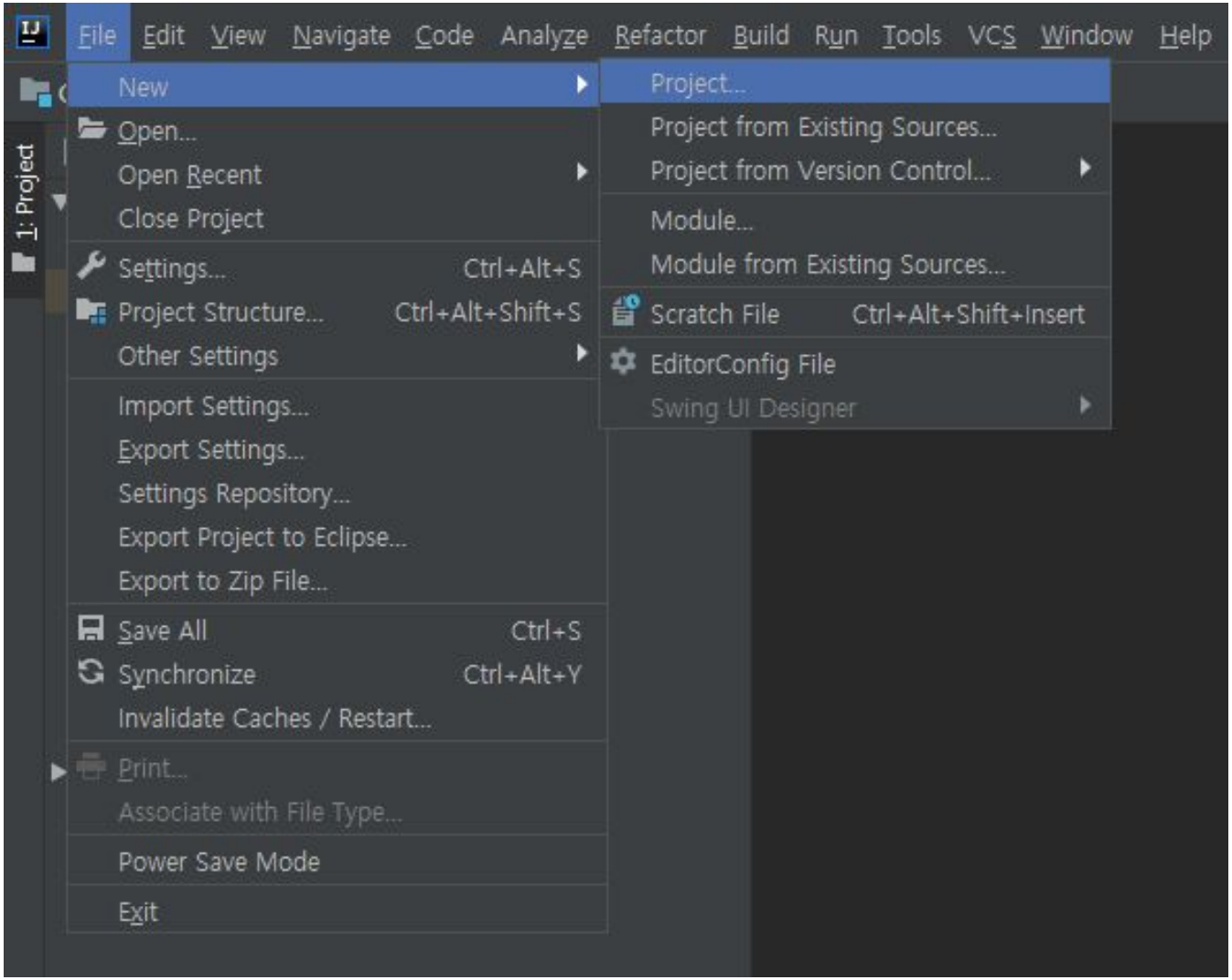
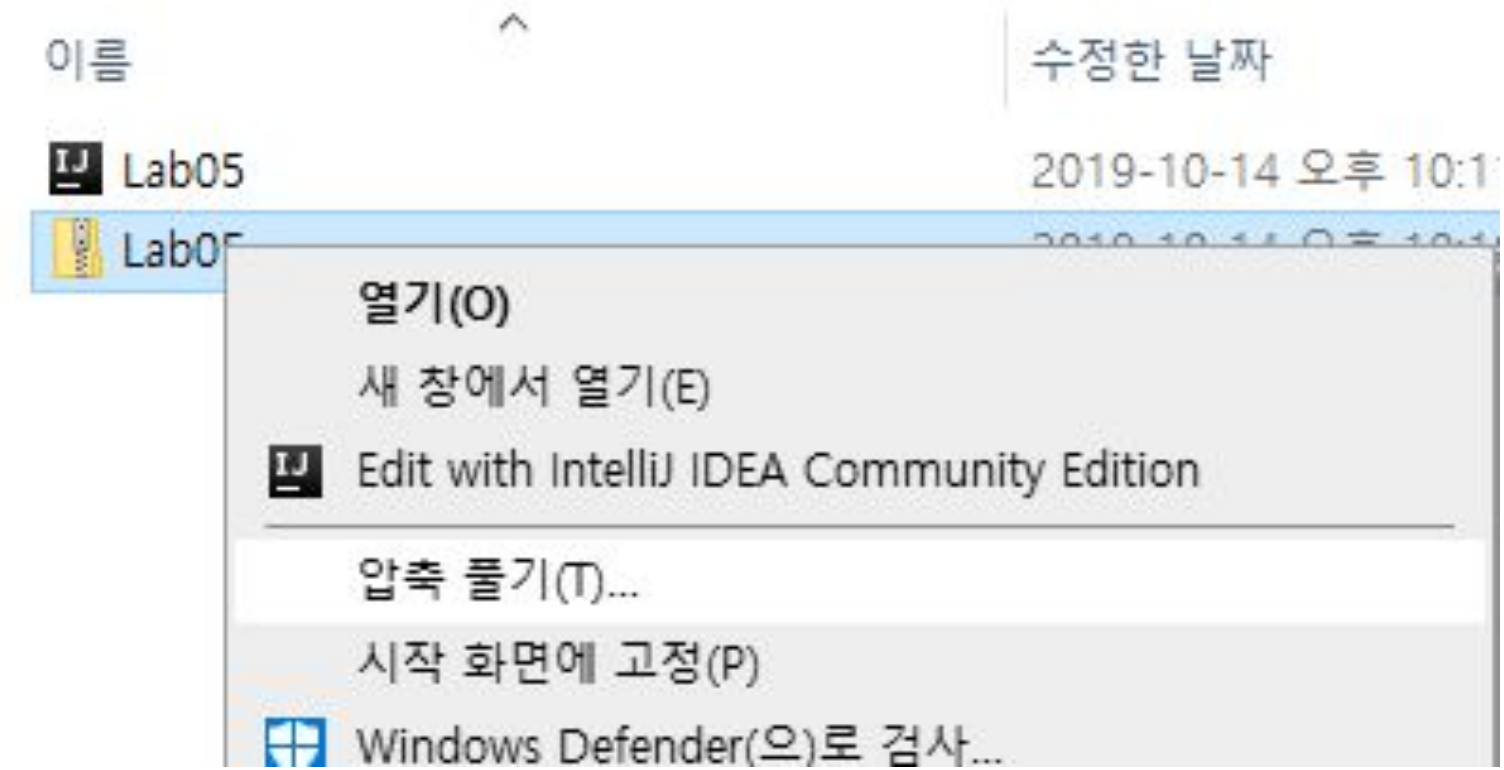
Polymorphism

