Java Remote Method Invocation
Parte 1

Programmazione Distribuita - A.A. 2020/2021

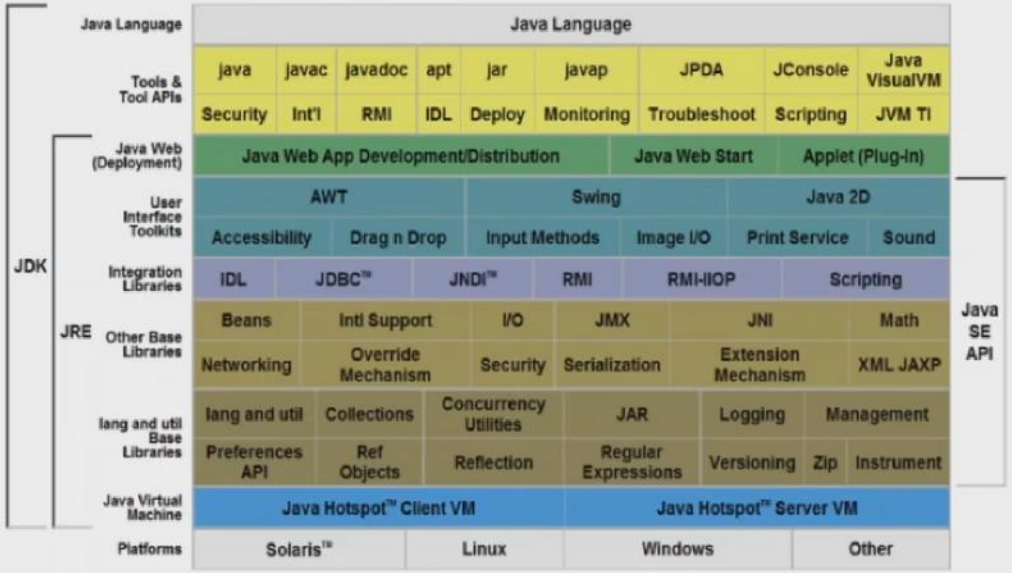
Biagio Cosenza
Dipartimento di Informatica
Università di Salerno
http://cosenza.eu/
bcosenza@unisa.it

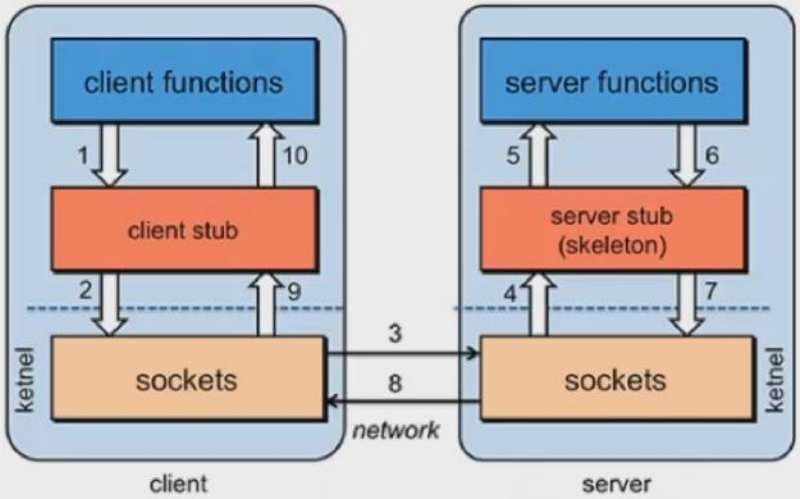
Organizzazione della Lezione

- Java Remote Method Invocation
  - Gli obiettivi della progettazione
- La sicurezza di Java
- Il modello ad oggetti distribuito di Java RMI
  - Interfacce ed eccezioni remote
  - Implementazioni remote
  - Riferimenti remoti
  - Localizzazione ed invocazione
  - Passaggio di parametri
  - Meccanismo di Marshalling
- Conclusioni

Java Remote Method Invocation

- Libreria che permette l'invocazione di metodi su oggetti remoti
- Il ruolo di Java RMI: integration library





