## 5 Classes and Objects

- Introduction
- Data encapsulation
- Example: Rational Numbers
- Wrapper classes
- Standard datatypes Queue, Stack, List
- Inheritance
- Chains of constructors
- Referencing superclasses and subclasses
- Override annotation
- Abstract methods & abstract classes
- Interfaces
- Inner Classes

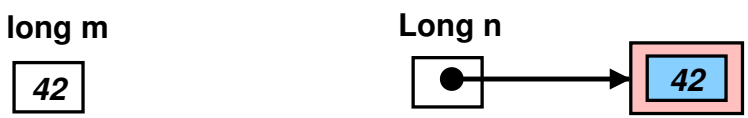
**'wrapper' classes ('Boxing')**:

In Java **`int`**, **`long`**, **`short`**, **`byte`**, **`char`**, **`float`**, **`double`** and **`boolean`** are primitive data types and not classes.Thus such types cannot be used where object references are expected.

Sometimes it can be useful to treat primitive values as objects. Java provides special classes ('wrapper') that allow to 'wrap' a value into an object:

| primitive type | wrapper class |
|---|---|
| **`boolean`** | **`Boolean`** |
| **`byte`** | **`Byte`** |
| **`short`** | **`Short`** |
| **`char`** | **`Character`** |
| **`int`** | **`Integer`** |
| **`long`** | **`Long`** |
| **`float`** | **`Float`** |
| **`double`** | **`Double`** |

Example:



## Example: `java.lang.Integer`

```
1 public final class Integer
2                    extends Number implements Comparable<Integer>
```

- The class **`Integer`** wraps an **`int`** value into an object.
- **`Integer`** objects contain a single **`int`** data field.

Constructors:

- **`Integer (int value)`**, i.e. **`int`**-Wert $\rightsquigarrow$ **`Integer`** object (‘wrap’)
- **`Integer (String s)`**, i.e. **`String`** $\rightsquigarrow$ **`Integer`** object

Constants (selection):

- **`static int MAX_VALUE`** ($= \mathbf{2^{31} - 1}$);
- **`static int MIN_VALUE`** ($= \mathbf{-2^{31}}$);

# Small selection of the methods of **public final class** Integer:

```
1  public static int parseInt(String s)
2              // generates int value from String
3
4  public static Integer valueOf(String s)
5              // generates Integer object from String
6
7  public int intValue ()
8              //generates int value from Integer object
9              //('unwrap')
10
11 public int compareTo(Integer iObj)
12             //numeric comparison of two Integer objects
13             // e.g.: x = int1.compareTo(int2);
14             //    x == 0 if: intValue(int1) == intValue(int2)
15             //    x < 0  if: intValue(int1) <  intValue(int2)
16             //    x > 0  if: intValue(int1) >  intValue(int2)
17
18 public static String toString(int i)
19             // generates String from int value
20
21 public String toString()
22             // generates String from Integer object
```

**Auto-Boxing, Auto-Unboxing:** Automatic conversion between wrapper class and primitive type (auto-boxing, auto-unboxing)

- explicit conversion (required until Java 1.4.2):

```
1 int i = 15;
2 Integer iObj = new Integer (i);
3 i = iObj.intValue();
4
5 Integer [] iArray = new Integer [10];
6 iArray [5] = new Integer (54286);
7 int value = iArray [5].intValue();
```

- Auto-(Un)boxing (since Java 1.5/5.0):

```
1 int i = 15;
2 Integer iObj = i;
3 i = iObj;
4
5 Integer [] iArray = new Integer [10];
6 iArray [5] = 54286;
7 int value = iArray [5];
```

## Corresponding Example `K5B04E_AutoBox`:

```
1  public class AutoBox {
2    public static void main (String args[]) {
3
4      int i = 54286;
5
6      Integer iObj = new Integer (i);
7      Integer jObj = new Integer (i);
8
9      if (i == iObj)
10       System.out.println( "i␣=␣iObj␣=␣" + iObj + "␣(values!)\n");
11
12     if (jObj != iObj)
13       System.out.println( "jObj␣!=␣iObj␣(references!)\n");
14
15     i = iObj / 2;
16     System.out.println( "iObj␣/␣2␣=␣" + i + "␣(Auto-Unboxing)");
17   }
18 }
```

## 5 Classes and Objects

- Introduction
- Data encapsulation
- Example: Rational Numbers
- Wrapper classes
- Standard datatypes Queue, Stack, List
- Inheritance
- Chains of constructors
- Referencing superclasses and subclasses
- Override annotation
- Abstract methods & abstract classes
- Interfaces
- Inner Classes

## Addresses, Pointer, References:

- Every data item stored in a computer has a (memory) **address**.
- A **pointer** is an address and is used to
  - locate and identify variables, Strings, arrays and other complex structures
  - construct more complex structures by chaining simple structures
- Some programming languages allow to compute with pointers (‘pointer swizzling’)
- **References** are pointers, but with restricted semantics: In Java references can only refer to existing objects, ‘pointer swizzling’ is not possible.

