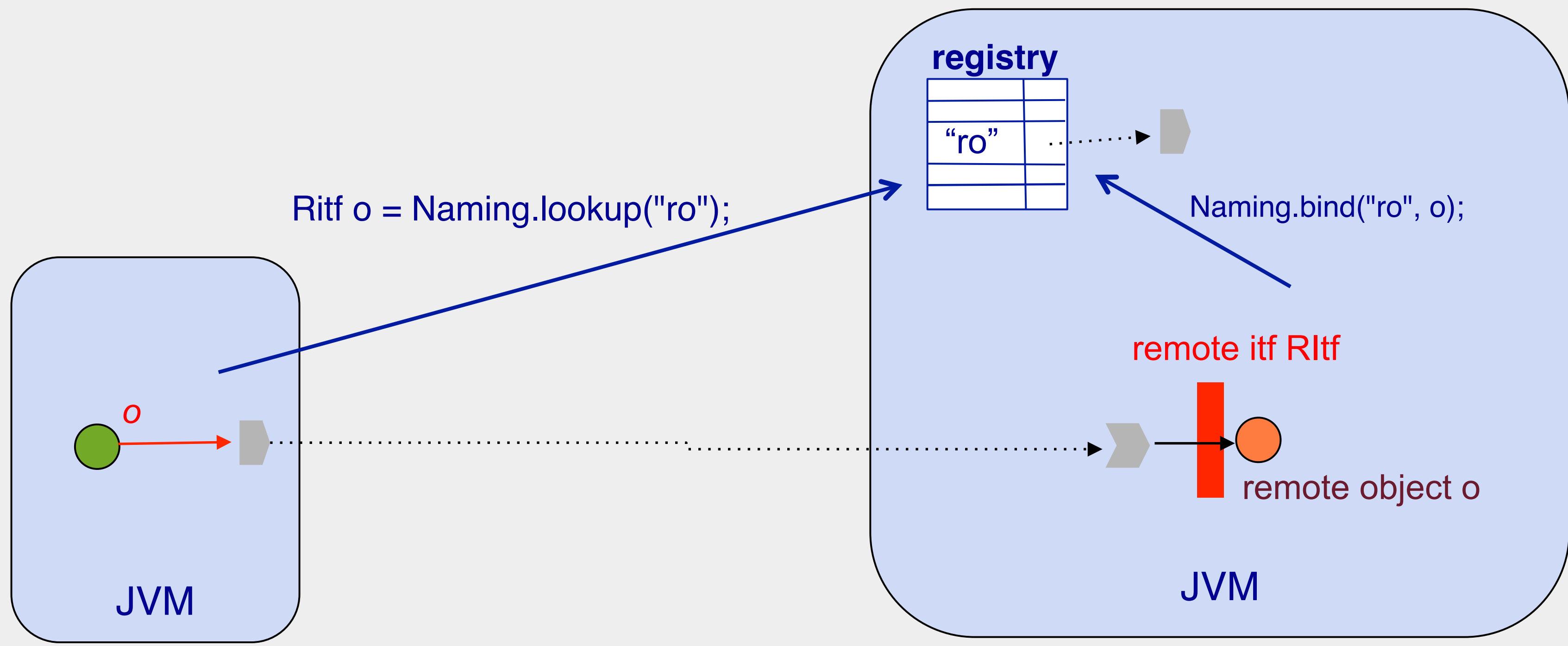
RMI Naming Service (Registry)
Allows to register / lookup a remote reference

- Launch the RMI Registry on a JVM associated to given (host, port)
- The JVM B register the reference of the remote object to the registry
- The JVM A gets the reference of the remote object through the registry



RMI Naming Service (Registry)
Allows to register / lookup a remote reference

- The RMI Registry can also be launched "inside" JVM B



RMI Name Service (rmiregistry)

- Embedded Registry
  - Class LocateRegistry

    static Registry createRegistry(int noport)

    static Registry getRegistry(String host, int noport)

    Class Registry provides methods *bind(String name, Remote Object)* & *Remote lookup(String name)*
- In any case

  Class Naming provides static methods *bind(String url, Remote r) & Remote lookup(String url)*
  - Should specify the url of the remote object

    //goedel.imag.fr:2001/Examples/WS

    //goedel.imag.fr:2001/WS

    //:2001/WS *(localhost by default)*

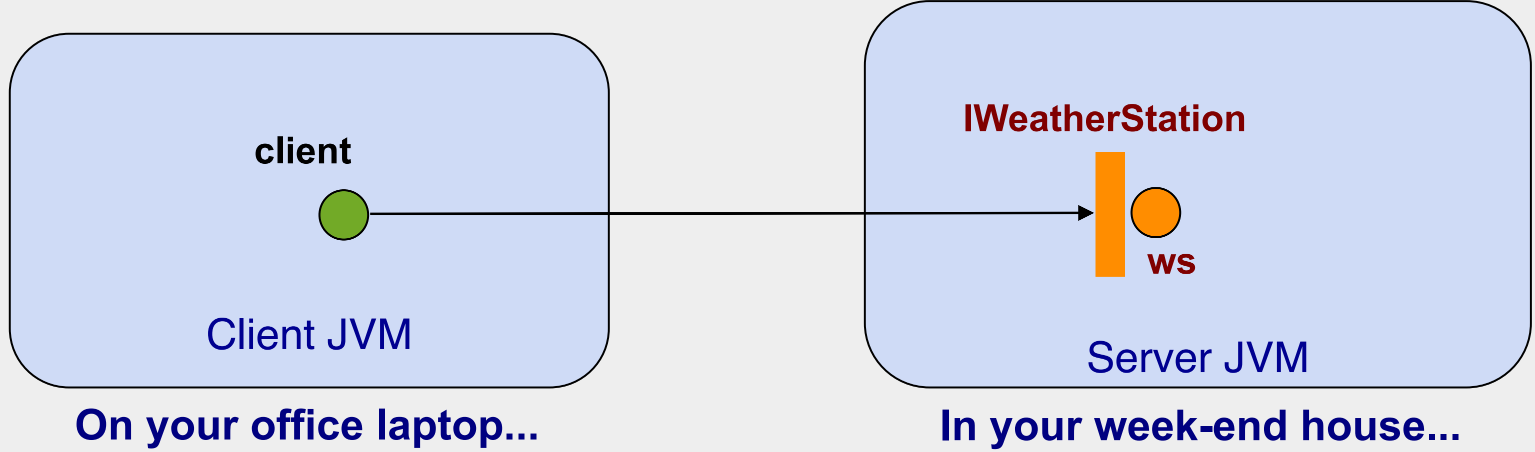
    //:WS *(localhost and port 1099 by default)*