Lab 5

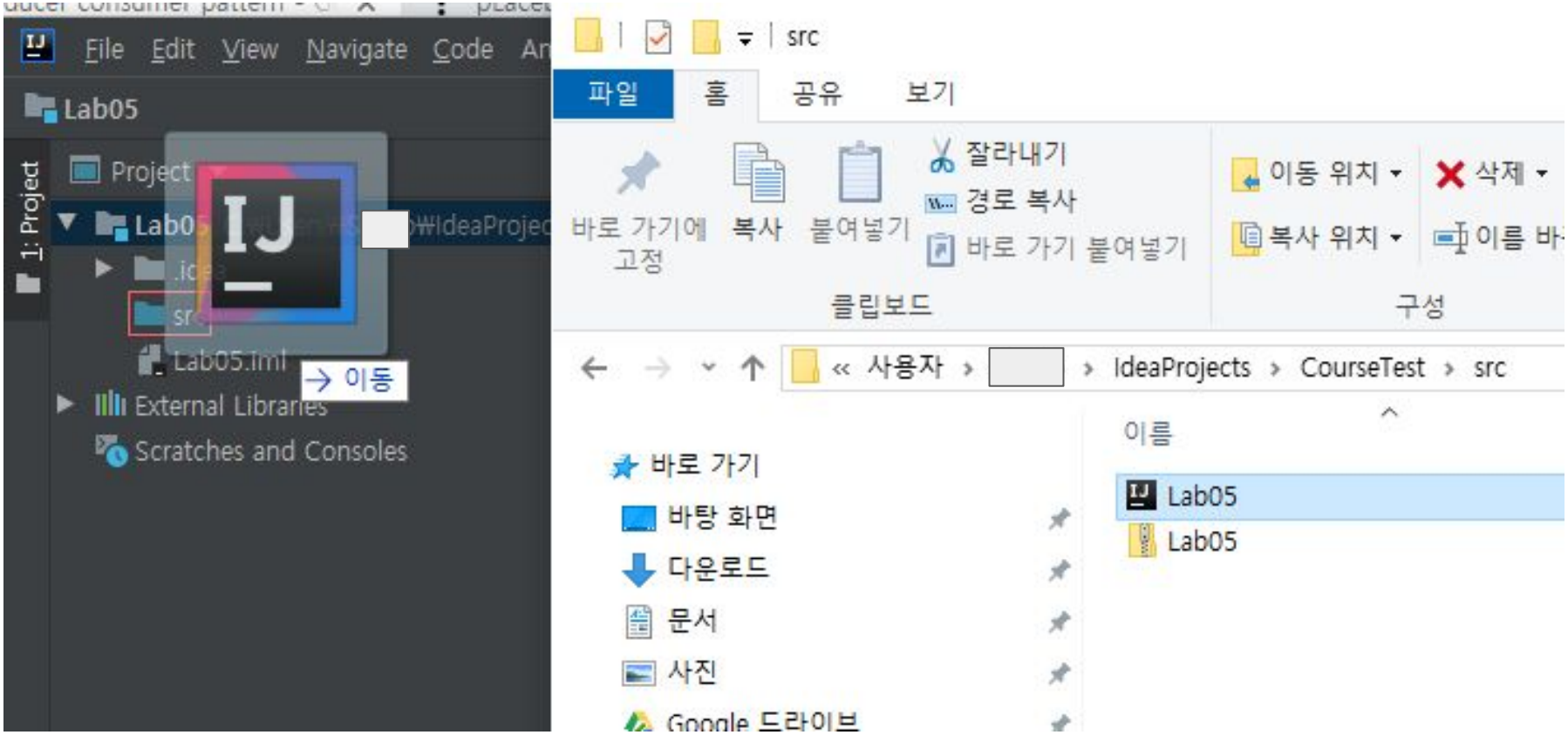
Create New Project



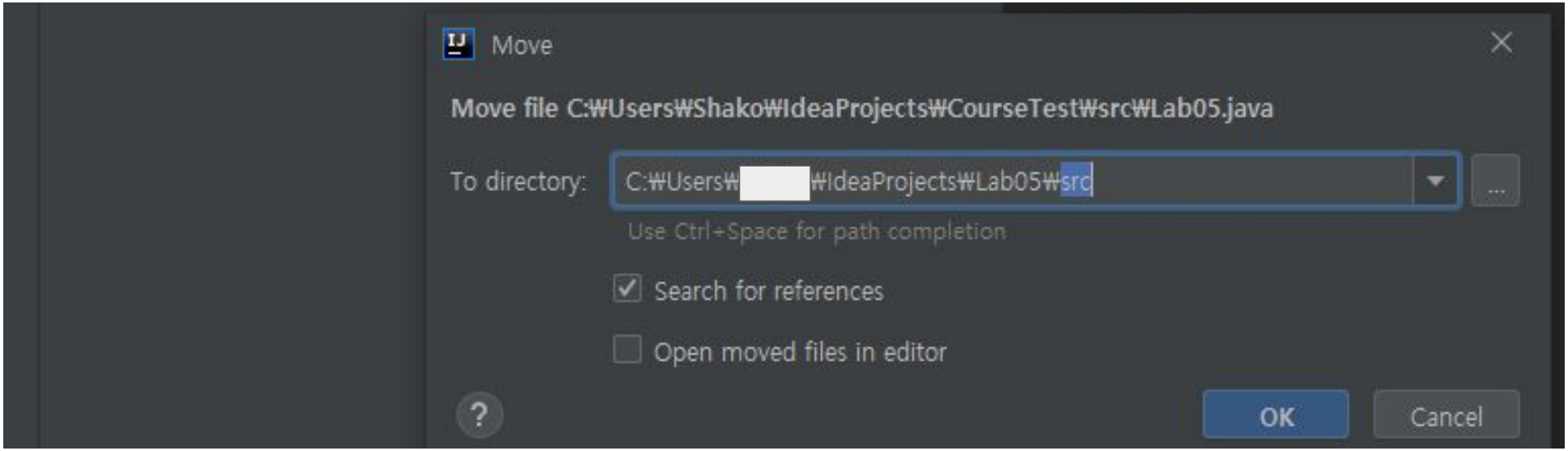