**References** can be used to combine simple elements to **dynamic data structures**.

Examples are

- **Queue**
  Collection of elements, organized following the **FIFO** principle (first in, first out).
  can be used when data or events should be processed in the order of their arrival.
- **Stack**
  Collection of elements, organized following the **LIFO** principle (last in, first out).
  can be used to for resolving recursion or for processing context-free languages
- **List**
  Linear collection of elements, may be sorted.

# Basic class '**Elem**' for the construction of complex structures (List, Stack, Queue, ...):

```
1  public class Elem {
2   public Elem () { } constructor
3   public Elem (Object obj) { setObject(obj); } constructor
4
5   private Object obj;  data (arbitrary object)
6   private Elem next; reference to next element
7
8   public void setObject (Object newObj) { data access method
9       obj = newObj;
10  }
11  public Object getObject () { data access method
12      return obj;
13  }
14  public void setNext (Elem nextElem) { access to reference
15      next = nextElem;
16  }
17  public Elem getNext () { access to reference
18      return next;
19  }
20
21  @Override public String toString () { return obj.toString ();  }
22 }
```

- Every instance `x` of `Elem` (i.e., every object of this class) includes as instance variables `Object obj` and `Elem next`.
- Here, `x` does not store the actual data of these objects, but only references to them.
- In particular, this can be the **`null`** reference, for which one must watch out, for example
  `y = x.getNext(); if (y != null ){...process y...}`.
- Self-reference is often intended, for example `x.setNext(x)`.
- `setObject` can take arbitrary *objects* even auto-boxing using wrapper classes is allowed, e.g., `x.setObject("text")` or `x.setObject(42)`.
- `getObject` returns an *object* that needs to be casted to the intended data type, e.g.,
  `String s = (String) x.getObject()`,
  `Integer i = (Integer) x.getObject()` or even
  **`int`** `i = (Integer) x.getObject()` (via auto-unboxing).
- More on casts and `@Override` in the chapter about *inheritance*!

**Queue**: The data structure **`Queue`** can be characterized by two operations:

- **`enQueue`**: inserts an object at the end of the queue.
- **`deQueue`**: removes the first object from the queue.

more useful operations:

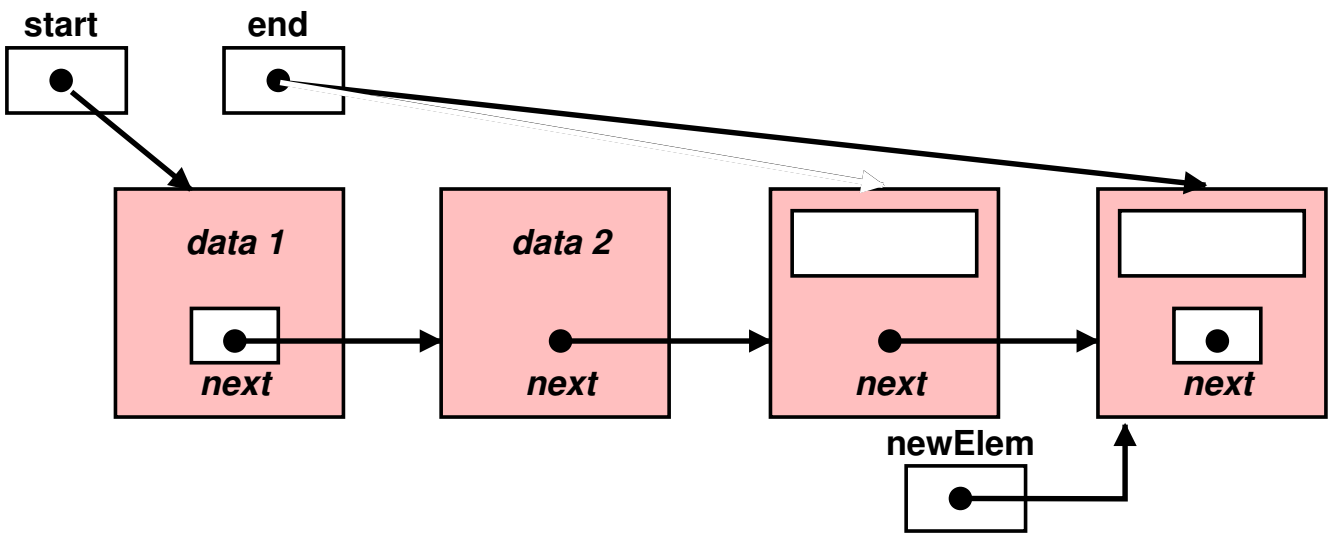
- **`isEmpty`**: checks if the queue is empty.
- **`isFull`**: checks if the queue is empty or if the maximal size is reached.
- **`peek`**: returns the first object, but does not remove it