Genesi di RMI

- Team condotto da Jim Waldo, con esperienza su
  - sulla progettazione di RPC da parte di Sun
  - sulla realizzazione dei Network Objects di Modula-3 ma anche su CORBA (Waldo per conto di HP)
- Obiettivi di massima:
  - semplicità della progettazione
    - favorisce la diffusione
    - ma impedisce anche errori ed abusi del sistema
  - integrazione con il linguaggio
    - ambiente che risulta naturale per il programmatore
    - che usa lo stesso ambiente usato per livello application e presentation

Obiettivi di RMI

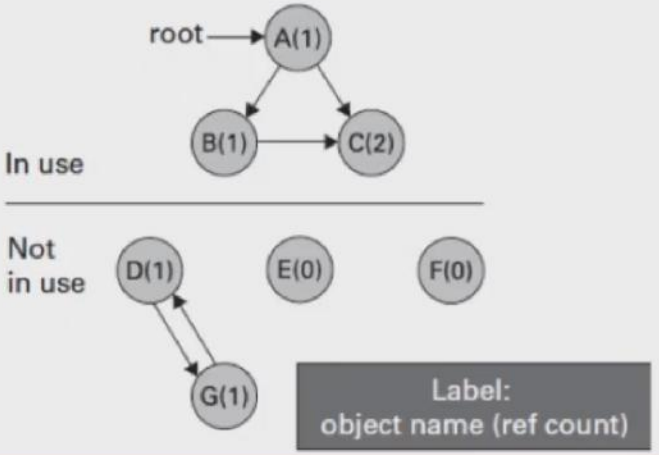
- Invocazione trasparente di metodi remoti
  - offrire l'illusione che l'invocazione avvenga su un oggetto interno alla JVM
  - JVM dalla quale parte l'invocazione remota
- Integrazione completa in Java
  - ambiente familiare
  - semantica di oggetti locali e distribuiti differente
    - ma in maniera limitata e resa esplicita
  - uso di politiche di garbage collection distribuita integrate con quelle locali
    - *memory leak* pericolosi per le applicazioni distribuite quanto (e più di) quelle locali

Garbage Collection

- Caratteristica importante di un linguaggio di programmazione:
  - al degrado di prestazioni (limitato) dovuto alle attività periodiche di collection
  - fa da contraltare la stabilità del sistema
- *Memory leak*: allocazione di memoria che non viene deallocata anche se non utilizzata
  - particolarmente pericolosa in caso di server attivi 24/24
- La Java Virtual Machine implementa politiche elaborate per la garbage collection
- Tecniche di base: *reference counting*

Reference Counting

- Ogni oggetto ha un contatore
- Quando un oggetto A contiene/usa il riferimento ad un altro oggetto B, il contatore di B viene incrementato
- Periodicamente viene effettuato il calcolo dei riferimenti e vengono eliminati (restituiti all'heap) le variabili con contatore a zero
  - problemi con riferimenti circolari: esempio D e G



Obiettivi di Java RMI

- Non-trasparenza della natura locale/remota di un oggetto
  - differenze che devono essere esplicite in fase di progettazione
  - il fatto che un oggetto sia remoto oppure locale deve essere chiaro ed evidente sia in fase di progettazione che in fase di implementazione
- Minima complessità di client e server
- Modalità di invocazioni multiple
  - unicast
  - multicast
  - attivazione e persistenza
- Livelli di trasporto multipli
- Preservare il modello di sicurezza fornito da Java

Sicurezza di Java

- Dai WhitePaper di Java
  - Semplice
  - Distribuito
  - Interpretato
  - Portabile
  - Alte prestazione
  - Object-oriented
  - Multithread
  - Dinamico
  - Sicuro