Unzip the given code



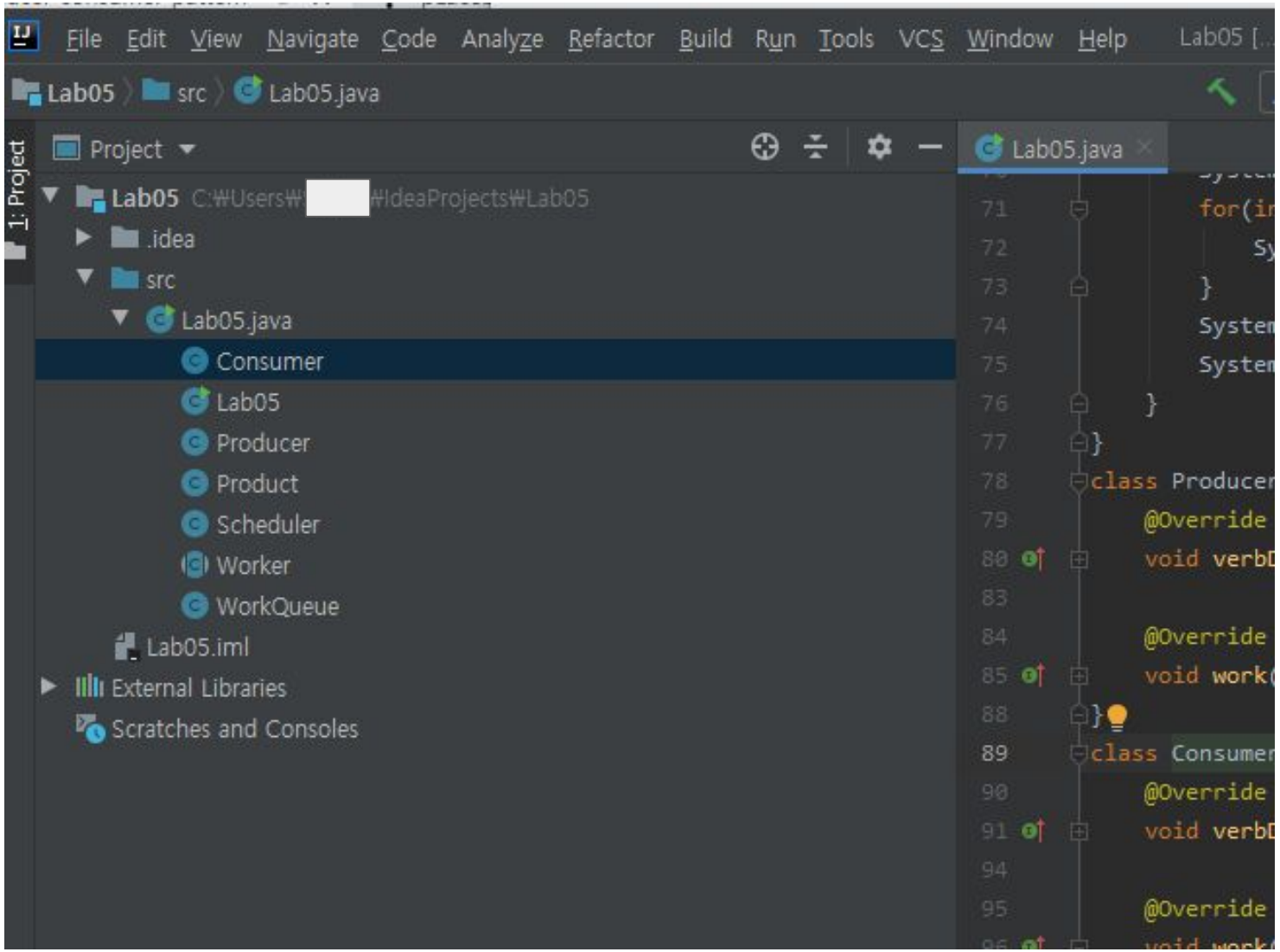
Bring the unzipped code to the src folder



