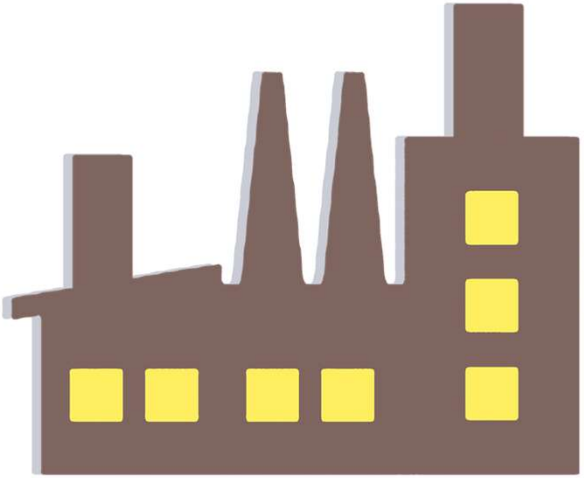
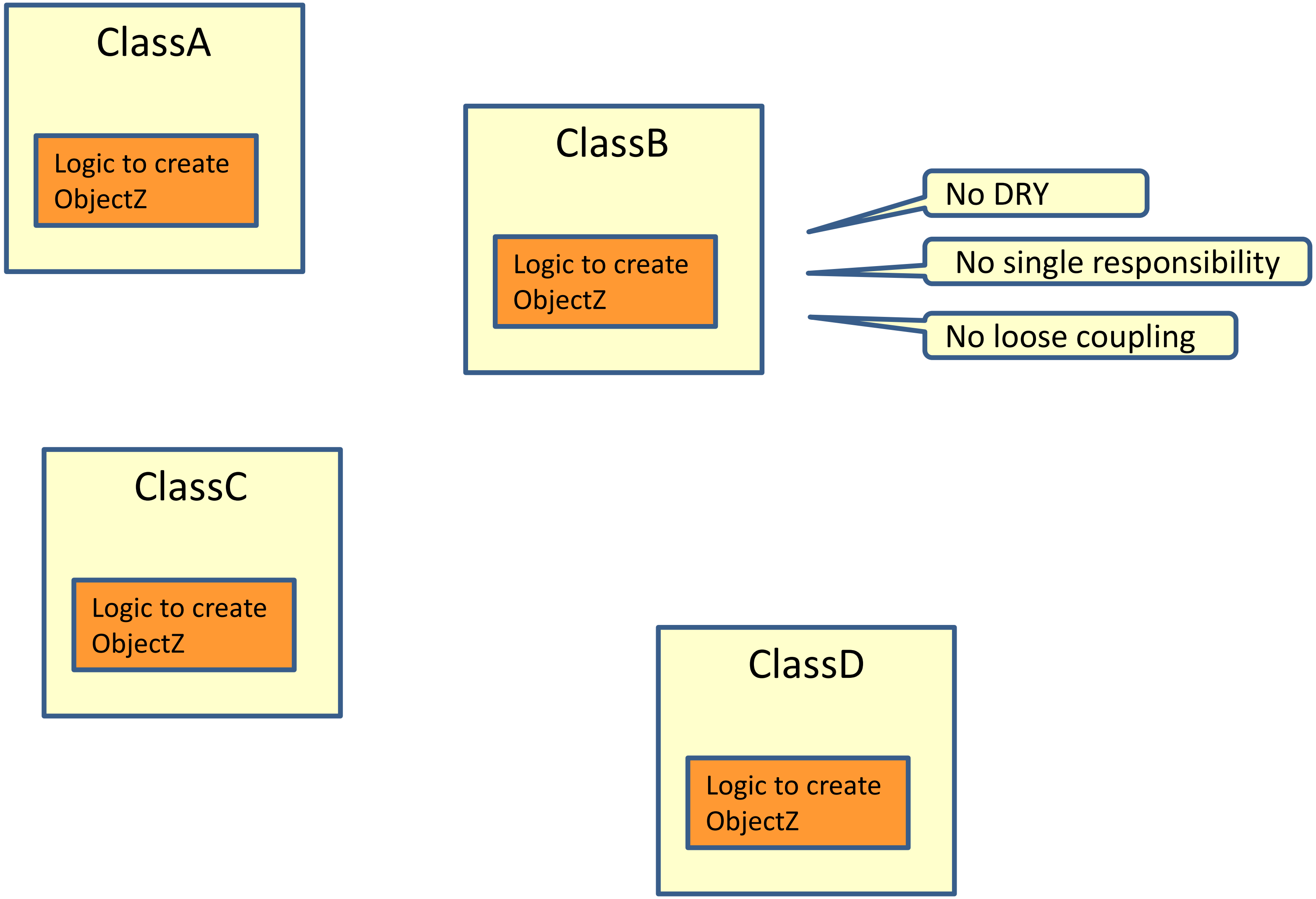
LESSON 9
FACTORY, BUILDER, SINGLETON PATTERN

Factory pattern

- A factory creates objects
    - Encapsulation of the logic to create objects



Without a factory

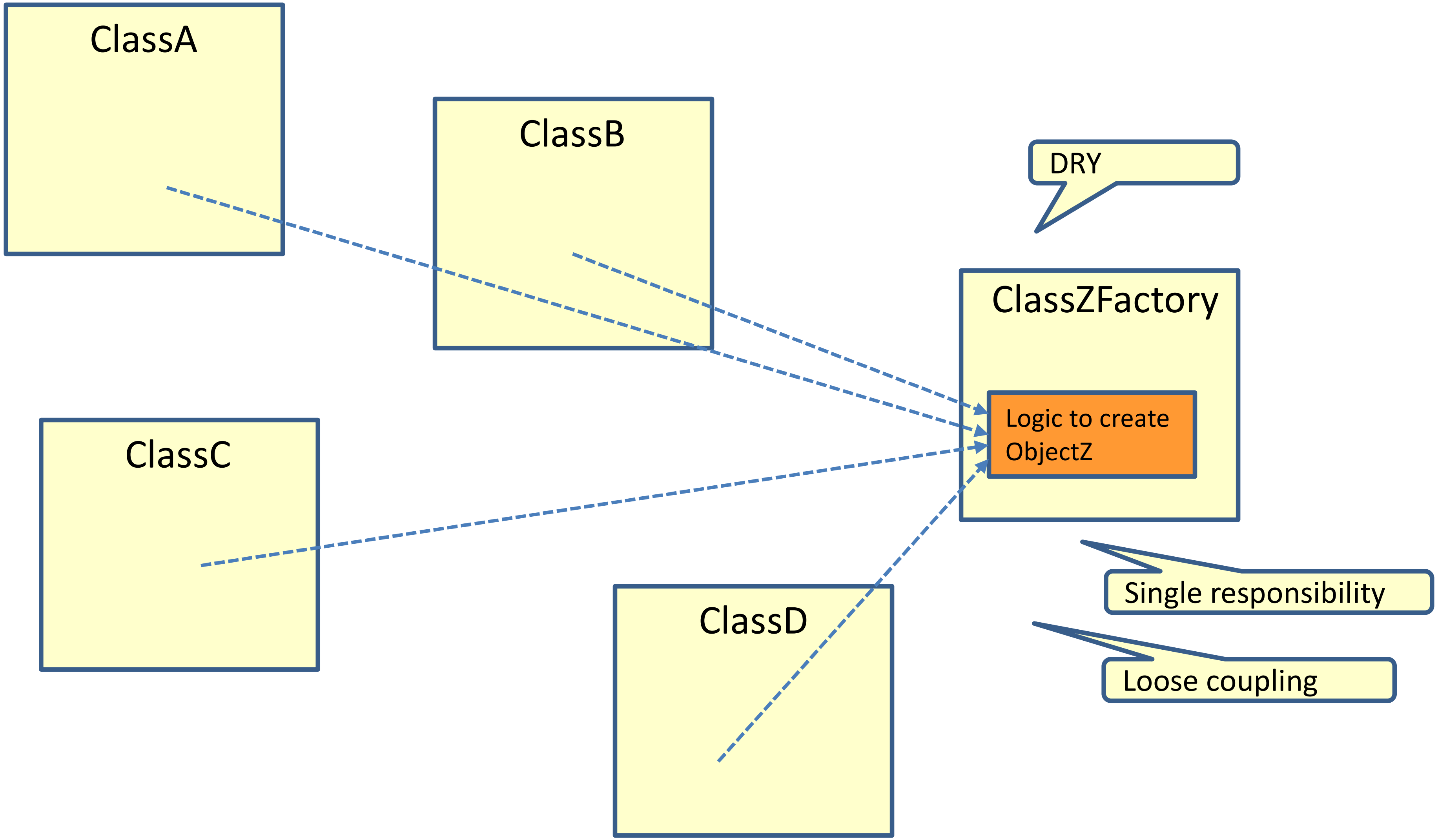


No DRY

No single responsibility

No loose coupling

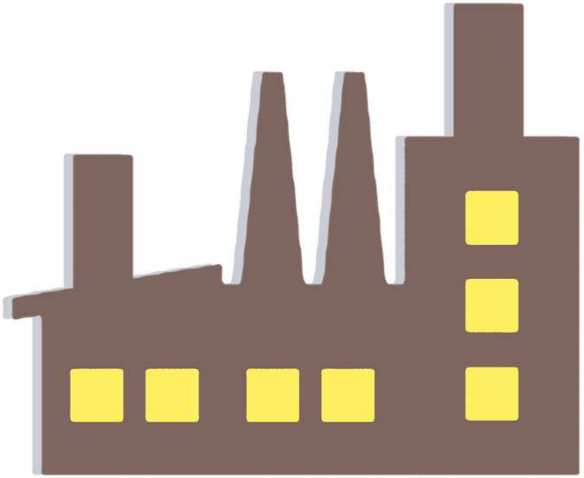
With a factory



© 2023

Different types of factories

- Simple factory method
  - Static or not static
- Factory method pattern
- Abstract factory pattern

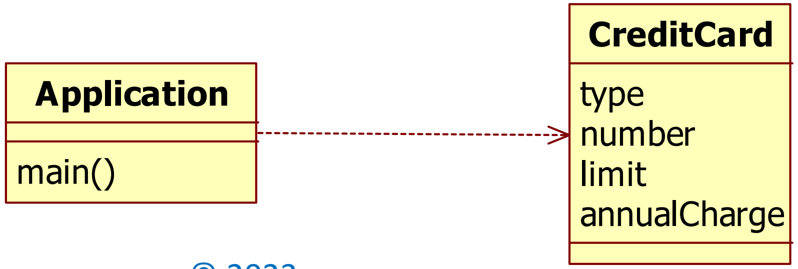


SIMPLE FACTORY METHOD

Using the constructor

```java
public class CreditCard {
  private String type;
  private String number;
  private double limit;
  private double annualCharge;

  public CreditCard(String type, String number, double limit, double annualCharge) {
    this.type = type;
    this.number = number;
    this.limit = limit;
    this.annualCharge = annualCharge;
  }
}
```

```java
public class Application {

  public static void main(String[] args) {
    // with constructor
    CreditCard creditCard = new CreditCard("visa", "1232786598763429", 2500.0, 10.0);
  }
}
```



