Polymorphism

Lab 6

TA : Hyuna Seo, Kichang Yang, Minkyung Jeong, Jaeyong Kim

SEOUL NATIONAL UNIVERSITY

Announcement

- You should finish the lab practice and submit your job to eTL before the next lab class starts(Wednesday, 7:00 PM).
- The answer of the practice will be uploaded after the due.

Overview

- **Recap:** Polymorphism
  - Overloading
  - Overriding
  - Generics
- **Preliminary:** Producer-Consumer Pattern
- **Problem**

Recap: Overloading

- Two or more methods in a class can have **the same name** but different parameter formats.
- Which function to call is determined based on types of parameters.

```
class Point {
    float x, y;
    void move(int dx, int dy) { x += dx; y += dy; }
    void move(float dx, float dy) { x += dx; y += dy; }
    public String toString() { return "("+x+", "+y+")"; }
}
```

Recap: Overloading

- **Parameters** have to be different.
  Return type doesn't matter

```
int add(int, int);
float add(float, float);
int add(int, int, int);
```

Different parameters
→ Overloading

```
int add(int, int);
double add(int, int);
```

Only return type different
→ Cannot coexist

Recap: Why Overloading?

- To handle the same task on **different data types.**

Supports both (int, int) and (float, float) Argument data types

```
class Point {
    float x, y;
    void move(int dx, int dy) { x += dx; y += dy; }
    void move(float dx, float dy) { x += dx; y += dy; }
    public String toString() { return "("+x+", "+y+")"; }
}
```

main Method

```
Point p = new Point();
p.move(3, 0);
p.move(0.0f, 4.0f);
System.out.println(p.toString());
```

Output

```
(3.0, 4.0)
```

Recap: Why Overloading?

- To handle slightly different, but closely related behaviors.

```
static double abs_add(double a, double b) {
   return Math.abs(a) + Math.abs(b);
}
static double abs_add(double[] arr) {
   double sum = 0;
   for(int index = 0; index < arr.length; ++index) {
       sum = abs_add(sum, arr[index]);
   }
   return sum;
}
```

main Method

```
double[] arr = new
   double[]{1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0,10.0};
System.out.println(abs_add(arr));
```

Output

```
55.0
```

Recap: Why Overloading?

- Supply default values for the parameters.

```
static double power(double input, int n){
    if(n <= 0) return 1;
    return input * power(input, n-1);
}
static double power(double input){
    return power(input, 2);
}
```

main Method

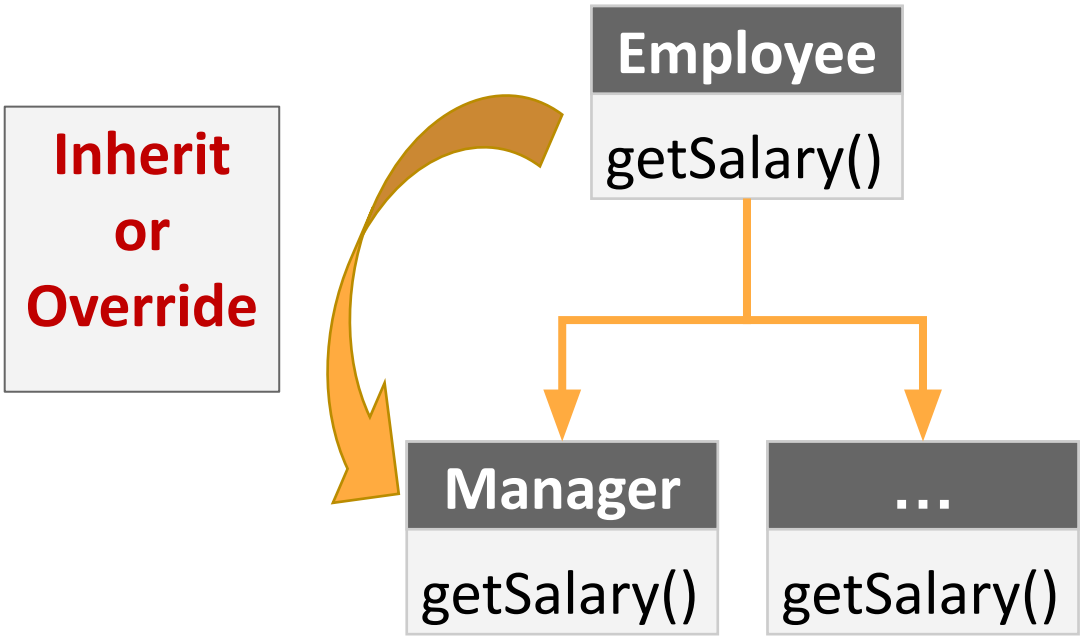
```
System.out.println(power(2, 10));
System.out.println(power(2));
```

Output

```
1024.0
4.0
```

Recap: Overriding

- To redefine an inherited **instance method** of a subclass to change or extend the behavior of the parent's corresponding method.



Name, parameters, return type have to be same!

