Konzepte objektorientierter Programmierung

Prof. Dr. Peter Müller

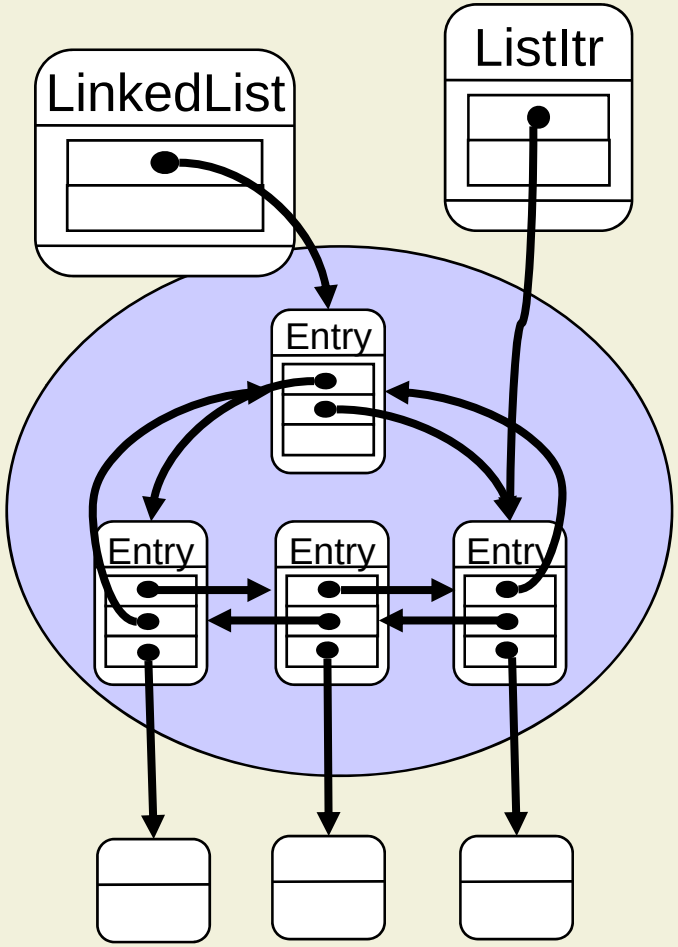
Chair of Programming Methodology

Exercises 8: Ownership

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Ownership Model

- The **object store** is partitioned into **contexts**
- Each object **belongs to exactly one context**
- Each context has at most one **owner object**
  - The owner does not belong to the context it owns
- Contexts are **hierarchical**



Types

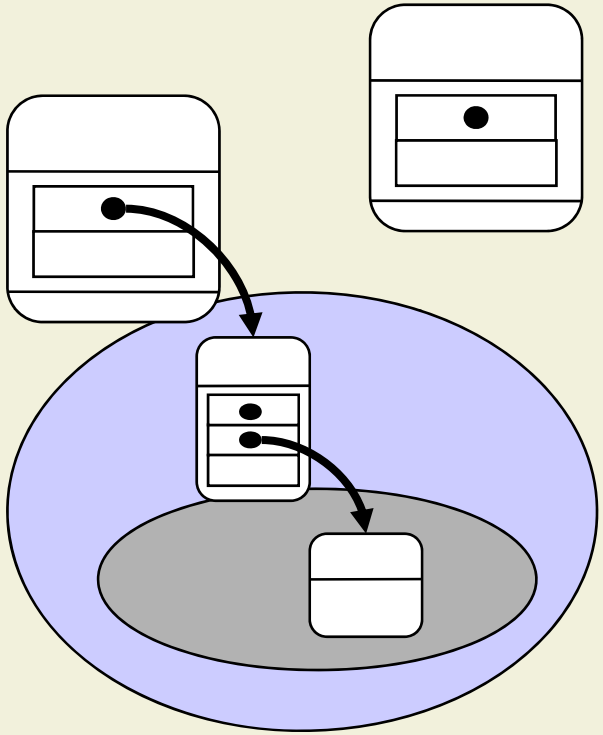
- Each class or interface T introduces **three types**
- Peer and readonly types
- **Rep type** *rep(T)*
  - Denoted by **rep** T in programs

```
class LinkedList {
  private rep Entry header;
  public void add( readonly Object o ) {
    rep Entry newE =
      new rep Entry( o, header, header.previous );
    … }
}
```

```
class Entry {
  private readonly Object element;
  private Entry previous, next;
  public Entry( readonly Object o,
                Entry p, Entry n ) { … }
}
```

