Advanced Topics in Programming

LAB 4 – JAVA CONTAINERS & DESIGN PATTERNS

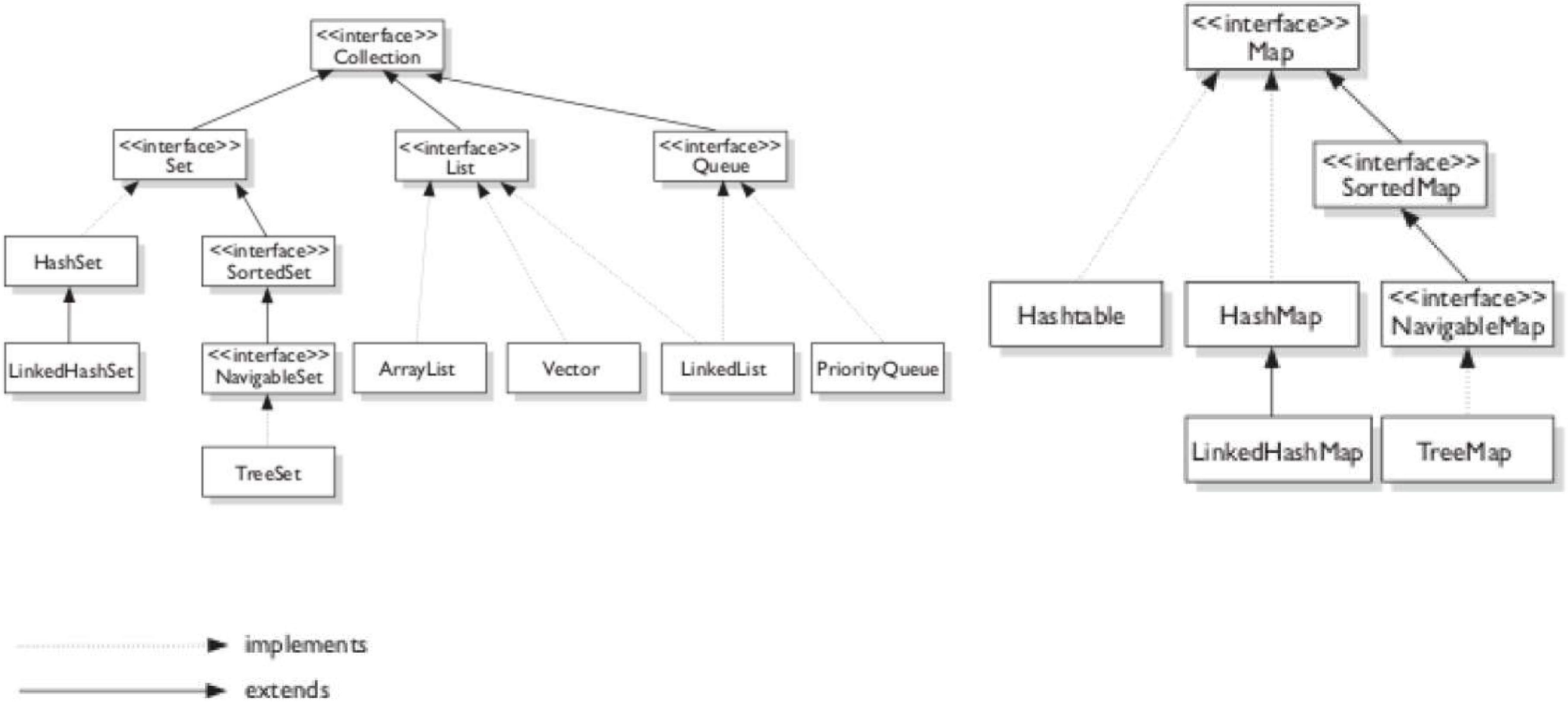
Java Containers

java.util.*

Useful Containers

- ❑Java has 2 types of containers:
  - ❑Collections – collect single **values**
    - ❑Lists – sequence is important
    - ❑Sets – each element appears only once
  - ❑Maps – map **keys** to **values**
- ❑The implementation is as you have learned in Data-Structures course.
- ❑Use them wisely

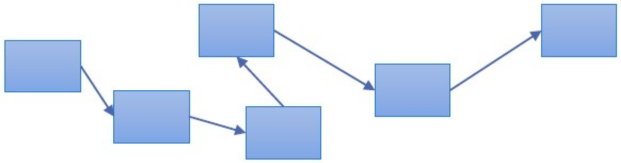
Useful Containers

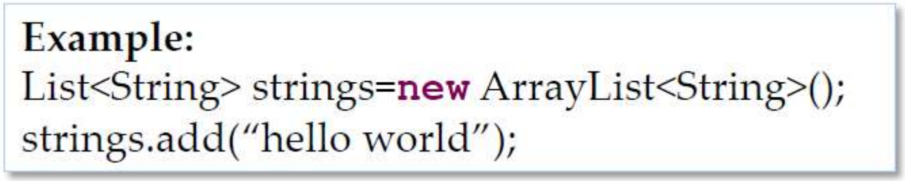


Useful Containers - collections

- Lists:
  - **ArrayList<E>** – uses an array
    - Fast random access: O(1)
    - Slow addition / deletion from the middle: O(1) amortized

