プログラミング環境特論

Javaによる
分散プログラミング入門

佐藤

Java

- すべてのプログラムはクラス定義の集まりで定義される。Cのように、関数だけ、データ定義だけというのはない。
- オブジェクト指向言語。メンバー関数、メンバーの可視化制御、継承ができる。
- Constructorはあるが、destructorはない。参照されなくなったオブジェクトは自動的にガベージコレクションされる。
- ポインタはない。すべてのオブジェクトは、C++でいえばポインターで表現されている。メンバー関数はすべてvirtualメンバー関数。
- ひとつのオブジェクトからしか、継承できない。
- interface定義。（C++の仮想クラス定義に相当する）
- オブジェクト型に演算子は定義できない。Operator overloadingなし。
- Template機能もなし。

C++

- オブジェクトを定義するためにclassを導入。データ型に対し、その操作を定義するメンバー関数を宣言できる。ちなみにCの構造体であるstructは、全メンバーが公開（public）なclassと同値。
- クラス定義において、継承(inheritance)関係を定義でき、メンバーの可視性を制御できる。２つ以上のベースクラスも持つことができる。（Multiple inheritance）
- クラス定義においては、クラスを生成する構築子（constructor）と消滅子(destructor)を宣言でき、クラスが生成・消滅するときに呼び出される。
- new / delete　演算子
- 仮想メンバー関数 (virtual function)
- オブジェクトに対し、演算子をできる（operator overloading）
- 多義関数名、int foo(int x)とint foo（double）は違う関数となる。ただし、「暗黙の型変換」が行われるので注意。
- defaultの引数が使える。
- 引数のReference渡しが使える。
- Template機能。Genericなプログラミングができる。

Javaについてコメント

- C++と比較して議論されることもあるjavaであるが、むしろ、その発想としてはsmalltalkに近い。
- プログラムは通常クラスファイルというjavaバイトコードからなる中間形式にコンパイルされ、java virtual machineと呼ばれるバイトコードインタープリタで実行される。
- この実行形式がネットワーク上の言語としてのjavaの柔軟性を与えているといえる。

Java入門

- 簡単な例
  - test.java
  - javac test.javaでコンパイル、test.classを作る
  - java test　で実行。test.classのmainから始まる。
  - staticは、クラスで共通の関数を定義

```
class test {
 public static void main(String arg[]){
   System.out.println(“hello”);
 }
}
```


java入門

- もうちょっと難しい例

```
class test {
 public static void main(String arg[]){
   hello h = new hello(“jack”);
   h.say();
 }
}
```

```
class hello {
 String who;
 public hello(String who){
   this.who = who;
}
 public void say(){
   System.out.println(“hello”+who);
}
}
```


java入門

- もうちょっと難しい例、継承の例

```
class test {
 public static void main(String arg[]){
   hello h = new konnichwa(“jack”);
   h.say();
 }
}
```

```
class konnichiwa extends hello {
 String who;
 public konnichiwa(String who){
   this.who = who;
}
 public void say(){
   System.out.println(“konnnichwa”+who);
}
}
```


java入門

- クラスパス
  - クラスを探すパスを指定する環境変数、通常はdirectory(jarでもOK)
  - jarとは、classファイルをzipしてあるファイル
  - javaのvmは動的にクラスを動的にロードする。
- interface
  - 実装していなくてはならないメンバー関数を指定するもの
  - javaでは継承は１つしかできないが、interfaceは複数もつことができる。

java入門

- もうちょっと難しい例、interfaceの例

```
class test {
 public static void main(String arg[]){
   oval ov = new oval(…);
   draw_it(ov);
 }
 static void draw_it(drawable o){
    … o.draw(); …
 }
}
```

```
interface drawable {
  void draw();
}
…
class circle implements drawable{
  void draw();
}
class oval implements drawable{
  void draw();
}
class polygon implements drawable{
  void draw();
}
```


