RMI pitfall: `equals` and `hashCode`

- To find out if two remote objects have the same contents, the call to `equals` would need to contact the servers containing the objects and comparing their contents
- That call could fail, but `equals` does not throw `RemoteException`
- The same holds for `hashCode`
- We must rely on the redefinitions of these two methods in `RemoteObject`, that is the superclass of all remote objects and stubs
- These methods actually compare the identities
- If we redefine the methods in the remote object, this will not affect stubs, as they are mechanically generated

RMI pitfall: `clone`

- `clone` does not throw `RemoteException`
- It does not make sense to clone a stub: we could simply duplicate its reference
- We may define a `remoteClone` method:

```
interface SomeInterface extends java.rmi.Remote
{    public Object remoteClone() throws RemoteException,
          cloneNotSupportedException;
     […]
}

class SomeInterfaceImpl extends UnicastRemoteObject
          implements SomeInterface
{    public Object remoteClone() throws RemoteException,
          cloneNotSupportedException
     {    return clone();
     }
}
```

RMI pitfall: inappropriate remote parameters

- **Consider a remote method with the following interface:**

```
void paint(Graphics g) throws RemoteException
```

- **`Graphics` (actually, its subclasses) interacts with native graphics code, so its serialized version would embed machine-dependent pointers**
- **`Graphics` is not `Serializable`, so it cannot be sent via RMI**

Dynamic class downloading in RMI

- **One of the most significant capabilities of the Java platform is the ability to dynamically download Java software from any Uniform Resource Locator (URL) to a JVM running in a separate process, usually on a different physical system**
- **The result is that a remote system can run a program, for example an applet, which has never been installed on its disk**
- **For example, a JVM running from within a Web browser can download the bytecodes for subclasses and any other classes needed by that applet**
- **The system on which the browser is running has most likely never run this applet before, nor installed it on its disk**
- **Once all the necessary classes have been downloaded from the server, the browser can start the execution of the applet program using the local resources of the system on which the client browser is running**

- Java RMI takes advantage of this capability to download and execute classes on systems where those classes have never been installed on disk
- Using the RMI API any JVM, not only those in browsers, can download any Java class file including specialized RMI stub classes, which enable the execution of method calls on a remote server using the server system's resources
- It could be needed to download also client-unknown subclasses of client-known classes
- When a Java program uses a `ClassLoader`, that class loader needs to know the location(s) from which it should be allowed to load classes
- Usually, a class loader is used in conjunction with an HTTP server that is serving up compiled classes for the Java platform
- A codebase can be defined as a source, or a place, from which to load classes into a Java virtual machine
- You can think of your `CLASSPATH` as a "local codebase", because it is the list of places on disk from which you load local classes. When loading classes from a local disk-based source, your `CLASSPATH` variable is consulted

How codebase is used in RMI

- Using RMI, applications can create remote objects that accept method calls from clients in other JVMs
- In order for a client to call methods on a remote object, the client must have a way to communicate with the remote object
- The `java.rmi.server.codebase` property value represents one or more URL locations from which the stubs (and any classes needed by the stubs) can be downloaded
- Like applets, the classes needed to execute remote method calls can be downloaded from "`file:///`" URLs, but like applets, a "`file:///`" URL generally requires that the client and the server reside on the same physical host, unless the file system referred to by the URL is made available using some other protocol, such as NFS
- Generally, the classes needed to execute remote method calls should be made accessible from a network resource, such as an HTTP or FTP server

- **Before starting the server, you need to start RMI's registry, using the `rmiregistry` command**
- **Before you start the `rmiregistry`, you must make sure that the shell or window in which you will run `rmiregistry` either has no `CLASSPATH` environment variable set or has a `CLASSPATH` environment variable that does not include the path to any classes, including the stubs for your remote object implementation classes, that you want downloaded to clients of your remote objects**
- **If you *do* start the `rmiregistry` and it *can* find your stub classes in `CLASSPATH` (or in the starting directory) it will not remember that the loaded stub class can be loaded from your server's code base, specified by the `java.rmi.server.codebase` property when you started up your server application**
- **Therefore, the `rmiregistry` will not convey to clients the true code base associated with the stub class and, consequently, your clients will not be able to locate and to load the stub class or other server-side classes**

1. **The remote object's codebase is specified by the remote object's server by setting the `java.rmi.server.codebase` property**
2. **The RMI server registers a remote object, bound to a name, with the RMI registry**
3. **The codebase set on the server JVM is annotated to the remote object reference in the RMI registry**
4. **The RMI client requests a reference to a named remote object. The reference (the remote object's stub instance) is what the client will use to make remote method calls to the remote object**
5. **The RMI registry returns a reference (the stub instance) to the requested class**