La Sandbox di Java

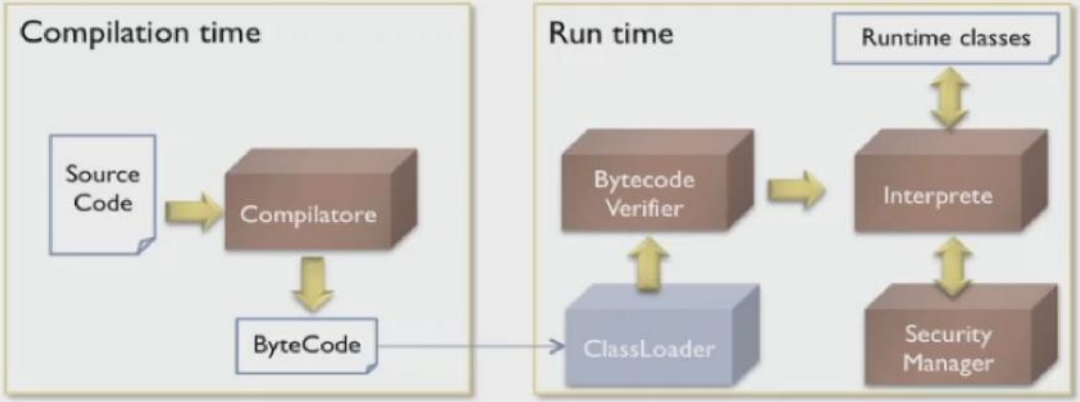
- Permette la esecuzione di programmi/applet in modo che le operazioni siano ristrette
- 4 livelli di sicurezza forniti dal linguaggio
  1. I livello: la sicurezza del linguaggio
  2. II livello: Class loader
  3. III livello: Bytecode Verifier
  4. IV livello: Security Manager
- Evoluzione dalle varie versioni di Java

I Livello di Sicurezza: la Sicurezza del Linguaggio

- Java è fortemente tipizzato: tipi definiti a tempo di compilazione e pochi casting automatici a run-time
  - gli altri casting vanno esplicitati
- Gestione della memoria mediante garbage collection
  - si evitano *memory leak*
- Assenza di puntatori non permette accessi illegali in memoria
  - i puntatori permettono accesso diretto ad indirizzi di memoria
- Accesso a memoria reale calcolato a tempo di esecuzione
  - non possibile scrivere codice per alterare zone di memoria reale
- Limiti di array controllati a run-time, per prevenire accessi ad elementi non esistenti

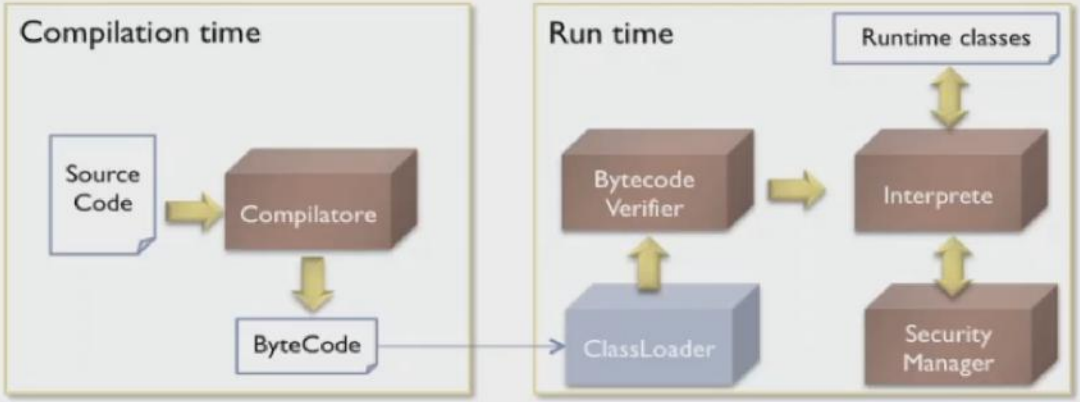
Il Livello di Sicurezza: il Class Loader

- Il suo compito principale è quello di caricare la classe in un namespace separato rispetto a quello delle classi locali,
  - in modo che classi del linguaggio built-in, locali, non possono essere rimpiazzate da altre
  - non c'è possibilità di "sovrascrivere" una classe di sistema
  - il `ClassLoader` controlla che non possiamo scrivere una classe `String` che, ogni volta che viene usata, manda il contenuto della stringa via mail a qualcuno, in modo da rivelare numeri di carta di credito ed altre informazioni
  - la classe utilizzata sarà sempre quella presente nel package `java.lang.*`