  - **LinkedList<E>** - uses a lined list
    - Slow random access: O(n)
    - Fast addition / deletion from the middle: O(1)

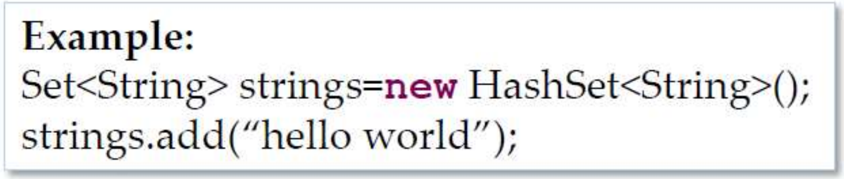




Example:
```
List<String> strings=new ArrayList<String>();
strings.add(“hello world”);
```

Useful Containers - collections

- Sets:
  - **HashSet<E>** – uses an hash table
    - Use when search time is important
    - Object’s int HashCode() method needs to be overridden
    - Usually we’ll use something ready as String’s hash code
  - **TreeSet<E>** - uses a balanced tree
    - O(log(n)) for random access
    - Can easily extract a sorted list

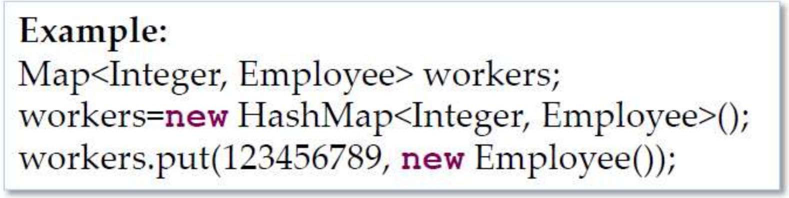


Example:

```
Set<String> strings=new HashSet<String>();
strings.add(“hello world”);
```

Useful Containers - maps

- Maps example:
  - **HashMap** – uses a hash table
    - The **key** object needs to implement *hashcode()* method
  - **LikedHashMap**
    - Also stores the order of entry
  - **TreeMap** – uses a red-black tree
    - Can easily extract a sorted list



Example:

```
Map<Integer, Employee> workers;
workers=new HashMap<Integer, Employee>();
workers.put(123456789, new Employee());
```

Design Patterns

Factory,Command

Types of Design Patterns

Creational: These patterns deal with object creation and initialization. Creational pattern gives the program more flexibility in deciding which objects need to be created for a given use case.

- **Singleton** , Factory and etc.

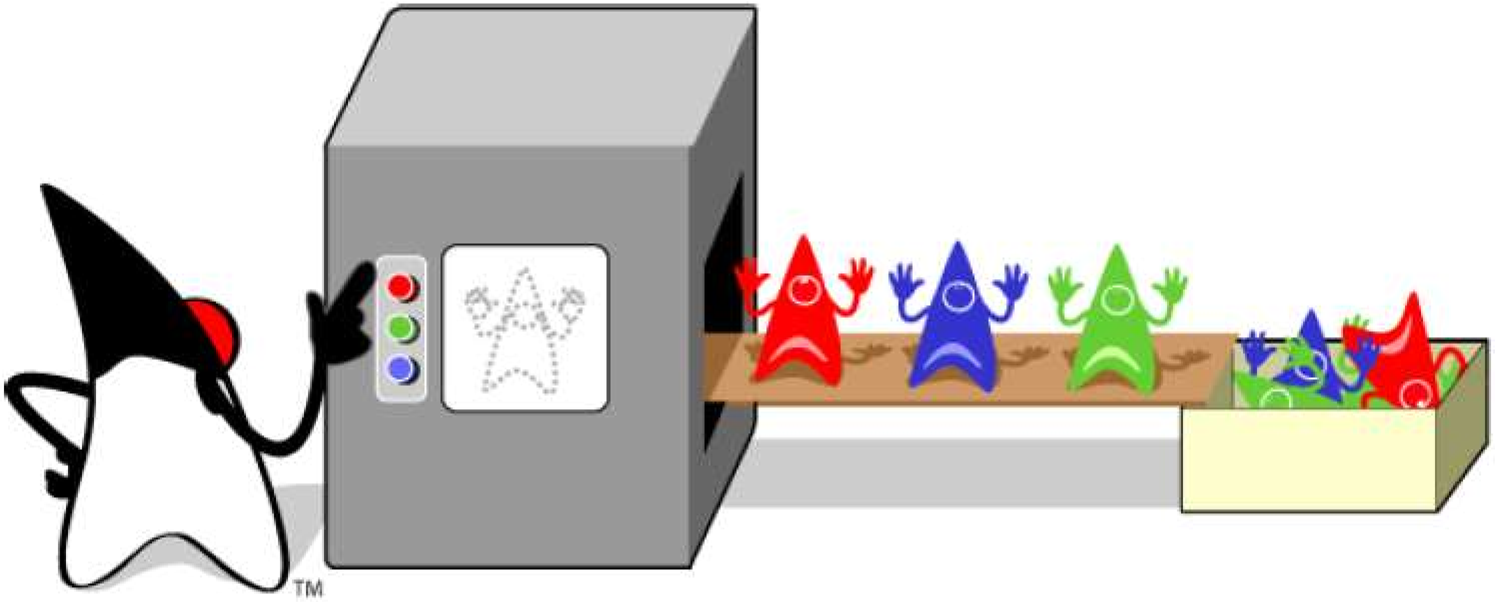
Structural: These pattern deals with class and object composition. In simple words, This pattern focuses on decoupling interface, implementation of classes and its objects.