Let's consider a simple example

Weather station

- You just bought a weather station for your week-end house
- It provides remote methods to access the weather sensors

```
public interface IWeatherStation extends Remote {
 int getTemperature(int unit) throws RemoteException ;
 int getWindSpeed() throws RemoteException  ;
 String getWindDirection() throws RemoteException  ;
}
```



Simple Example: server side

```
class WeatherStation implements IWeatherStation extends UnicastRemoteObject {
  public int getTemperature(int unit) throws RemoteException {..}
  public int getWindSpeed() throws RemoteException {..}
  public String getWindDirection() throws RemoteException {..}
}
```

```
// SERVER side
 static void main(String argv[]){
  IWeatherStation ws = new WeatherStation();
  Registry registry = LocateRegistry.createRegistry(9999);
  registry.bind("WS", ws);
}
```

Notice that the server code ends there, but the server will stay alive while some of its objects are reachable from remote clients (internal RMI feature)

Simple Example: client side

```
// CLIENT side
 static void main(String argv[]){
  Registry registry = LocateRegistry.getRegistry("myweekendhost.xxx.yy", 9999);
  IWeatherStation ws = (IWeatherStation) registry.lookup("WS");
  int temp=ws.getTemperature(..);
  String dir=ws.getWindDirection();
  int speed=ws.getWindSpeed();
  System.out.println("Temp="+temp+" Wind speed="+speed+" Direction="+dir);
}
```

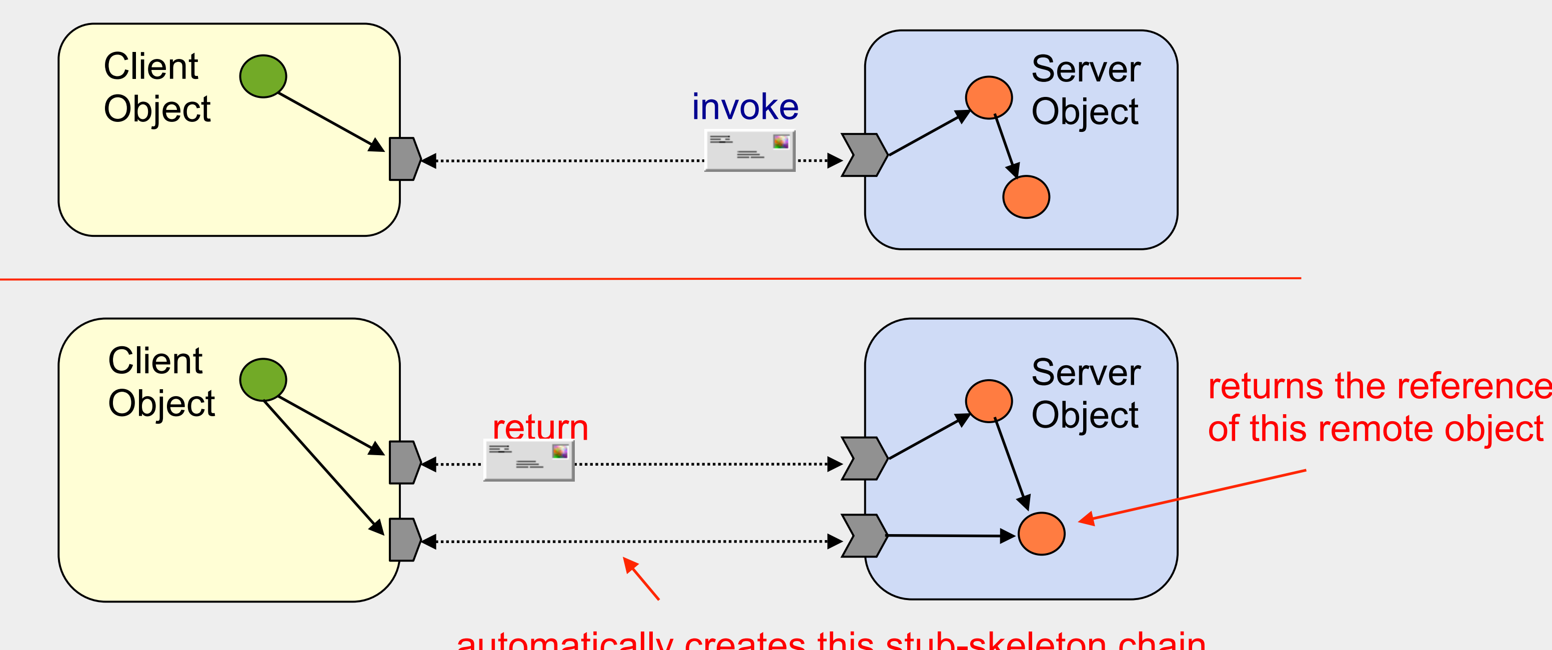
Passing Arguments

- **Primitive types**
  - boolean, byte, char, short, int, float, double
  - Always transferred by value
- **What about objects?**
  - Can be either by-value or by-reference
    - By-value means a copy
    - By-reference means no copy ***
  - Applies to arguments and to results