オブジェクト指向プログラミングの原則

- オブジェクト指向設計　Object oriented design
  - 保守性：後から、見たとき、あるいはデバック中にも容易に理解できるようなプログラムを作ること。他の人が見たときにわかりやすいこと（可読性）も重要である。
  - 拡張性：プログラムの機能を加えるときに、なるべくほかのコードを変更せずに機能を加えることができることが望ましい。
  - 再利用性：ほかのプログラムに転用できるような部品として設計しておけば、プログラムの価値は高まる。
  - 効率：そして、プログラムは速くなくてはならない。

オブジェクト指向設計

- publicな継承が” is a”関係であることをしっかり理解する
- インタフェースの使い方、インタフェースと継承の違い
- 層化によって”has a”関係や”is implemented in terms of”関係を表現する（項目４０）
- Privateな継承は、正しくつかう（項目４１）

Javaによる分散プログラミング

- RMIとはRemote Method Invocationの略であり、Javaの分散プログラミングのための仕掛けである。
  - この仕掛けをつかうことによって、いろいろなマシンにオブジェクトのインスタンスを生成し、これらの間でRMIを使って他のマシンのオブジェクトのメソッドを呼び出すことによって、分散システムを構築することができる。

Remote Procedure Call

- 基本的には分散システムをプログラミングするためにはTCP/IPやUDPなど低レベルの通信レイヤを使つかう。しかし、いちいち、機能ごとにプロトコルを設計して、通信しなくてはならない。
- このプロトコルを関数呼び出しに抽象化したのが、RPC(remote procedure call）
  - SUN　RPC
  - CORBA (Java、C++)
  - Web Service
  - RMI, Jini….
  - JAX RPC
  - …

ネットワークプログラミング

- サーバー側

```
s = socket(); /* socketを作る*/
bind(s,address); /* 名前を与える */
listen(s,backlog); /* backlogの指定 */
ss = accept(s); /* connectionが発生したら
          新しいfile descriptorを返す */
close(s); /* 必要なければ、もとのsはclose */
recv(ss,…); /* read 開始 */
```

- クライアント側

```
s = socket(); /* socketを作る*/
connect(s,address); /* connectionする*/
send(s,…); /* send開始 */
```


Javaのデータ転送

サーバー側:

```
ServerSocket ss = new ServerSocket(port);
 Socket s = ss.accept();
 DataOutputStream out =
          new DataOutputStream(s.getOutputStream());
 out.writeInt(123);  /* write …*/
```

クライアント側:

```
Socket s = new Socket(host, port);
DataInputStream in = new DataInputStream(s.getInputSteram());
y = in.readInt(); /* … read …*/
```


Javaのオブジェクト転送

Javaでは、オブジェクトそのものを書き出すSerialization機能を持っている。
これをつかえば、Serializableインタフェースを実装しているオブジェクト
そのものを転送することができる。

```
ObjectOutputStream out = new ObjectOutputStream(s.getOutputStream());
out.writeObject(obj);
```

```
ObjectInputStream in = new ObjectInputStream(s.getInputStream());
Object obj = in.readObject();
```

ＳｈｏｗＤａｔｅのインタフェースの定義

```
public interface ShowDate {
  public long getCurrentMillis();
  public long getMillis();
}
```


◆ ShowDateのインタフェースの実装

```
public class ShowDateImpl implements Serializable, ShowDate {
  long millis = 0;
  Date date = null;
  public ShowDateImpl(){
    millis = getCurrentMillis();
    date = new Date(millis);
  }
  public long getCurrentMillis(){
    System.out.println("getCurrentMillis called!");
    return System.currentTimeMillis();
  }
  public long getMillis() {
    System.out.println("getMillis called!");
    return millis;
  }
  public static void main(String argv[]){
    ShowDateImpl sdi = new ShowDateImpl();
    System.out.println(sdi.date);
    System.out.println(sdi.getCurrentMillis());
  }
}
```