Recap: Overriding

```
class Employee {
    public double getSalary() { return 180; }
}
class Manager extends Employee {
    @Override
    public double getSalary() { return 250; }
}
```

main Method

```
Employee e = new Employee();
Manager m = new Manager();
System.out.println("Employee : "
        + e.getSalary()
        + " , Manager : "
        + m.getSalary());
```

Output

```
Employee : 180.0 , Manager
: 250.0
```

Recap: Overriding Object Methods

- We can override methods of the class *Object* to suit the purpose of our own classes.

```
class Point2D {
double x, y;
Point2D(double x, double y) { this.x = x; this.y = y; }
    public string toString() { return "(" + x + "," + y + ")“; }
    public boolean equals (Object o) {
      if(x == ((Point2D)o).x && y ==((Point2D)o).y) return true;
      return false;
   }
}
```

main Method

```
System.out.println(new Point2D(66,54));
System.out.println((new Point2D(1,2)).equals(new Point2D(4,5)));
```

Output

```
(66,54)
False
```

Generics

- A method, class or interface defined with a **type variable**, and thus applicable to arbitrary types.
- ***Generics*** can be considered as Java implementation of ‘parametric polymorphism’**.**
- Advantages
  - Robustness
  - No type casting

Generic Class

- Class declaring one or more type variables.
  - Syntax: class C<T1,...,Tn>.
  - Referring a type parameter on static member declaration incurs a compile-time error.

```
class Wrapper<T> {
    T obj;
    void add(T obj) { this.obj = obj; }
    T get() { return obj; }
}
```

main Method

```
Wrapper<Integer> m = new Wrapper<Integer>();
m.add(2);
System.out.println(m.get());
```

Output

```
2
```

```
class Pair<T, S> {
    T first; S second;
    Pair(T a, S b) {
        this.first = a;
        this.second = b;
    }
    public String toString() {
        return "(" + first.toString() + ", " + second.toString() + ")";
    }
}
```

Type Variables

main Method

```
System.out.println(new Pair<Integer, String>(6, "Six"));
System.out.println(new Pair<Boolean, String>(true, "True"));
```

Output

```
(6, Six)
(true, True)
```

Generic Methods

- A method declaring one or more type variables.
  - Type arguments to the method may either be inferred or passed explicitly.

```
class ArrayPrinter {
   public static <E> void printArray(E[] elements) {
      for(E element : elements) {
          System.out.print(element.toString() + " ");
      }
   }
}
```

main Method

```
Integer[] intArray = { 1, 1, 2, 3, 5, 8, 13 };
Character[] charArray = { 'C', 'S', 'E', 'C', 'P' };
ArrayPrinter.printArray(intArray);
ArrayPrinter.<Character>printArray(charArray);
```

Output

```
1 1 2 3 5 8 13
C S E C P
```

Advantages: Robustness

- Resolving types are all done in compile-time.
- Runtime type error does not occur at all, and it is robust. (Type safety)

```
import java.util.ArrayList;
import java.util.List;
```

main Method

```
List<String> list = new ArrayList<>();
list.add("Compile");
list.add(11235);
```

Output

```
java:
incompatible
types: int cannot
be converted to
java.lang.String
```

Advantages: No Type Casting

```
import java.util.ArrayList;
import java.util.List;
```

main Method (Before Generics)

```
List list = new ArrayList();
list.add("String");
System.out.println((String)list.get(0));
```

Type “Object”

Output

```
String
```

main Method (After Generics)

```
List<String> list = new ArrayList<>();
list.add("String");
System.out.println(list.get(0));
```

Wildcards in Generics

- The ? (question mark) symbol is a wildcard element. It is used to restrict a type together with the “extends” or “super” keywords.

- It is used to specify a method’s return type and parameter types.

Upper-Bounded Wildcard <? extends >

- Restricts the type to be a subtype of a class.

```
import java.util.ArrayList;
private static Double reduce(ArrayList<? extends Number> num) {
    double sum = 0.0;
    for(Number n:num) { sum = sum + n.doubleValue(); }
    return sum;
}
```

Now, num can access members of class Number

main Method

```
ArrayList<Integer> l1 = new ArrayList<>();
l1.add(10); l1.add(20);
System.out.println("sumint "+reduce(l1));
```

Output

```
sumint 30.0
```

Lower-Bounded Wildcards <? super >

- Restrict the type to be a supertype of a class.

```
public static void addNumbers(List<? super Integer> list) {
    for (Object n : list) {
        System.out.print(n.toString() + " ");
    }
}
```

main Method

```
List<Integer> l1 = Arrays.asList(1, 2, 3);
addNumbers(l1);
List<Number> l2 = Arrays.asList(1.0, 2.0, 3.0);
addNumbers(l2);
```

Output

```
1 2 3 1.0 2.0 3.0
```

Unbounded Wildcards <?>

- Represents an arbitrary type.
  - Object is assumed for the bound.

```
import java.util.Collection;
static void printCollection(Collection<?> c) {
   for (Object o : c) { System.out.println(o); }
}
```

main Method

```
Collection<String> cs = new
ArrayList<String>();
cs.add("hello");
cs.add("world");
printCollection(cs);
```