Approve the move of the file



The code is brought to the new project



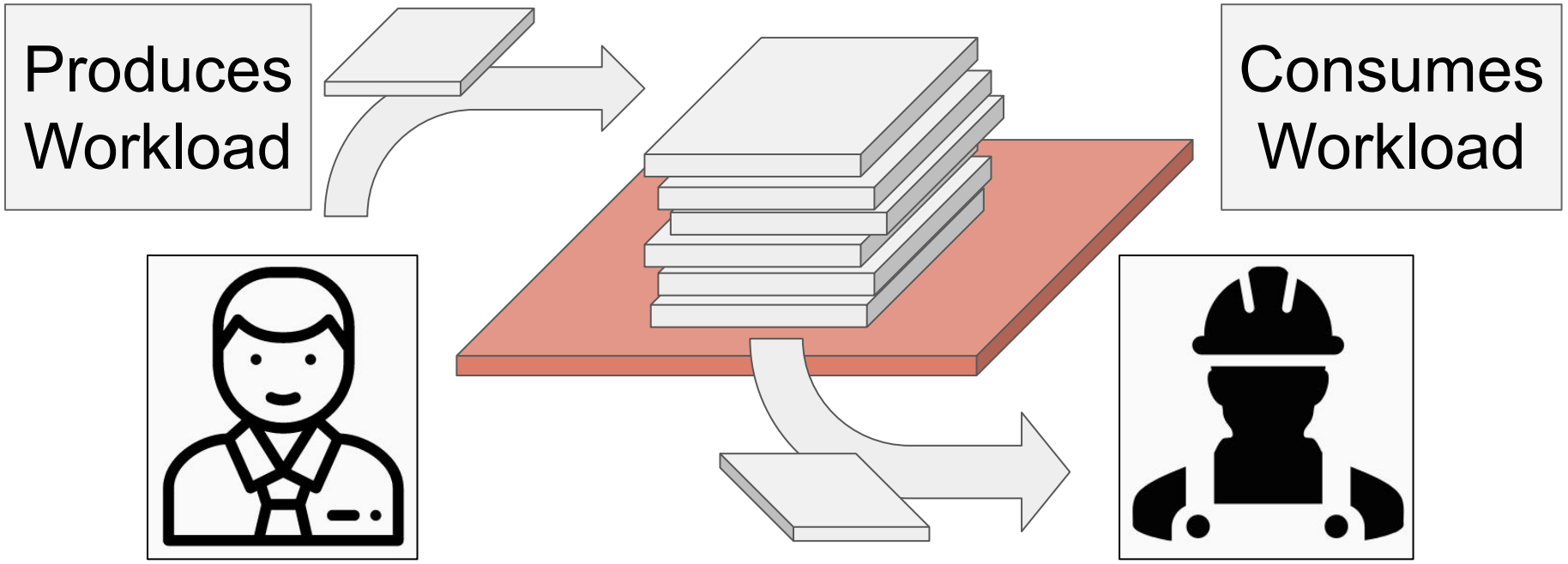
Goal

- Understand the differences of static and instance members, and observe the practical usage of these differences.
- Understand the Generics and its advantage.
- Understand the advantages of polymorphism in java(Overriding)
  - Easy-to-read, Expressive programming