オブジェクトを転送するサーバの例

```
public class ObjectServer {
  public static void main(String argv[]){
    try {
      int port = 8080;

      ServerSocket ss = new ServerSocket(port);
      while(true){
        Socket s = ss.accept();
        System.out.println("Object Server accept!!!");
        ObjectOutputStream oos =
          new ObjectOutputStream(s.getOutputStream());
        ShowDateImpl sd = new ShowDateImpl();
        System.out.println("write "+sd);
        oos.writeObject(new ShowDateImpl());
        s.close();
      }
    } catch(Exception e){
      System.out.println("object write err:"+ e);
    }
  }
}
```


オブジェクトを受け取るクライアントの例

```
public class client0 {
  public static void main(String argv[]){
    try {
      client1 cl = new client1();
      String host = "localhost";
      int port = 8080;

      Socket s = new Socket(host,port);

      ObjectInputStream ois =
                  new ObjectInputStream(s.getInputStream());
      ShowDate sd = (ShowDate)(ois.readObject());
      System.out.println(sd.getCurrentMillis());
      System.out.println(sd.getMillis());
      System.out.println(sd);
    } catch(Exception e){
      System.out.println(e);
    }
  }
}
```

ネットワーククラスローダの例（1）

```
public class NetworkClassLoader extends ClassLoader {
  InputStream in;
  ByteArrayOutputStream out = new ByteArrayOutputStream(1024);
  public NetworkClassLoader() {
    this("localhost",8081);
  }
  public NetworkClassLoader(String host, int port){
    try {
      Socket s = new Socket(host,port);
      in = s.getInputStream();
    } catch(Throwable e){
      System.err.print("cannot open socket");
      System.exit(1);
    }
  }
  protected class findClass(String name)
              throws ClassNotFoundException {
…
}
```


ネットワーククラスローダの例(2)

```
protected Class findClass(String name)
                  throws ClassNotFoundException {
   try {
     byte buff[] = new byte[1024];
     int n,m;
     int len = 0;
     while((n = in.read(buff,0,1024)) > 0){
      out.write(buff,0,n);
      len += n;
     }
     byte data[] = new byte[len];
     data = out.toByteArray();
     return defineClass(null,data,0,len);
   } catch(Throwable e){
     System.err.println("read err");
     throw new ClassNotFoundException();
   }
  }
 }
```


ShowDataImplだけをサービスするクラスサーバ

```
public class ClassServer {
  public static void main(String argv[]){
    try {
      String classFile = "ShowDateImpl.class";
      int port = 8081;
      ServerSocket ss = new ServerSocket(port);
      while(true){
         Socket s = ss.accept();
         System.out.println("Class Server accept!!!");
         BufferedOutputStream bos =
           new BufferedOutputStream(s.getOutputStream());
         BufferedInputStream bis =
           new BufferedInputStream(new FileInputStream(classFile));
         int len;
         byte buff[] = new byte[256];
         while((len = bis.read(buff,0,256)) >= 0){
           bos.write(buff,0,len);
         }
         bos.flush();
         bos.close();
         bis.close();
      }
    } catch(Exception e){
      System.out.println("class file err:"+ e);
    }
  }
```


ネットワーククラスローダを使って、クラス情報をロードする例

```
public class client {
  public static void main(String argv[]){
    try {
      NetworkClassLoader loader = new NetworkClassLoader();
      Class cl = loader.loadClass("ShowDateImpl");
      ShowDate sd = (ShowDate)(cl.newInstance());
      System.out.println(sd.getCurrentMillis());
      System.out.println(sd);
    } catch(Exception e){
      System.out.println(e);
    }
  }
}
```

オブジェクトクラスローダをObjectStreamに加えた例（1）

```
public class client1 {
  public static void main(String argv[]){
    try {
      client1 cl = new client1();
      String host = "localhost";
      int port = 8080;

      Socket s = new Socket(host,port);

      MyObjectInputStream ois =
        cl. new MyObjectInputStream(s.getInputStream(),
                                    new NetworkClassLoader());
      ShowDate sd = (ShowDate)(ois.readObject());
      System.out.println(sd.getCurrentMillis());
      System.out.println(sd.getMillis());
      System.out.println(sd);
    } catch(Exception e){
      System.out.println(e);
    }
  }
```


