General Problems - 1

Analysis of solutions

António Rui Borges

Summary

Palindrome

- *Base solution*
- *Generalizing storage capacity*
- *Using the inheritance mechanism*
- *Treating error conditions*
- *Introducing parametric data types*

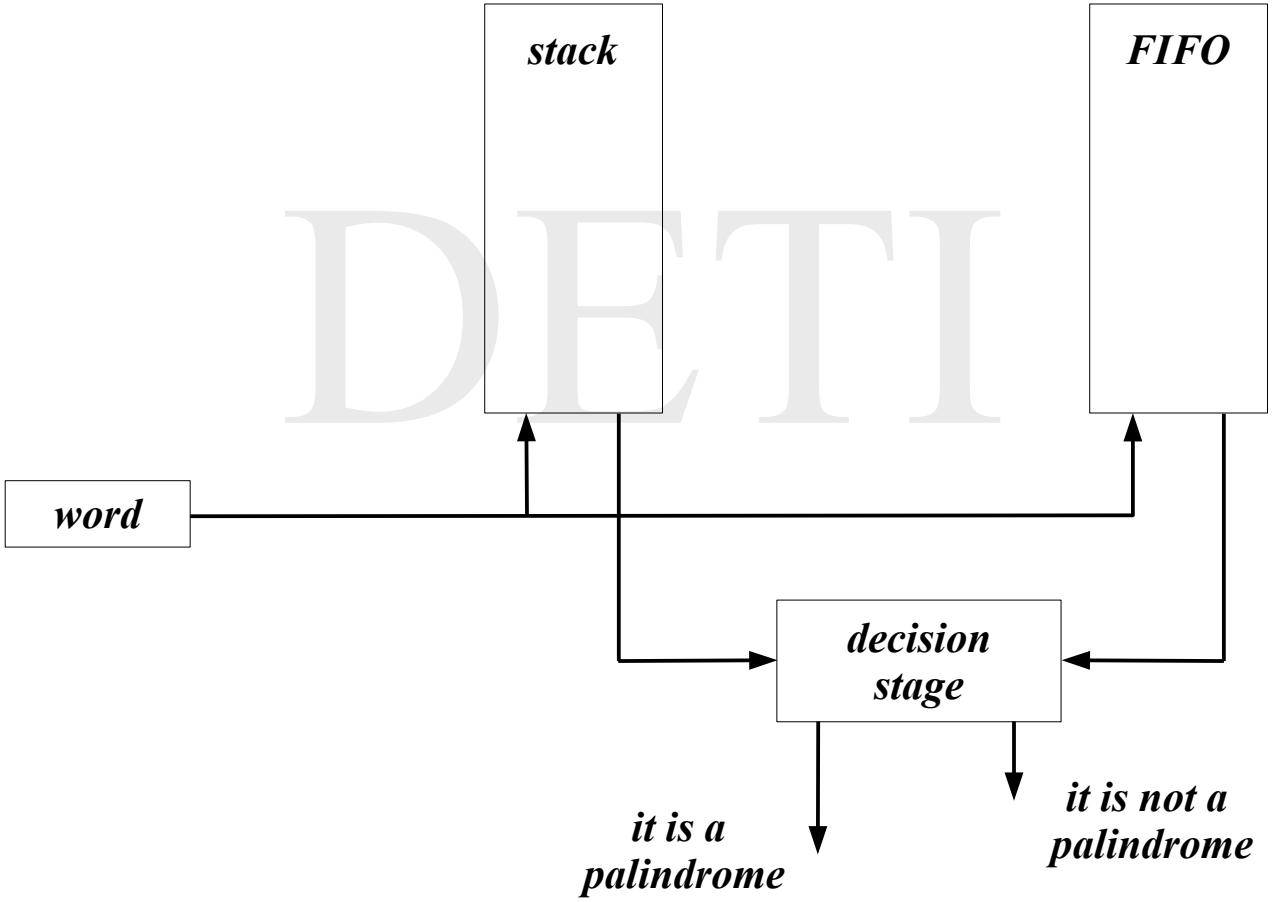
Palindrome - 1

A *palindrome* is a word which is read in the same fashion starting either from left to right, or from right to left.