Producer-Consumer Pattern

These specific terms or concepts are **not** on exam

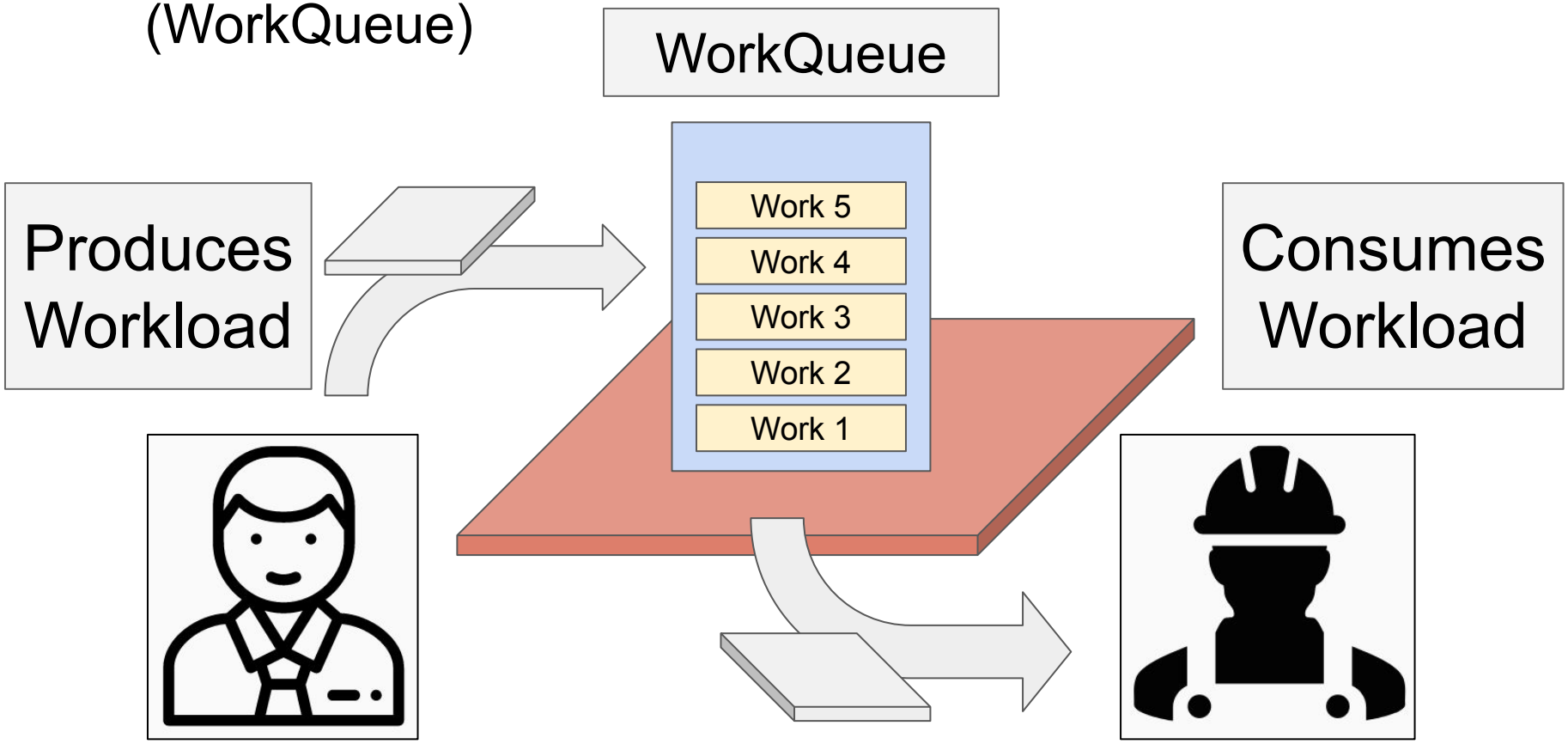
- Workload (Pile of works to be done) exists.
- *Producer* produces works and pile up in the workload.
- *Consumer* consumes works from the workload.



Producer-Consumer Pattern

These specific terms or concepts are **not** on exam

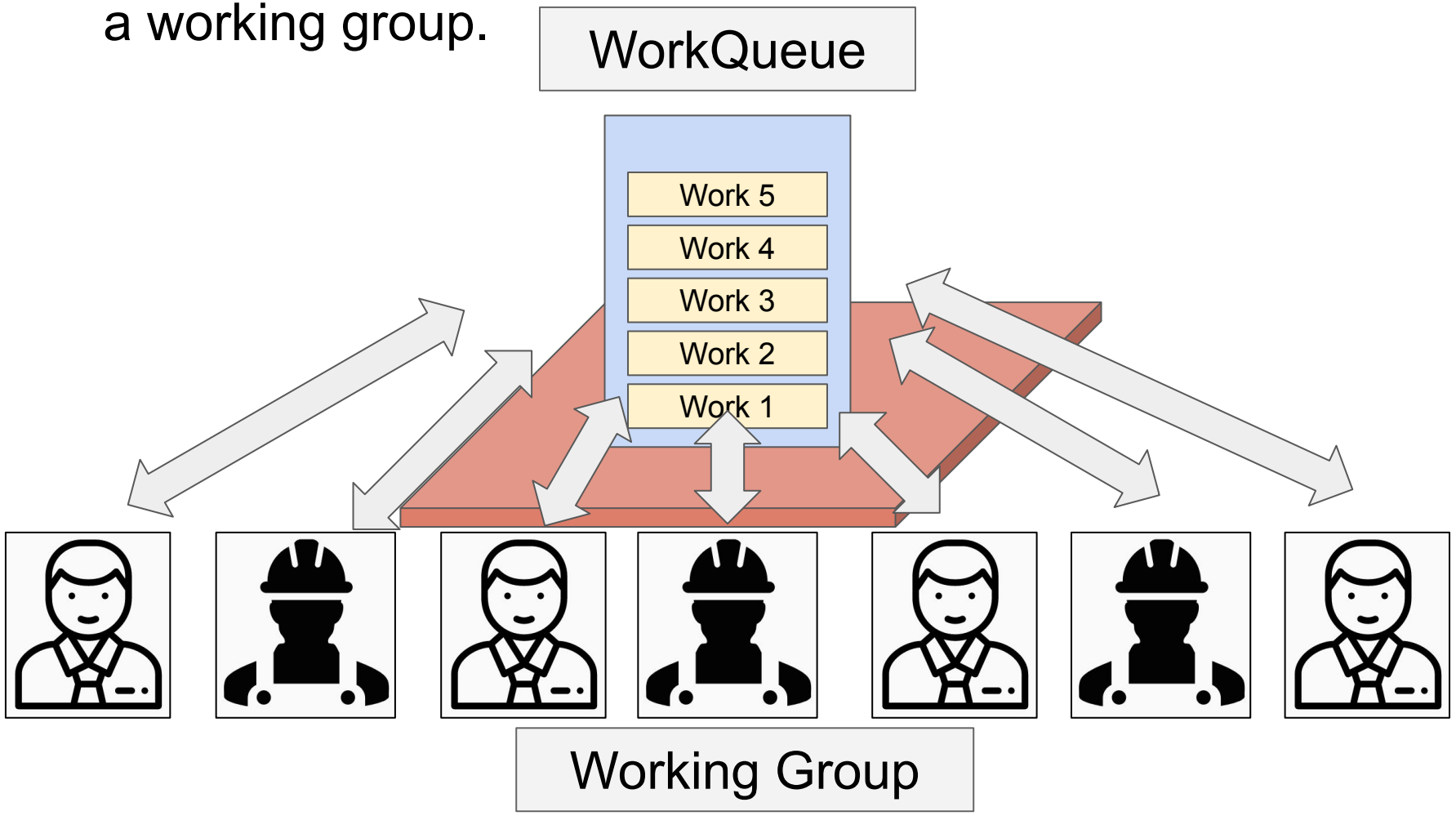
- Workload could be represented as a Queue of Works (WorkQueue)