オブジェクトクラスローダをObjectStreamに加えた例（2）

```
public class MyObjectInputStream extends ObjectInputStream {
    public ClassLoader cl;
    public MyObjectInputStream(InputStream im, ClassLoader cl)
        throws IOException {
          super(im);
          this.cl = cl;
    }
    protected Class resolveClass(ObjectStreamClass v)
                                    throws IOException {
      try {
       return super.resolveClass(v);
      } catch(ClassNotFoundException e){
       try {
         return cl.loadClass("ShowDateImpl");
       } catch(Exception e2){
         System.out.println(e2);
       }
      }
      return null;
    }
  }
}
```


MarshalInputStreamを使ったクライアントの例

```
public class client0 {
  public static void main(String argv[]){
    try {
      String host = "localhost";
      int port = 8080;
      if(System.getSecurityManager() == null){
              System.setSecurityManager(new RMISecurityManager());
      }
      System.out.println("security done...");
      Socket s = new Socket(host,port);
      System.out.println("socket="+s);
      MarshalInputStream ois =
             new MarshalInputStream(s.getInputStream());
      ShowDate sd = (ShowDate)(ois.readObject());
      System.out.println(sd.getCurrentMillis());
      System.out.println(sd.getMillis());
      System.out.println(sd);
    } catch(Exception e){
      System.out.println(e);
    }
  }
}
```


MarshalOutputStreamを用いたサーバの例

```
public class ObjectServer {
  public static void main(String argv[]){
    try {
      int port = 8080;

      if(System.getSecurityManager() == null){
         System.setSecurityManager(new RMISecurityManager());
      }
      System.out.println("security manager done ...");

      ServerSocket ss = new ServerSocket(port);
      System.out.println("accept ..."+ss);
      while(true){
         Socket s = ss.accept();
         System.out.println("Object Server accept!!!");
         MarshalOutputStream oos =
           new MarshalOutputStream(s.getOutputStream());
         ShowDateImpl sd = new ShowDateImpl();
         System.out.println("write "+sd);
         oos.writeObject(sd);
         s.close();
      }
    } catch(Exception e){
      System.out.println("object write err:"+ e);
    }
```

MarshalObjectを用いたクライアントの例

```
public class client {
  public static void main(String argv[]){
    try {
      String host = "localhost";
      int port = 8080;
      if(System.getSecurityManager() == null){
        System.setSecurityManager(new RMISecurityManager());
      }
     client cl = new client();
      Socket s = new Socket(host,port);
      ObjectInputStream ois =
                  new ObjectInputStream(s.getInputStream());
      MarshalledObject mo = (MarshalledObject)ois.readObject();
      System.out.println("Marshalled Object ="+mo);
      System.out.println("            Object ="+mo.get());
      ShowDate sd = (ShowDate)(mo.get());
      System.out.println(sd.getCurrentMillis());
      System.out.println(sd.getMillis());
      System.out.println(sd);
    } catch(Exception e){
      System.out.println(e);
    }
```


MarshalObjectを用いたサーバの例

```
public class ObjectServer {
  public static void main(String argv[]){
    try {
      int port = 8080;
      if(System.getSecurityManager() == null){
        System.setSecurityManager(new RMISecurityManager());
      }
       ObjectServer os = new ObjectServer();
      ServerSocket ss = new ServerSocket(port);
      System.out.println("accept ..."+ss);
      while(true){
        Socket s = ss.accept();
        System.out.println("Object Server accept!!!");
        ObjectOutputStream oos =
          new ObjectOutputStream(s.getOutputStream());
        ShowDateImpl sd = new ShowDateImpl();
        System.out.println("write "+sd);
        oos.writeObject(new MarshalledObject(sd));
        s.close();
      }
    } catch(Exception e){
      System.out.println("object write err:"+ e);
```