Type Rules: The Type Combinator

*	*peer(T)*	*rep(T)*	*ro(T)*
peer(S)	*peer(T)*	*ro(T)*	*ro(T)*
rep(S)	*rep(T)*	*ro(T)*	*ro(T)*
ro(S)	*ro(T)*	*ro(T)*	*ro(T)*



Home-work

Universes: Lightweight Ownership for JML

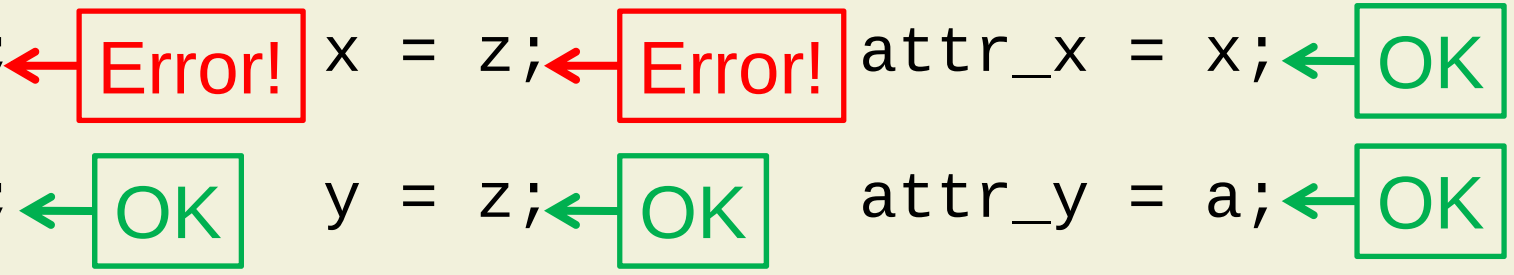
Dietl, W. and Müller, P

Journal of Object Technology (JOT), 2005.

http://www.jot.fm/issues/issue_2005_10/article1.pdf

Exercise 1

```
public class Assignments {
 private rep       Integer attr_x;
 private readonly  Integer attr_y;
 private peer      Integer attr_z;
 public void m(readonly Integer a ) {
  rep       Integer x = new rep integer(4);
  readonly  Integer y = new Integer(3);
  peer      Integer z = new Integer(2);

    x = y;   x = z;   attr_x = x;
    y = x;   y = z;   attr_y = a;
 }
}
```



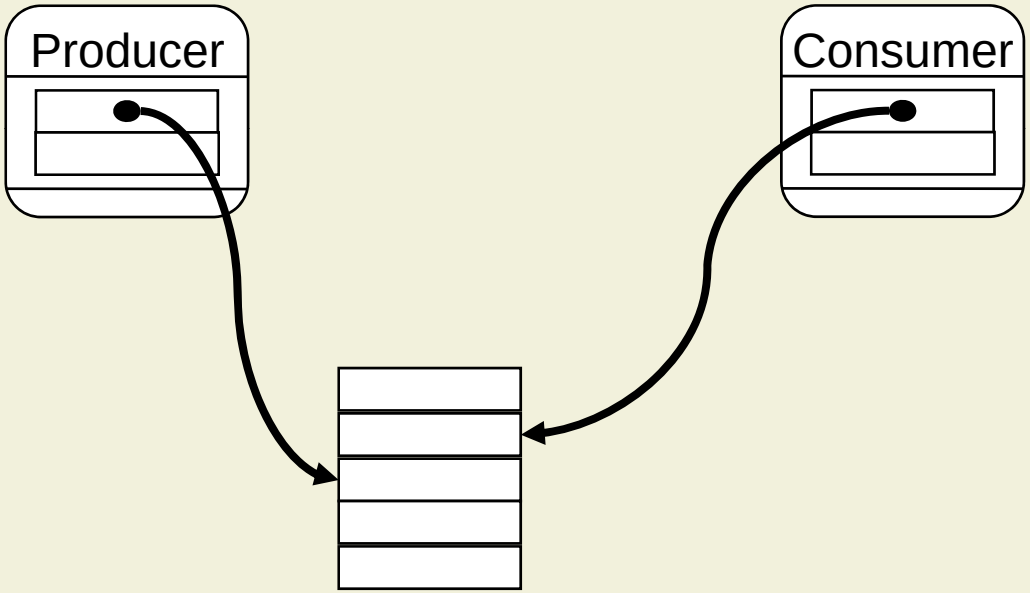
Exercise 2

```
class Room {
    private Human m;
    void enter( Human m ) {}
    Human leave() {}
}
public class Building {
    private Room[] rooms;
    void enter( Human m )
        {rooms[0].enter( m ); }
    Human leave()
        {return rooms[0].leave(); }
}
```