*** this is a way to get remote references: through arguments or results of remote invocations

Objects passed by-reference

- **All objects whose classes implement Remote**
- **Creates a stub-skeleton chain**



Back to our example

- **Introduce weather stats**
  - We introduce a logger object that periodically collect weather information
  - The logger is a remote object, so it can be returned to the client by-reference
  - Once the client has get the logger reference, it can invoke it

Back to our example

```
public interface IWeatherStation
extends Remote {
int getTemperature(int unit)
throws RemoteException ;
IWeatherLogger getStatLogger()
throws Remote Exception;
..
}
```

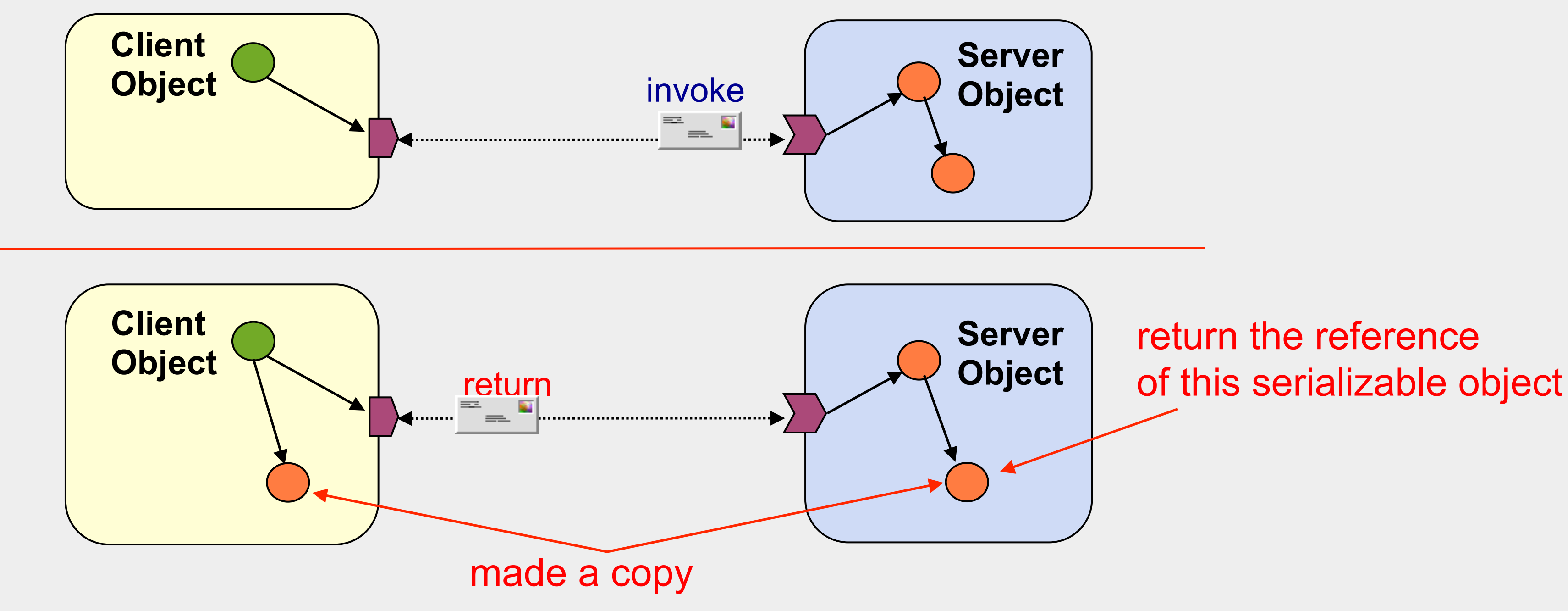
```
class WeatherStation implements
IWeatherStation extends UnicastRemoteObject{
WeatherLogger logger;
public WeatherStation(){
  logger = new WeatherLogger(this);
}
public IWeatherLogger getStatLogger() throws
RemoteException {
 return logger;  // passed by reference
}
...
```

```
public interface IWeatherLogger
extends Remote {
  ..
}
```

```
class WeatherLogger implements
IWeatherLogger extends
UnicastRemoteObject{
 ..
}
```

Objects passed by-value

- **All objects whose classes implement Serializable**
- **Copy semantics**
  - Yields at least two objects: on server and client

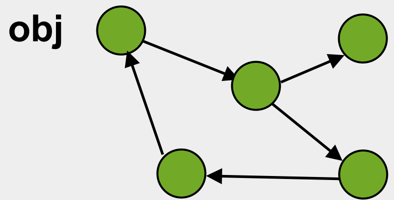


Serialization

- **Convert an object into a sequence of bits**
  - Can be persisted on a storage medium (such as a file)
  - Can be transmitted across the network

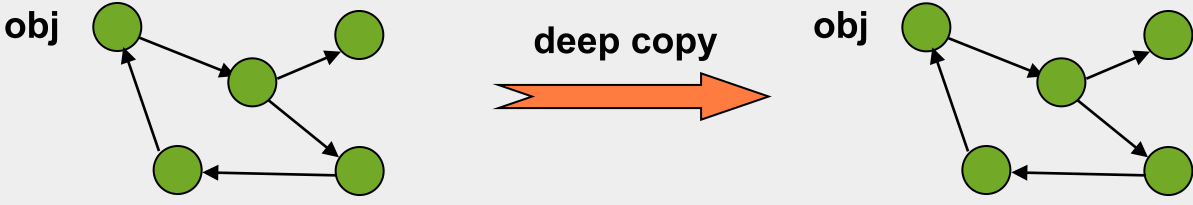
Serialization

- **Convert an object into a sequence of bits**
  - Can be persisted on a storage medium (such as a file)
  - Can be transmitted across the network