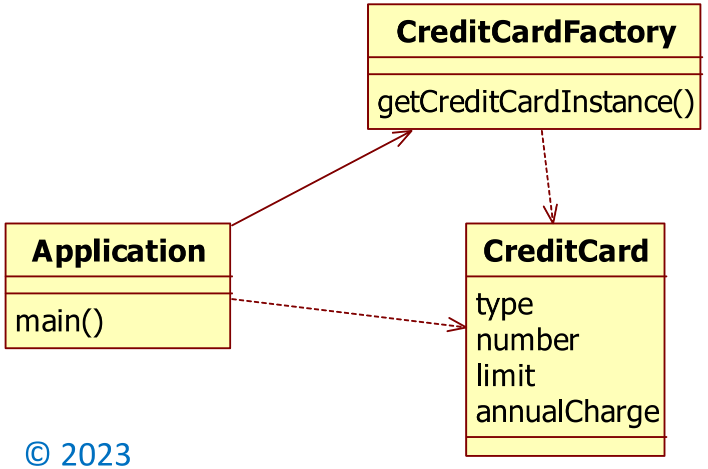
© 2023

Using a static factory method

```java
public class CreditCardFactory {
  static CreditCard getCreditCardInstance(String type, String number, double limit,
                                           double annualCharge) {
    return new CreditCard(type, number, limit, annualCharge);
  }
}
```

Static factory method

```java
public class Application {

  public static void main(String[] args) {
    //with factory
    CreditCard creditCard2 = CreditCardFactory.getCreditCardInstance("visa",
                                "1232786598763429", 2500.0, 10);
  }
}
```



What is the difference?

```
public class Application {

  public static void main(String[] args) {
    // with constructor
    CreditCard creditCard = new CreditCard("visa", "1232786598763429", 2500.0, 10);

    //with factory
    CreditCard creditCard2 = CreditCardFactory.getCreditCardInstance("visa",
                                "1232786598763429", 2500.0, 10);
  }
}
```

- In this simple case: not much
  - But when creating objects get more complex, we can encapsulate this complexity in the factory method

Constructor

```
public class RandomIntGenerator {
    private final int min;
    private final int max;

    public RandomIntGenerator(int min, int max) {
        this.min = min;
        this.max = max;
    }

    public RandomIntGenerator(int min) {
        this.min = min;
        this.max = Integer.MAX_VALUE;
    }

    public RandomIntGenerator(int max) {
        this.max = min;
        this.min = Integer.MIN_VALUE;
    }

    public int next() {...}
}
```

Constructors do not have meaningful names

Constructors cannot return anything else:

- A subclass
- A cached class

Compilation error

```
RandomIntGenerator randomIntGenerator = new RandomIntGenerator(40, 100);

RandomIntGenerator randomIntGenerator = new RandomIntGenerator(50);
```

Static factory method

```java
public class RandomIntGenerator {
    private final int min;
    private final int max;

    private RandomIntGenerator(int min, int max) {
        this.min = min;
        this.max = max;
    }

    public static RandomIntGenerator between(int max, int min) {
        return new RandomIntGenerator(min, max);
    }

    public static RandomIntGenerator biggerThan(int min) {
        return new RandomIntGenerator(min, Integer.MAX_VALUE);
    }

    public static RandomIntGenerator smallerThan(int max) {
        return new RandomIntGenerator(Integer.MIN_VALUE, max);
    }

    public int next() {...}
}
```

Private !

Factory methods can return anything:

- A subclass
- A cached class

Meaningful names

We can have multiple factory methods with the same argument(s)

```java
RandomIntGenerator randomIntGenerator = RandomIntGenerator.between(40, 100);
RandomIntGenerator randomIntGenerator = RandomIntGenerator.smallerThan(50);
RandomIntGenerator randomIntGenerator = RandomIntGenerator.biggerThan(50);
```

Prefer factory methods over constructors

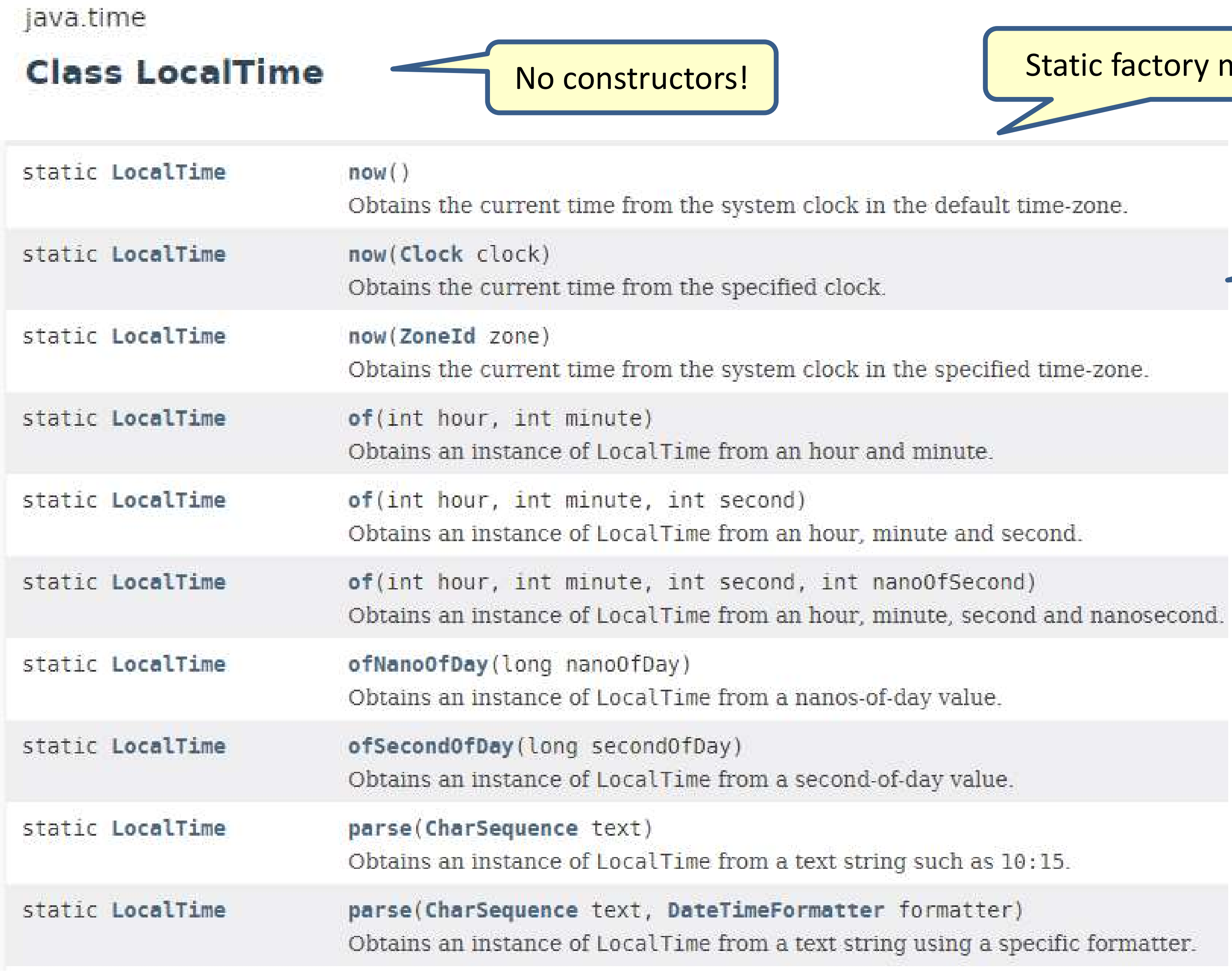
```
// with constructor
Range range = new Range( 0 , n-1);

//with factory
Range range = = RangeFactory.getUpto(n);
```

More descriptive

More flexible: Can return also subclasses of Range

Testability: Can return also MockRange which subclasses Range

Java 8 LocalTime



java.time

Class LocalTime

static **LocalTime**	**now**() Obtains the current time from the system clock in the default time-zone.
static **LocalTime**	**now**(**Clock** clock) Obtains the current time from the specified clock.
static **LocalTime**	**now**(**ZoneId** zone) Obtains the current time from the system clock in the specified time-zone.
static **LocalTime**	**of**(int hour, int minute) Obtains an instance of LocalTime from an hour and minute.
static **LocalTime**	**of**(int hour, int minute, int second) Obtains an instance of LocalTime from an hour, minute and second.
static **LocalTime**	**of**(int hour, int minute, int second, int nanoOfSecond) Obtains an instance of LocalTime from an hour, minute, second and nanosecond.
static **LocalTime**	**ofNanoOfDay**(long nanoOfDay) Obtains an instance of LocalTime from a nanos-of-day value.
static **LocalTime**	**ofSecondOfDay**(long secondOfDay) Obtains an instance of LocalTime from a second-of-day value.
static **LocalTime**	**parse**(**CharSequence** text) Obtains an instance of LocalTime from a text string such as 10:15.
static **LocalTime**	**parse**(**CharSequence** text, **DateTimeFormatter** formatter) Obtains an instance of LocalTime from a text string using a specific formatter.

No constructors!