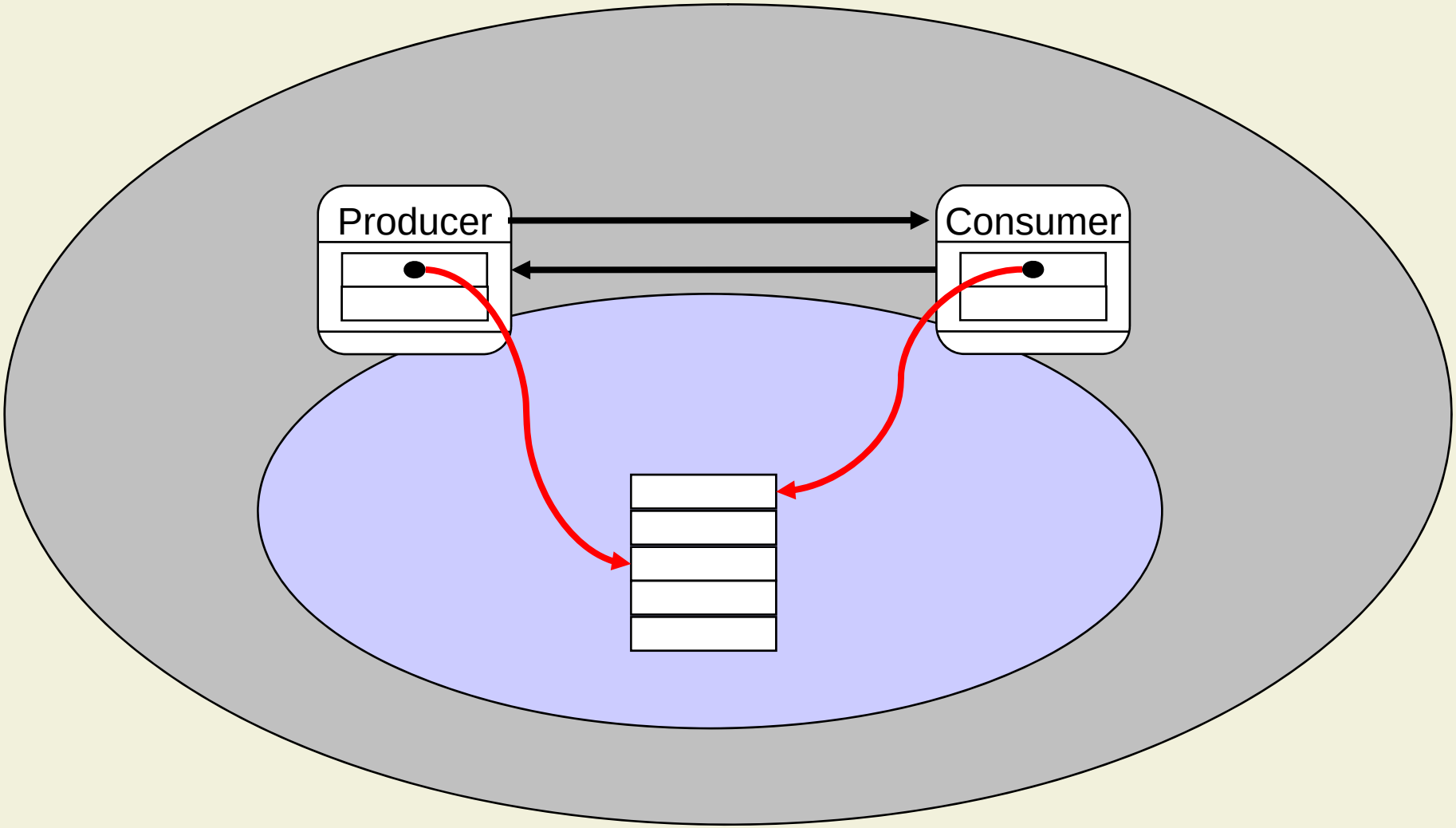
Exercise 2

```
class Room {
    private readonly Human m;
    void enter(readonly Human m ) {}
    readonly Human leave() {}
}
public class Building {
    private rep Room[] rooms;
    void enter(readonly Human m )
        {rooms[0].enter( m ); }
    readonly Human leave()
        {return rooms[0].leave(); }
}
```