# Queue based on objects of the class **Elem**:

```
1  public class Queue {
2   public Queue() { }
3   private Elem start, end;  chaining
4
5   public void enQueue (Elem newElem) {  insert an element
6      if (start == null)     start = newElem;
7          else          end.setNext (newElem);
8      end = newElem;
9   }
10  public Elem deQueue () {  remove the 'oldest' element
11     if (start == null) return null;
12     Elem temp = start;
13     start = start.getNext();
14     if (start == null) end = null;
15     return temp;
16  }
17  @Override public String toString () {   ... for debugging ...
18     Elem position = start;
19     String str = "";
20     while (position != null) {
21        str += position.getObject().toString() + "␣␣";
22        position = position.getNext();
23     }
24     return str;
25 } }
```

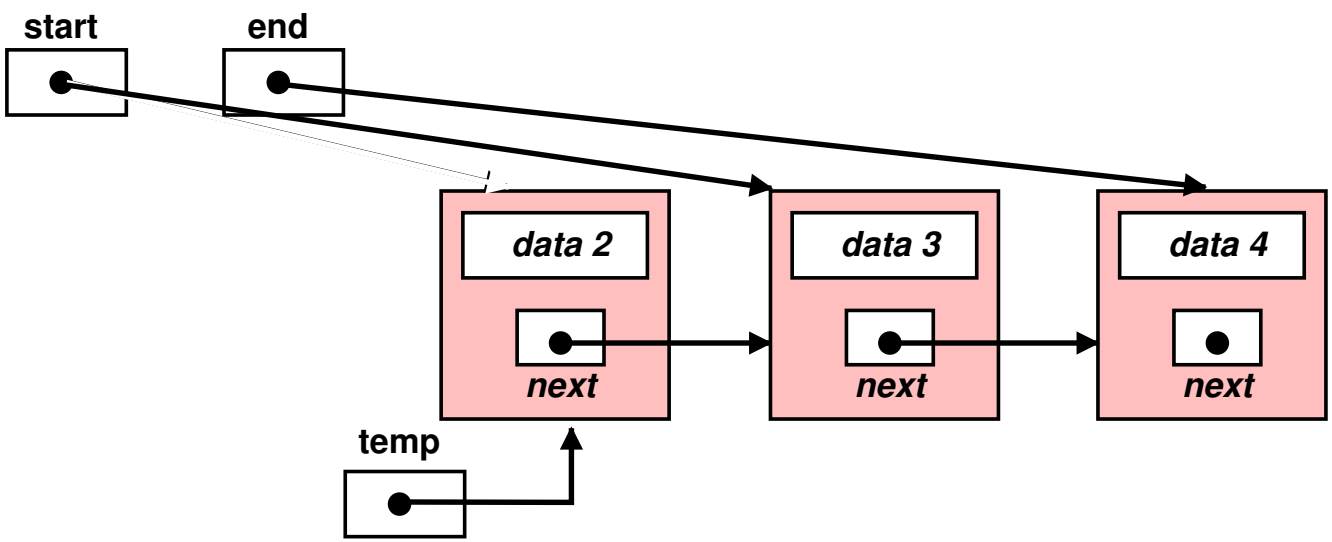
# **enQueue**: Insert an element at the end of the queue

```
1    public void enQueue (Elem newElem) {
2       if (start == null)
3          start = newElem;
4       else
5          end.setNext (newElem);
6       end = newElem;
7    }
8
9    ...
```



# `deQueue`: remove the first element

```
1  public Elem deQueue () {
2        if (start == null)
3           return null;
4        Elem temp = start;
5        start = start.getNext();
6        if (start == null)
7           end = null;
8        return temp;
9     }
```