Solution to be sought

- read the word from from the standard input device
- write the word one character at the time both to a *stack* and a *FIFO*
- read the word one character at the time both from the stack and the FIFO
  - compare the read values to take the decision.

Palindrome - 2



Base solution - 1

It is a naive implementation which meets the specifications, but that does try to develop more general data types for later reuse

- *stack* and *FIFO* memories are modeled as concrete data types that only store characters
- the only feature of the object oriented paradigm that is included, is the faculty of reserving storage space for the memories in *runtime*.

Base solution - 2

```
public class StackChar
{
   private int stackPnt;
   private char [] stack;
   public StackChar (int nElem)
   {
     if (nElem > 0)
        { stack = new char [nElem];
          stackPnt = 0;
        }
   }
   public void push (char val)
   {
     if (stackPnt < stack.length)
        { stack[stackPnt] = val;
          stackPnt += 1;
        }
   }
   public char pop ()
   {
     char val = '\0';
     if (stackPnt > 0)
        { stackPnt -= 1;
          val = stack[stackPnt];
        }
     return val;
   }
}
```

stack is implemented as a memory for character storage

Base solution - 3

```
public class FIFOChar
{
   private int inPnt, outPnt;
   private char [] fifo;
   private boolean empty;
   public FIFOChar (int nElem)
   {
     if (nElem > 0)
        { fifo = new char [nElem];
          inPnt = outPnt = 0;
          empty = true;
        }
   }
   public void in (char val)
   {
     if ((inPnt != outPnt) || empty)
        { fifo[inPnt] = val;
          inPnt = (inPnt + 1) % fifo.length;
          empty = false;
        }
   }
   public char out ()
   {
     char val = '\0';
     if (!empty)
        { val = fifo[outPnt];
          outPnt = (outPnt + 1) % fifo.length;
          empty = (inPnt == outPnt);
        }
     return val;
   }
}
```

FIFO is implemented as a memory for character storage

Base solution - 4

```
import genclass.GenericIO;

public class Palindrome
{
   public static void main (String [] args)
   {
     String word;

     GenericIO.writeString ("Qual é a palavra? ");
     word = GenericIO.readlnString ();
     if (word == null) word = "";

     StackChar stack = new StackChar (word.length ());
     FIFOChar fifo = new FIFOChar (word.length ());

     for (int i = 0; i < word.length (); i++)
     { stack.push (word.charAt (i));
       fifo.in (word.charAt (i));
     }

     for (int i = 0; i < word.length (); i++)
       if (stack.pop () != fifo.out ())
          { GenericIO.writelnString ("It is not a palindrome!");
            return;
          }
     GenericIO.writelnString ("It is a palindrome!");
   }
}
```

reserving storage space in *runtime*

Making the storage capability more general - 1

The first questions to be asked, bearing in mind a later reuse, are:

Why should one implement stack and FIFO memories which are only able to store characters?

Would not be more productive to model them as memories capable of storing any type of values?

This can be achieved by defining the internal storage area as an *array* of variables of type `Object` (the *supertype* of all reference data types in Java).

Notice that, when one intends to store values of a *primitive* data type, one can always replace them by the associated *wrapping* data type (here, `Character` instead of `char`).

Making the storage capability more general - 2

```
public class StackGen
{
   private int stackPnt;
   private Object [] stack;
   public StackGen (int nElem)
   {
     if (nElem > 0)
        { stack = new Object [nElem];
          stackPnt = 0;
        }
   }
   public void push (Object val)
   {
     if (stackPnt < stack.length)
        { stack[stackPnt] = val;
          stackPnt += 1;
        }
   }
   public Object pop ()
   {
     Object val = null;
     if (stackPnt > 0)
        { stackPnt -= 1;
          val = stack[stackPnt];
        }
     return val;
   }
}
```