Serialization

- **Convert an object into a sequence of bits**
  - Can be persisted on a storage medium (such as a file)
  - Can be transmitted across the network
- **Recursive deep copy**
  - By default, all fields are copied except those declared transient
  - If any referenced object is not serializable, an exception is thrown (unless the reference is declared transient)
  - Cycles are properly handled



Serialization example

```
..
FileOutputStream fos = new FileOutputStream(“myFile.txt");
ObjectOutputStream oos = new ObjectOutputStream(fos);
MyObject obj = new myObject();
oos.writeobject(obj);
oos.close();
..
FileInputStream fis = new FileInputStream(“myFile.txt");
ObjectInputStream ois = new ObjectInputStream(fis);
MyObject obj = (MyObject) ois.readObject();
..
```

Serialization

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

Back to our example

- **Improving performance**
  - Reducing the number of remote method invocations
  - Return a WeatherData object that gathers all information (wind & temperature)
  - Passed by-value

Back to our example

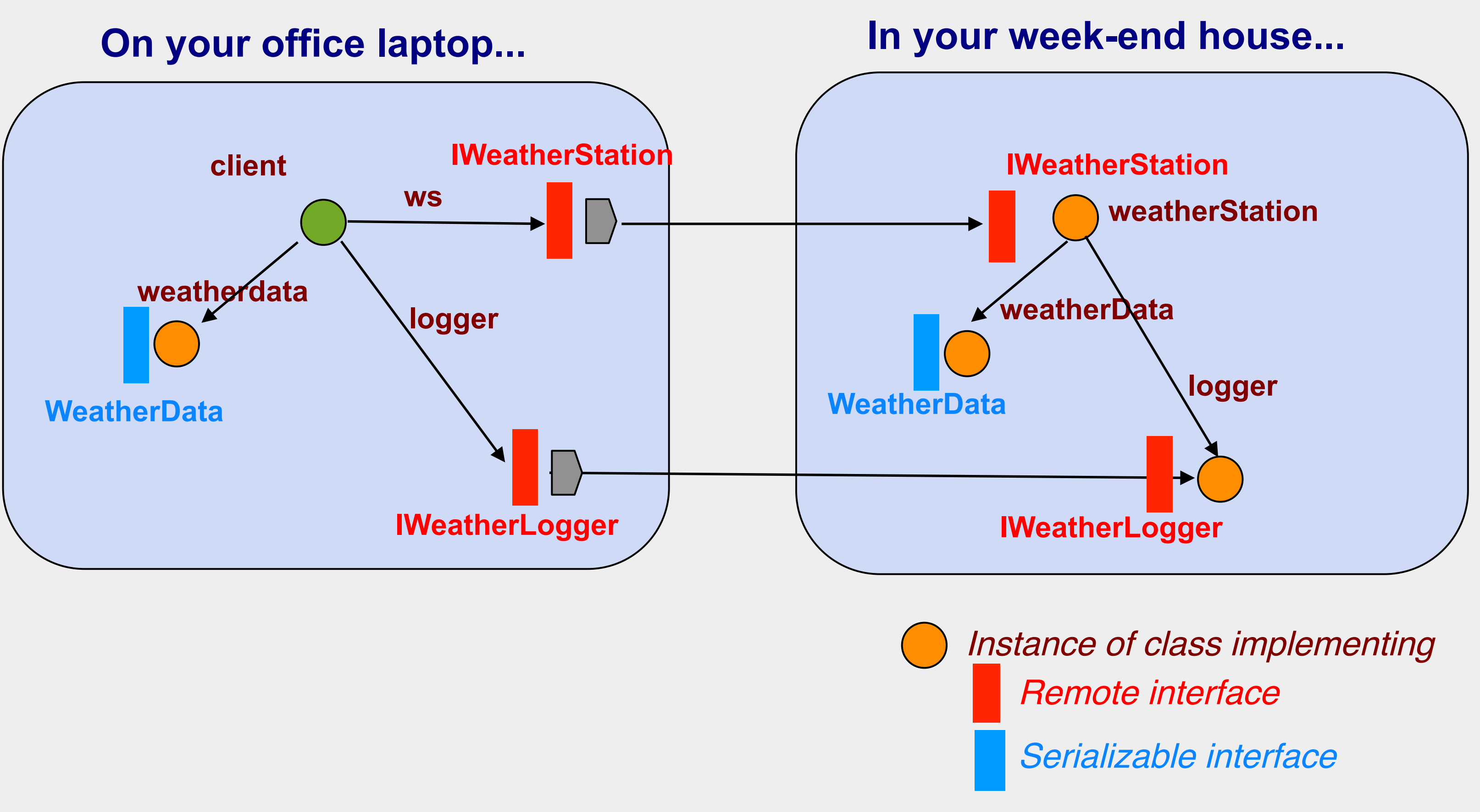
```
public interface IWeatherStation
extends Remote {
int getTemperature(int unit)
throws RemoteException ;
WeatherData getData() throws
Remote Exception;
..
}
```

```
class WeatherStation implements
IWeatherStation extends UnicastRemoteObject{
..

Public WeatherData getData() throws
RemoteException {
 return new WeatherData(temp,wind,dir);
                            // passed by value
}
```

```
public class WeatherData
implements Serializable{
 int getTemperature();
 ..
}
```