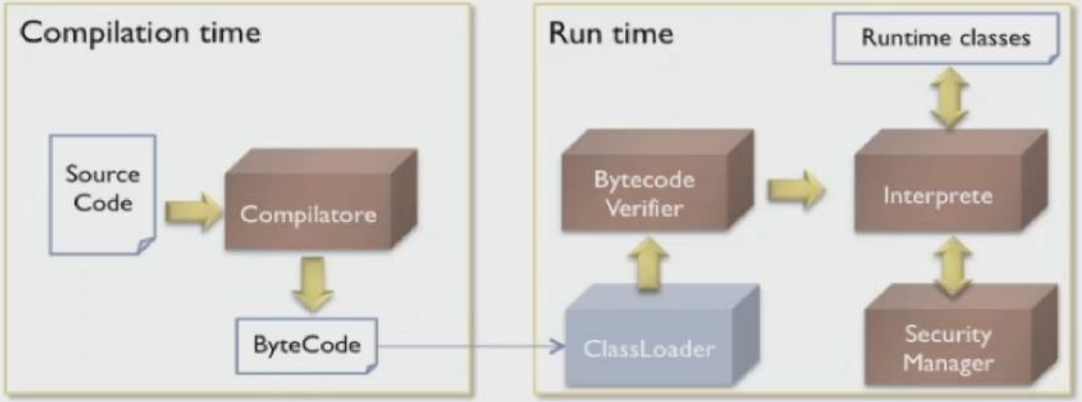
III Livello di Sicurezza: il Bytecode Verifier

- Classe conforme alle specifiche (correttezza operandi, casting illegali, etc.)
- Stack underflow/overflow
- Violazioni modificatori di accesso
- Necessario per il caricamento da remoto
  - possibili modifiche ad-hoc del bytecode



IV Livello di Sicurezza: il Security Manager

- Definisce i confini della sandbox, usando le *policy* definite dall'utente
- Applica la politica alle operazioni "potenzialmente" pericolose
  - scrivere un file non è sempre pericoloso, dipende dall'applicazione che lo fa



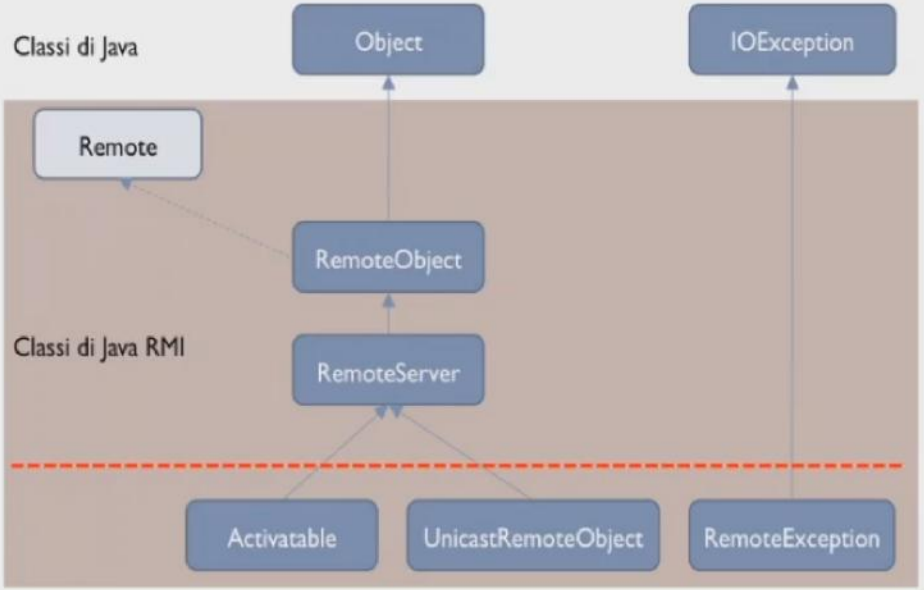
Il modello ad Oggetti Distribuito di Java RMI

- Un oggetto remoto è un oggetto i cui metodi possono essere acceduti da un altro spazio di indirizzamento, e potenzialmente da un'altra macchina
- La descrizione dei servizi offerti da remoto da un oggetto remoto è contenuta all'interno di un'interfaccia remota che è un'interfaccia Java che dichiara i metodi remoti
- Un'invocazione di metodi remoti (*Remote Method Invocation*) rappresenta l'invocazione di un metodo su un oggetto remoto (specificato nell'interfaccia remota) e ha la stessa sintassi di una invocazione di un metodo locale