Output

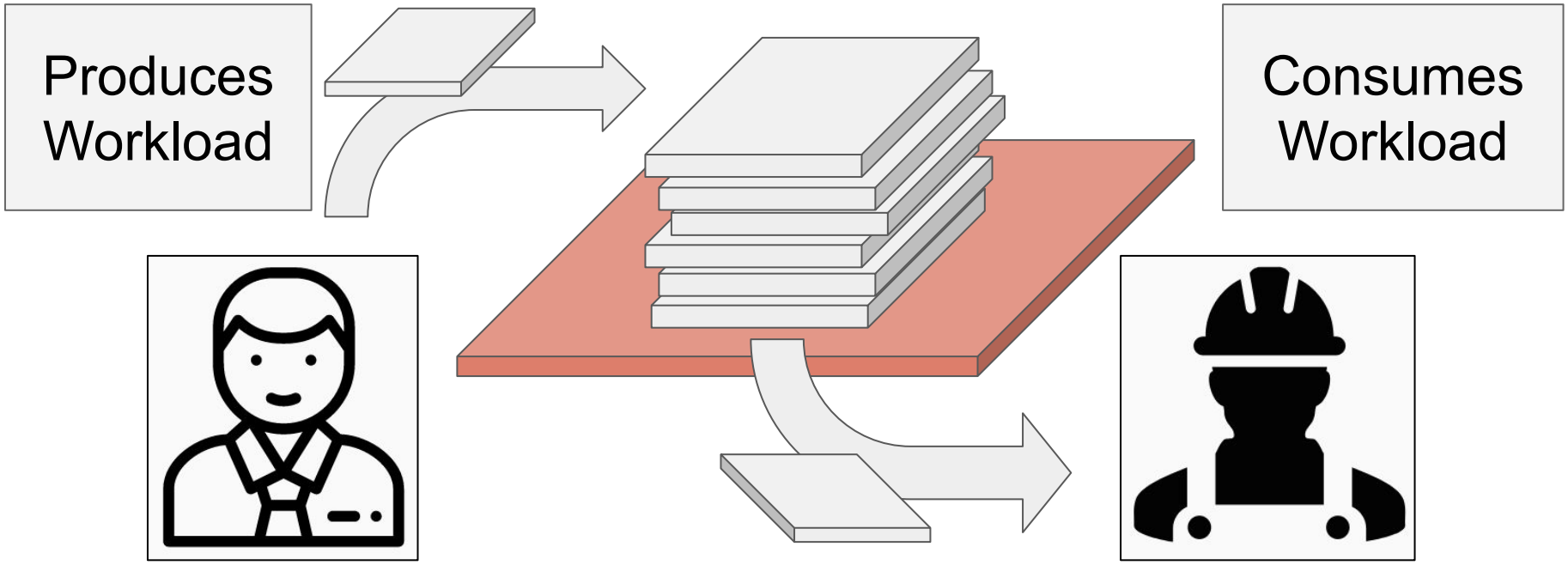
```
hello
world
```

Overview

- ~~Recap: Polymorphism~~
  - ~~Overloading~~
  - ~~Overriding~~
  - ~~Generics~~
- Preliminary: Producer-Consumer Pattern
- Problem