WeatherStation global picture



RMI: additional concerns

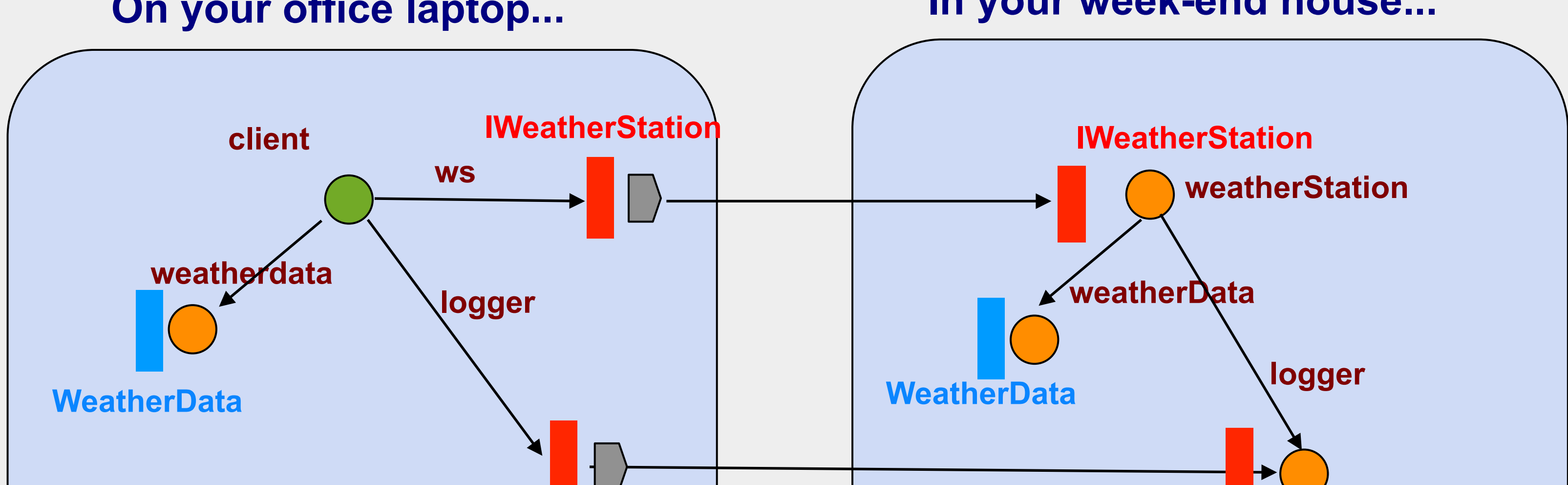
- **Packaging**
  - What classes and interfaces should be deployed at client side, at server side?
- **Execution model**
  - Which thread performs a remote invocation?
- **Garbage aspects**
  - An object may be no more referenced locally but still referenced remotely..
- **VM lifecycle**
  - Does a VM exit as soon as all non-daemon threads terminate?

Packaging distributed applications

- **Clients must have access to**
  - Interfaces for remote objects
  - Implementation classes for objects received by-value **
- **Server must have access to**
  - Interfaces & classes for (server-side) remote objects
  - Implementation classes for objects received by-value **

** A JVM should have access to the class of an object to deserialize

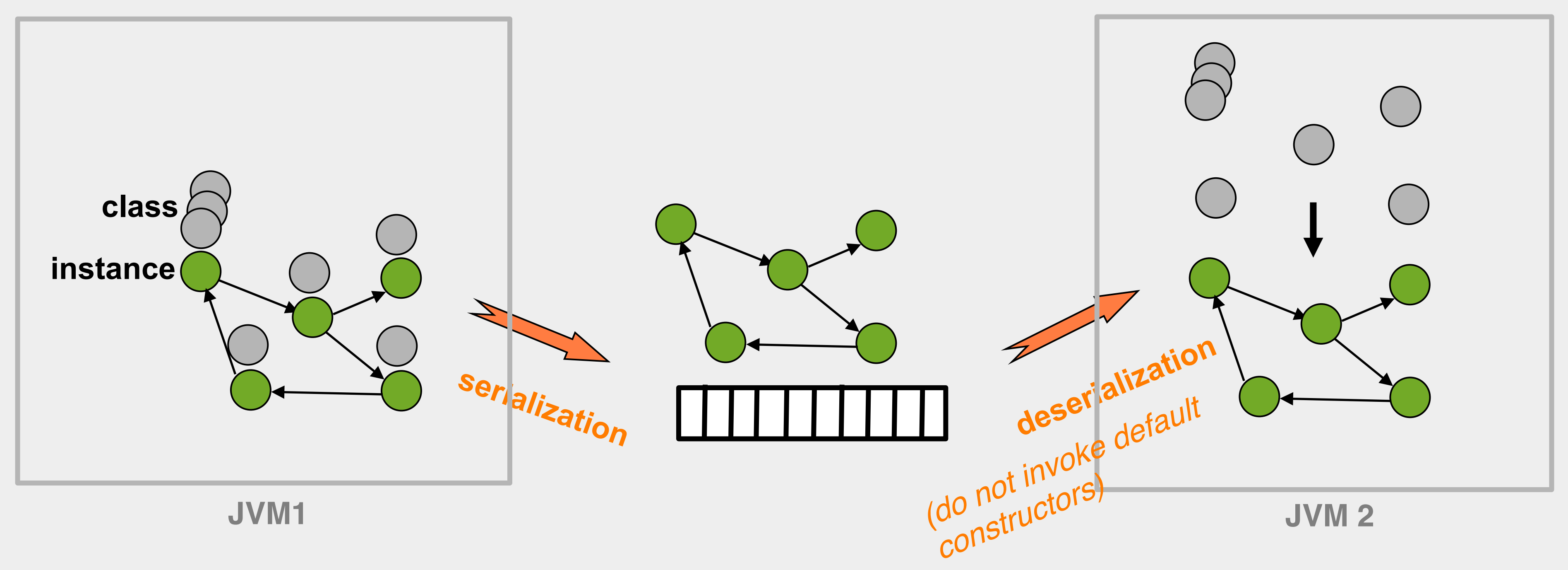
Packaging the WeatherStation



- At client side:
  - IWeatherStation, IWeatherLogger
  - WeatherData
- At server side:
  - IWeatherStation, IWeatherLogger
  - WeatherStation, WeatherData, WeatherLogger