Struttura

- Oggetto remoto: i cui metodi sono accessibili da un altro spazio di indirizzamento (quindi da una altra macchina)
- Interfaccia remota: interfaccia Java che descrive i servizi

- Struttura delle classi in 5 package:
  1. `java.rmi` funzionalità di base
  2. `java.rmi.server` funzionalità di base
  3. `java.rmi.activation` oggetti attivabili
  4. `java.rmi.dgc` garbage collection
  5. `java.rmi.registry` servizio di naming



Aspetti Importanti di RMI

- Interfacce ed eccezioni remote
- Implementazioni remote
- Riferimenti remoti
- Localizzazione ed invocazione
- Passaggio di parametri
- Meccanismo di marshalling

Aspetti Importanti di RMI

- Interfacce ed eccezioni remote
- Implementazioni remote
- Riferimenti remoti
- Localizzazione ed invocazione
- Passaggio di parametri
- Meccanismo di marshalling

Interfacce ed Eccezioni Remote (1)

- Una interfaccia remota per Java RMI deve estendere (implementare) l'interfaccia `java.rmi.Remote`
  - che è una cosiddetta interfaccia marker
  - una interfaccia vuota
  - in questo caso, serve solamente per poter segnalare che essa definisce dei metodi accessibili da remoto

Interfacce ed Eccezioni Remote (2)

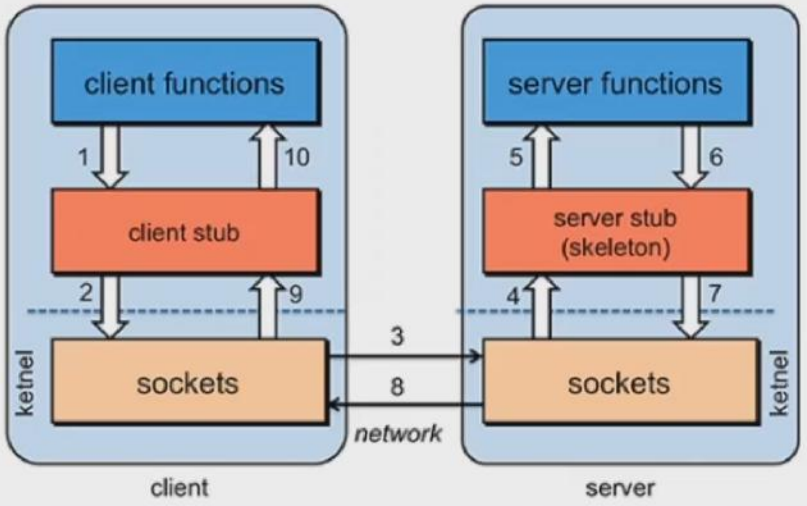
- Ogni metodo descritto in una interfaccia remota deve essere un metodo remoto cioè deve soddisfare le seguenti condizioni:
  - dichiarare esplicitamente `java.rmi.RemoteException`
    - checked exception dal compilatore
    - la semantica diversa dei malfunzionamenti in locale impone al programmatore di rendere esplicita la natura del metodo
  - parametri remoti dichiarati tramite interfaccia remota
    - permette l'uso di riferimenti remoti come parametri/valori restituiti
    - l'accesso remoto in qualche maniera aggiunge modifica di accesso ai tradizionali public, private, etc

Implementazioni Remote

- Per realizzare l'implementazione remota che deriva (`implements`) da una interfaccia remota per offrire verso l'esterno i metodi remoti in essa definiti, si può procedere in 2 modi:
  - riuso della implementazione remota
    - prevede che la classe che contiene l'implementazione dell'oggetto derivi esplicitamente da `java.rmi.server.UnicastRemoteObject`
    - ereditando di conseguenza il comportamento definito dalle classi `java.rmi.server.RemoteObject` e `java.rmi.server.RemoteServer`
  - classe di implementazione locale
  - permette che la classe derivi il comportamento da qualche altra classe (non remota) e che si debba quindi occupare esplicitamente di
    - esportare l'oggetto: tramite il metodo `statico exportObject()` di `java.rmi.server.UnicastRemoteObject`
    - implementare la semantica di alcune operazioni di Object per oggetti remoti che sono ridefiniti in `java.rmi.server.RemoteObject` e `java.rmi.server.RemoteServer`