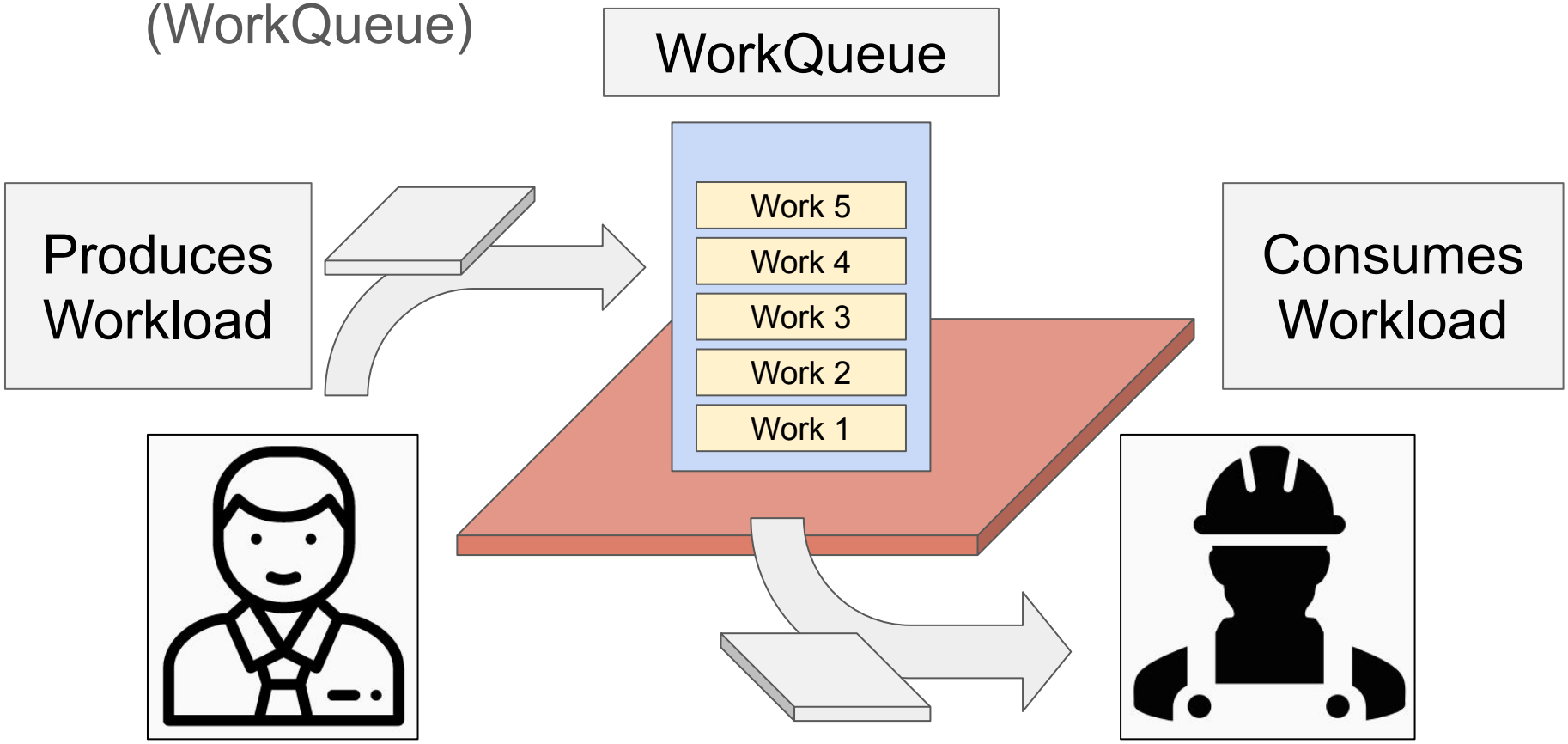
Producer-Consumer Pattern

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



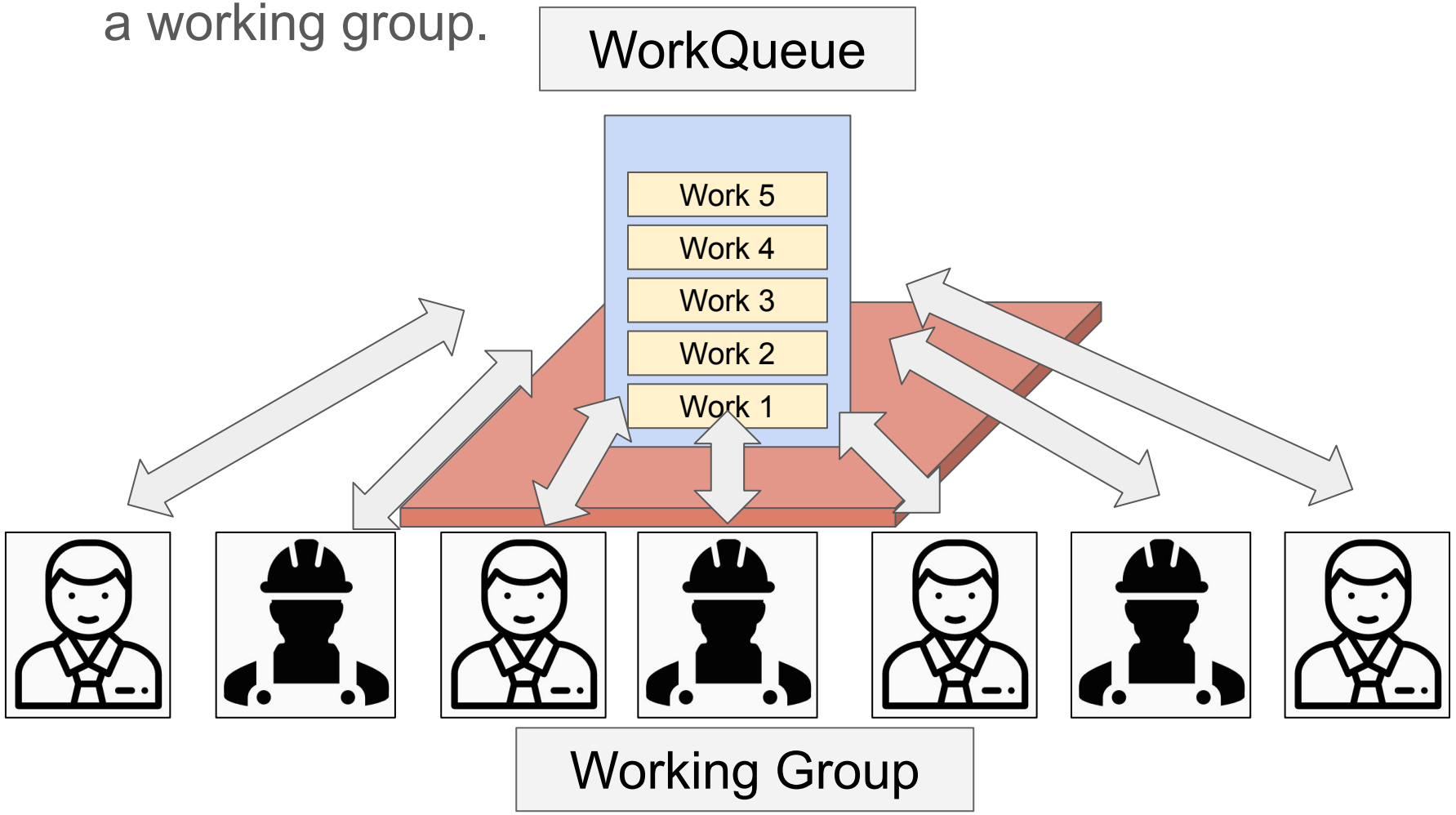