Packaging distributed applications

- **JVM should have access to classes for received by-value objects**
  - Why? Because Java serialization only serializes the state of objects, not their classes



Packaging distributed applications

- **How can the JVM be sure that a class is the right class for a received by-value object?**
  - serialVersionUID: a kind of checksum that identifies the compatibility of the class for a given object

```
class WeatherData implements Serializable{
 private static final long serialVersionUID = 1190386516911681470L;
 int Temperature;
 int wind;
 String windDirection;
 ...
}
```

Dynamic class download

- A class may be dynamically downloaded from a given url (given as *codebase* option when launching the server)
- Useful feature when
  - The class of an object received by-value is unknown by the receiver
  - The receiver only knows a super-class

Examples

```
java … -D java.rmi.server.codebase=http://mywebsite.com/classes/
weatherstation.jar WeatherServer
```

```
java … -D java.rmi.server.codebase=ftp://myname@mywebsite.com:/classes/
weatherstation.jar WeatherServer
```

Dynamic class download

- Security MUST be enabled for dynamic code downloading
- Security is configured in .policy file

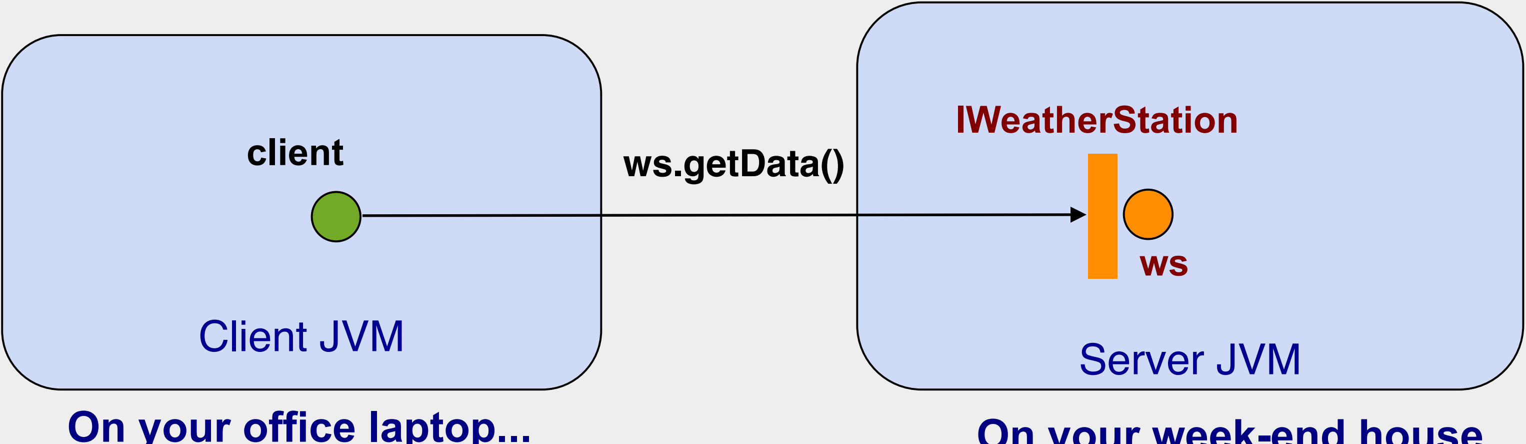
```
java -Djava.security.manager -Djava.security.policy=./weatherstation.policy
-Djava.rmi.server.codebase=http://mywebsite.com/classes/weatherstation.jar
WeatherServer
```

Example of file weatherstation.policy:

```
grant { permission java.net.SocketPermission "*:1024-65535","connect,accept";
        permission java.net.SocketPermission "*:80", "connect";
      // permission java.security.AllPermission; /* full authorization*/ };
```