Static factory methods

More descriptive

Logging static factory method

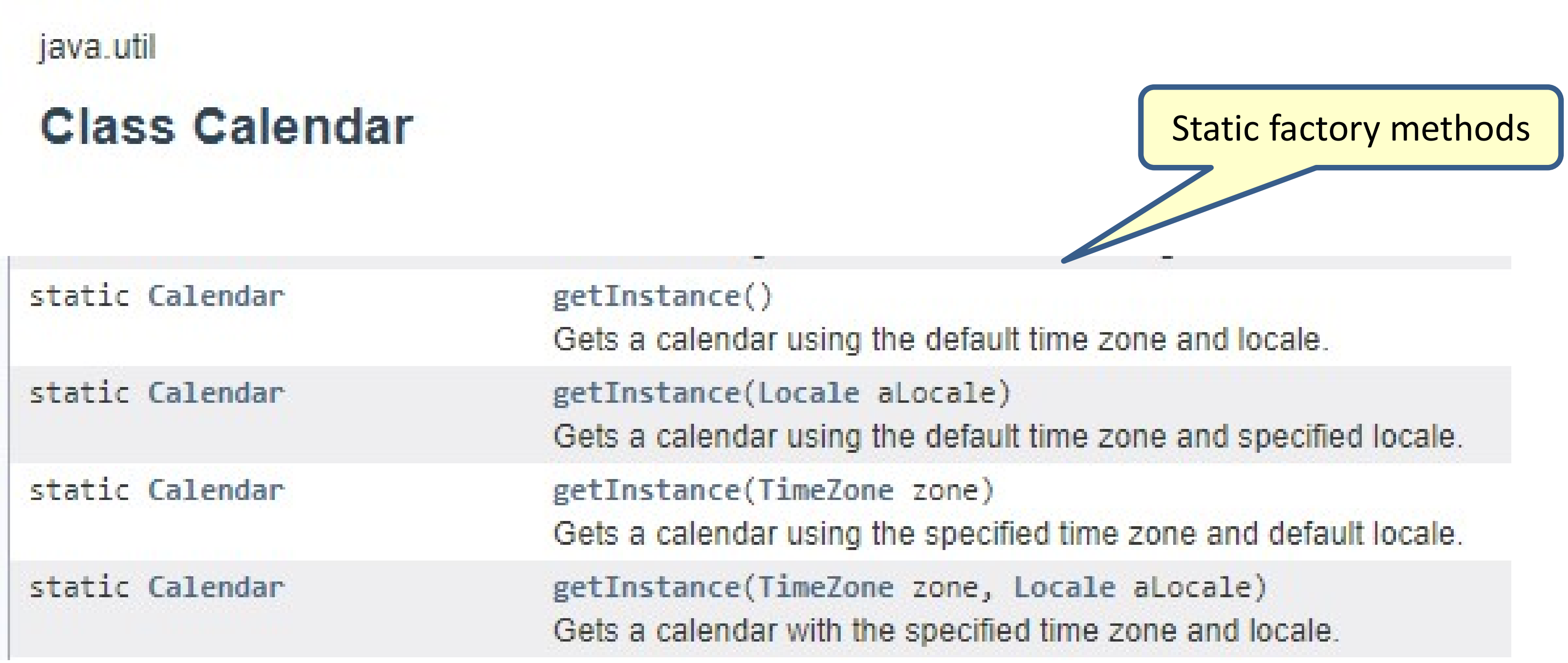
```
public class Application {
  public static void main(String[] args) {
    ProductService productService = new ProductService();
    productService.addProduct();
  }
}
```

Static factory method

```
import java.util.logging.Logger;

public class ProductService {
  static Logger logger = Logger.getLogger(ProductService.class.getName());

  public void addProduct() {
    logger.info("Add a product");
  }
}
```

```
Aug 19, 2019 12:24:26 PM test.ProductService addProduct
INFO: Add a product
```

Calendar static factory methods



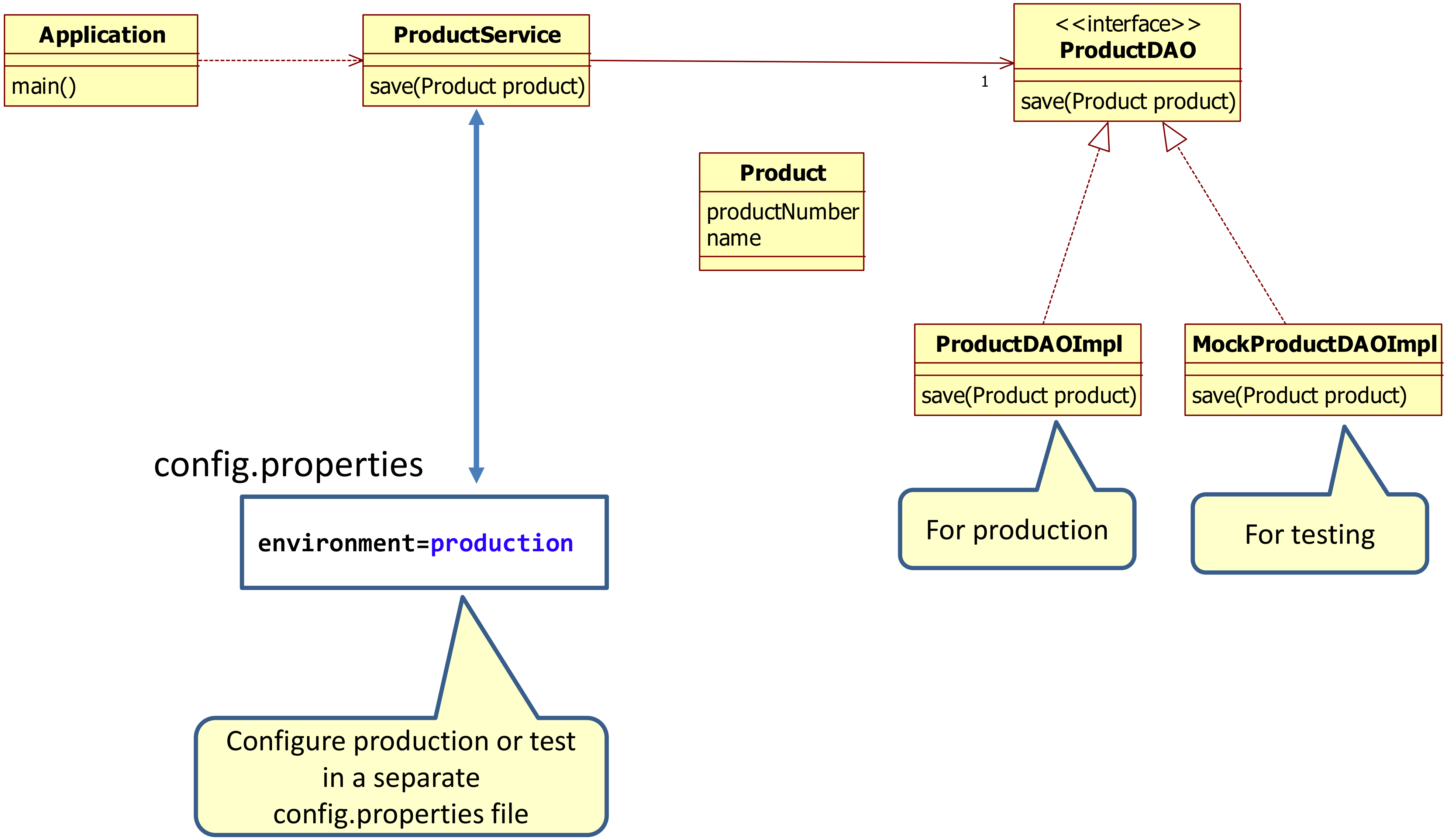
java.util

Class Calendar

`static Calendar`	`getInstance()` Gets a calendar using the default time zone and locale.
`static Calendar`	`getInstance(Locale aLocale)` Gets a calendar using the default time zone and specified locale.
`static Calendar`	`getInstance(TimeZone zone)` Gets a calendar using the specified time zone and default locale.
`static Calendar`	`getInstance(TimeZone zone, Locale aLocale)` Gets a calendar with the specified time zone and locale.

Static factory methods

Example application



config.properties

```
environment=production
```

For production

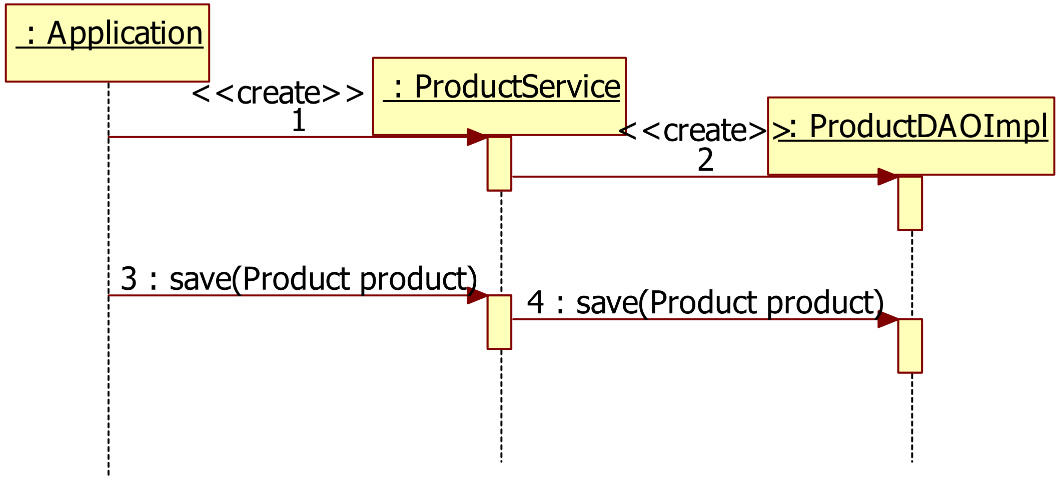
For testing

Configure production or test in a separate config.properties file

Example application

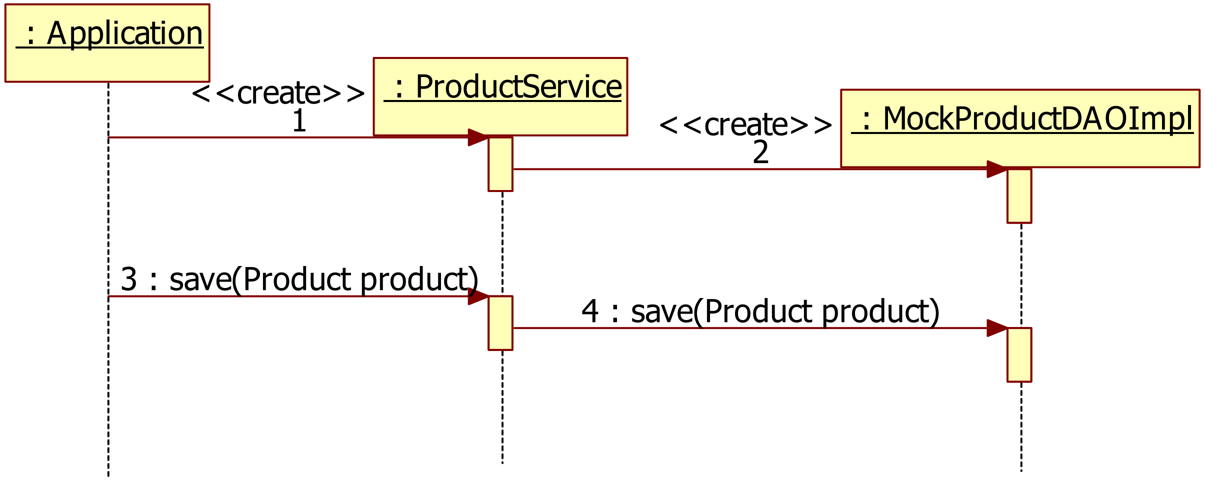
config.properties

```
environment=production
```



config.properties

```
environment=test
```



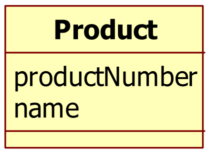
Product and DAO

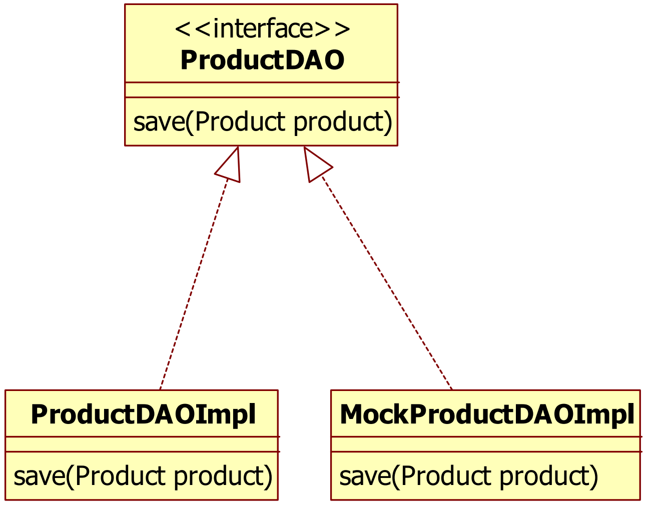
```
public interface ProductDAO {
  void save(Product product);
}
```

```
public class Product {
  private int productNumber;
  private String name;

  ....
}
```

```
public class ProductDAOImpl implements ProductDAO{

  public void save(Product product) {
    System.out.println("ProductDAOImpl saves product");
  }
}
```

```
public class MockProductDAOImpl implements ProductDAO{

  public void save(Product product) {
    System.out.println("MockProductDAOImpl saves product");

  }
}
```