Producer-Consumer Pattern

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



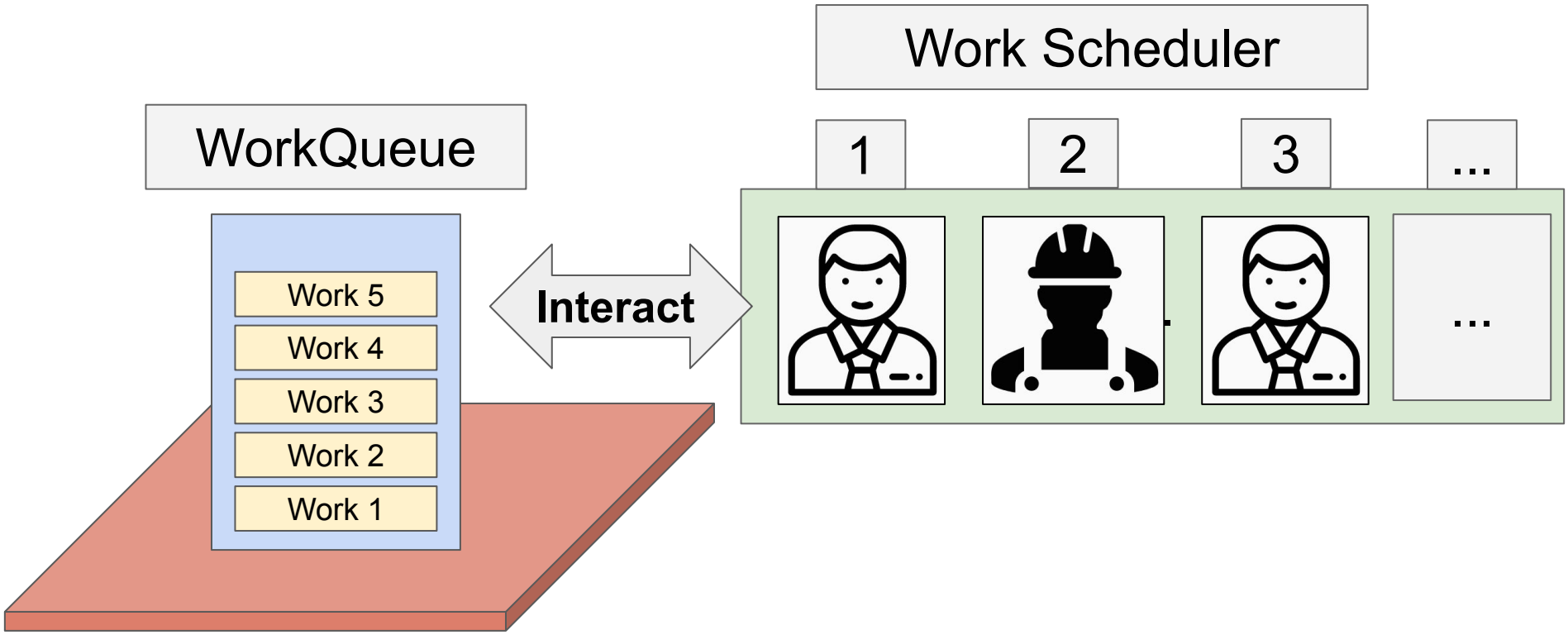
Producer-Consumer Pattern