stack is implemented as a memory for generic data type storage

Making the storage capability more general - 3

```
public class FIFOGen
{
   private int inPnt, outPnt;
   private Object [] fifo;
   private boolean empty;
   public FIFOGen (int nElem)
   {
     if (nElem > 0)
        { fifo = new Object [nElem];
          inPnt = outPnt = 0;
          empty = true;
        }
   }
   public void in (Object val)
   {
     if ((inPnt != outPnt) || empty)
        { fifo[inPnt] = val;
          inPnt = (inPnt + 1) % fifo.length;
          empty = false;
        }
   }
   public Object out ()
   {
     Object val = null;
     if (!empty)
        { val = fifo[outPnt];
          outPnt = (outPnt + 1) % fifo.length;
          empty = (inPnt == outPnt);
        }
     return val;
   }
}
```

FIFO is implemented as a memory for generic data type storage

Making the storage capability more general - 4

```
import genclass.GenericIO;
import java.util.Objects;
public class Palindrome
{
   public static void main (String [] args)
   {
     String word;
     GenericIO.writeString ("Qual é a palavra? ");
     word = GenericIO.readlnString ();
     if (word == null) word = "";
     StackGen stack = new StackGen (word.length ());
     FIFOGen fifo = new FIFOGen (word.length ());
     for (int i = 0; i < word.length (); i++)
     { stack.push (word.charAt (i));
       fifo.in (word.charAt (i));
     }
     for (int i = 0; i < word.length (); i++)
       if (!Objects.equals ((Character) stack.pop (), (Character) fifo.out ()))
          { GenericIO.writelnString ("It is not a palindrome!");
            return;
          }
     GenericIO.writelnString ("t is a palindrome!");
   }
}
```

inboxing

Applying the inheritance mechanism - 1

A further question to be asked is that, being *stacks* and *FIFOs* just memory devices distinguished only by their access mode, one could think of defining first a generic memory, characterized by its internal storage space and by the conceptual operations of *writing* and *reading a value*, and only then to proceed to the specification of a particular access mode.

This means, according to the object oriented paradigm, that

- one should proceed to define an abstract data type which describes only the internal storage space and the conceptual operations of *writing* and *reading a value*
- applying next the inheritance mechanism to specialize the base data type into concrete data types where the access mode is specified.

Applying the inheritance mechanism - 2

```
public abstract class MemObject
{
   protected Object [] mem;
   protected MemObject (int nElem)
   {
     if (nElem > 0) mem = new Object [nElem];
   }
   protected abstract void write (Object val);
   protected abstract Object read ();
}
```

virtual methods

Applying the inheritance mechanism - 3

```
public class MemStack extends MemObject
{
   private int stackPnt;
   public MemStack (int nElem)
   {
     super (nElem);
     stackPnt = 0;
   }
   @Override
   public void write (Object val)
   {
     if (stackPnt < stack.length)
        { mem[stackPnt] = val;
          stackPnt += 1;
        }
   }
   @Override
   public Object read ()
   {
     Object val = null;
     if (stackPnt > 0)
        { stackPnt -= 1;
          val = mem[stackPnt];
        }
     return val;
   }
}
```

calling the supertype constructor

virtual method implementation

virtual method implementation

Applying the inheritance mechanism - 4

```
public class MemFIFO extends MemObject
{
   private int inPnt, outPnt;
   private boolean empty;
   public MemFIFO (int nElem)
   {
     super (nElem);
     inPnt = outPnt = 0;
     empty = true;
   }
   @Override
   public void write (Object val)
   {
     if ((inPnt != outPnt) || empty)
        { mem[inPnt] = val;
          inPnt = (inPnt + 1) % mem.length;
          empty = false;
        }
   }
   @Override
   public Object read ()
   {
     Object val = null;
     if (!empty)
        { val = mem[outPnt];
          outPnt = (outPnt + 1) % mem.length;
          empty = (inPnt == outPnt);
        }
     return val;
   }
}
```

calling the supertype constructor

virtual method implementation

virtual method implementation

Treating error conditions - 1

Although the code that has been presented is robust, no means were provided to signal directly if the calling of the operations upon the memories is carried out in a correct or incorrect fashion. The object oriented paradigm deals with this kind of signaling through the casting of *exceptions*.