## Queue with **visible** element structure

```
1  public class Queue {
2   public Queue() {}
3   private Elem start, end;
4
5   public void enQueue
6                (Elem newElem){
7
8
9    if (start == null)
10      start = newElem;
11   else
12      end.setNext (newElem);
13   end = newElem;
14  }
15   public Elem deQueue () {
16   if (start == null)
17      return null;
18   Elem temp = start;
19   start = start.getNext();
20   if (start == null)
21      end = null;
22   return temp;
23  }
```

## modified queue with **hidden** element structure

```
1  public class Queue {
2   public Queue() {}
3   private Elem start, end;
4
5   public void enQueue
6                (Object newObj}){
7    Elem newElem =
8                new Elem(newObj);
9    if (start == null)
10      start = newElem;
11   else
12       end.setNext (newElem);
13   end = newElem;
14  }
15  public Object deQueue () {
16   if (start == null)
17      return null;
18   Elem temp = start;
19   start = start.getNext();
20   if (start == null)
21      end = null;
22   return temp.getObject();
23  }
```

**interface vs. implementation**:

A class and its objects provide a unique **interface**, defined by the methods and variables visible from the outside.

A class $C_1$ can be replaced by a class $C_2$, if $C_2$

- provides a syntactically and semantically identical interface to the outside world
- uses only services of existing classes

Thus: the implementation of a class can be changed locally, if

- the interface is not changed
- no services of additional classes are needed

# Examples K5B05E_... with two implementations of Queue:

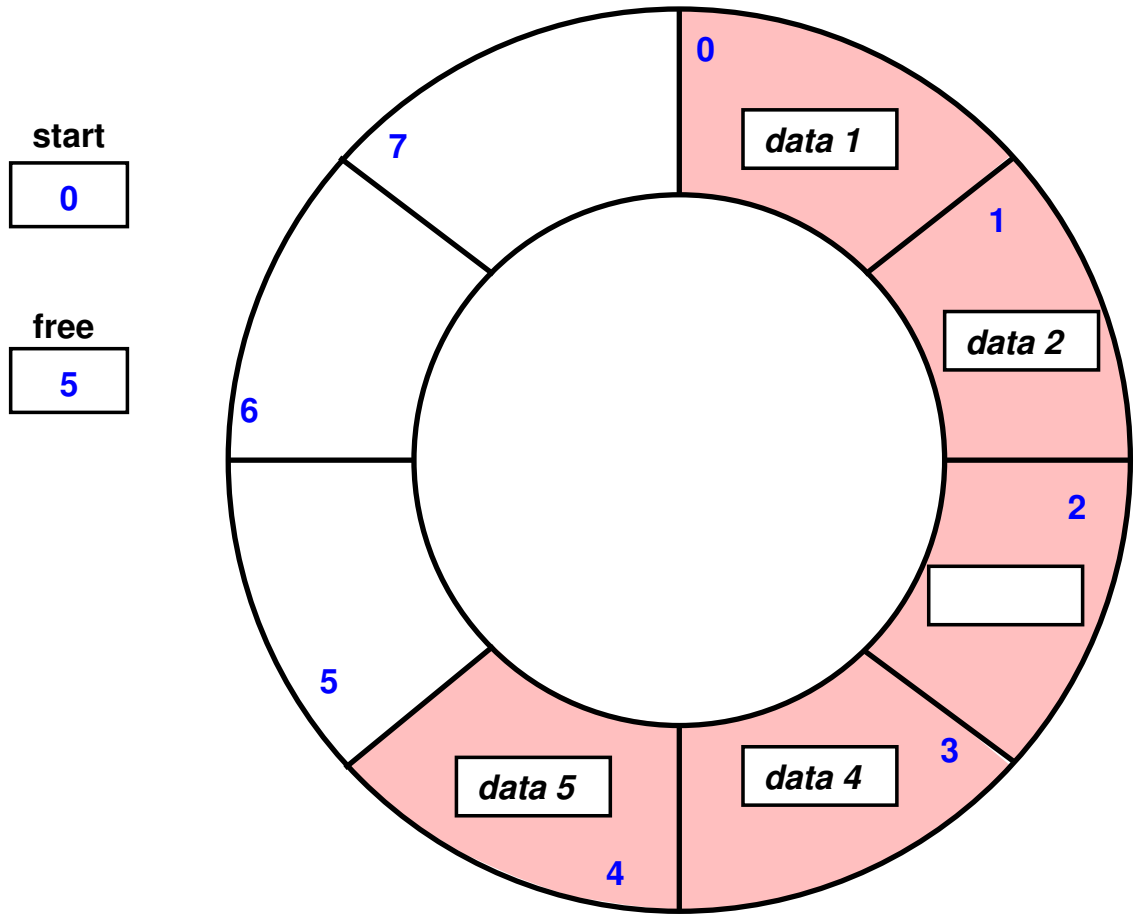
## K5B05E_Queue_Linked (with linked elements)

```
1  public class Queue {
2
3   public Queue ()  {}
4
5   private Elem start, end;
6
7
8
9
10
11
12  public void enQueue
13                (Object newObj) {
14    Elem newElem =
15                new Elem(newObj);
16    if (start == null)
17      start = newElem;
18    else
19      end.setNext (newElem);
20    end = newElem;
21  }
22  ...
```