Product service

```
public class ProductService {
  ProductDAO productDAO;

  public ProductService() {
    String rootPath = Thread.currentThread().getContextClassLoader().getResource("").getPath();
    try {
      Properties prop = new Properties();
      // load the properties file
      prop.load(new FileInputStream(rootPath+"/config.properties"));
      // get the property value
      String environment= prop.getProperty("environment");

      if (environment.equals("production")) {
        productDAO = new ProductDAOImpl();
      } else  if (environment.equals("test")) {
        productDAO = new MockProductDAOImpl();
      } else {
        System.out.println("environment property not set correctly");
      }
    } catch (FileNotFoundException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }

  public void save(Product product) {
    productDAO.save(product);
  }
}
```

Example application

```java
public class Application {

  public static void main(String[] args) {
    Product product = new Product(3324, "DJI Mavic 2 Pro drone");

    ProductService productService = new ProductService();
    productService.save(product);
  }
}
```

config.properties

```
environment=production
```

```
ProductDAOImpl saves product
```

config.properties

```
environment=test
```

```
MockProductDAOImpl saves product
```

What is the problem?

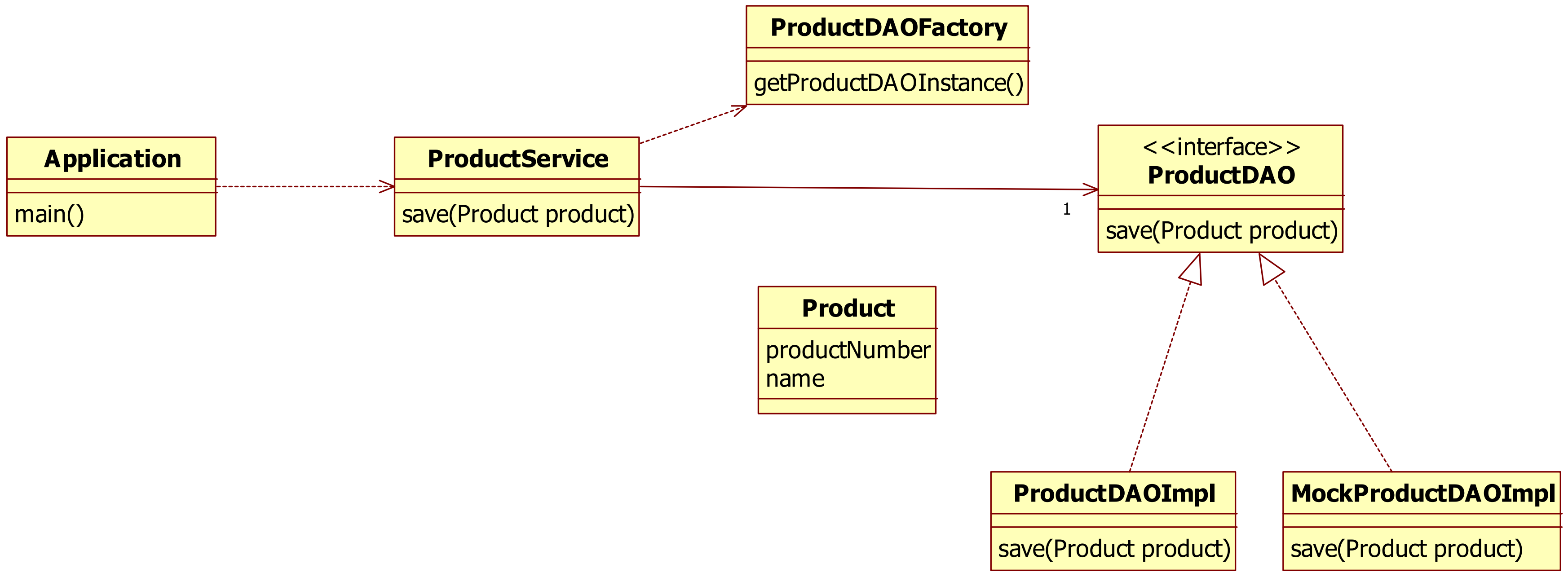
```java
public class ProductService {
  ProductDAO productDAO;

  public ProductService() {
    String rootPath = Thread.currentThread().getContextClassLoader().getResource("").getPath();
    try {
      Properties prop = new Properties();
      // load the properties file
      prop.load(new FileInputStream(rootPath+"/config.properties"));
      // get the property value
      String environment= prop.getProperty("environment");

      if (environment.equals("production")) {
        productDAO = new ProductDAOImpl();
      } else  if (environment.equals("test")) {
        productDAO = new MockProductDAOImpl();
      } else {
        System.out.println("environment property not set correctly");
      }
    } catch (FileNotFoundException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }

  public void save(Product product) {
    productDAO.save(product);
  }
}
```

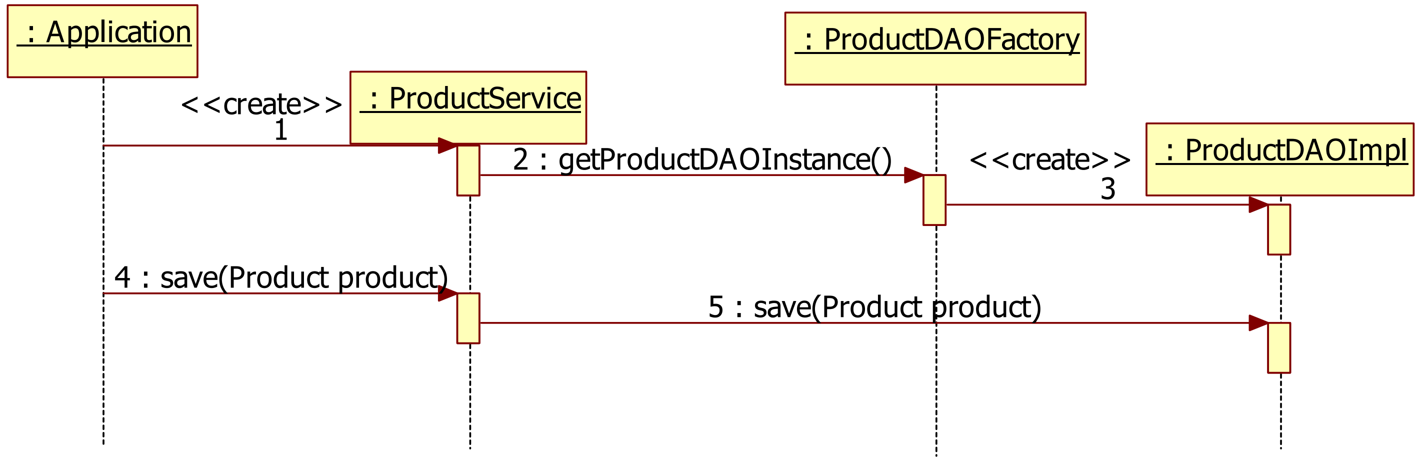
ProductService contains complex logic about creating the ProductDAO

This code has to be copied to every class that needs the ProductDAO

Every service class that needs a DAO needs to have code like this

Solution: Factory method





Solution: Factory method

```
public class ProductDAOFactory {
  static ProductDAO getProductDAOInstance() {
    String rootPath = Thread.currentThread().getContextClassLoader().getResource("").getPath();
    try {
      Properties prop = new Properties();
      // load the properties file
      prop.load(new FileInputStream(rootPath + "/config.properties"));
      // get the property value
      String environment = prop.getProperty("environment");

      if (environment.equals("production")) {
        return new ProductDAOImpl();
      } else if (environment.equals("test")) {
        return new MockProductDAOImpl();
      } else {
        System.out.println("environment property not set correctly");
      }
    } catch (FileNotFoundException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
    return null;
  }
}
```

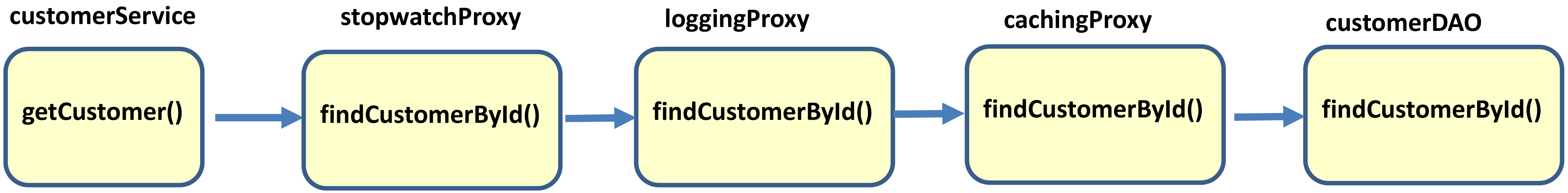
Encapsulate the logic to create objects

```
public class ProductService {
  ProductDAO productDAO;

  public ProductService() {
    productDAO=ProductDAOFactory.getProductDAOInstance();
  }

  public void save(Product product) {
    productDAO.save(product);
  }
}
```