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



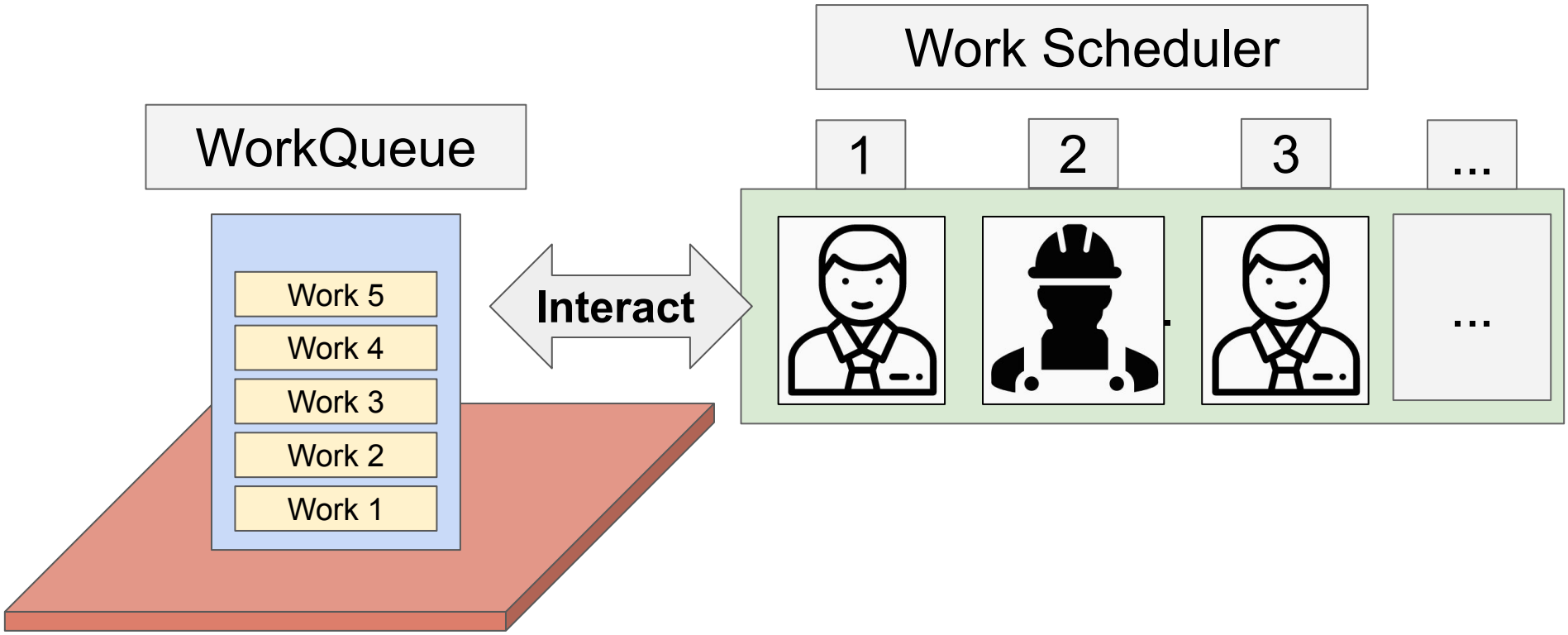
Producer-Consumer Pattern

- 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.