Exceptions are modeled in Java by using data types derived from the reference data type `Trowable`. Two other data types, `Exception` and `Error`, are then defined using it as its common supertype. The former concerns cases that an well-organized application should be able to deal with; the latter concerns more serious situations typically associated with operational malfunctions of the computer system, the Java virtual machine, or with code incompatibilities.

Thus, user-defined exceptions should be defined as derived data types of the reference data type `Exception`.

Treating error conditions - 2

```
public class MemException extends Exception
{
   public MemException (String errorMessage)
   {
     super (errorMessage);
   }

   public MemException (String errorMessage, Throwable cause)
   {
     super (errorMessage, cause);
   }
}
```

associating a message to the exception

associating not only a message to the exception, but also an underlying exception

Treating error conditions - 3

signaling the casting of an exception if the operation is called in abnormal situations

```
public abstract class MemObject
{
   protected Object [] mem;

   protected MemObject (int nElem) throws MemException
   {
     if (nElem > 0)
        mem = new Object [nElem];
        else throw new MemException ("Illegal storage size!");
   }

   protected abstract void write (Object val) throws MemException;

   protected abstract Object read () throws MemException;
}
```

casting of the exception with specification of its cause

Treating error conditions - 4

signaling the casting of an exception if the operation is called in abnormal situations

```
public class MemStack extends MemObject
{
   private int stackPnt;
   public MemStack (int nElem) throws MemException
   {
     super (nElem);
     stackPnt = 0;
   }
   @Override
   public void write (Object val) throws MemException
   {
     if (stackPnt < mem.length)
        { mem[stackPnt] = val;
          stackPnt += 1;
        }
        else throw new MemException ("Stack full!");
   }
   @Override
   public Object read () throws MemException
   {
     if (stackPnt > 0)
        { stackPnt -= 1;
          return (mem[stackPnt]);
        }
        else throw new MemException ("Stack empty!");
   }
}
```

casting of the exception with specification of its cause

casting of the exception with specification of its cause

Treating error conditions - 5

```
public class MemFIFO extends MemObject
{
   private int inPnt, outPnt;
   private boolean empty;
   public MemFIFO (int nElem) throws MemException
   {
     super (nElem);
     inPnt = outPnt = 0;
     empty = true;
   }
   @Override
   public void write (Object val) throws MemException
   {
     if ((inPnt != outPnt) || empty)
        { mem[inPnt] = val;
          inPnt = (inPnt + 1) % mem.length;
          empty = false;
        }
        else throw new MemException ("Fifo full!");
   }
   @Override
   public Object read () throws MemException
   {
     Object val;
     if (!empty)
        { val = mem[outPnt];
          outPnt = (outPnt + 1) % mem.length;
          empty = (inPnt == outPnt);
        }
        else throw new MemException ("Fifo empty!");
     return val;
   }
}
```

signaling the casting of an exception if the operation is called in abnormal situations

casting of the exception with specification of its cause

casting of the exception with specification of its cause

Treating error conditions - 6

```
import genclass.GenericIO;
import java.util.Objects;
public class Palindrome
{
   public static void main (String [] args)
   {
     String word;
     int i = 0;
     GenericIO.writeString ("Qual é a palavra? ");
     word = GenericIO.readlnString ();
     if (word == null) word = "";
     try
     { MemStack stack = new MemStack (word.length ());
       MemFIFO fifo = new MemFIFO (word.length ());
       try
       { for (i = 0; i < word.length (); i++)
         { stack.write (word.charAt (i));
           fifo.write (word.charAt (i));
         }
       }
       catch (MemException e)
       { GenericIO.writelnString ("Erro: ", e.getMessage (),
                                  " na iteração " + (i+1));
         GenericIO.writelnString ("Erro: ", e.toString ());
         e.printStackTrace ();
         System.exit (1);
       }
```

variable declaration and initialization is required due to the chance of error during execution

catching the exception resulting from memory instantiation

catching the exception resulting from writing a word character at a time to the memories

dealing with the exception resulting from writing a word character at a time to the memories