- **Adapter**, and etc.

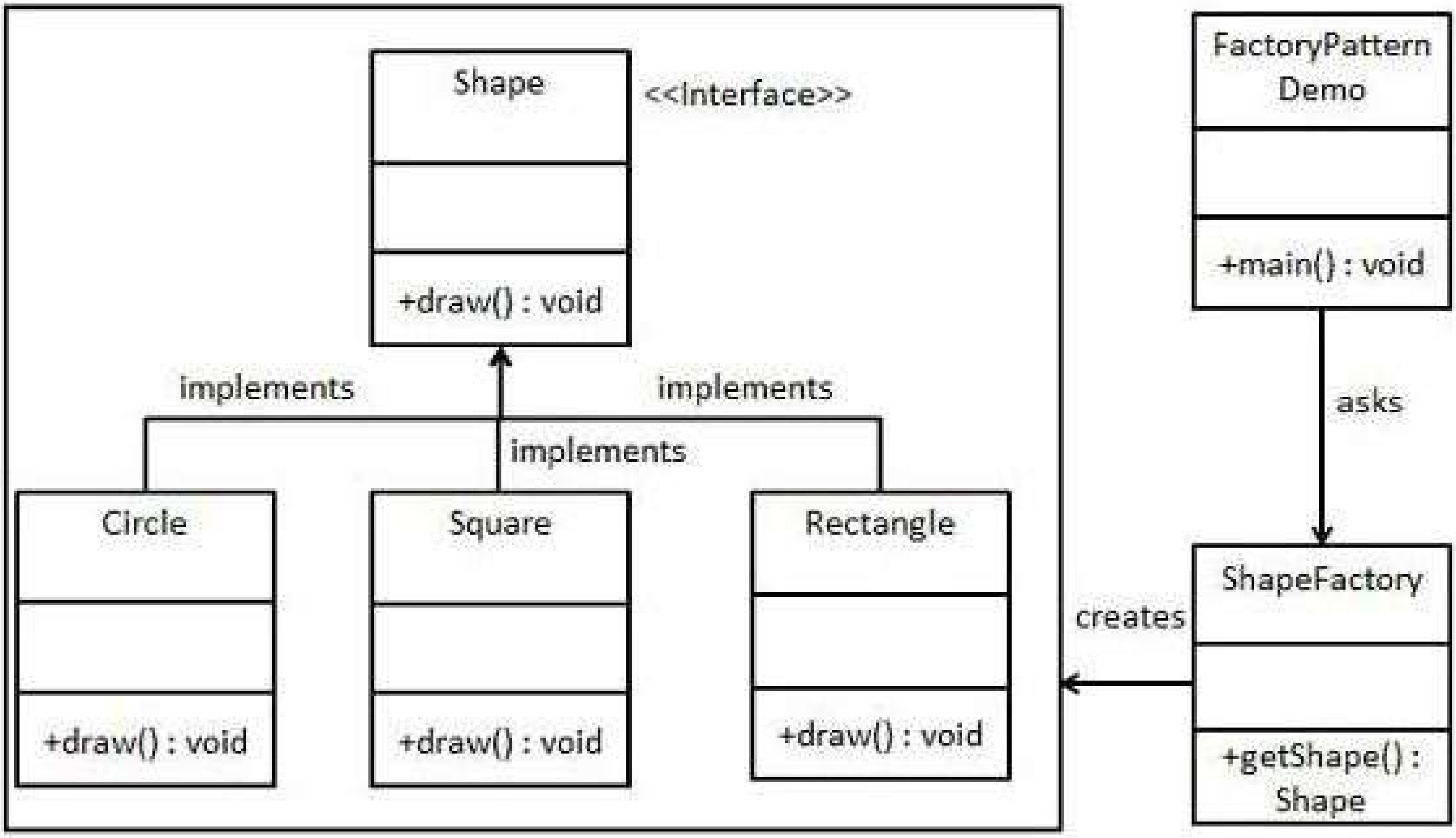
Behavioral: These patterns deal with communication between classes and objects.

- **Strategy**, Command etc.