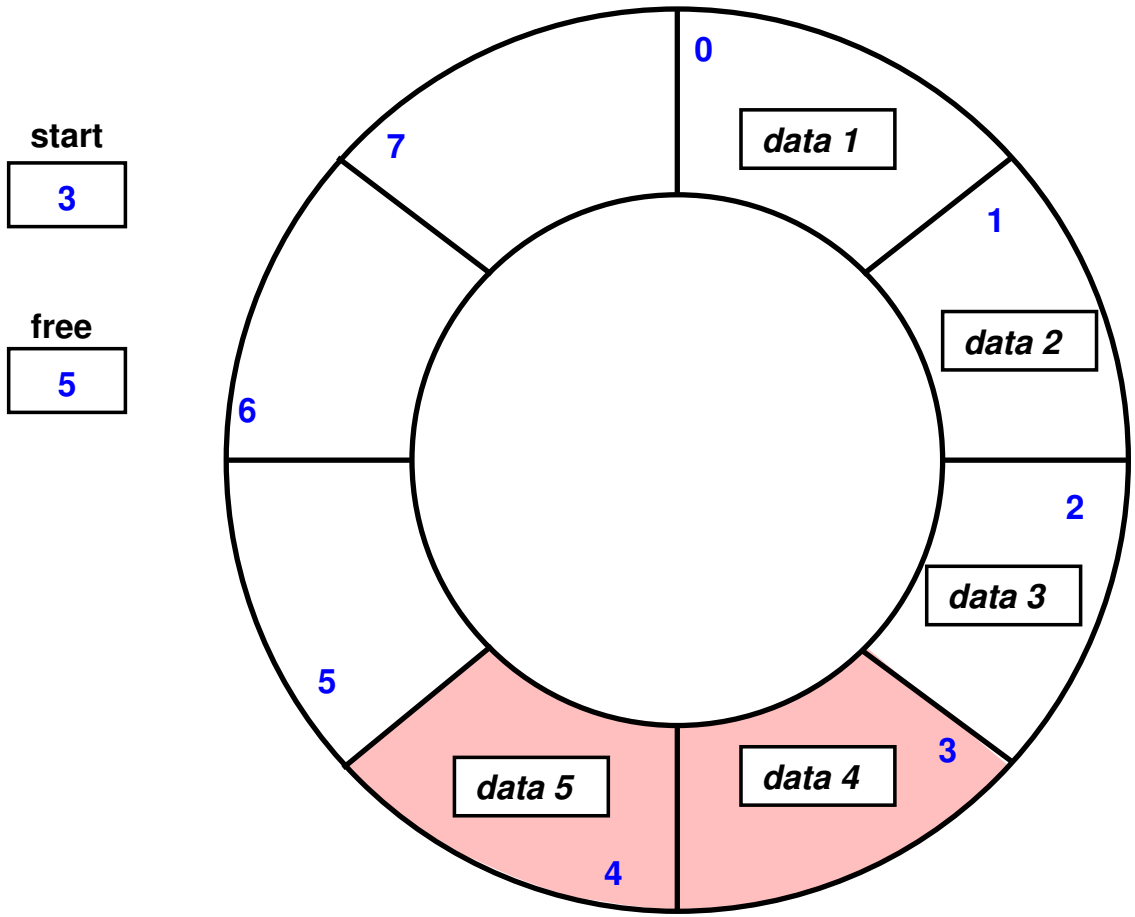
## K5B05E_Queue_Array (based on array)

```
1  public class Queue {
2
3   public Queue () {}
4
5   private int start, free,
6                 length = 8;
7   private boolean empty = true,
8                     full = false;
9   private Object array []
10          = new Object[length];
11
12  public void enQueue
13                (Object newObj) {
14    if ( !full ) {
15      array [free] = newObj;
16      free = (free+1) % length;
17      empty = false;
18      full = ( start == free );
19    }
20  }
21
22  ...
```