Creating a dynamic proxy

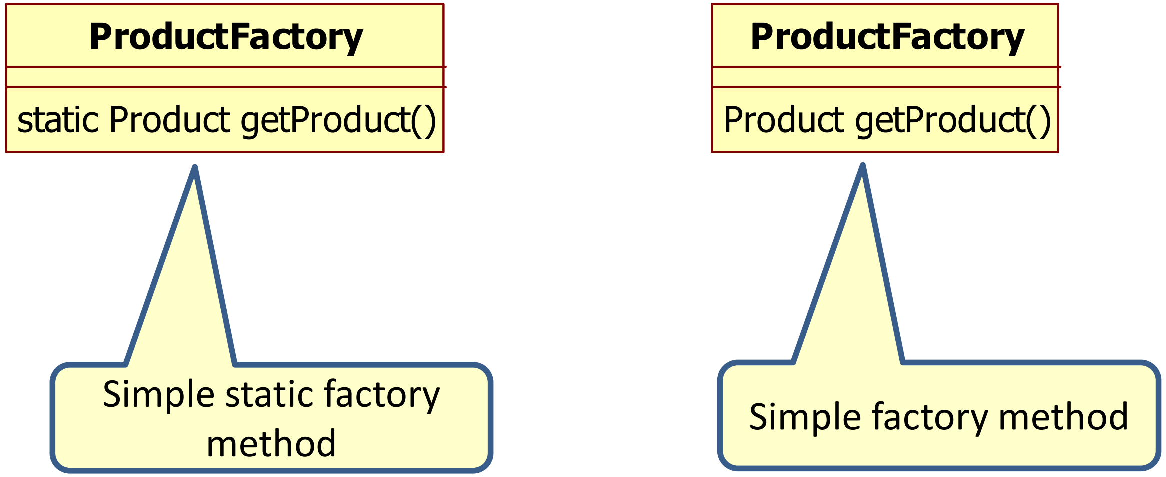
```
public class CustomerService {
  CustomerDAO customerDAO = new CustomerDAOImpl();
  ClassLoader classLoader = CustomerDAO.class.getClassLoader();
  CustomerDAO cachingProxy =
    (CustomerDAO) Proxy.newProxyInstance(classLoader,
                                         new Class[] { CustomerDAO.class },
                                         new CachingProxy(customerDAO));
  CustomerDAO loggingProxy =
    (CustomerDAO) Proxy.newProxyInstance(classLoader,
                                         new Class[] { CustomerDAO.class },
                                         new LoggingProxy(cachingProxy));
  CustomerDAO stopwatchProxy =
    (CustomerDAO) Proxy.newProxyInstance(classLoader,
                                         new Class[] { CustomerDAO.class },
                                         new StopWatchProxy(loggingProxy));

  public Customer getCustomer(int customerId) {
    return stopwatchProxy.findCustomerById(customerId);
  }
}
```

Move complex logic for creating dynamic proxies into a factory



Factory method that is not static

- Similar as static factory method, only now you instantiate the factory object, and then call the factory method.
  - Factory class needs state
    - Caching

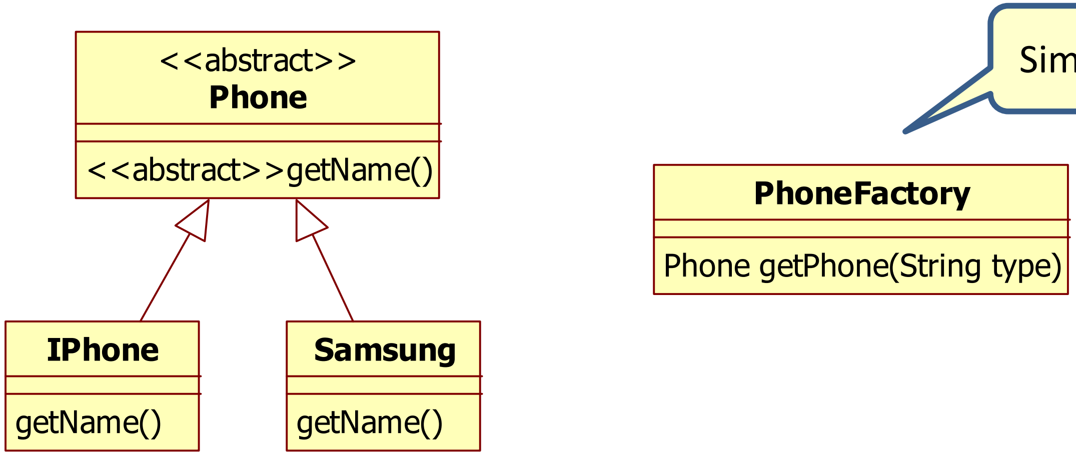


FACTORY METHOD PATTERN

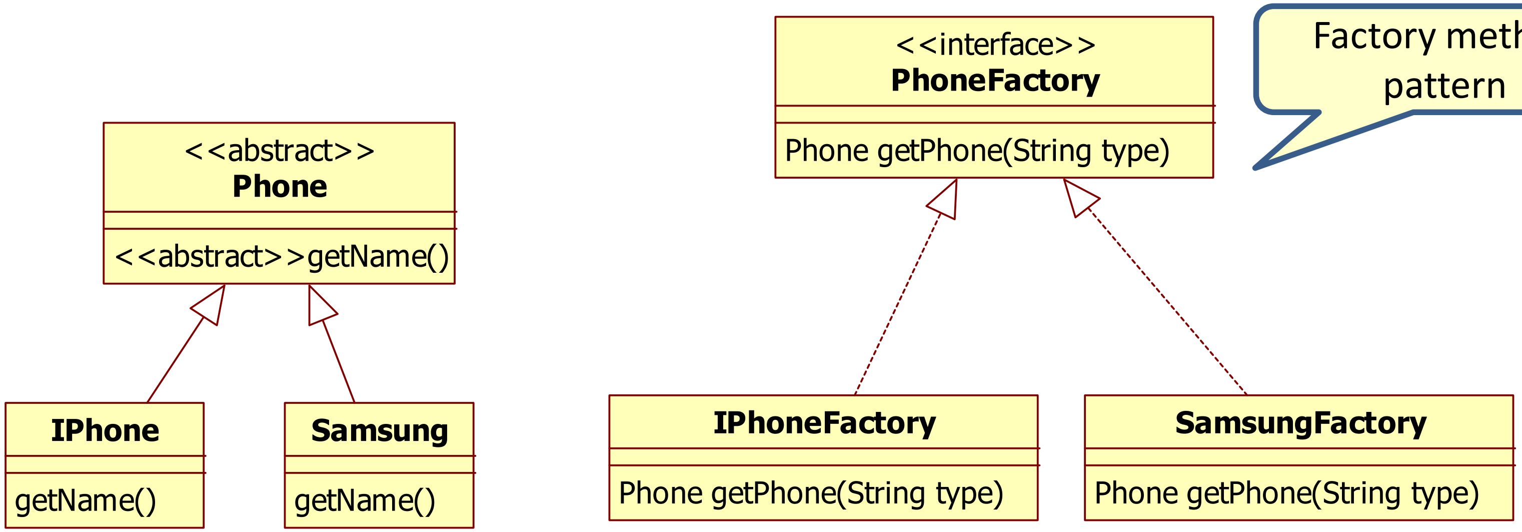
Factory method pattern

- Defines an interface for creating an object, but leaves the choice of its type to the subclasses,
- Factory method lets the class creation being deferred at run-time.
  - Polymorphic factory

Simple factory method vs. Factory method pattern



Simple factory method



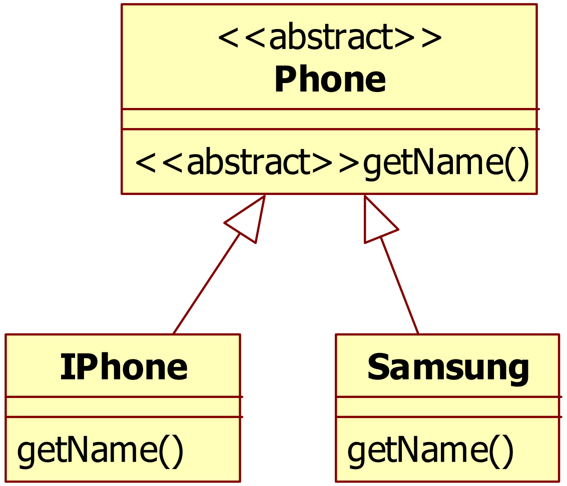
Factory method pattern

The phones

```
public abstract class Phone {
  public abstract String getName();
}
```

```
public class IPhone extends Phone{

  @Override
  public String getName() {
    return "Iphone";
  }
}
```

```
public class Samsung extends Phone{

  @Override
  public String getName() {
    return "Samsung phone";
  }
}
```

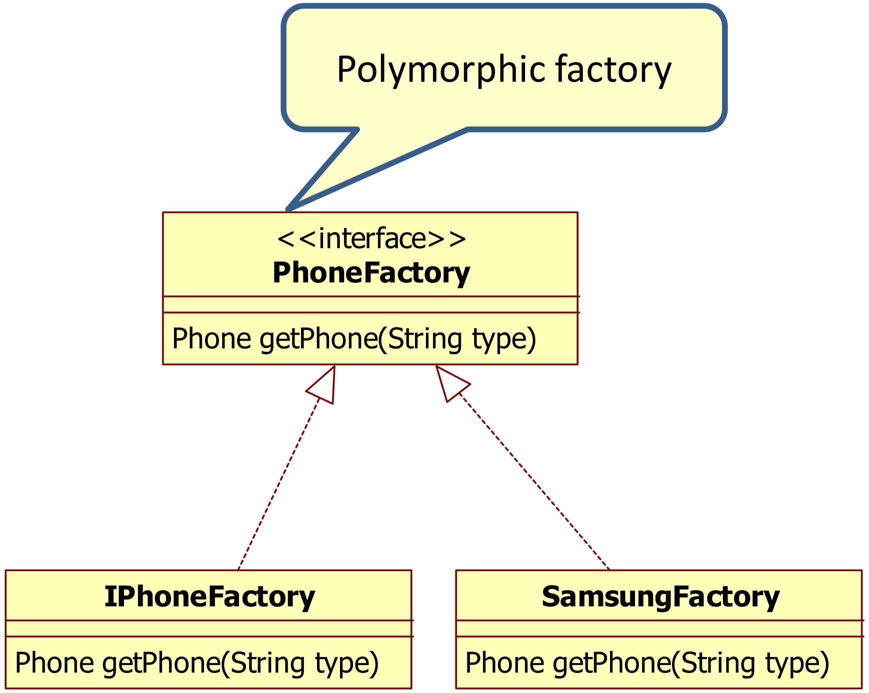


The phone factories

```java
public interface PhoneFactory {
   Phone getPhone();
}
```

```java
public class IPhoneFactory implements PhoneFactory{

   @Override
   public Phone getPhone() {
     return new IPhone();
   }
}
```

```java
public class SamsungFactory implements PhoneFactory{

   @Override
   public Phone getPhone() {
     return new Samsung();
   }
}
```



Polymorphic factory

<<interface>>
PhoneFactory
Phone getPhone(String type)
IPhoneFactory
Phone getPhone(String type)
SamsungFactory
Phone getPhone(String type)