Producer-Consumer Pattern

These specific terms or concepts are **not** on exam

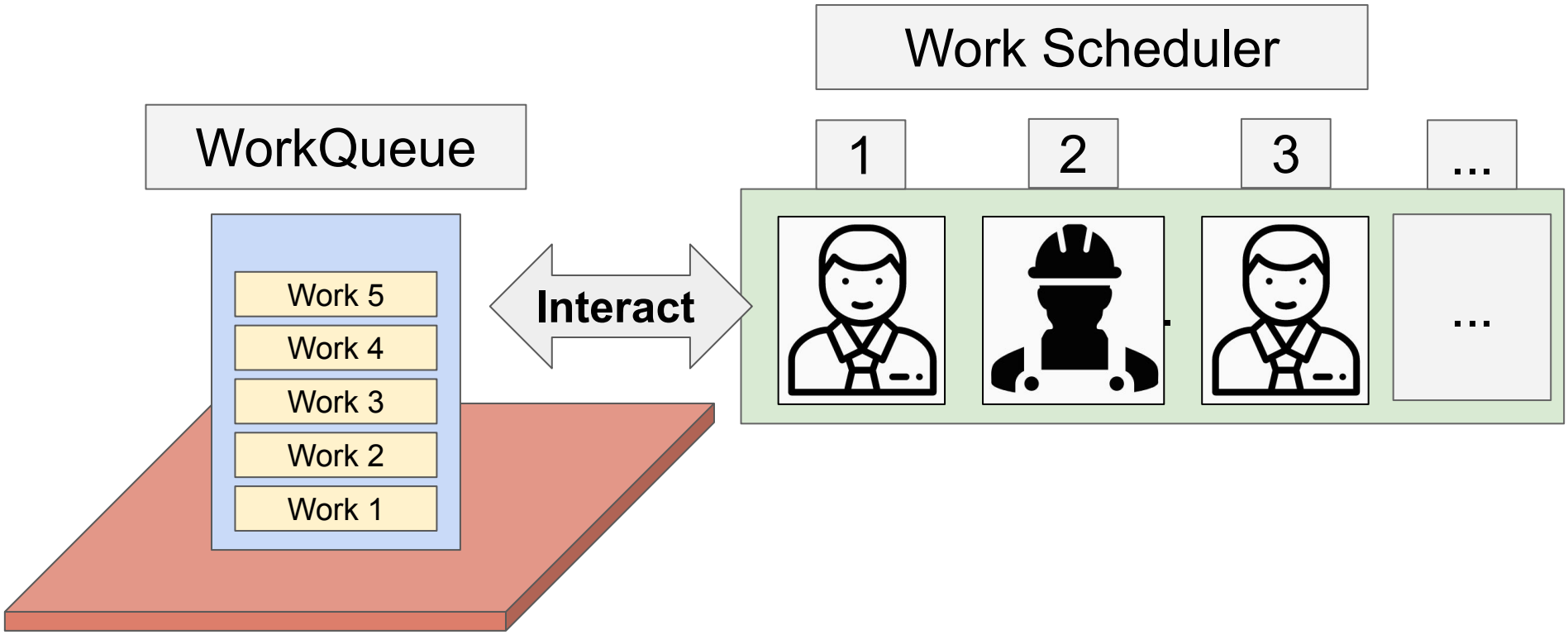
- Several Producers and Consumers could exist. They form a working group.