RMIでのデータ転送の手順

- まずあらかじめ、ネットワークのクラスサーバー（webサーバーでもよい）を立ち上げておく。(http://localhost:8081)
- 送るべきプログラムをjarファイルにしておく。(dl.jar)
- 送信側のプログラムには、どこからクラスをロードするか（codebase）を指定する。
- 双方のプログラムについて、セキュリティマネジャーを設定し、起動時にはセキュリティポリシーを指定する。

java –Djava.rmi.sever.codebase=http://localhost:8081/dl.jar
–Djava.security.policy=policy ObjectSever

オブジェクトの転送のまとめ

- 転送先でオブジェクトを参照するためには、インタフェースのみを共有しておけばよい。これは、Javaのinterfaceを用いて実現されている。実際のコード（の実装）に関しては転送される側は知る必要はない。
- Javaのオブジェクトの転送機構であるObjectStreamはオブジェクトのクラス名とデータのみを転送する。したがって、転送されたオブジェクトを実際に動作させる（例えば、メソッドを呼び出す）場合にはコードを転送する必要がある。
- コードを転送するためにクラスファイルを転送する機構を用意する必要がある。通常、このためにhttpサーバを用いる。これを自動的に行うクラスがMarshalledObjectStreamである。実行時にjava.rmi.server.codebaseに指定する。

RMIの概要

- インタフェースを、Remoteインタフェースをextendして定義する。これをクライアント、サーバ、双方に置く。
- サーバ側にはリモートのオブジェクトを管理するプロセスであるrmiregistryを起動しておく。
- また、サーバ側に仲介するプログラムであるstubを生成するプログラムであるrmicをつかって、stubを生成しておく。このプログラムは、Remoteインタフェースから、スタブをプログラムを生成する。スケルトン_Skel.class とスタブ_Stub.classが生成される。
- サーバー側のオブジェクトは、UnicastRemoteObjectをsuperクラスとして作成し、サーバ側ではリモートのオブジェクトを登録する。
- クライアント側では登録されているオブジェクトを取り出し、インタフェースを使って呼び出す。

RMIリモートオブジェクトへのインタフェースの定義

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface ShowDate extends Remote {
  public long getCurrentMillis() throws RemoteException;
  public long getMillis() throws RemoteException;
}
```


RMIリモートオブジェクトの実装・登録（1）

```
public class ShowDateImpl extends
      UnicastRemoteObject implements ShowDate {
  long millis = 0;
  Date date = null;
  public ShowDateImpl() throws RemoteException {
    super();
    millis = getCurrentMillis();
    date = new Date(millis);
  }
  public long getCurrentMillis() throws RemoteException {
    System.out.println("getCurrentMillis called!");
    return System.currentTimeMillis();
  }
  public long getMillis() throws RemoteException {
    System.out.println("getMillis called!");
    return millis;
  }
  public static void main(String argv[]){
    if(System.getSecurityManager() == null){
      System.setSecurityManager(new RMISecurityManager());
    }
```


RMIリモートオブジェクトの実装・登録（2）

```
public class ShowDateImpl extends
       UnicastRemoteObject implements ShowDate {
  ……
  public static void main(String argv[]){
    if(System.getSecurityManager() == null){
      System.setSecurityManager(new RMISecurityManager());
    }
    try {
      ShowDateImpl sdi = new ShowDateImpl();
      Naming.rebind("//localhost/TimeServer",sdi);
      System.out.println("TimeServer bound in registry");
    } catch(Exception e){
      System.out.println(e.getMessage());
    }
  }
}
```

RMIリモートオブジェクトを用いたクライアントの例