The service and application

```
public class PhoneService {
  private PhoneFactory phoneFactory;

  public void setPhoneFactory(PhoneFactory phoneFactory) {
    this.phoneFactory = phoneFactory;
  }

  public Phone getPhone() {
    return phoneFactory.getPhone();
  }
}
```

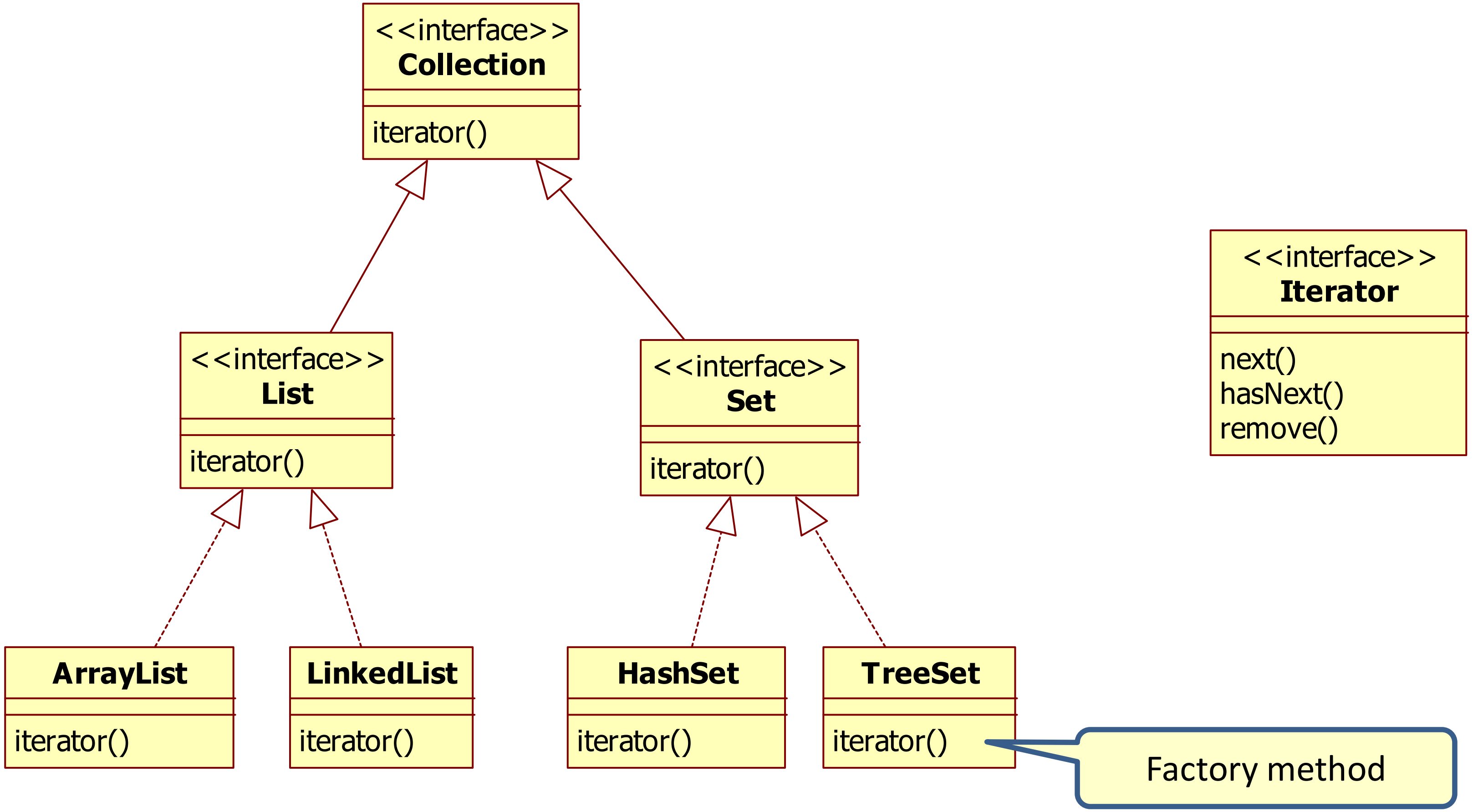
Flexibility: You can set (inject) any PhoneFactory

Testability: inject a MockPhoneFactory

```
public class Application {

  public static void main(String[] args) {
    PhoneService phoneService = new PhoneService();
    phoneService.setPhoneFactory(new IPhoneFactory());
    System.out.println(phoneService.getPhone().getName());

    phoneService.setPhoneFactory(new SamsungFactory());
    System.out.println(phoneService.getPhone().getName());
  }
}
```

```
Iphone
Samsung phone
```

iterator() factory method

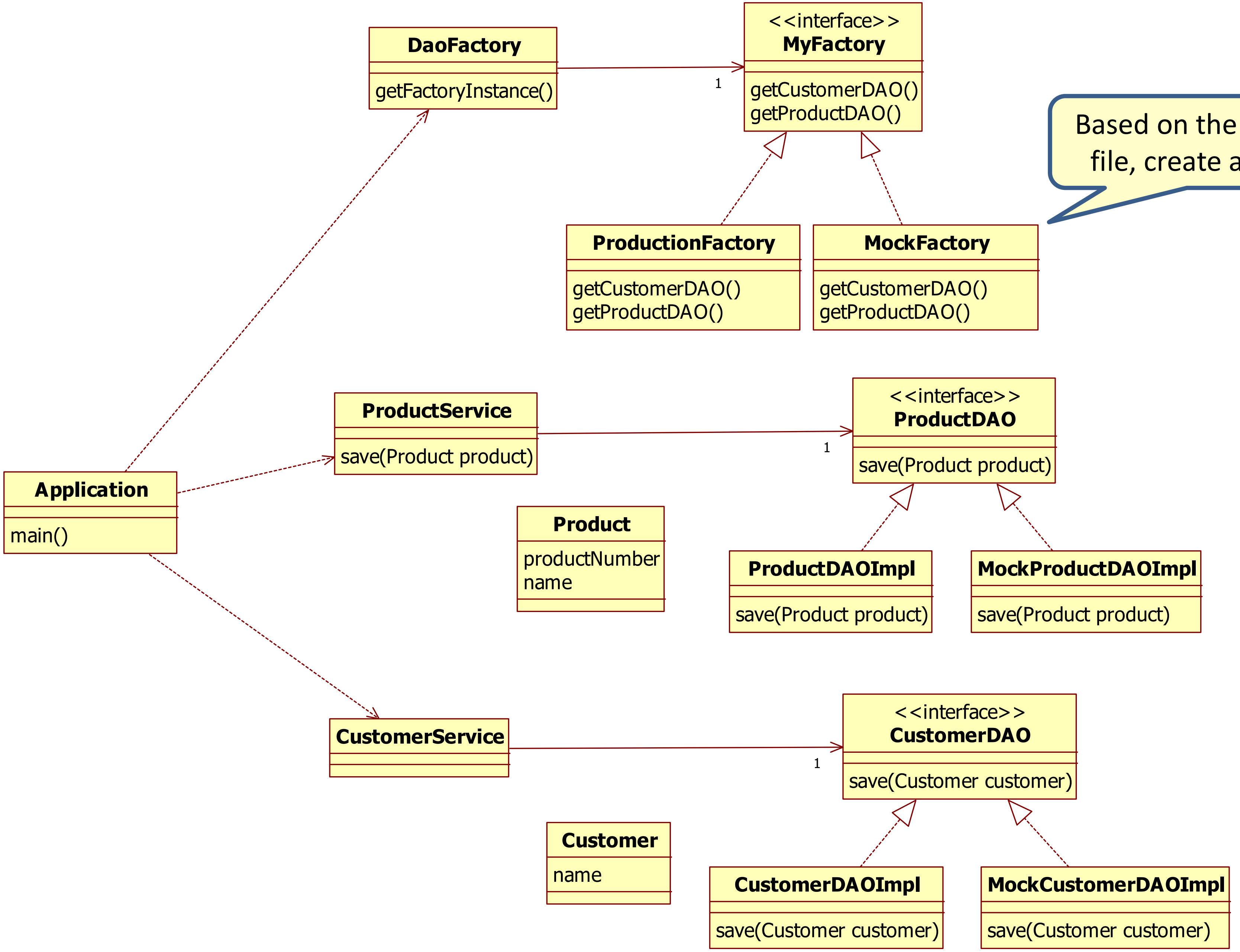


ABSTRACT FACTORY PATTERN

Abstract factory pattern

- Provides an interface for creating families of related objects without specifying their concrete classes.
  - Factory of factories

Abstract factory pattern example



Based on the property file, create a factory

Abstract factory example

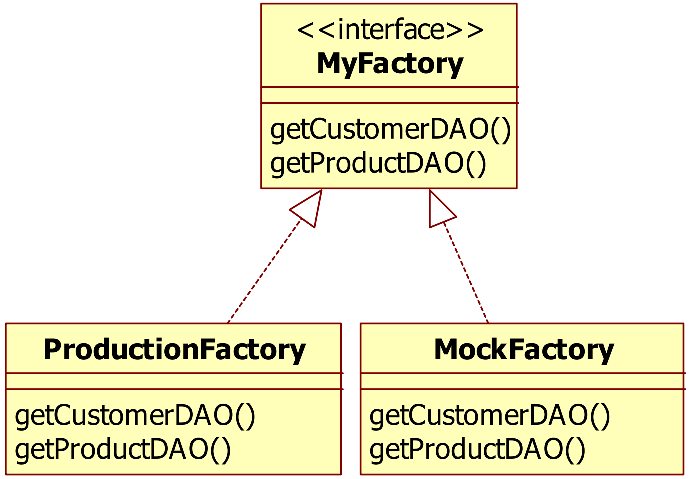
```java
public class DaoFactory {
  private MyFactory factory;

  public DaoFactory() {
    String rootPath = Thread.currentThread().getContextClassLoader().getResource("").getPath();
    try {
      Properties prop = new Properties();
      // load the properties file
      prop.load(new FileInputStream(rootPath + "/config.properties"));
      // get the property value
      String environment = prop.getProperty("environment");

      if (environment.equals("production")) {
        factory= new ProductionFactory();
      } else if (environment.equals("test")) {
        factory= new MockFactory();
      } else {
        System.out.println("environment property not set correctly");
      }
    } catch (FileNotFoundException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }

  public MyFactory getFactoryInstance() {
    return factory;
  }
}
```

Abstract factory example

```
public interface MyFactory {
  public CustomerDAO getCustomerDAO();
  public ProductDAO getProductDAO();
}
```

```
public class ProductionFactory implements MyFactory{
  public CustomerDAO getCustomerDAO() {
    return new CustomerDAOImpl();
  }

  public ProductDAO getProductDAO() {
    return new ProductDAOImpl();
  }
}
```

```
public class MockFactory implements MyFactory{
  public CustomerDAO getCustomerDAO() {
    return new MockCustomerDAOImpl();
  }

  public ProductDAO getProductDAO() {
    return new MockProductDAOImpl();
  }
}
```