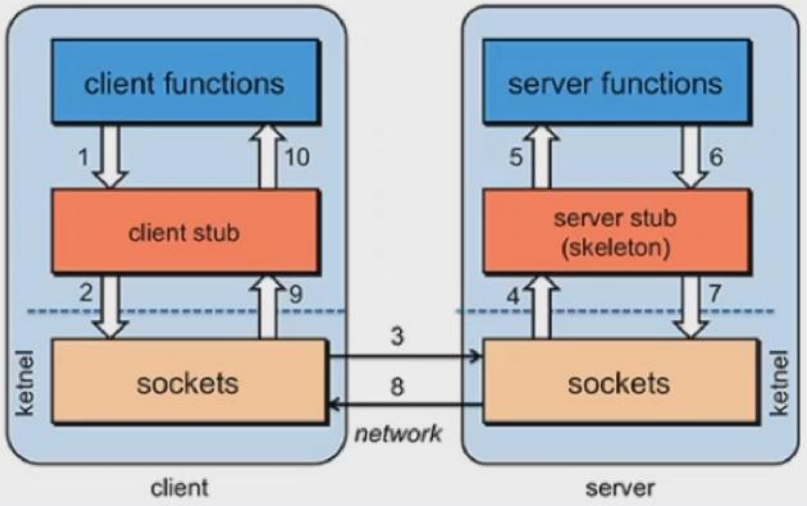
Riferimenti Remoti

- Lo stub e lo skeleton forniscono la rappresentazione dell'oggetto remoto
- Lo stub
  - gestisce la simulazione locale sul client e agendo come proxy consente la comunicazione con l'oggetto remoto
- Lo skeleton
  - ne consente l'esecuzione sul server



Riferimenti Remoti

- I client interagiscono con l'oggetto stub
  - che rappresenta l'interfaccia remota dell'oggetto remoto in locale sulla macchina (virtuale) del client
  - espone localmente esattamente le stesse interfacce remote definite dall'oggetto remoto
- Possibile caricare dinamicamente lo stub
  - da un server WWW
  - a tempo di esecuzione, quando si ottiene il riferimento remoto



Caricamento Dinamico delle Classi (1)

- Java Remote Method Invocation permette il passaggio di oggetti come parametri, valori restituiti o eccezioni, attraverso la serializzazione
  - permette di poter mantenere il sistema di tipi del linguaggio offre, continuando ad offrire il familiare ambiente di programmazione
- Questo si scontra con una altra caratteristica di Java, quella del caricamento delle classi a tempo di esecuzione
  - risulta essere più complesso nel momento in cui stiamo passando ad un metodo offerto da un server remoto (ad esempio) un parametro che è una istanza di una sottoclasse della classe dichiarata nella firma del metodo
  - in questo caso, l'oggetto remoto può trovarsi nella situazione in cui non conosce esattamente come è strutturata la classe di cui l'oggetto passato è istanza

Caricamento Dinamico delle Classi (2)

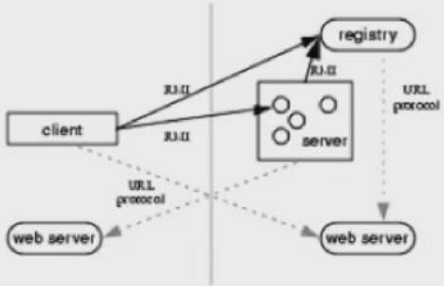
- Questo viene risolto da Java RMI attraverso il caricamento dinamico delle classi
- Quando si fa il *marshalling* degli oggetti per la trasmissione (ad esempio, come parametri nella invocazione da client a server), essi vengono anche annotati con il codebase
  - cioè con la Uniform Resource Locator (URL) di un server WWW da dove è possibile trovare la definizione della classe (cioè il file .class)
- Quando viene effettuato l'*unmarshalling* dell'oggetto, il `ClassLoader` cerca di risolvere il nome della classe nel suo contesto
  - in caso non sia possibile, viene acceduta la definizione della classe per poter ricreare l'oggetto all'altro capo della comunicazione (nel nostro esempio, sul server)

Caricamento Dinamico delle Classi (3)