Execution Model

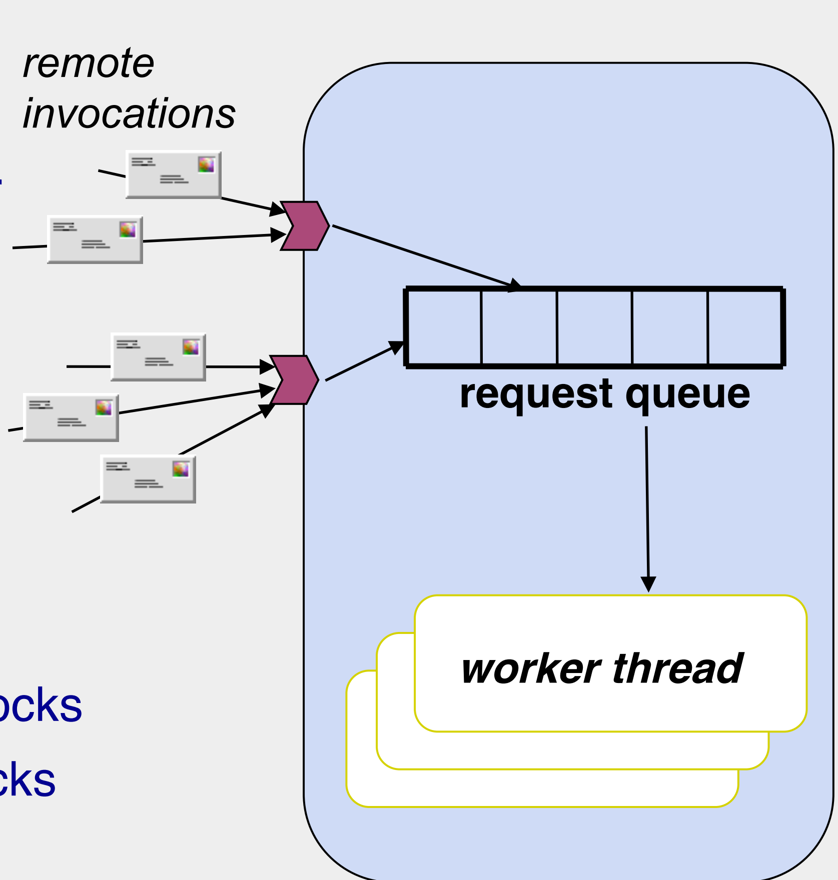
- **Which thread processes a remote invocation?**




```
// SERVER side
 static void main(String argv[]){
  IWeatherStation ws = new WeatherStation();
  Registry reg = LocateRegistry.createRegistry(9999);
  reg.bind("WS", ws);
}
```

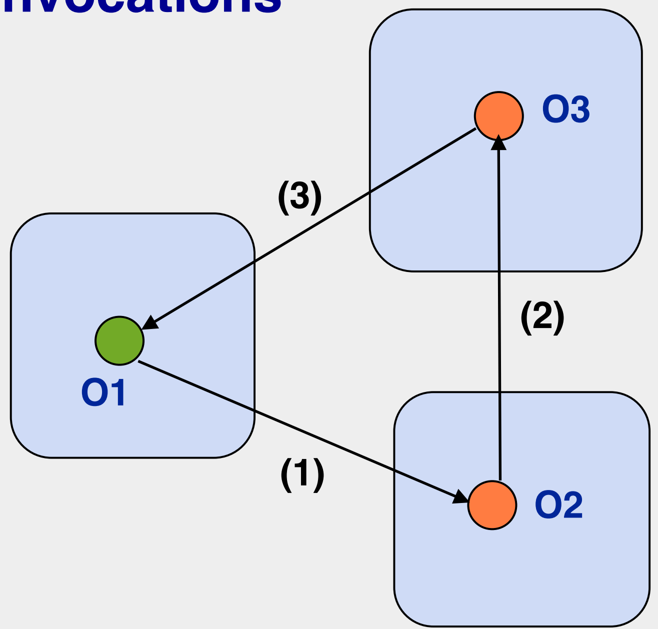
Execution Model

- **RMI thread pool**
  - Entirely managed by the RMI layer
  - Pick one thread to carry one invocation
- **Concurrent execution model**
  - A remote object may be invoked concurrently from several clients
    - Use synchronized methods or blocks
    - Pay attention to potential deadlocks (no remote reentrance)



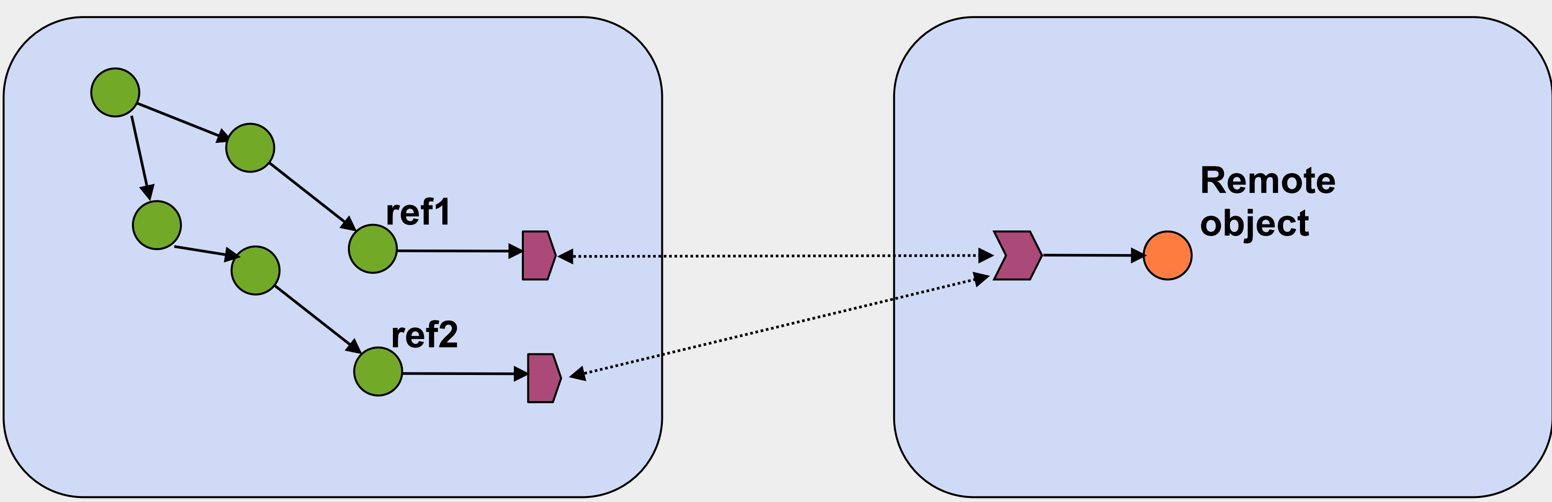
Execution Model

- **Remote objects are often implemented as monitors**
  - All methods are synchronized
- **Care about deadlocks in case of cyclic invocations**
  - Consider the figure
  - At least 2 threads executing in o1
    - The initial one invoking o2
    - The one invoking o1 from o3
- **Reduce synchronized sections**
  - Manage deadlock issues
  - Better performances