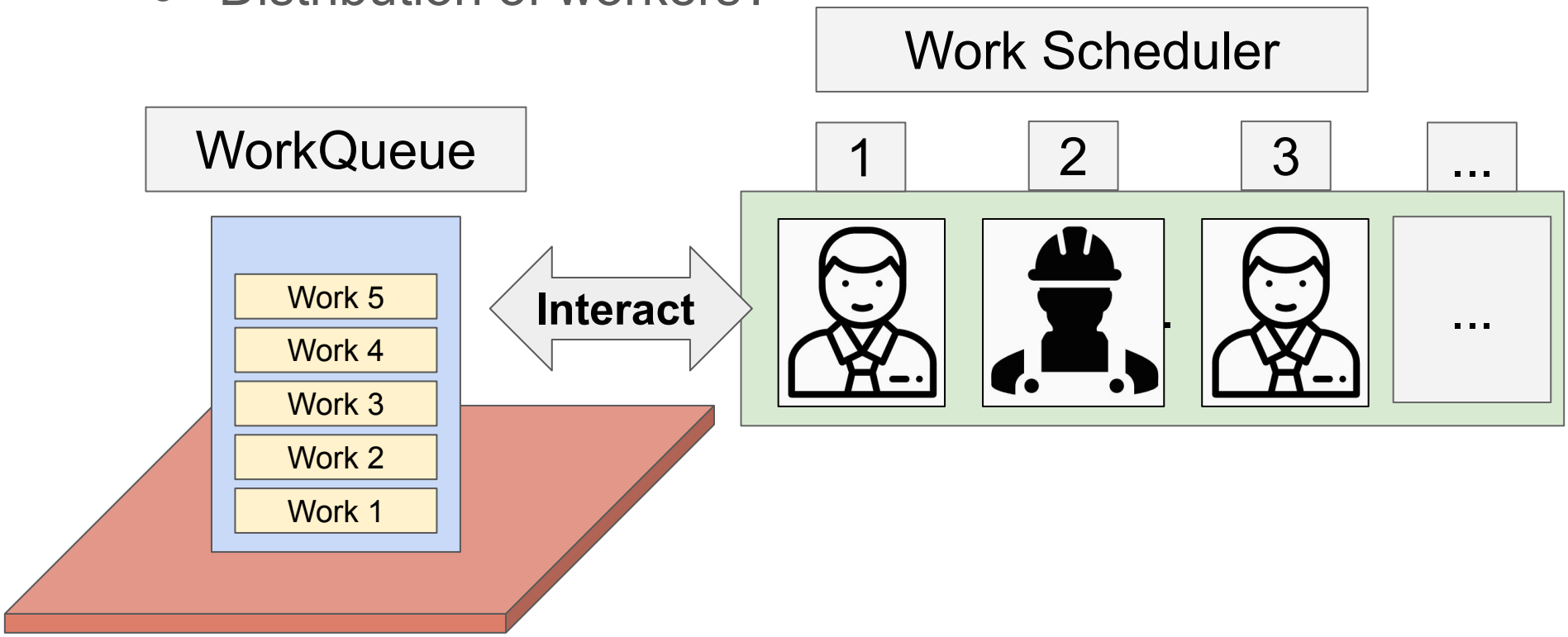
Producer-Consumer Pattern

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



Producer-Consumer Pattern

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



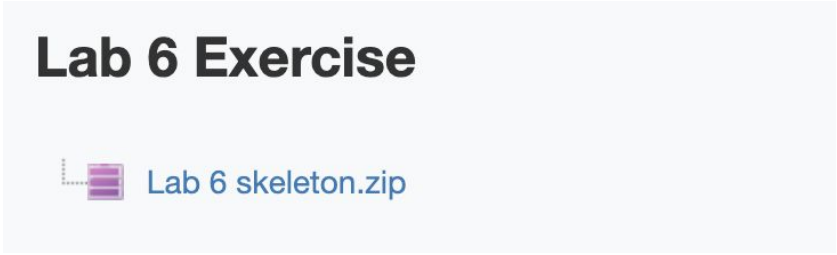
Problem Objectives

- Understand the differences of static and instance members, and observe the practical usage of these differences.
- Understand the advantages of Overriding in java.
- Understand the Generics and its advantage.

Problem Overview

- Problem1 - Work ID and Worker ID (5 min)
  - Managing WorkQueue
  - Managing Workers
- Problem2 - Producer and Consumer (10 min)
  - Worker class
- Problem3.1 - Scheduling (5 min)
- Problem3.2 - Fair Scheduling (10 min)

Before going in to the problem..

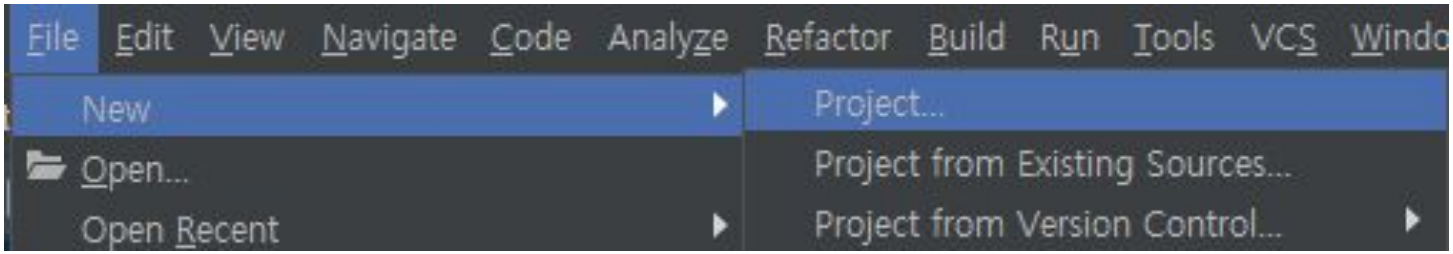


1. Download the skeleton

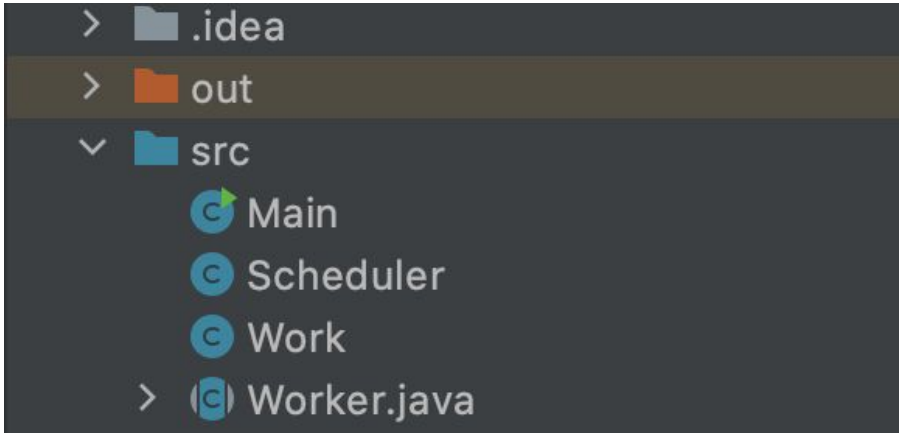
2. Extract the Zip file

3. Check the .java files and Copy them

Before going in to the problem..