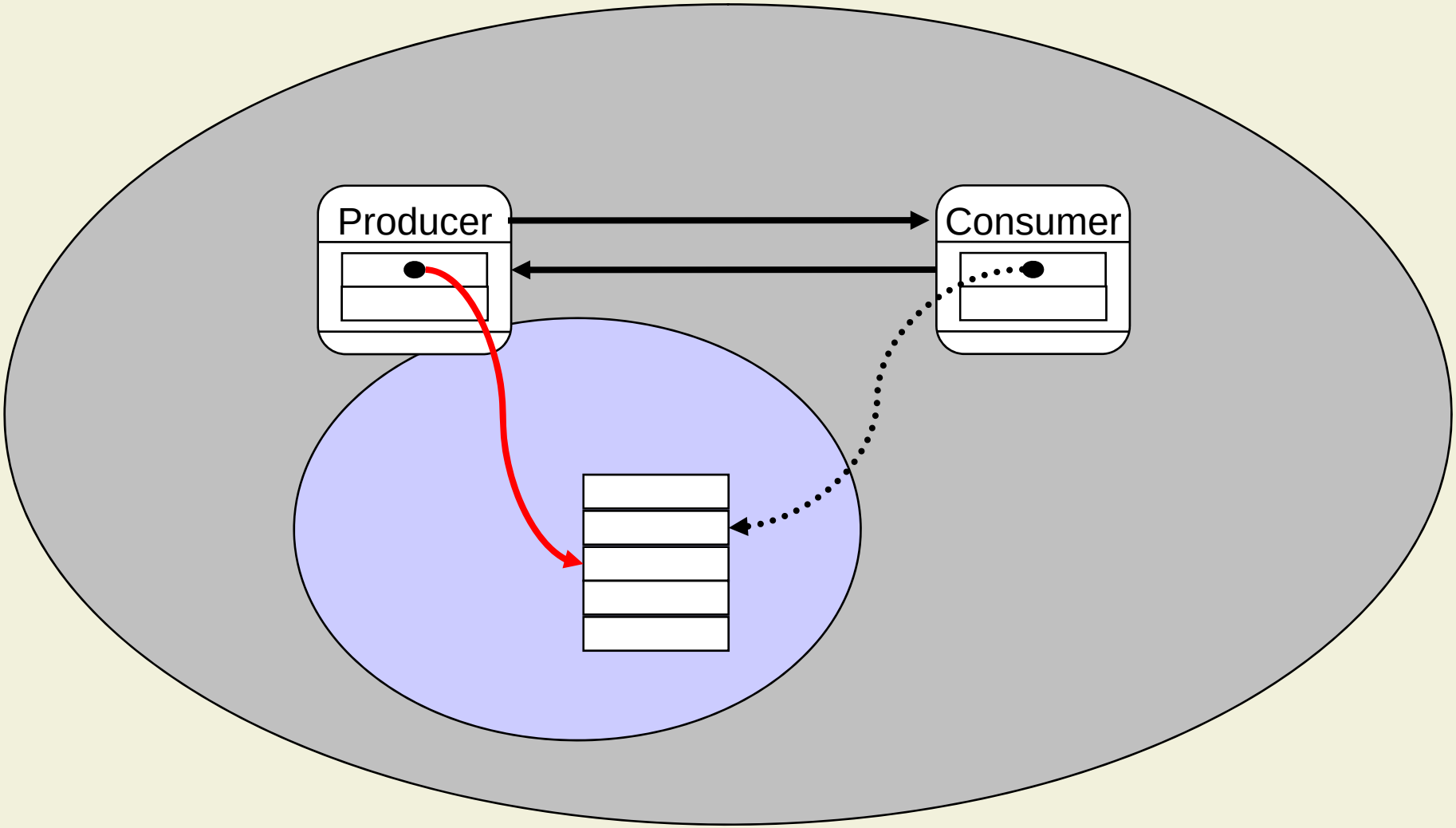
Exercise 3: Producer-Consumer



Exercise 3: Producer-Consumer



Exercise 3: Producer-Consumer



Producer – Original

```
class Producer {
    int[] buf;
    int n;
    Consumer con;

    Producer() {
        buf = new int[10];
    }

    void produce(int x) {
        buf[n] = x;
        n = (n+1) % buf.length;
    }
}
```

Producer – Annotated

```
class Producer {
    rep int[] buf;
    int n;
    peer Consumer con;

    Producer() {
        buf = new rep int[10];
    }

    void produce(int x) {
        buf[n] = x;
        n = (n+1) % buf.length;
    }
}
```

Consumer – Original

```
class Consumer {
    int[] buf;
    int n;
    Producer pro;

    Consumer(Producer p) {
        buf = p.buf;
        pro = p;
        p.con = this;
    }

    int consume() {
        n = (n+1) % buf.length;
        return buf[n];
    }
}
```