# Queue based on array: `enQueue` operations:



# Queue based on array: `deQueue` operation:



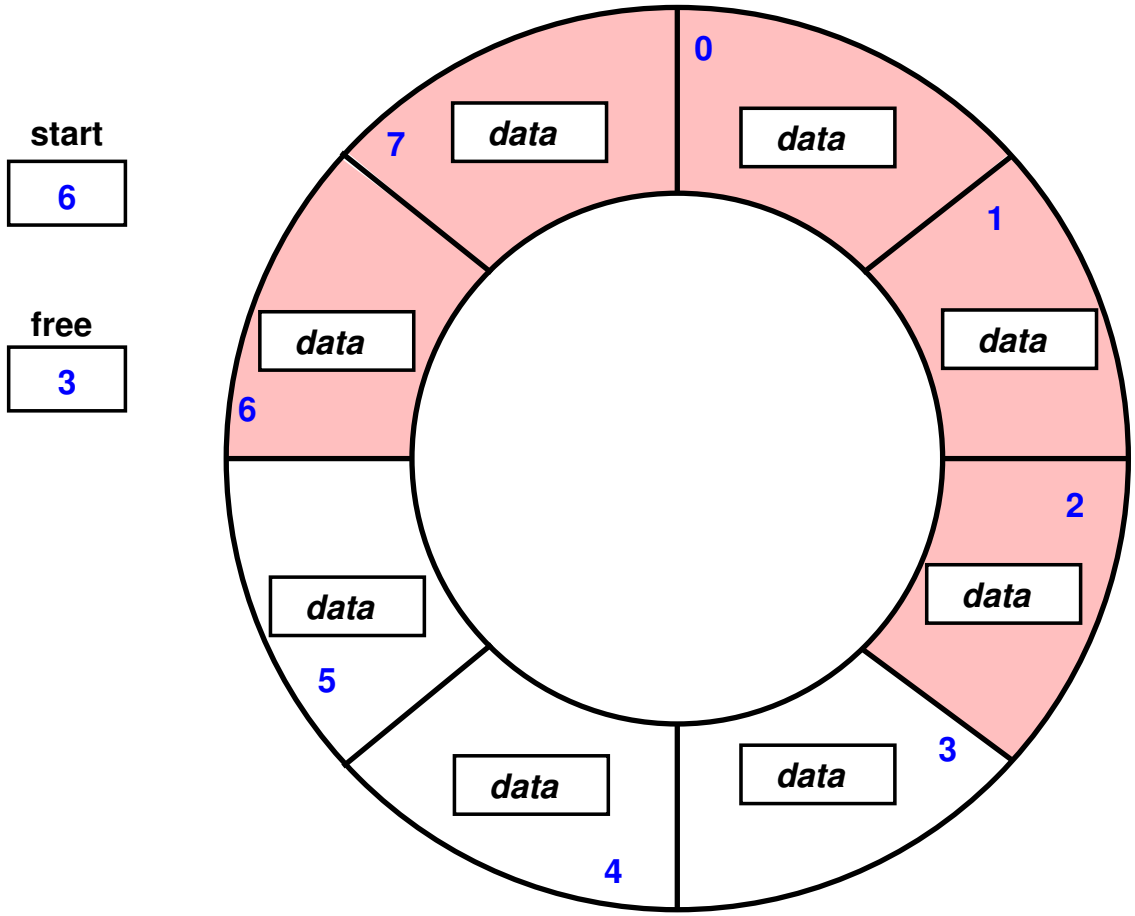
```
1  ...
2
3  public Object deQueue () {
4   if (start != null) {
5    Elem temp = start;
6    start = start.getNext();
7    if (start == null)
8       end = null;
9    return temp.getObject();
10   }
11  else return null;
12 }
13
14 public boolean isEmpty () {
15  return (start == null);
16 }
17
18 ...
```

```
1  ...
2
3  public Object deQueue () {
4   if ( !empty ) {
5    Object temp = array [start];
6    start = (start+1) % length;
7    full = false;
8    empty = ( start == free );
9    return temp;
10  }
11   else return null;
12 }
13
14 public boolean isEmpty () {
15  return empty;
16 }
17
18 ...
```

```
1  ...
2
3  public boolean isFull () {
4   return false;
5  }
6
7  @Override
8  public String toString () {
9   String str = "";
10  Elem position = start;
11  while (position != null) {
12   str += position.
13          getObject().
14          toString() + "␣";
15   position = position.
16              getNext();
17  }
18  return str;
19 }
20
21 }
```

```
1  ...
2
3  public boolean isFull () {
4   return full;
5  }
6
7  @Override
8  public String toString () {
9   String str = "";
10  int i = start;
11  if (!empty)
12   do {
13    str += array [i] + "␣";
14    i = (i + 1) % length;
15   }
16   while (i != free);
17
18   return str;
19 }
20
21 }
```