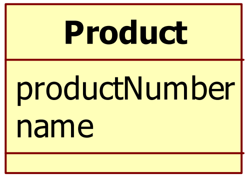
Product and DAO

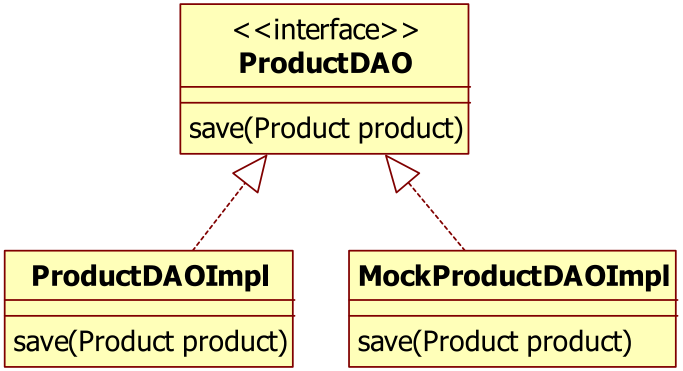
```
public interface ProductDAO {
  void save(Product product);
}
```

```
public class ProductDAOImpl implements ProductDAO{

  public void save(Product product) {
    System.out.println("ProductDAOImpl saves product");
  }
}
```

```
public class MockProductDAOImpl implements ProductDAO{

  public void save(Product product) {
    System.out.println("MockProductDAOImpl saves product");

  }
}
```

```
public class Product {
  private int productNumber;
  private String name;

  ....
}
```





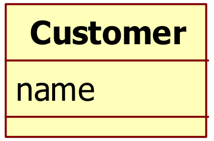
Customer and DAO

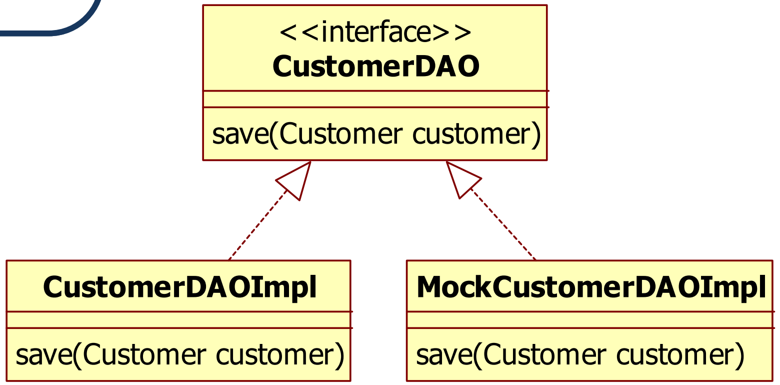
```
public interface CustomerDAO {
  void save(Customer customer);
}
```

```
public class CustomerDAOImpl implements CustomerDAO{

  public void save(Customer customer) {
    System.out.println("CustomerDAOImpl saves customer");
  }
}
```

```
public class MockCustomerDAOImpl implements CustomerDAO{

  public void save(Customer customer) {
    System.out.println("MockCustomerDAOImpl saves customer");
  }
}
```

```
public class Customer {
  private String name;

  ....
}
```





Service classes

```
public class CustomerService {
  private CustomerDAO customerDAO;

  public CustomerService(CustomerDAO customerDAO) {
    this.customerDAO= customerDAO;
  }

  public void save(Customer customer) {
    customerDAO.save(customer);
  }
}
```

```
public class ProductService {
  private ProductDAO productDAO;

  public ProductService(ProductDAO productDAO) {
    this.productDAO= productDAO;
  }

  public void save(Product product) {
    productDAO.save(product);
  }
}
```

Application

```java
public class Application {

  public static void main(String[] args) {
    Product product = new Product(3324, "DJI Mavic 2 Pro drone");
    Customer customer = new Customer("Frank Brown");

    DaoFactory mainfactory = new DaoFactory();
    MyFactory factory = mainfactory.getFactoryInstance();

    ProductDAO productDao = factory.getProductDAO();
    CustomerDAO customerDao = factory.getCustomerDAO();

    ProductService productService = new ProductService(productDao);
    productService.save(product);
    CustomerService customerService = new CustomerService(customerDao);
    customerService.save(customer);
  }
}
```

Main point

- In the factory pattern, the logic of object creation is encapsulated in the factory.

- Whatever we put our attention on will grow stronger in our life.

Builder

- Builds a complex object using a step by step approach

Immutable class

- Once created, an immutable object can never be changed

```
public class Money {
  private BigDecimal value;

  public Money(BigDecimal value) {
    this.value = value;
  }

  public Money add(Money money){
    return new Money(value.add(money.getValue()));
  }

  public Money subtract(Money money){
    return new Money(value.subtract(money.getValue()));
  }

  public BigDecimal getValue() {
    return value;
  }
}
```

No setter methods

Mutation leads to the creation of new instances

Why immutable classes?

- Reasons to make a class immutable:
  - Less prone to errors
  - Easier to share
  - Thread safe

- Immutable classes in Java
  - java.lang.String
  - java.io.File
  - java.util.Locale
  - Almost all classes in java.time

Constructor with many parameters

Constructor is not expressive

```
Customer customer = new Customer("Mary", "Jones", "0623416754",
"mjones@gmail.com", 34, 3, 8, true, 50000.0, 2000.0);
```

What do these parameters mean?

Easy to make mistakes

If you have optional parameters, you need many constructors

```
public class Customer {
  private String firstName;
  private String lastname;
  private String phone;
  private String email;
  private int age;
  private int numberOfChildren;
  private int shoesize;
  private boolean isMarried;
  private double yearlyIncome;
  private double yearlyAmountSpendOnShoes;

  public Customer(String firstName, String lastname, String phone, String email, int age, int
    numberOfChildren, int shoesize, boolean isMarried, double yearlyIncome, double
    yearlyAmountSpendOnShoes) {
    this.firstName = firstName;
    this.lastname = lastname;
    this.phone = phone;
    this.email = email;
    this.age = age;
    this.numberOfChildren = numberOfChildren;
    this.shoesize = shoesize;
    this.isMarried = isMarried;
    this.yearlyIncome = yearlyIncome;
    this.yearlyAmountSpendOnShoes = yearlyAmountSpendOnShoes;
  }
}
```

Class can be immutable

Using setters

```
public class ApplicationUsingSetters {
  public static void main(String[] args) {
    Customer customer = new Customer();
    customer.setFirstName("Mary");
    customer.setLastname("Jones");
    customer.setPhone("0623416754");
    customer.setEmail("mjones@gmail.com");
    customer.setAge(34);
    customer.setNumberOfChildren(3);
    customer.setShoesize(8);
    customer.setMarried(true);
    customer.setYearlyIncome(50000.0);
    customer.setYearlyAmountSpendOnShoes(2000.0);
    System.out.println(customer);
  }
}
```

Clear what the parameters mean

Class is not immutable

What if we want

- Expressive code
- Immutable class

- Solution: Builder

Builder example

```
public class Customer {
  private String firstName;
  private String lastname;
  private String phone;
  private String email;
  private int age;
  private int numberOfChildren;
  private int shoesize;
  private boolean isMarried;
  private double yearlyIncome;
  private double yearlyAmountSpendOnShoes;

  public static class Builder {

    private String firstName="";
    private String lastname="";
    private String phone="";
    private String email="";
    private int age = 0;
    private int numberOfChildren = 0;
    private int shoesize = 0;
    private boolean isMarried = false;
    private double yearlyIncome = 0.0;
    private double yearlyAmountSpendOnShoes = 0.0;

    public Builder withFirstName(String firstName) {
      this.firstName = firstName;
      return this;
    }
```

Builder inner class

‘Setter’ method on the builder

Return ‘this’ for method chaining

Builder example

```java
public Builder withLastname(String lastname) {
  this.lastname = lastname;
  return this;
}
public Builder withPhone(String phone) {
  this.phone = phone;
  return this;
}
public Builder withEmail(String email) {
  this.email = email;
  return this;
}
public Builder withAge(int age) {
  this.age = age;
  return this;
}
public Builder withNumberOfChildren(int numberOfChildren) {
  this.numberOfChildren = numberOfChildren;
  return this;
}
public Builder withShoesize(int shoesize) {
  this.shoesize = shoesize;
  return this;
}
public Builder isMarried() {
  this.isMarried = true;
  return this;
}
```

Builder example

```
  public Builder isNotMarried() {
    this.isMarried = false;
    return this;
  }
  public Builder withYearlyIncome(double yearlyIncome) {
    this.yearlyIncome = yearlyIncome;
    return this;
  }
  public Builder withYearlyAmountSpendOnShoes(double yearlyAmountSpendOnShoes) {
    this.yearlyAmountSpendOnShoes = yearlyAmountSpendOnShoes;
    return this;
  }

  public Customer build() {
    return new Customer(this);
  }
}
```

The build() method does the actual creation of the object

Builder example

```
 private Customer(Builder builder) {
   this.firstName = builder.firstName;
   this.lastname = builder.lastname;
   this.phone = builder.phone;
   this.email = builder.email;
   this.age = builder.age;
   this.numberOfChildren = builder.numberOfChildren;
   this.shoesize = builder.shoesize;
   this.isMarried = builder.isMarried;
   this.yearlyIncome = builder.yearlyIncome;
   this.yearlyAmountSpendOnShoes = builder.yearlyAmountSpendOnShoes;
 }

 @Override
 public String toString() {
   return "Customer [firstName=" + firstName + ", lastname=" + lastname + ", phone=" + phone + ",
       email=" + email + ", age=" + age + ", numberOfChildren=" + numberOfChildren + ", shoesize="
       + shoesize + ", isMarried="+ isMarried + ", yearlyIncome=" + yearlyIncome + ",
       yearlyAmountSpendOnShoes=“ + yearlyAmountSpendOnShoes + "]";
 }

}
```

The constructor has a Builder as argument

The client code

```java
public class Application {

  public static void main(String[] args) {
    Customer customer1 = new Customer.Builder()
      .withFirstName("Mary")
      .withLastname("Jones")
      .withEmail("mjones@gmail.com")
      .withAge(34)
      .isMarried()
      .withNumberOfChildren(3)
      .withPhone("0623416754")
      .withShoesize(8)
      .withYearlyIncome(50000.0)
      .withYearlyAmountSpendOnShoes(2000.0)
      .build();
    System.out.println(customer1);

    Customer customer2 = new Customer.Builder()
      .withFirstName("Lucy")
      .withLastname("Jhonson")
      .isNotMarried()
      .withPhone("0698345234")
      .build();
    System.out.println(customer2);
  }
}
```

Clear code

Customer is immutable

Builder used in Quartz

Quartz 1.0

```
SchedulerFactory schedFact = new StdSchedulerFactory();
Scheduler sched = schedFact.getScheduler();
sched.start();
// define the job and tie it to our HelloJob class
JobDetail job = JobDetail("myJob", "group1", HelloJob.class);

// Trigger the job to run now, and then every 40 seconds
Trigger trigger = newTrigger(("myTrigger", "group1", new Date(), null,
       SimpleTrigger.REPEAT_INDEFINITELY, 40)

// Tell quartz to schedule the job using our trigger
sched.scheduleJob(job, trigger);
```

Quartz 2.0

```
SchedulerFactory schedFact = new StdSchedulerFactory();
Scheduler sched = schedFact.getScheduler();
sched.start();
// define the job and tie it to our HelloJob class
JobDetail job = newJob(HelloJob.class)
    .withIdentity("myJob", "group1")
    .build();
// Trigger the job to run now, and then every 40 seconds
Trigger trigger = newTrigger()
    .withIdentity("myTrigger", "group1")
    .startNow()
    .withSchedule(simpleSchedule()
        .withIntervalInSeconds(40)
        .repeatForever())
    .build();
// Tell quartz to schedule the job using our trigger
sched.scheduleJob(job, trigger);
```

Main point

- The builder pattern is a great help if you want to create objects with many different parameters.