- Questo meccanismo di caricamento dinamico permette anche il caricamento dinamico degli stub
  - con lo stesso meccanismo di annotazione che viene effettuato nel *marshalling*
- Questo, ad esempio, avviene quando si passa un riferimento remoto, che consiste nell'inviare lo stub

Localizzazioni e Invocazione

- Per poter invocare il metodo remoto di un oggetto remoto, l'oggetto client deve avere a disposizione il riferimento remoto, chepuò essere reperito in due maniere:
  - ottenuto come risultato di altre invocazioni (locali o remote) di metodi
  - attraverso un servizio di directory
- Servizio fornito da `java.rmi.Naming`
  - Metodi statici
    - `lookup()`
    - `bind()` / `rebind()` / `unbind()`
    - `list()`
  - Accetta string della forma: `//host:port/name`



- Questo meccanismo è utilizzabile anche come tool da linea di comando (`rmiregistry`)
  - eseguito sulla macchina sulla quale l'oggetto server si trova
- Un meccanismo di localizzazione più robusto verrà proposto successivamente
  - quando tratteremo della implementazione di RMI su IIOP, basato su Java Naming and Directory Interface

Localizzazioni e Invocazione

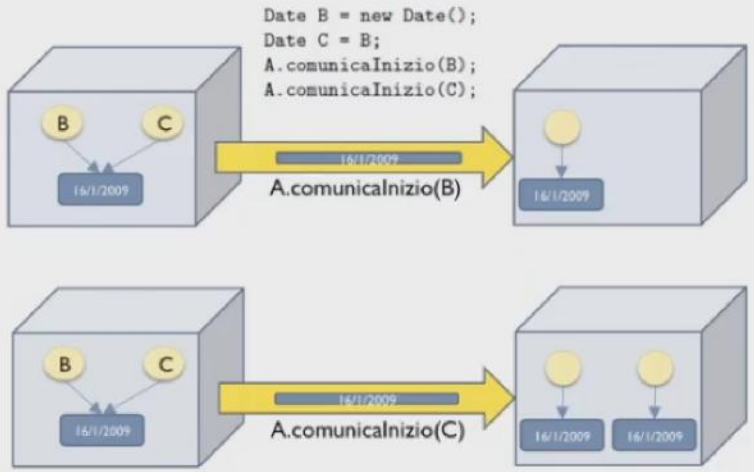
- L'invocazione di un metodo remoto ha la stessa sintassi di una invocazione locale
  - Poiché i metodi remoti devono includere `RemoteException` nella propria firma, il codice dell'oggetto client viene forzato dal compilatore a gestire questo possibile malfunzionamento della chiamata remota
  - in aggiunta ad altre eccezioni che dipendono dalla semantica dell'applicazione

Passaggio di Parametri

- Devono implementare l'interfaccia `Serializable`
- Oggetto locale: passato per copia
  - cambiamento rispetto passaggio di parametri in metodi locali
- IMPORTANTE
  - Quando si passano riferimenti allo stesso oggetto mediante invocazioni diverse sulla macchina server saranno oggetto diversi
  - Integrità referenziale:
    - se più riferimenti allo stesso oggetto vengono passati nella stessa invocazione allora viene garantito che punteranno allo stesso oggetto sulla macchina server

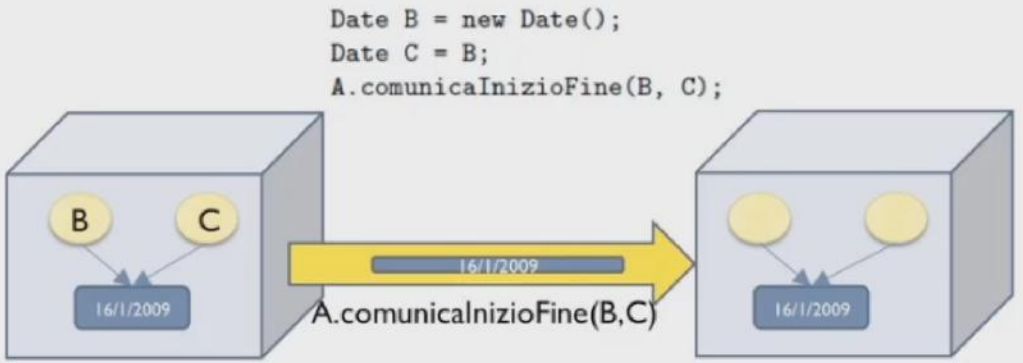
Passaggio in metodi diversi

- Codice per controllo uguaglianza deve dipendere dalla natura locale/remota del server
  - Cambia l'uso di = ma anche di `.equals()`
    - di necessaria eventuale definizione o specializzazione