Treating error conditions - 7

```
      try
      { for (i = 0; i < word.length (); i++)
          if (!Objects.equals ((Character) stack.read (),
                               (Character) fifo.read ()))
            { GenericIO.writelnString ("It is not a palindrome!");
              return;
            }
      }
      catch (MemException e)
      { GenericIO.writelnString ("Erro: ", e.getMessage (),
                                 " na iteração " + (i+1));
        GenericIO.writelnString ("Erro: ", e.toString ());
        e.printStackTrace ();
        System.exit (1);
      }
      GenericIO.writelnString ("It is a palindrome!");
    }
    catch (MemException e)
    { GenericIO.writelnString ("Erro: ", e.getMessage ());
      GenericIO.writelnString ("Erro: ", e.toString ());
      e.printStackTrace ();
      System.exit (1);
    }
  }
}
```

catching the exception resulting from reading a word character at a time from the memories

dealing with the exception resulting from reading a word character at a time from the memories

dealing with the exception resulting from memory instantiation

Treating error conditions - 8

```
Erro: Illegal storage size!
Erro: MemException: Illegal storage size!
MemException: Illegal storage size!
at MemObject.<init>(MemObject.java:23)
at MemStack.<init>(MemStack.java:21)
at Palindrome.main(Palindrome.java:32)
Java Result: 1
```

error resulting from instantiating a memory of size zero

```
Erro: Fifo full! na iteração 6
Erro: MemException: Fifo full!
MemException: Fifo full!
at MemFIFO.write(MemFIFO.java:39)
at Palindrome.main(Palindrome.java:40)
Java Result: 1
```

error resulting from trying to write to a full *FIFO*

```
Erro: Stack empty! na iteração 18
Erro: MemException: Stack empty!
MemException: Stack empty!
at MemStack.read(MemStack.java:50)
at Palindrome.main(Palindrome.java:56)
Java Result: 1
```

error resulting from trying to read from an empty *stack*

Treating error conditions - 9

Alternative version

```
public class MemStack extends MemObject
{
   private int stackPnt;
   public MemStack (int nElem) throws MemException
   {
     super (nElem);
     stackPnt = 0;
   }
   @Override
   public void write (Object val) throws MemException
   {
     try
     { mem[stackPnt] = val;
       stackPnt += 1;
     }
     catch (ArrayIndexOutOfBoundsException e)
     { throw new MemException ("Stack full!", e);
     }
   }
   @Override
   public Object read () throws MemException
   {
     try
     { stackPnt -= 1;
       return (mem[stackPnt]);
     }
     catch (ArrayIndexOutOfBoundsException e)
     { stackPnt += 1;
       throw (MemException) (new MemException ("Stack empty!")).initCause (e);
     }
   }
}
```

dealing with the underlying exception and generating the exception resulting from writing a character to a full *stack*

dealing with the underlying exception and generating the exception resulting from reading a character from a full *stack*

Treating error conditions - 10

```
Erro: Stack full! na iteração 9
Erro: MemException: Stack full!
MemException: Stack full!
at MemStack.write(MemStack.java:37)
at Palindrome.main(Palindrome.java:39)
Caused by: java.lang.ArrayIndexOutOfBoundsException: 8
at MemStack.write(MemStack.java:33)
... 1 more
Java Result: 1
```

error resulting from trying to write to a full *stack*

```
Erro: Stack empty! na iteração 29
Erro: MemException: Stack empty!
MemException: Stack empty!
at MemStack.read(MemStack.java:54)
at Palindrome.main(Palindrome.java:56)
Caused by: java.lang.ArrayIndexOutOfBoundsException: -1
at MemStack.read(MemStack.java:50)
... 1 more
Java Result: 1
```

error resulting from trying to read from an empty *stack*

Introducing parametric data types - 1

In building data types that represent *stack* and *FIFO* memories, an extra degree of abstraction may be introduced. In the current version of the code, the instantiation of variables of these data types allow the *simultaneous* storage of values of any reference data types. However, in most applications, one aims to store values of a *single* data type or, at most, of a set of somewhat related data types.