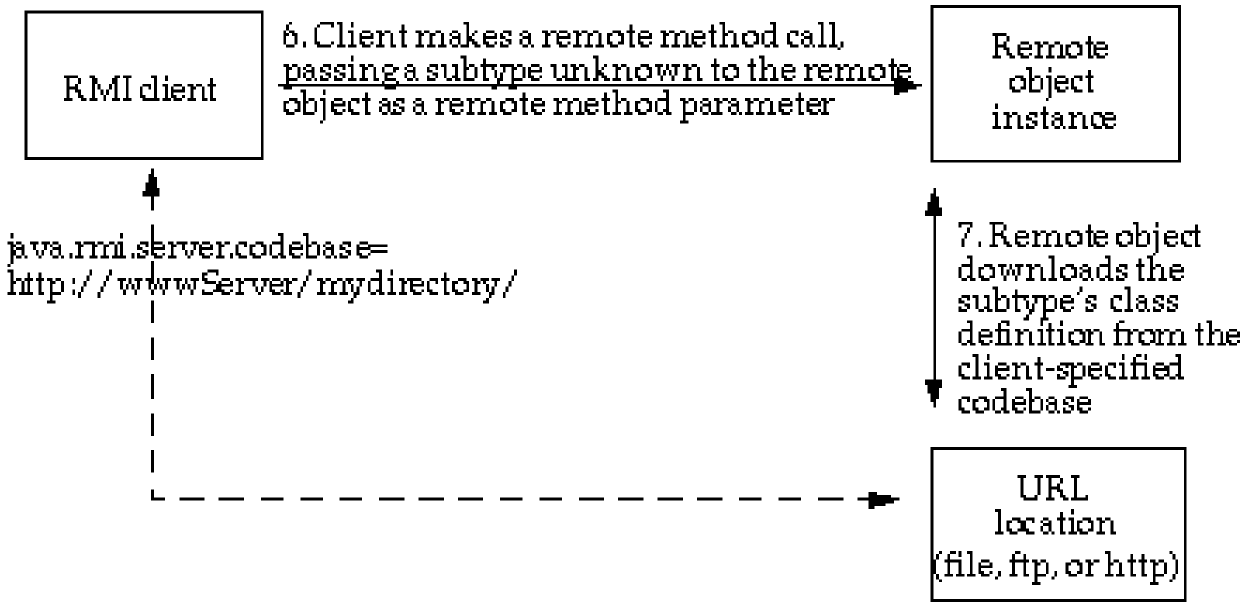
6. **If the class definition for the stub instance can be found locally in the client's `CLASSPATH` , which is always searched before the codebase, the client will load the class locally**
7. **If the definition for the stub is not found in the client's `CLASSPATH`, the client will attempt to retrieve the class definition from the remote object's codebase**
8. **The client requests the class definition from the codebase. The codebase the client uses is the URL that was annotated to the stub instance when the stub class was loaded by the registry**
9. **The class definition for the stub (<u>and any other class(es) that it needs</u>) is downloaded to the client**
10. **Now the client has all the information that it needs to invoke remote methods on the remote object**
11. **The stub instance acts as a proxy to the remote object that exists on the server; so unlike the applet which uses a codebase to execute code in its local JVM, the RMI client uses the remote object's codebase to execute code in another, potentially remote JVM**

Using codebase in RMI for more than just stub downloading

- **In addition to downloading stubs and their associated classes to clients, the `java.rmi.server.codebase` property can be used to specify a location from which any class, not only stubs, can be downloaded**
- **When a client makes a method call to a remote object, there are three distinct cases that may occur, based on the data type(s) of the method argument(s)**

  1. **In the first case, all of the method parameters (and return value) are primitive data types, so the remote object knows how to interpret them as method parameters, and there is no need to check its `CLASSPATH` or any codebase.**
  2. **In the second case, at least one remote method parameter or the return value is an object, for which the remote object can find the class definition locally in its `CLASSPATH`**

3. **In the third case the remote method receives an object instance, for which the remote object cannot find the class definition locally in its `CLASSPATH`. The class of the object sent by the client is a subtype of the declared parameter type**


RMI client
6. Client makes a remote method call, passing a subtype unknown to the remote object as a remote method parameter
Remote object instance
java.rmi.server.codebase= http://wwwServer/mydirectory/
7. Remote object downloads the subtype's class definition from the client-specified codebase
URL location (file, ftp, or http)

- Like the applet's codebase, the client-specified codebase is used to download `Remote` classes, non-remote classes, and interfaces to other JVMs
- If the codebase property is set on the client application, then that codebase is annotated to the subtype instance when the subtype class is loaded by the client
- If the codebase is not set on the client, the remote object will mistakenly use its own codebase

Command-line examples

Because the remote object's codebase can refer to any URL, not just one that is relative to a known URL, the value of the RMI codebase must be an absolute URL to the location of the stub class and any other classes needed by the stub class

This value of the codebase property can refer to:

- The URL of a directory in which the classes are organized in package-named sub-directories
- The URL of a JAR file in which the classes are organized in package-named directories
- A space-delimited string containing multiple instances of JAR files and/or directories that meet the criteria above

Note: When the codebase property value is set to the URL of a *directory*, the value must be terminated by a "/"