Passaggio nell stesso metodo

- Meccanismo di *marshalling* usato da Java RMI
  - i riferimenti a stessi oggetti nella stessa invocazione verranno risolti da un handle singolo
  - risultando in un unico oggetto sul server



Meccanismo di Marshalling (1)

- Fare il *marshalling* di un oggetto in Java significa
  - effettuare una serializzazione modificando la semantica dei riferimenti remoti
    - invece di un riferimento remoto viene inserito lo stub dell'oggetto remoto
    - e aggiungendo informazioni all'oggetto (il codebase della classe dell'oggetto)
- Il meccanismo di *marshalling* di Java RMI si basa sulla specializzazione del meccanismo tradizionale di serializzazione effettuata da `ObjectOutputStream`
  - questa classe offre la possibilità di poter modificare il comportamento tramite il quale gli oggetti vengono scritti come stream di byte

Meccanismo di Marshalling (2)

- La specializzazione del meccanismo di serializzazione per il marshalling avviene modificando tre metodi della classe `ObjectOutputStream`:
  - Il metodo `replaceObject()` che può definire un metodo alternativo per serializzare un oggetto sullo stream
  - Il metodo `enableReplaceObject()` che restituisce un booleano e stabilisce se la istanza deve oppure no specializzare il meccanismo di serializzazione, usando il metodo `replaceObject()`
  - Il metodo `annotateClass()`, che permette di inserire informazioni addizionali sulla classe
    - viene usato per specificare il codebase e permettere quindi il caricamento dinamico

Meccanismo di Marshalling (3)

- L'operazione più complessa è quella di `replaceObject()`
- Il meccanismo di serializzazione di RMI specifica (abilitando il flag booleano restituito da `enableReplaceObject()`) che questo metodo deve essere richiamato ogni qualvolta viene invocato `writeObject()`
  - **prima** che questo provveda alla serializzazione dando la possibilità di sostituire l'oggetto da serializzare
- Il metodo `replaceObject()` fa le seguenti operazioni:
  - Se l'oggetto da serializzare è una istanza di `java.rmi.Remote` e quindi è un riferimento remoto, e l'oggetto risulta esportato al runtime di RMI, **allora viene restituito dallo stub** per quell'oggetto remoto utilizzando il metodo `java.rmi.server.RemoteObject.toStub()`
    - nel caso in cui l'oggetto remoto non sia esportato, allora si restituisce l'oggetto remoto stesso
  - Se l'oggetto da serializzare non è una istanza di `java.rmi.Remote` allora viene restituito a `writeObject()`

Meccanismo di Marshalling (4)

- Tramite questo meccanismo viene risolta una apparente incongruità nelle invocazioni:
  - quando viene passato un riferimento remoto come parametro, questo viene copiato sullo stream, ma non abbiamo il vincolo di dichiarare un riferimento remoto come serializzabile
  - infatti, quando si passa un riferimento remoto che è esportato (quindi attivo), esso viene sostituito dal suo stub che è una istanza di `java.rmi.server.RemoteStub`, che è una classe che implementa la interfaccia `Serializable`
- Questo meccanismo spiega anche la modalità con la quale viene assicurata la integrità referenziale:
  - poichè i parametri di una stessa invocazione remota utilizzano lo stesso stream di output, parametri che si riferiscono allo stesso oggetto nella stessa invocazione verranno serializzati nel flusso come facenti riferimento allo stesso oggetto, e verranno deserializzati nella stessa maniera all'altro capo dello stream

Conclusioni

- Java Remote Method Invocation
  - Gli obiettivi della progettazione
- La sicurezza di Java
- Il modello ad oggetti distribuito di Java RMI
  - Interfacce ed eccezioni remote
  - Implementazioni remote
  - Riferimenti remoti
  - Localizzazione ed invocazione
  - Passaggio di parametri
  - Meccanismo di marshalling
- Conclusioni