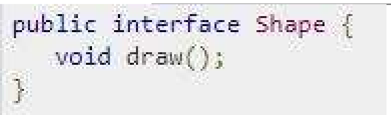
Factory Design Pattern



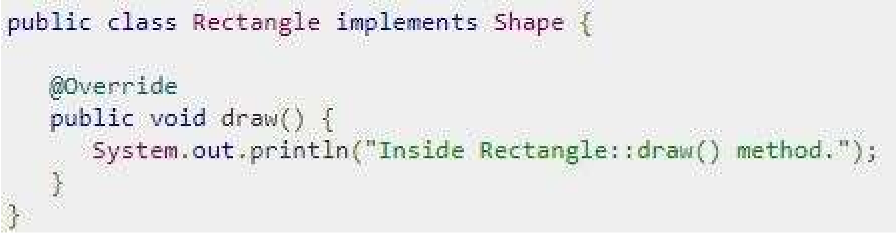
Factory Design Pattern



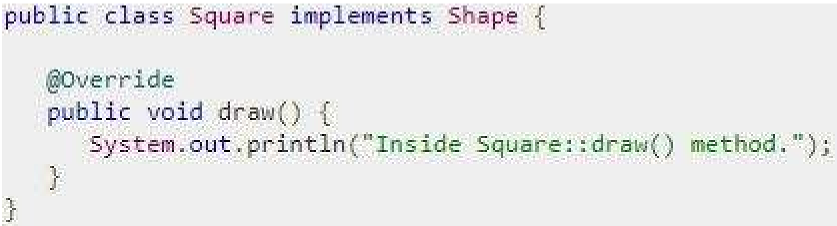
Factory Design Pattern – The Code



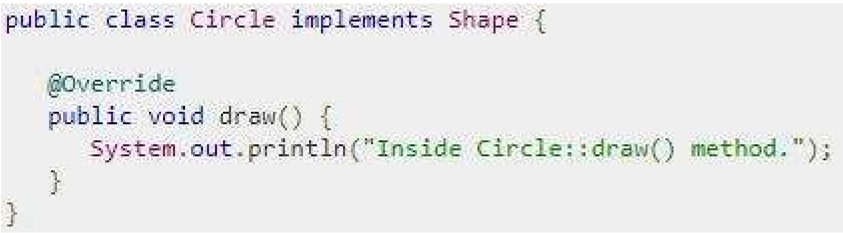
```
public interface Shape {
   void draw();
}
```



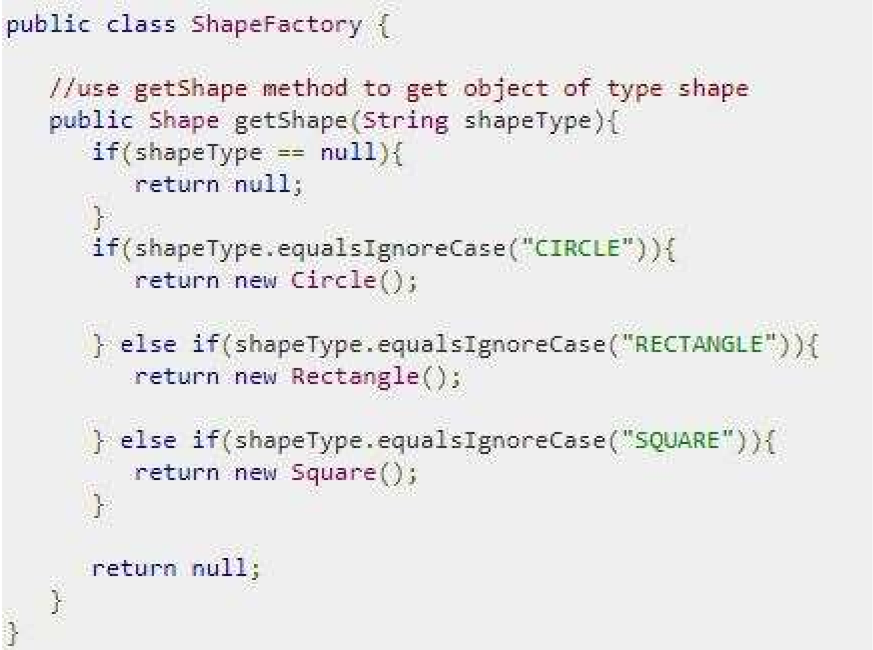
```
public class Rectangle implements Shape {

   @Override
   public void draw() {
      System.out.println("Inside Rectangle::draw() method.");
   }
}
```



```
public class Square implements Shape {

   @Override
   public void draw() {
      System.out.println("Inside Square::draw() method.");
   }
}
```