- **If the location of your downloadable classes is on an HTTP server named "webserver", in the directory "export" (under the web root), your codebase property setting might look like this:**

```
-Djava.rmi.server.codebase=http://webserver/export/
```

- **If the location of your downloadable classes is in a JAR file named "mystuff.jar", in the directory "public" (under the Web root), your codebase property setting might look like this:**

```
-Djava.rmi.server.codebase=http://webserver/public/mystuff.jar
```

- **If the location of your downloadable classes has been split between two JAR files, "myStuff.jar" and "myOtherStuff.jar". If these JAR files are located on different servers (named "webserver1" and " webserver2"), your codebase property setting might look like this:**

```
-Djava.rmi.server.codebase="http://webserver1/myStuff.jar
                      http://webserver1/myOtherStuff.jar"
```

Security

- The Java 2 security model is more sophisticated than the model used for JDK 1.1
- Java 2 contains enhancements for finer-grained security and requires code to be granted specific permissions to be allowed to perform certain operations
- You need to specify a policy file
- Here is a general policy file that allows downloaded code, from any code base, to do two things:
    - Connect to or accept connections on unprivileged ports (ports greater than 1024) on any host
    - Connect to port 80 (the port for HTTP)
- Here is the code for the general policy file (client.policy):

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

- **You can start the client, specifying the `java.security.policy` property, used to specify the policy file that contains the permissions you intend to grant:**

  ```
  java -Djava.security.policy=client.policy Client
  ```

- **To set the default RMI security manager:**

  ```
  System.setSecurityManager(new RMISecurityManager());
  ```

Client-side callbacks

- In many architectures, a server may need to make a remote call to a client
- Examples include progress feedback, time tick notifications, warnings of problems, etc.
- To accomplish this, a client must also act as an RMI server
- There is nothing really special about this as RMI works equally well between all computers
- It may be impractical for a client to extend `java.rmi.server.UnicastRemoteObject`
- In these cases, a remote object may prepare itself for remote use by calling the static method:

```
UnicastRemoteObject.exportObject(remote_object)
```

- To support a callback, the client must act as an RMI server
- It does this by exporting and implementing a remote interface
- Our client will define and implement the `TimeMonitor` interface, that is designed to be called by a time service that supplies the current date and time
- The server cannot call back to the client until it knows where to find it
- It is the client's responsibility to register itself with the server. It does this by using the server's `registerTimeMonitor` method in the `TimeServer` and passes a reference to itself to the server.
- In this exercise, you will need to define the interfaces and the implementations for both the server and the client.

NOTE: We will also use an alternative way to set up an RMI registry

```
import java.rmi.*;
import java.util.Date;

public interface TimeMonitor extends java.rmi.Remote
{   public void time(Date d) throws RemoteException;
}
```

```
import java.rmi.*;

public interface TimeServer extends java.rmi.Remote
{   public void registerTimeMonitor(TimeMonitor tm)
        throws RemoteException;
}
```

```
import java.util.Date;

class TimeTicker extends Thread
{   private TimeMonitor tm;

    TimeTicker( TimeMonitor tm )
    {   this.tm = tm;
    }

    public void run()
    {   while(true)
        try
        {   sleep( 2000 );
            tm.time(new Date());
         }
         catch ( Exception e )
         {   System.out.println(e);
         }
    }
}
```

```
import java.net.*;
import java.io.*;
import java.util.Date;

import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.LocateRegistry;

public class TimeServerImpl implements TimeServer
{  private static TimeServerImpl  tsi;

   public static void main (String[] args)
   {     try
         {   tsi = new TimeServerImpl();
             LocateRegistry.createRegistry(1099);
             System.out.println("Registry created");

             UnicastRemoteObject.exportObject(tsi);
             Naming.rebind("TimeServer", tsi);
             System.out.println( "Bindings Finished" );
             System.out.println( "Waiting for Client requests" );
         } [to be continued…]
```

```
        catch (Exception e)
        {    System.out.println(e);
        }
    }

   public void registerTimeMonitor( TimeMonitor tm )
   {    System.out.println( "Client requesting a connection" );

        TimeTicker tt;
        tt = new TimeTicker( tm );
        tt.start();
        System.out.println( "Timer Started" );
   }

}  // class TimeServerImpl
```

```
import java.util.Date;
import java.net.URL;
import java.rmi.*;
import java.rmi.server.*;

public class TimeClient implements TimeMonitor
{   private TimeServer ts;

    public TimeClient()
    {   try
        {   System.out.println( "Exporting the Client" );
            UnicastRemoteObject.exportObject(this);
            ts = (TimeServer)Naming.lookup(
                "rmi://localhost:1099/TimeServer");
            ts.registerTimeMonitor(this);
         }
         catch (Exception e)
         {  System.out.println(e);
         }
    }

    [to be continued…]
```

```
   public void time( Date d )
   {    System.out.println(d);
   }

} //class TimeClient
```

```
public class Main
{  public static void main (String[] args)
   {    new TimeClient();
   }
}
```