```
public class client {
  public static void main(String argv[]){
    if(System.getSecurityManager() == null){
      System.setSecurityManager(new RMISecurityManager());
    }
     ShowDate obj = null;
    try {
      String location = "rmi://localhost/TimeServer";
      obj = (ShowDate)Naming.lookup(location);

      long remote_millis = obj.getCurrentMillis();
      long local_millis = System.currentTimeMillis();
      System.out.println("remote =" + remote_millis);
      System.out.println("local =" + local_millis);
    } catch(Exception e){
      System.out.println(e);
    }
  }
}
```


activation

- java.rmi.activation.Actvatableをextendsしてクラスを作る。
- コンストラクタとして、IDと引数データを引数とするコンストラクターを定義する。
- activationGroupのインスタンスを生成する。これは、policyや実行環境を定義するものである。
- activation groupに登録し、IDを取得し、これを使ってグループを生成する。デフォールトのグループに登録。
- activation descriptorを生成する。これには、クラスの名前、クラスがロードされるべきcodebase、コンストラクタに渡される引数を指定する。activationGroupが指定しない場合にはデフォールトのgroupが使われる。
- descriptorをrmidに登録する。ここにstubが返される。
- これをName.bindで、rmiregistryに登録する。
- あとは、プログラムは終了してよい。

RMIのactivationを使ったサーバの例

```
public class ShowDateImpl extends Activatable implements ShowDate {
  public static void main(String argv[]){
    if(System.getSecurityManager() == null){
      System.setSecurityManager(new RMISecurityManager());
    }
    try {
      Properties props = new Properties();
      props.put("java.security.policy",
                "/home/msato/java/tmp/rm-test5/serv/policy.txt");
      ActivationGroupDesc myGroup =
                     new ActivationGroupDesc(props,null);
      ActivationGroupID agi =
         ActivationGroup.getSystem().registerGroup(myGroup);
      ActivationGroup.createGroup(agi,myGroup,0);
      String location = "file:/home/msato/java/tmp/rm-test5/serv/";
      ActivationDesc desc =
              new ActivationDesc("ShowDateImpl",location,null);
      ShowDate rmi = (ShowDate)Activatable.register(desc);
      System.out.println("Got the stub for the ShowDateImpl = "+rmi);
      Naming.rebind("//localhost/TimeServer",rmi);
      System.out.println("Exported ShowDateImpl...");

      System.exit(0);
    } catch(Exception e){
      System.out.println(e.getMessage());
```


Jiniとは

- Jiniはこの分散オブジェクトプログラミングをベースに、いろいろなコンピュータ、家電に入っているプロセッサからスーパーコンピュータまで、ネットワーク上のあらゆる機器（コンピュータ）を「連合(federation)」させるための仕組みを提唱したものである。

- Jiniのもっとも重要な概念として「サービス」がある。
  - ネットワーク上に接続されているコンピュータを単なるデータを交換する対象と考えるのではなく、なんらかのサービスを提供する対象と考える。
  - そのサービスをお互いに交換することによって、分散システムはなんらかの仕事をする。
  - これまで、いわゆるサーバはサービスを提供する担い手であり、クライアントはそのサーバからサービスを受ける形態が一般的であったが、Jiniが想定しているのはネットワーク上の分散システムを構成するコンピュータがお互いにサービスを提供することによって協調作業をするシステムを想定している。

Jiniのlookup サービス

- Jiniでサービスをネットワーク上のどこからでも利用できる。
- サービスはネットワーク上を移動するオブジェクトによって提供される。
- いろいろなサービスがあるとするJiniでは、そのサービスを見つけるための機構「Lookupサービス」が提供されている。
- これによって、ネットワーク上に提供されているサービスを検索し、そのサービスを利用できる
- たとえば、DHCP・・・
- RMIは個々のコンピュータで提供するオブジェクトを管理する（registry）機能を提供しているが、Jiniはこれをネットワーク全体に拡張し、すべてのコンピュータで提供されている機能を検索する機能を提供するものということもできる。

その他の機能

- イベントリスナーモデルを分散環境に拡張した分散イベント(distributed event)の仕組み
  - イベントも分散オブジェクトとして登録され、lookup サービスを通じてやり取りが行われる。
- データベースの更新などの同期を取るために、トランザクションをサポートする機構などもサポートされている。