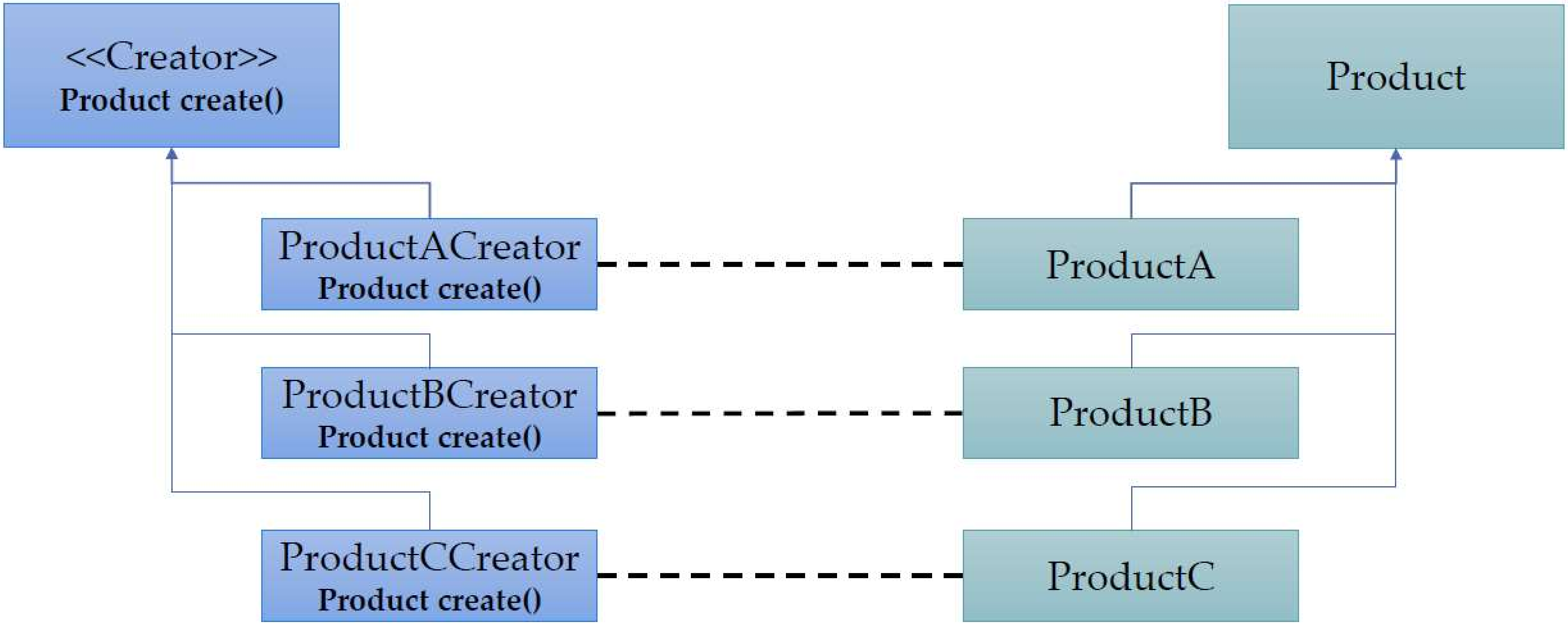


```
public class Circle implements Shape {

   @Override
   public void draw() {
      System.out.println("Inside Circle::draw() method.");
   }
}
```



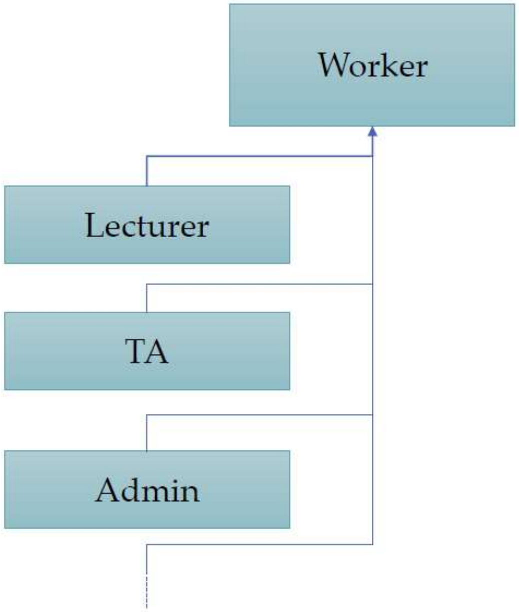
```
public class ShapeFactory {

   //use getShape method to get object of type shape
   public Shape getShape(String shapeType){
      if(shapeType == null){
         return null;
      }
      if(shapeType.equalsIgnoreCase("CIRCLE")){
         return new Circle();

      } else if(shapeType.equalsIgnoreCase("RECTANGLE")){
         return new Rectangle();

      } else if(shapeType.equalsIgnoreCase("SQUARE")){
         return new Square();
      }

      return null;
   }
}
```

Factory Design Pattern

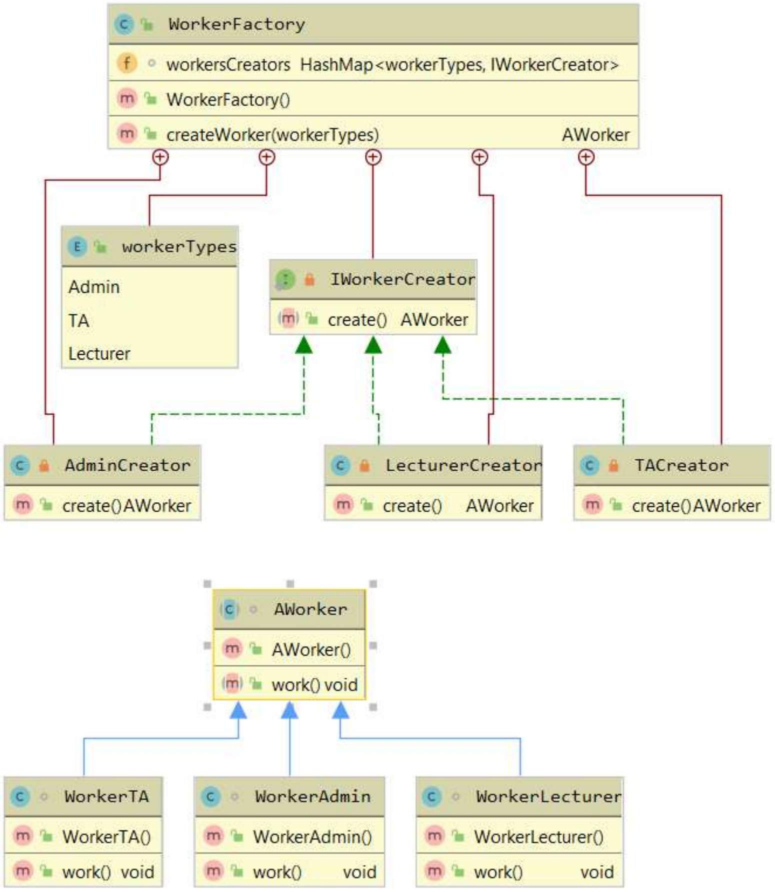


Quiz…

- ❑Let’s say we have n types of workers
- ❑And when the user inputs the type, the right object needs to be instanced.
- ❑Creating n “if” statements takes O(n) time
- ❑It is also not very object oriented…
- ❑Utilize the factory pattern and container to return the new worker in O(1).



