Tutorial 1: using Bottles
How do we exchange data over YARP easily?

A Bottle is a **Portable** object on which a set of operations are defined: add/retrieve data types (string, int, double, vocab, list)

```c
Bottle b1;
b1.addString("color");
b1.addString("red");
printf("Bottle b1 is: %s\n", b1.toString().c_str());
// should give: color red

Bottle b2;
b2.addString("height");
b2.addInt(15);
printf("Bottle b2 is: %s\n", b2.toString().c_str());
// should give: height 15

Bottle b3;
b3.addList() = b1;
b3.addList() = b2;
printf("Bottle b3 is: %s\n", b3.toString().c_str());
// should give: (color red) (height 15)
```
// alternative way to create a Bottle from textual representation
Bottle b5("(pos left top) (size 10)");
printf("Bottle b5 is: %sn", b5.toString().c_str());
// should give: (pos left top) (size 10)

Bottle b6;
b6 = b5;
b6.addAll((Vocab::encode("req"))); // convert a string into a vocabulary identifier and add it to the bottle
b6.addVocab(Vocab::encode("req"));
// should give: (pos left top) (size 10) (nested ((color red) (height 5))

printf("size check: %d\n", b6.find("size").asInt());
printf("pos check: %s\n", b6.find("pos").asString().c_str());
// find assumes key->value pairs; for lists, use findGroup
printf("pos group check: %s\n", b6.findGroup("pos").toString().c_str());
// see documentation for Bottle::findGroup
printf("nested check: %s\n", b6.find("nested").toString().c_str());
printf("nested height check: %d\n", b6.find("nested").find("height").asInt());

printf("b2 elements %s %d\n", b2.get(0).asString(), b2.get(1).asInt());
// should give: height 15

b6.clear();

Tutorial 2: using ports
A (very) simple example: read data to/from a port

[on terminal 1] yarp server
[on terminal 2] yarp read /read
[on terminal 3] yarp write /write /read

$ yarp write /write /read
Port /write listening at tcp://127.0.0.1:10012
yarp: Sending output from /write to /read using tcp
Added output connection from "/write" to "/read"
hello yarp
1 2 3

$ yarp read /read
Port /read listening at tcp://127.0.0.1:10002
yarp: Receiving input from /write to /read using tcp
hello yarp
1 2 3
How do we get this?

Let’s now to write a simple “relay” executable which takes whatever comes from a port and forwards it to another one.

```c
int main(int argc, char *argv) {
    Network yarp;
    Port inPort;
    inPort.open("/relay/in");

    Port outPort;
    outPort.open("/relay/out");

    while (true) {
        cout << "waiting for input" << endl;
        Bottle input, output;
        inPort.read(input);
        output = input;
        cout << "writing " << output.toString().c_str() << endl;
        outPort.write(output);
    }
    return 0;
}
```
Connect the new module to our network

-yarp disconnect /write /read

-yarp write
   /write

-yarp read
   /read

-yarp connect /write /relay/in

-yarp connect /relay/out /read

-yarp connect /relay/in
It is easy to add, for example, another reader…
Processes can run on different machines, with different OS

```
yarp write
/write
```
```
relay
/relay/in
/relay/out
```
```
yarp read
/read
```
```
yarp connect /relay/out /read
```
```
yarp connect /relay/out /read2
```
```
yarp connect /relay/out /read3
```
```
...
Tutorial 3: decoupling timing
BufferedPort

- In the previous example timing between ports is coupled:
  - The reader waits until data arrives to the port
  - The writer waits until data is transmitted
- Buffered ports allow decoupling time:
  - non blocking read
  - non blocking write
- May loose messages
• Read:
  BufferedPort<Bottle> p; // Create a port.
p.open("/in"); // Give it a name on the network.
while (true) {
  Bottle *b = p.read(); // Read/wait for until data arrives. ...
  // Do something with data in *b
}

• Write:
  BufferedPort<Bottle> p; // Create a port.
p.open("/out"); // Give it a name on the network.
while (true) {
  Bottle& b = p.prepare(); // Get a place to store things. ...
  // Generate data.
  p.write(); // Send the data.
}
• Polling: when you do not want to wait for input data:

```c
BufferedPort<Bottle> p;
...
Bottle *b = p.read(false);
if (b!=NULL) {
    // data received in *b
}
```
Tutorial 3: getting callbacks
• Callbacks: useful if you want to be notified when data arrives
• Easy to do with BufferedPorts

class DataPort : public BufferedPort<Bottle> {
    virtual void onRead(Bottle& b) {
        // process data in b
    }
};

...  
DataPort p;
p.useCallback(); // input should go to onRead() callback
p.open("/in");
Things are a bit more complicated with normal ports

class DataProcessor : public PortReader {
    virtual bool read(ConnectionReader& connection) {
        Bottle b;
        bool ok = b.read(connection);
        if (!ok) return false;
        // process data in b
        return true;
    }
};

Port p;
p.open(..)
DataProcessor processor;
...
p.setReader(processor); // no need to call p.read() on port any more.
Tutorial 4: getting replies
Port p;                  // Create a port.
p.open("/out");         // Give it a name on the network.
while (true) {
    Bottle in, out;     // Make places to store things.
    ...                // prepare command "out".
    p.write(out, in);   // send command, wait for reply.
    ...                // process response "in".
}
Server side

Port p;  // Create a port.
p.open("/in");  // Give it a name on the network.
Bottle in, out;  // Make places to store things.
while (true) {
    p.read(in,true);  // Read and warn that we'll be replying.
    ...  // Do something with data, prepare reply
    p.reply(out);  // send reply.
}
Replies in a callback

class DataProcessor : public PortReader {
   virtual bool read(ConnectionReader& connection) {
      Bottle in, out;
      bool ok = in.read(connection);
      if (!ok) return false;
      ... // process data "in", prepare "out"
      ConnectionWriter *returnToSender = connection.getWriter();
      if (returnToSender!=NULL) {
         out.write(*returnToSender);
      }
      return true;
   }
};
DataProcessor processor;
...
p.setReader(processor);  // no need to call p.read() on port any more.