In object oriented programming, this can be achieved by specifying parametrically the data type, that is, one builds a data type capable of storing values of an yet undisclosed data type `R`, which is later on materialized when variables of the memory data type are declared.

In Java, this is carried out by the *Generics* construct.

Introducing parametric data types - 2

```
public abstract class MemObject<R>
{
   protected R [] mem;
   protected MemObject (R [] storage) throws MemException
   {
     if (storage.length > 0)
        mem = storage;
        else throw new MemException ("Illegal storage size!");
   }
   protected abstract void write (R val) throws MemException;
   protected abstract R read () throws MemException;
}
```

parametrization of `MemObject` data type

notice that the storage area can not be instantiated inside (it is still unknown at this stage)

Introducing parametric data types - 3

parametrization of `MemStack` data type

```
public class MemStack<R> extends MemObject<R>
{
   private int stackPnt;
   public MemStack (R [] storage) throws MemException
   {
     super (storage);
     stackPnt = 0;
   }
   @Override
   public void write (R val) throws MemException
   {
     if (stackPnt < mem.length)
        { mem[stackPnt] = val;
          stackPnt += 1;
        }
        else throw new MemException ("Stack full!");
   }
   @Override
   public R read () throws MemException
   {
     if (stackPnt != 0)
        { stackPnt -= 1;
          return mem[stackPnt];
        }
        else throw new MemException ("Stack empty!");
   }
}
```

Introducing parametric data types - 4

parametrization of `MemFIFO` data type

```
public class MemFIFO<R> extends MemObject<R>
{
   private int inPnt, outPnt;
   private boolean empty;
   public MemFIFO (R [] storage) throws MemException
   {
     super (storage);
     inPnt = outPnt = 0;
     empty = true;
   }
   @Override
   public void write (R val) throws MemException
   {
     if ((inPnt != outPnt) || empty)
        { mem[inPnt] = val;
          inPnt = (inPnt + 1) % mem.length;
          empty = false;
        }
        else throw new MemException ("Fifo full!");
   }
   @Override
   public R read () throws MemException
   {
     R val;
     if (!empty)
        { val = mem[outPnt];
          outPnt = (outPnt + 1) % mem.length;
          empty = (inPnt == outPnt);
        }
        else throw new MemException ("Fifo empty!");
     return val;
   }
}
```

Introducing parametric data types - 5

```
import genclass.GenericIO;
import java.util.Objects;

public class Palindrome
{
   public static void main (String [] args)
   {
     String word;
     int i = 0;

     GenericIO.writeString ("Qual é a palavra? ");
     word = GenericIO.readlnString ();
     if (word == null) word = "";
     try
     { MemStack<Character> stack = new MemStack<> (new Character [word.length ()]);
       MemFIFO<Character> fifo = new MemFIFO<> (new Character [word.length ()]);
       try
       { for (i = 0; i < word.length (); i++)
         { stack.write (word.charAt (i));
           fifo.write (word.charAt (i));
         }
       }
       catch (MemException e)
       { GenericIO.writelnString ("Erro: ", e.getMessage (),
                                  " na iteração " + (i+1));
         GenericIO.writelnString ("Erro: ", e.toString ());
         e.printStackTrace ();
         System.exit (1);
       }
```

making parametric data type concrete

Introducing parametric data types - 6

making parametric data type concrete

```
        try
        { for (i = 0; i < word.length (); i++)
            if (!Objects.equals (stack.read (), fifo.read ()))
               { GenericIO.writelnString ("It is not a palindrome!");
                 return;
               }
        }
        catch (MemException e)
        { GenericIO.writelnString ("Erro: ", e.getMessage (),
                                   " na iteração " + (i+1));
          GenericIO.writelnString ("Erro: ", e.toString ());
          e.printStackTrace ();
          System.exit (1);
        }
        GenericIO.writelnString ("It is a palindrome!");
      }
      catch (MemException e)
      { GenericIO.writelnString ("Erro: ", e.getMessage ());
        GenericIO.writelnString ("Erro: ", e.toString ());
        e.printStackTrace ();
        System.exit (1);
      }
    }
  }
```