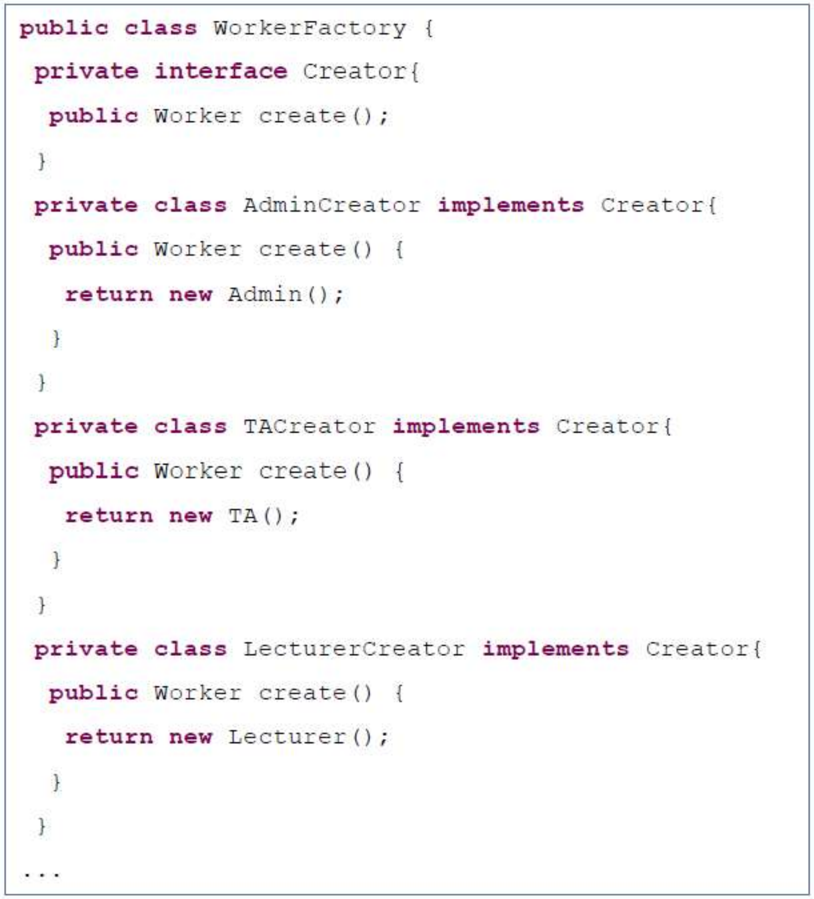
Factory Design Pattern



Factory Design Pattern

- First we implement the interface and the classes inside the factory

- For each type of *AWorker*, we create a Creator class.

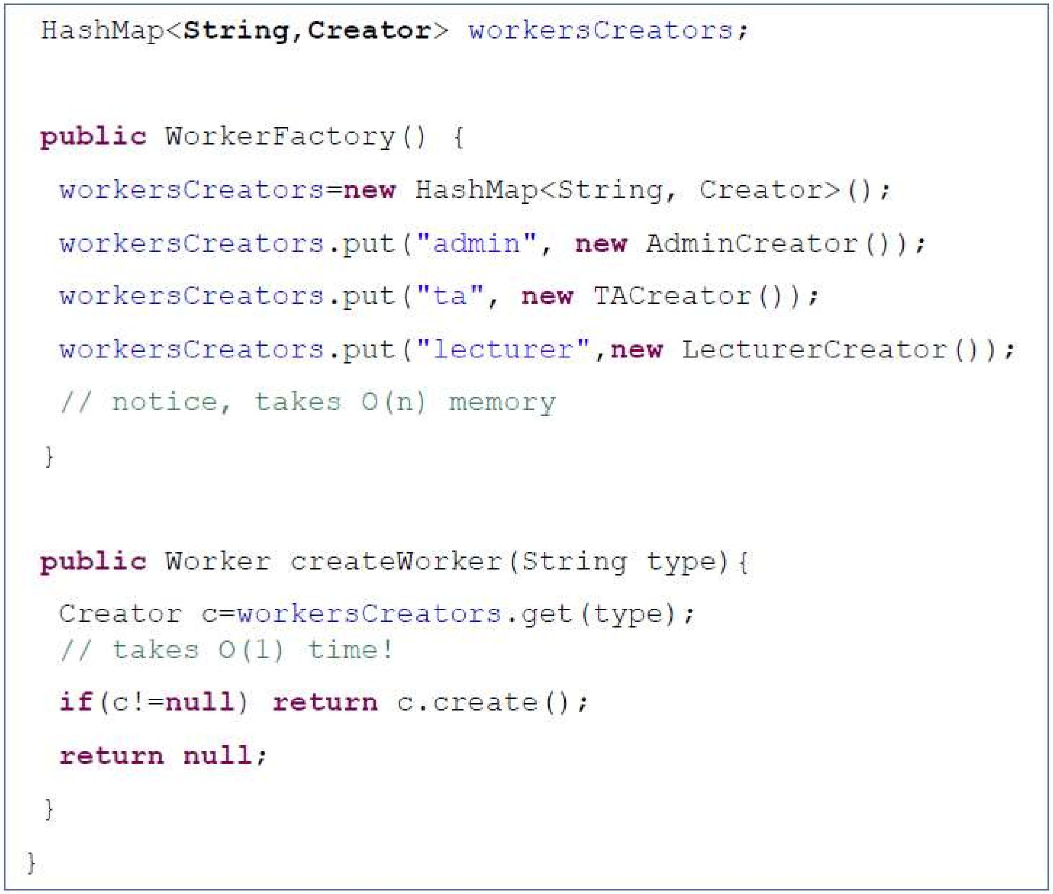


```
public class WorkerFactory {
 private interface Creator{
  public Worker create();
 }
 private class AdminCreator implements Creator{
  public Worker create() {
   return new Admin();
  }
 }
 private class TACreator implements Creator{
  public Worker create() {
   return new TA();
  }
 }
 private class LecturerCreator implements Creator{
  public Worker create() {
   return new Lecturer();
  }
 }
...
```