Consumer – Annotated

```
class Consumer {
    readonly int[] buf;
    int n;
    peer Producer pro;

    Consumer(peer Producer p) {
        buf = p.buf;
        pro = p;
        p.con = this;    }

    int consume() {
        n = (n+1) % buf.length;
        return buf[n];
    }
}
```

Context – Original

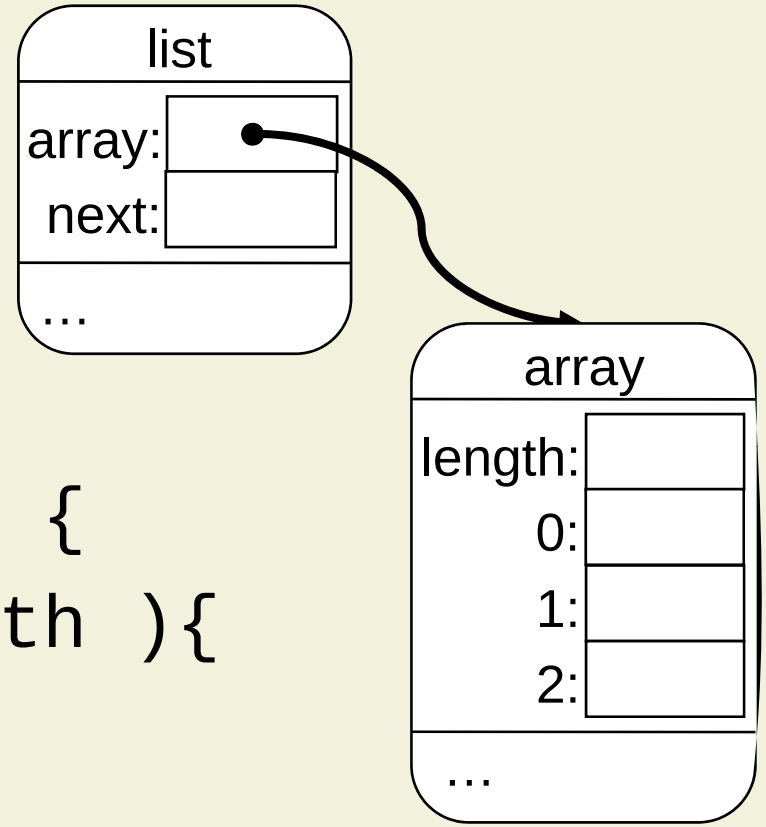
```
class Context {
    Producer p;
    Consumer c;

    Context(){
        p = new Producer();
        c = new Consumer(p);
    }

    public void run() {
        for(int i=-5; i <=5; ++i){
            p.produce(i);
            if(i%2 == 0) c.consume();
        }
    }
}
```

Context – Annotated

```
class Context {
    rep Producer p;
    rep Consumer c;

    Context(){
        p = new rep Producer();
        c = new rep Consumer(p);
    }

    public void run() {
        for(int i=-5; i <=5; ++i){
            p.produce(i);
            if(i%2 == 0) c.consume();
        }
    }
}
```

Exercise 4: ArrayList

```
class ArrayList {
  protected int[] array;
  protected int next;

  public void add(int i) {
    if( next==array.length ){
      resize( ); }
    array[ next ] = i;
    next++;
  }
```



ArrayList

```
public void setElems( int[] ia ) {
  array = ia;
}

public int[] getElems() {
  return array;
}
```

- `setElems` is *Capturing* an external array
- `getElems` is *Leaking* the internal representation

ArrayList – Main Program

```
public static void main( String[] args ) {
  int[] myarr = new int[10];

  for(int i=0; i < myarr.length; ++i)
    myarr[i] = i;

  ArrayList al = new ArrayList();

  al.setElems( myarr );

  myarr[0] = 42;
}
```

Annotated ArrayList

```
class ArrayList {
  protected rep int[] array;
  protected int next;
```

- Array is part of the internal representation
- Marking it as **rep** makes this explicit for the programmer

ArrayList – setElems

```
public void setElems( int[] ia ) {
  array = new rep int[ ia.length ];
  System.arraycopy
     (ia, 0, array, 0, ia.length );
  next = ia.length;
}
```

- The input array is of default type and can not directly be assigned to the `/* rep */` array
- We have to create our own copy that can be used as internal representation → no *Capturing*

ArrayList – getElems

```
public int[] getElems() {
  return (int[]) array.clone();
}
```

- The **rep** array can not be passed out directly as default array
- We create a clone of the internal array and return it as result
- The internal representation is still encapsulated → no *Leaking*

Questions?