Queue based on array: **ring buffer technique**:



```
1  public class QueueTest {
2    public static void main(String[] args) {
3      Queue qu = new Queue ();
4      while ( true ) {
5        String str = System.console().readLine
6                       ("Input [ key|-|0]: ");
7        if ( str.equals("0")  ) return;
8        if ( str.equals("-") ) {
9          int key = (Integer)qu.deQueue();
10         System.out.println("deQueue: " + key);
11         System.out.println("Queue:   " + qu);
12       } else {
13         int key = Integer.parseInt(str);
14         qu.enQueue (key);
15         System.out.println("enQueue: " + key);
16         System.out.println("Queue:   " + qu);
17       }
18     }
19   }
20 }
```

In the examples `K5B05E_Queue_Linked` und `K5B05E_Queue_Array` the `QueueTest.java` files are identical! All differences are in the file corresponding to `Queue.java`.

The data structure **stack** can be characterized by two operations:

- **push**:
  puts an object on the stack.
- **pop**:
  removes the top-most object from the stack.

Other useful operations:

- **isEmpty**:
  checks if the stack is empty.
- **isFull**:
  checks if the stack is full or its capacity is reached.
- **peek**:
  returns the top-most element of the stack without removing it.
  (can be replaced by a sequence of pop and push)

# Example `K5B06E_Stack_...` with two implementations of a stack:

`K5B06E_Stack_Linked`
(of linked elements)

```
1  public class Stack {
2
3   public Stack ()  {}
4
5   private Elem top;
6
7
8
9
10
11  public void push
12                (Object newObj) {
13    Elem newElem =
14                new Elem(newObj);
15    newElem.setNext(top);
16    top = newElem;
17  }
18
19  ...
```

`K5B06E_Stack_Array`
(based on array)

```
1  public class Stack {
2
3   public Stack () {}
4
5   private int free, length=50;
6   private boolean empty=true,
7                   full=false;
8   private Object array [] =
9             new Object [length];
10
11  public void push
12                (Object newObj) {
13    if ( !full ) {
14     array [free++] = newObj;
15     empty = false;
16     full = (free == length );
17    }
18  }
19  ...
```

```
1  ...
2
3  public Object pop () {
4    if (top != null) {
5      Elem temp = top;
6      top = top.getNext();
7      return temp.getObject();
8    }
9    else return null;
10 }
11
12
13
14 public boolean isEmpty () {
15   return (top == null);
16 }
17
18 ...
```

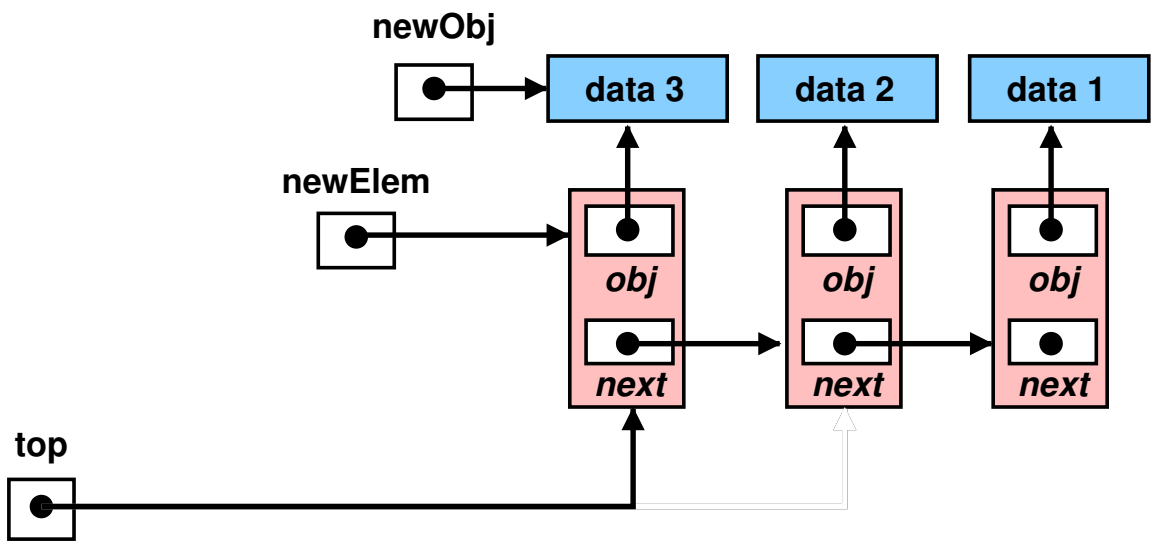
```
1  ...
2
3  public Object pop () {
4    if ( !empty ) {
5      free--;
6      Object temp = array[free];
7      full = false;
8      empty = ( free == 0 );
9      return temp;
10   } else return null;
11 }
12
13
14 public boolean isEmpty () {
15   return empty;
16 }
17
18 ...
```

```
1  ...
2
3  public boolean isFull () {
4    return false;
5  }
6
7  @Override
8  public String toString () {
9    String str = "";
10   Elem position = top;
11
12   while (position != null) {
13     str += position.
14            getObject().
15            toString() + "␣";
16     position = position.
17                getNext();
18   }
19
20   return str;
21 }
22 }
```

```
1  ...
2
3  public boolean isFull () {
4    return full;
5  }
6
7  @Override
8  public String toString () {
9
10   String str = "";
11
12
13   for (int i = free-1;
14            i >= 0;
15            i--) {
16     str += array[i] + "␣";
17   }
18
19
20   return str;
21 }
22 }
```