Factory Design Pattern

- Next we create a HashMap!
- String → Creator
- The **key** is exactly the user's parameter.
- The **value** is creator.
- We instantiate each class once, O(n) memory

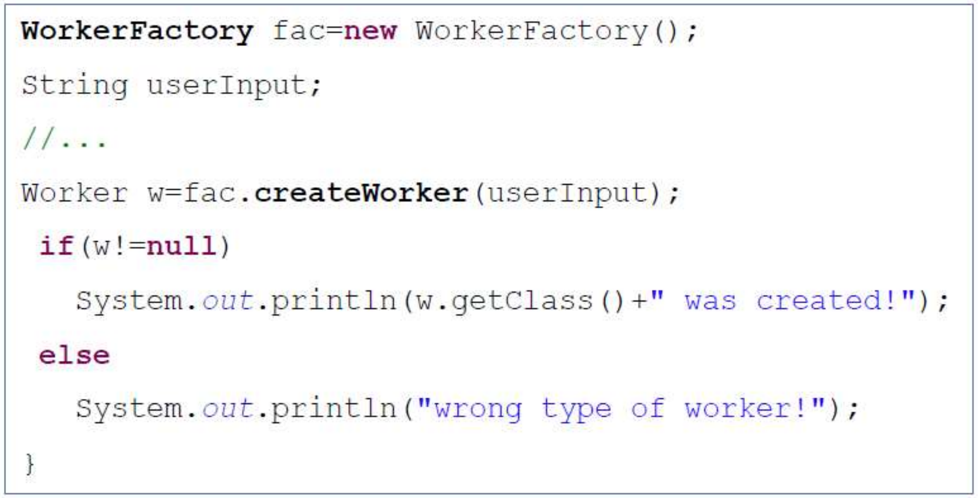
- Notice how createWorker takes O(1) instance of Worker of the given type!



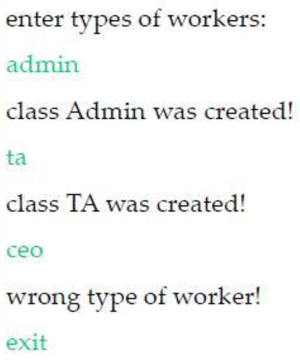
```
HashMap<String,Creator> workersCreators;


public WorkerFactory() {
 workersCreators=new HashMap<String, Creator>();
 workersCreators.put("admin", new AdminCreator());
 workersCreators.put("ta", new TACreator());
 workersCreators.put("lecturer",new LecturerCreator());
 // notice, takes O(n) memory
}


public Worker createWorker(String type){
 Creator c=workersCreators.get(type);
 // takes O(1) time!
 if(c!=null) return c.create();
 return null;
}
}
```

Factory Design Pattern

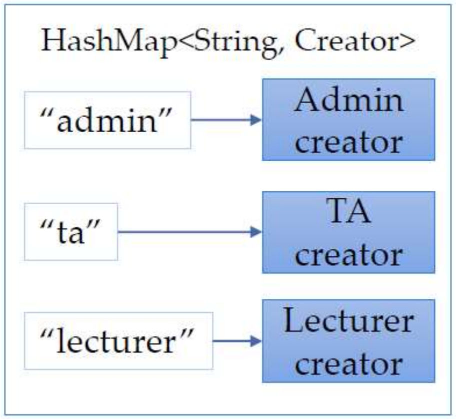
- Usage example

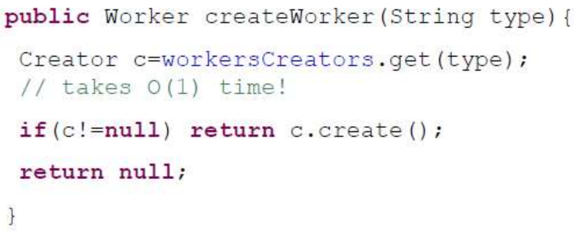


```
WorkerFactory fac=new WorkerFactory();
String userInput;
//...
Worker w=fac.createWorker(userInput);
 if(w!=null)
   System.out.println(w.getClass()+" was created!");
 else
   System.out.println("wrong type of worker!");
}
```



```
enter types of workers:
admin
class Admin was created!
ta
class TA was created!
ceo
wrong type of worker!
exit
```





```
public Worker createWorker(String type){
 Creator c=workersCreators.get(type);
 // takes O(1) time!
 if(c!=null) return c.create();
 return null;
}
```