Producer-Consumer Pattern

These specific terms or concepts are **not** on exam

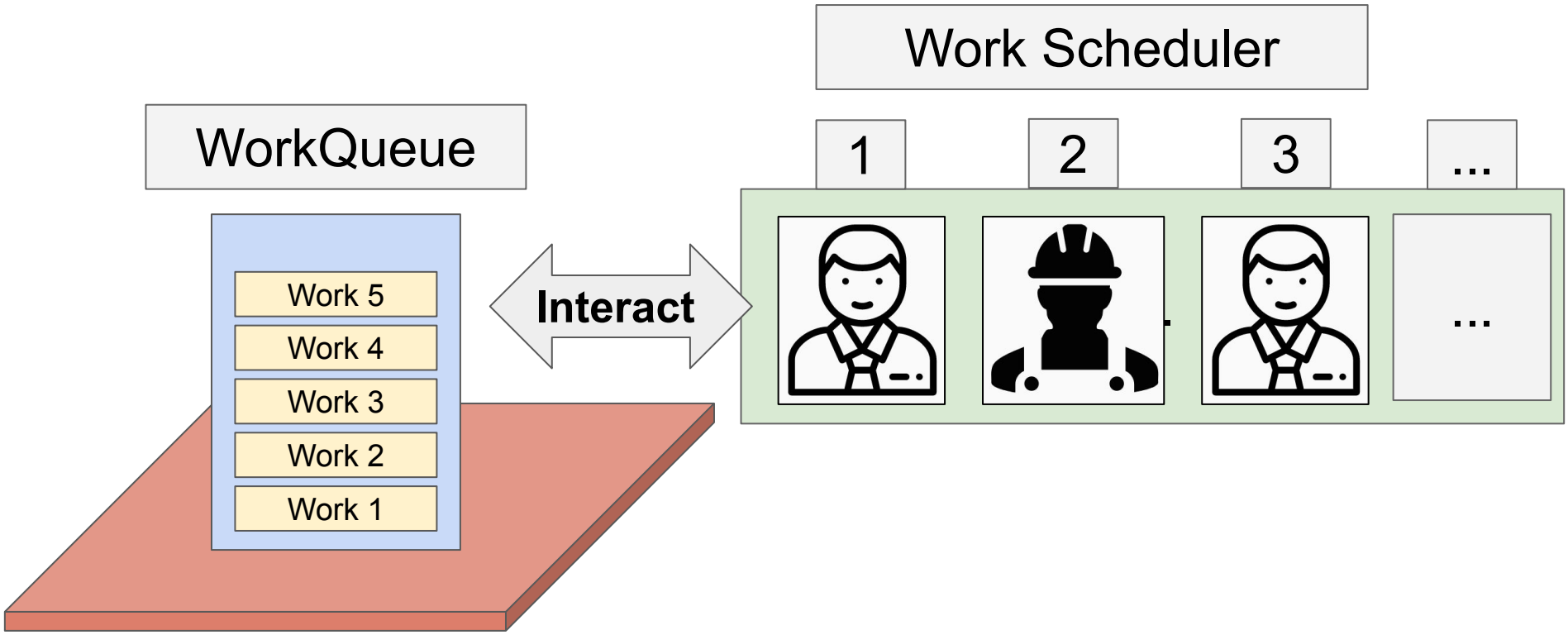
- Scheduler gives one of them the opportunity to do their work for a given period of time. (scheduling)
- The policy of selecting the worker could be very diverse.



Producer-Consumer Pattern

These specific terms or concepts are **not** on exam

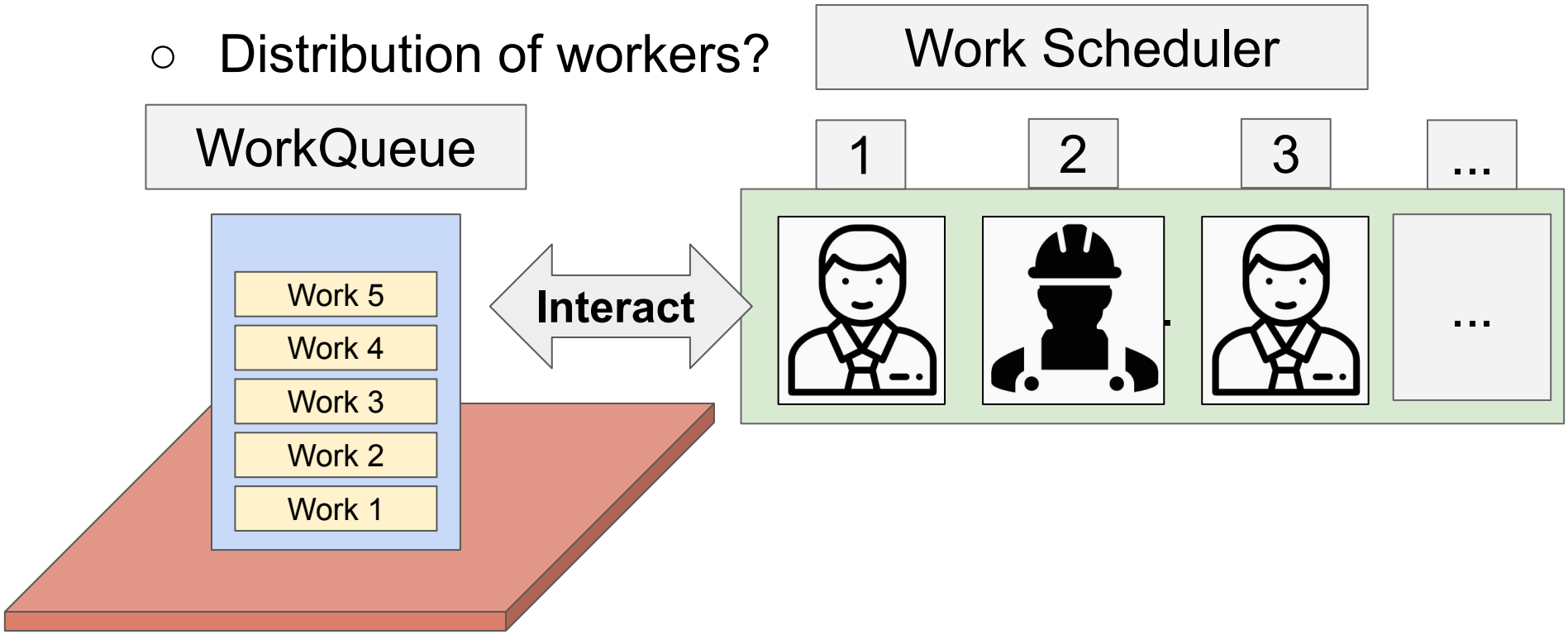
- Both producer and consumer being a class, and a workqueue being a computational workload, it is called a producer-consumer pattern.