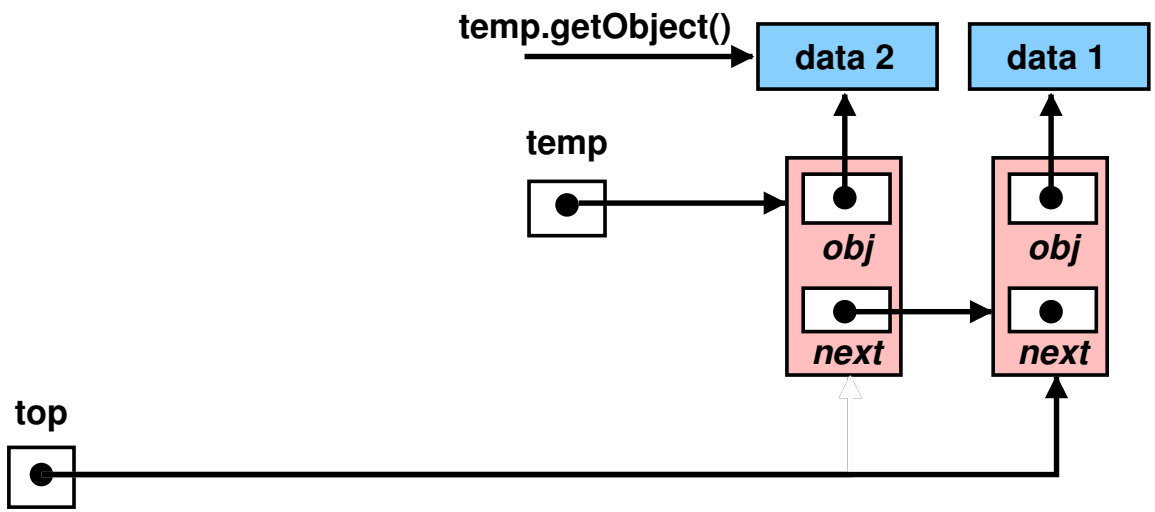
# **push**: Put an element on the stack

```
1  public void push
2                (Object newObj) {
3    Elem newElem =
4                new Elem(newObj);
5    newElem.setNext(top);
6    top = newElem;
7  }
8    ...
```



# `pop`: Take the top-most element from the stack

```
1  public Object pop () {
2    if (top != null) {
3      Elem temp = top;
4      top = top.getNext();
5      return temp.getObject();
6    }
7    else return null;
8  }
```



```
1  public class StackTest {
2    public static void main(String[] args) {
3      Stack st = new Stack();
4      while ( true ) {
5        String str = System.console().readLine
6                       ("Input [key|-|0]: ");
7        if ( str.equals("0")  ) return;
8        if ( str.equals("-") ) {
9          int key = (Integer)st.pop();
10         System.out.println("pop:␣␣␣" + key);
11         System.out.println("Stack:␣" + st);
12       } else {
13         int key = Integer.parseInt(str);
14         st.push(key);
15         System.out.println("push:␣␣" + key);
16         System.out.println("Stack:␣" + st);
17       }
18     }
19   }
20 }
```

In both `K5B06E_Stack_Linked` and `K5B06E_Stack_Array` the file `StackTest.java` is identical! All differences are in the file `Stack.java`.