Command Design Pattern

Command Design Pattern

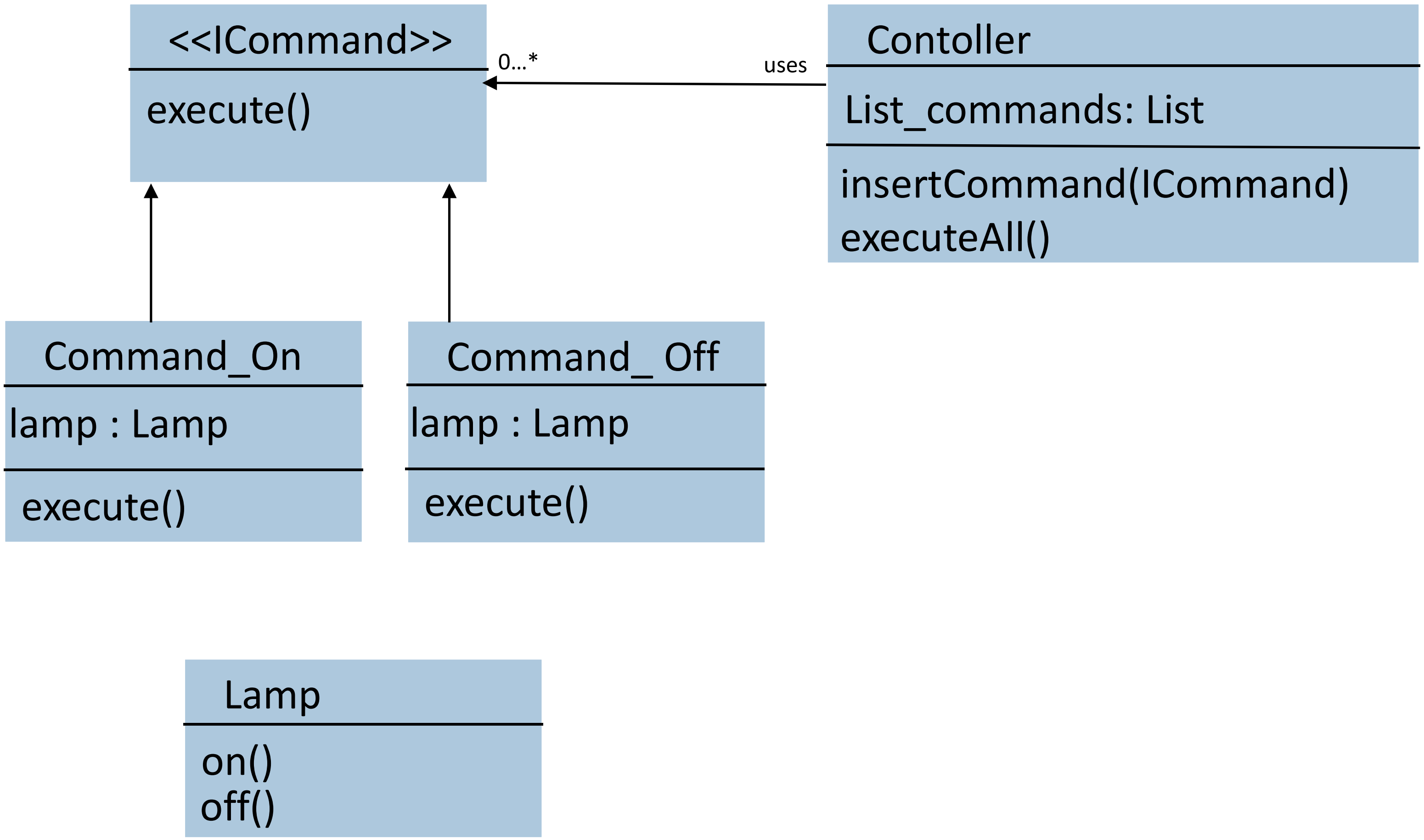
❑A request is wrapped under an object as command

❑Then passed to controller object.

❑Controller looks for the appropriate object which handle this command

❑And passes the command to the corresponding object which executes the command.

Command Design Pattern



<<ICommand>>

execute()

0…*

uses

Contoller

List_commands: List

insertCommand(ICommand)
executeAll()

Command_On

lamp : Lamp

execute()

Command_ Off

lamp : Lamp

execute()

Lamp

on()
off()

Lab Exercise

- Part 1 :
  - Create Class FileReader
- Part 2 :
  - Get to know Data Structures
  - Working stop watch