Producer-Consumer Pattern

These specific terms or concepts are **not** on exam

- What kind of characteristic of workqueue will be observed as we differentiate
  - the policy of this scheduling?
  - Distribution of workers?



Producer-Consumer Pattern

These specific terms or concepts are **not** on exam

- Lots of trials could be done
  - What if number of producers > number of customers?
  - What if the initial amount of workload changes?
- Progress of the size of the workload is always different

Syntactic aspects

- Generics
- polymorphic aspects of Java
- mixed use of static and instance member
- expressive encapsulation

Generics in WorkQueue<E> class

- *WorkQueue* for managing the workload, each work of type *E*.
  - *WorkQueue<Product>* : Queue of *Product* production
- *WorkQueue* need not know the specific type *E* in advance.

```
class WorkQueue<E>{
    ArrayList<E> data = new ArrayList<>();
    void in ( E elem){ data.add(elem);  }
    E out(){
        if(this.empty()){
            throw new IndexOutOfBoundsException();
        }
        return data.remove(0);
    }
...
```

Generics in Scheduler<T> class

Scheduler class defined under unknown type variable *T*.

- *Scheduler<Worker>* : worker group scheduler
- On *schedule*(), it samples one of the *T* from the group with its internal policy and returns it.
- *Scheduler* need not know further knowledge about the input type *T*.

```
class Scheduler<T>{
    private static final int waitms = 100;
    private T[] group;
    Scheduler(T[] array){
        group = array;
    }
    T schedule(){
        int sample = (int)(Math.random()
            * group.length);
        return group[sample];
    }
```

...

Overriding abstract method

Subclasses override abstract methods to do their work.

- *main* method uses these methods of Worker without knowing the actual implementation of it.

```
abstract class Worker {
    abstract void verbDefine();
    abstract void work();
}
class Producer extends Worker {
    void verbDefine() { verb = "produce"; }
    void work(){ workload.in(new Product()); }
}
class Consumer extends Worker {
    void verbDefine() { verb = "consume"; }
    void work() {
        if(workload.empty()) return;
            workload.out();
   }
}
```

Static definition of the workload

- A workload WorkQueue<Product> is defined static in the Worker class, so several Worker instances share the same workload.
  - Initiated in the initial execution of the program
- Static variable makes it easy for several instances to share a same value.

```
abstract class Worker {
    final static Product sampleProduct = new Product();
    static WorkQueue<Product> workload =  new WorkQueue<>();
```

Mixed use of static and instance variable

- Static *numWorkers* increases as a new Worker is instantiated.
  - Total number of workers are shared among workers
- *Id* is given for each instance using *numWorkers*.
  - Each individual instances are assigned a unique *id.*

```
// In Worker class
static int numWorkers = 0;
public int id;

Worker(){
    id = numWorkers;
    numWorkers++;
```

Expressiveness

Using an encapsulation ‘not’ instead of !(exclamation mark) could be used to make code more easier to understand.

```
private static boolean not (boolean in){  return !in;  }
```

This way of a program being more readable and intuitive is called ‘Expressive’.

Which one do you think represents the statement “While workload is not empty”much better?

```
while(!(Worker.workload.empty())){
```

```
while(not(Worker.workload.empty())){
```

Expressiveness

A complex for loop of iterating 0 to some N can be more easily represented with the additional method *range*.

```
private static ArrayList<Integer> range(int N){
    ArrayList<Integer> out = new ArrayList<>(N);
    for(int i = 0 ; i < N; i++) {
        out.add(i);
    }
    return out;
}
```

```
// for(int i = 0 ; i < N; i++) {

for(int i : range(initial)){
```

Expressiveness

A group of workers with one Producer and one Consumer.

Define a scheduler for the Worker group.

Set Initial workloads

While workload is not empty :

Schedule a new worker.

Let a new worker work.

Wait for the work.

Report what it has done.

```
Worker[] group = new Worker[]{
        new Producer(), new Consumer()};

Scheduler<Worker> scheduler
                = new Scheduler<>(group);
for(int i : range(initial)){
  Worker.workload.in(new Product());
}
while(not(Worker.workload.empty())){
  Worker worker = scheduler.schedule();
  worker.work();
  scheduler.delay();
  worker.report();
}
```

End of the Lab 05