4. Create a new project



5. Paste the skeleton code classes to the src dir

Main Class: Managing WorkQueue

- Initialize with Works
- Workers will insert and pop works.
- Queue API
  - Insert : `workQueue.add(new Work());`
  - Take out : `Work work = workQueue.poll();`

```
Queue<Work> workQueue;
workQueue = new LinkedList<>();
for(int i = 0; i < numWorks; i++){
    workQueue.add(new Work());
}
```

Main Class: Managing Workers

- Working Group for managing the workers, each worker of type `Producer` or `Consumer`.
  - `List<Worker>` : List of Workers
- Working Group contains derived classes of the Worker.

```
List<Worker> workers;
workers = new LinkedList<>();
for (int i = 0; i < numProducers; i++) {
    workers.add(new Producer(workQueue));
}
for (int i = 0; i < numConsumers; i++) {
    workers.add(new Consumer(workQueue));
}
```

Main Class: Managing Workers

- The scheduler schedules a worker.
- Selected worker (producer / consumer) runs.

```
Scheduler<Worker> scheduler = new Scheduler<>(workers);
while(!(workQueue.isEmpty())){
    System.out.printf("Scheduling work... (Remaining work:
%d)\n", workQueue.size());
    Worker worker = scheduler.schedule();
    if (worker != null) {
        worker.run();
    }
    Scheduler.delay();
}
```

5 mins

Problem1 - Work ID and Worker ID

- Add id attribute to `Work` and `Worker` class
- Consider static attribute.
- Every `Work` have their own id.
  - The first `Work` object id is 0, the second `Work` object id is 1, ...
- Every `Worker` object (including `Producer` and `Consumer`) have their own id.
  - The first `Worker` object id is 0, the second `Worker` object id is 1, ...

Work class

```
public class Work {
    Work() {
        // TODO: problem1
    }
}
```

Worker class

```
public abstract class Worker {
    Queue<Work> workQueue;
    public abstract void run();
    Worker(Queue<Work> workQueue) {
        this.workQueue = workQueue;
        // TODO: problem1
    }
}
class Producer extends Worker {
    Producer(Queue<Work> workQueue) { super(workQueue); }
    public void run() { }
}
class Consumer extends Worker {
    Consumer(Queue<Work> workQueue) { super(workQueue); }
    public void run() { }
}
```

Q&A

10 mins

Problem2 - Producer and Consumer

- Implement `public void run()` method of `Producer` and `Consumer` class.
- The `run` method of the `Producer` class always adds a `Work` object to the workQueue.
- Assume that the capacity of workQueue is 20. If the workQueue is full, the producer cannot add a work to the workQueue.
- The `run` method of the `Consumer` class probabilistically (50%) takes out a `Work` object to the workQueue.

10 mins

Problem2 - Producer and Consumer

- Add report message to the `public void run()` method of `Producer` and `Consumer` class.
  - Implement the `report()` method in `Worker` class and use it in the `run()` method of `Producer` and `Consumer` class.
  - When the producer produces a work, print
    “worker<Worker ID> produced work<Work ID>”
  - When the producer fails to produce a work (workQueue is full), print
    “worker<Worker ID> failed to produce work”
  - When the consumer consumes a work, print
    “worker<Worker ID> consumed work<Work ID>”
  - When the consumer fails to consume a work (2/3 probability failure or no work in the workQueue), print
    “worker<Worker ID> failed to consume work”

Worker class

```
public abstract class Worker {
    ...
    public abstract void run();
    void report(String msg){
        // TODO: problem2
    }
}
class Producer extends Worker {
    Producer(Queue<Work> workQueue) { super(workQueue); }
    public void run() {
        // TODO: problem2
    }
}
class Consumer extends Worker {
    Consumer(Queue<Work> workQueue) { super(workQueue); }
    public void run() {
        // TODO: problem2
    }
}
```

Q&A

5 mins

Problem3.1 - Scheduler

`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`.

```
public class Scheduler<T> {}
```

5 mins

Problem3.1 - Scheduler

- Currently, `T schedule()` method in `Scheduler` class returns the first worker from the list.
- Implement `T schedule(int index)` method in `Scheduler` class.
  - Return the worker of the given index from the list. If the index is out of list range, return the first worker.
- Implement `T scheduleRandom()` method in `Scheduler` class.
  - Return random worker.

Problem3.1 - Scheduler

```
T schedule() {
    return workers.get(0);
}

T schedule(int index) {
    // TODO: problem3.1 - Add scheduling logic
}

T scheduleRandom() {
    // TODO: problem3.1 - Add scheduling logic
}
```

10 mins

Problem3.2 - Fair Scheduler

- Implement `T scheduleFair()` method in `Scheduler` class.
  - Return the worker who wasn't scheduled for the longest time.
  - To implement this scheduling policy, define additional member variable in Worker class that tracks how long the worker wasn't scheduled.
  - (Hint) Change the `Scheduler<T>` to `Scheduler<T extends Worker>` so that you can access the members of Worker.

Problem3.2 - Fair Scheduler

```
public class Scheduler<T extends Worker> {

    ...

    T scheduleFair() {

        // TODO: problem3.2 - Add scheduling logic

    }

}
```

Q&A

Submission

- Compress your `src` directory into a zip file.
  - After unzip, the ‘src’ directory must appear.
- Rename your zip file as 20XX-XXXXX_{name}.zip - for example, 2021-12345_YangKichang.zip
- Upload it to eTL - Lab 6 assignment.