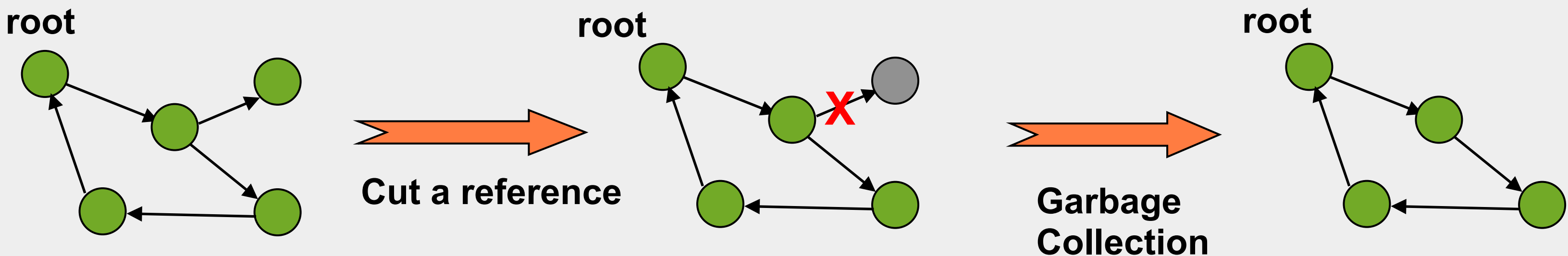
Object model

- **Stub unicity is not ensured**
  - Two objects in a same JVM referencing a same remote object may use distinct stub objects
  - Pay attention to use *ref1.equals(ref2)* to compare remote references



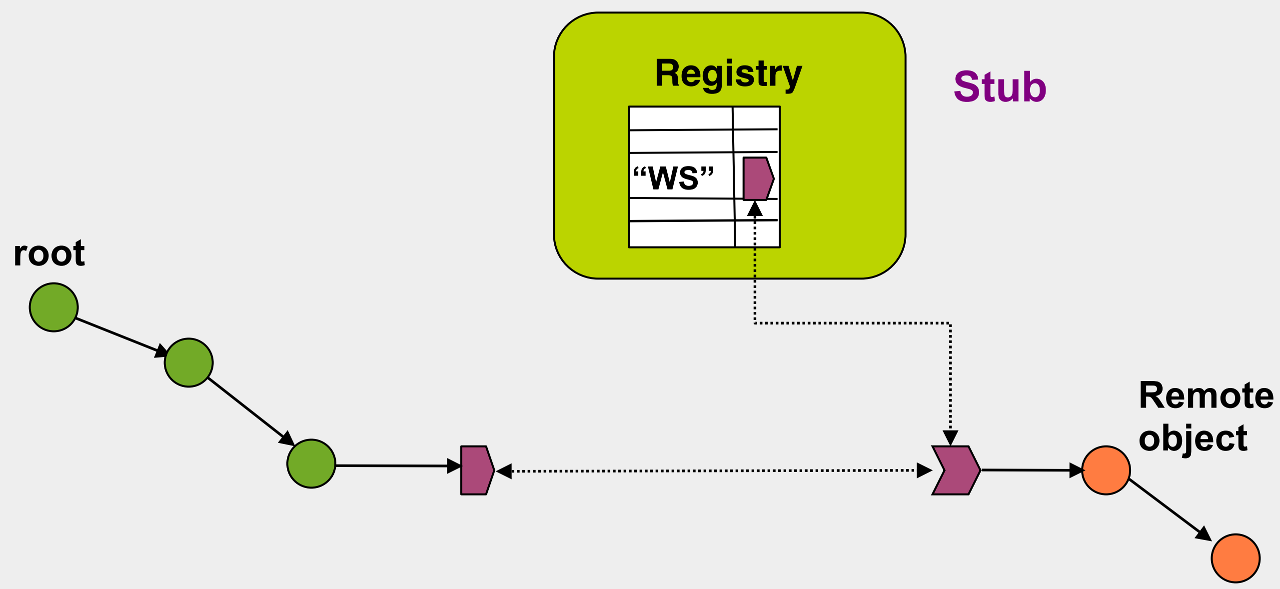
Garbage Collection

- **Local garbage collection**
  - Java is a garbage collected language
    - An object is garbage when it is no longer reachable from roots
    - Roots are thread stacks and class statics
  - The garbage collector detects and recycles garbage objects
    - This is done automatically and periodically



Garbage Collection

- **Distributed garbage collection**
  - Natural extension to the local case
  - If a stub is reachable, so is the skeleton, and so is the remote object
  - So any remote object referenced by the RMI Registry is reachable



Distributed Garbage Collection

- **Implementation Principles**
  - A skeleton keeps a reference counter (*ref-count*)
  - When a stub enters a VM, the skeleton is informed (*ref-count++*)
  - The VM has to ping the skeleton periodically
  - When the GC frees a stub, the finalize() method informs the skeleton (*ref-count--*)
  - When the reference counter is null (*ref-count==0*), the skeleton informs the GC
  - The GC frees the object if it is no more referenced locally

Execution model - VM lifecycle

- **New termination rule: a VM exits when**
  - All non daemon threads terminate
  - There are no more objects that are remotely referenced

```
// Weather station - SERVER side
 static void main(String argv[]){
  IWeatherStation ws = new WeatherStation();
  Registry registry = Registry.createRegistry();
  registry.bind("WS", ws);
  // the VM will stay alive until ws is no more referenced remotely
}
```