- All the intelligence of Nature is available at the level of the Unified Field

Singleton

- A singleton class can have only one instance.

- The office of the President of the United States is a *Singleton*. The United States Constitution specifies that there can be at most one active president at any given time.

Examples of singleton classes

- ConnectionPool
- PrinterBuffer
- Cache
- Configuration from configuration file

Classes that contain shared resources

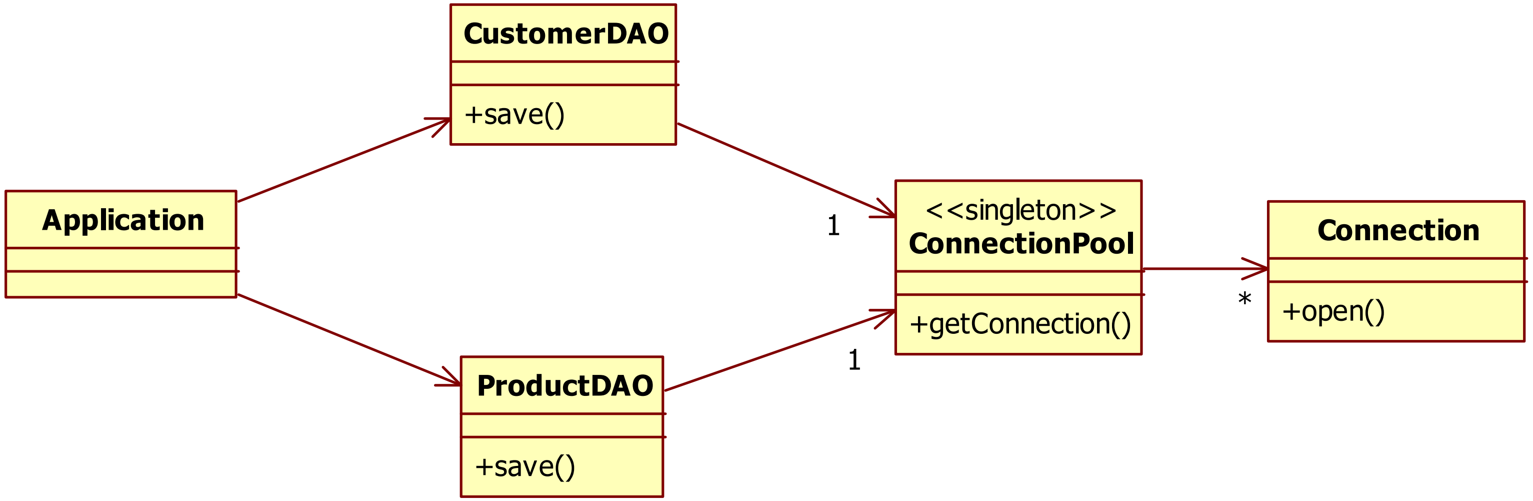
Singleton example

```
public class ConnectionPool {
  private static ConnectionPool pool = new ConnectionPool();
  //this is a pool with only 1 connection
  private Connection connection = new Connection();

  private ConnectionPool() {}

  public static ConnectionPool getPool() {
    return pool;
  }
  public Connection getConnection(){
    return connection;
  }
}
```

Declare a private static instance of the class

Make the constructor private

Add a static method to get an instance of the singleton class



The application

```
public class CustomerDAO {
  Connection conn;

  public CustomerDAO() {
    conn = ConnectionPool.getPool().getConnection();
  }
  public void save(){
    conn.open();
  }
}
```

```
public class Connection {
  public void open(){
    System.out.println("open connection to DB");
  }
}
```

```
public class ProductDAO {
  Connection conn;

  public ProductDAO() {
    conn = ConnectionPool.getPool().getConnection();
  }
  public void save(){
    conn.open();
  }
}
```

```
public class Application {

  public static void main(String[] args) {
    CustomerDAO cdao = new CustomerDAO();
    cdao.save();
    ProductDAO pdao = new ProductDAO();
    pdao.save();
  }
}
```

Eager and lazy instantiation

```java
public class ConnectionPool {
  private static ConnectionPool pool = new ConnectionPool();
  //this is a pool with only 1 connection
  private Connection connection = new Connection();

  private ConnectionPool() {}

  public static ConnectionPool getPool() {
    return pool;
  }
  public Connection getConnection(){
    return connection;
  }
}
```

Eager instantiation

```java
public class ConnectionPool {
  private static ConnectionPool pool;
  // this is a pool with only 1 connection
  private Connection connection = new Connection();

  private ConnectionPool() {}

  public static ConnectionPool getPool() {
    if (pool == null) {
      pool = new ConnectionPool();
    }
    return pool;
  }

  public Connection getConnection() {
    return connection;
  }
}
```

Lazy instantiation

Issues with singleton

- With reflection you can still create more instances of the singleton
  - Make the singleton reflection safe
- If 2 threads create a singleton at almost the same time, you might end up with 2 instances
  - Make the singleton thread safe
- With serialization and deserialization we might end up with 2 instances
  - Make the singleton serialization safe

Reflection

```
public class ReflectionSingletonTest {

  public static void main(String[] args) {
    ConnectionPool instanceOne = ConnectionPool.getPool();
    ConnectionPool instanceTwo = null;
    try {
       Constructor[] constructors = ConnectionPool.class.getDeclaredConstructors();
       for (Constructor constructor : constructors) {
         //Below code will break the singleton pattern
         constructor.setAccessible(true);
         instanceTwo = (ConnectionPool) constructor.newInstance();
         break;
       }
    } catch (Exception e) {
            e.printStackTrace();
    }
    System.out.println(instanceOne.getClass().getName()+" with hascode: " + instanceOne.hashCode());
    System.out.println(instanceTwo.getClass().getName()+" with hascode: " + instanceTwo.hashCode());
  }
}
```

Two instances of the ConnectionPool

```
reflection.ConnectionPool with hascode: 366712642
reflection.ConnectionPool with hascode: 1829164700
```

Make the singleton reflection safe

```java
public class ConnectionPool {
  private static ConnectionPool pool;
  // this is a pool with only 1 connection
  private Connection connection = new Connection();

  private ConnectionPool() {
    // Prevent form the reflection api.
    if (pool != null) {
      throw new RuntimeException("Use getInstance() method to get the single instance of this
                                  class.");
    }
  }

  public static synchronized ConnectionPool getPool() {
    if (pool == null) {
      pool = new ConnectionPool();
    }
    return pool;
  }

  public Connection getConnection() {
    return connection;
  }
}
```

Throw exception if constructor is called twice

```
java.lang.reflect.InvocationTargetException
at sun.reflect.NativeConstructorAccessorImpl.newInstance0(Native Method)
... 5 more
Exception in thread "main" java.lang.NullPointerException
reflection.safe.ConnectionPool with hascode: 2018699554at
reflection.safe.ReflectionSingletonTest.main(ReflectionSingletonTest.java:22)
```

Thread safety

```
public class SingletonTest {
 public static void main(String[] args) {
    //Thread 1
    Thread t1 = new Thread(new Runnable() {
      @Override
      public void run() {
      ConnectionPool instance1 = ConnectionPool.getPool();
        System.out.println("Instance 1 hash:" + instance1.hashCode());
      }
    });

    //Thread 2
    Thread t2 = new Thread(new Runnable() {
      @Override
      public void run() {
      ConnectionPool instance2 = ConnectionPool.getPool();
        System.out.println("Instance 2 hash:" + instance2.hashCode());
      }
    });

    //start both the threads
    t1.start();
    t2.start();
  }
}
```

```
Instance 1 hash:1870487130
Instance 2 hash:354710606
```

Two instances of the ConnectionPool

Thread safety solution 1

```java
public class ConnectionPool {
  private static ConnectionPool pool;
  // this is a pool with only 1 connection
  private Connection connection = new Connection();

  private ConnectionPool() {
    // Prevent form the reflection api.
    if (pool != null) {
      throw new RuntimeException("Use getInstance() method to get the single instance
                                 of this class.");
    }
  }

  public static synchronized ConnectionPool getPool() {
    if (pool == null) {
      pool = new ConnectionPool();
    }
    return pool;
  }

  public Connection getConnection() {
    return connection;
  }
}
```

synchronized

Performance problem

```
Instance 2 hash:892687863
Instance 1 hash:892687863
```

Thread safety solution 2

```
public class ConnectionPool {
  private static ConnectionPool pool;
  // this is a pool with only 1 connection
  private Connection connection = new Connection();

  private ConnectionPool() {
    // Prevent form the reflection api.
    if (pool != null) {
      throw new RuntimeException("Use getInstance() method to get the single instance
                                 of this class.");
    }
  }

  public static ConnectionPool getPool() {
    // Double check locking pattern
    if (pool == null) { // Check for the first time
      synchronized (ConnectionPool.class) { // Check for the second time.
        if (pool == null)  pool = new ConnectionPool();
      }
    }
    return pool;
  }

  public Connection getConnection() {
    return connection;
  }
```

Only use synchronized if the pool is not created yet

```
Instance 2 hash:892687863
Instance 1 hash:892687863
```

Serialization

```java
public class SingletonTest {
  public static void main(String[] args) {
    try {
      ConnectionPool instance1 = ConnectionPool.getPool();
      ObjectOutput out = new ObjectOutputStream(new FileOutputStream("filename.ser"));
      out.writeObject(instance1);
      out.close();

      // deserialize from file to object
      ObjectInput in = new ObjectInputStream(new FileInputStream("filename.ser"));
      ConnectionPool instance2 = (ConnectionPool) in.readObject();
      in.close();

      System.out.println("instance1 hashCode=" + instance1.hashCode());
      System.out.println("instance2 hashCode=" + instance2.hashCode());

    } catch (IOException | ClassNotFoundException e) {
      e.printStackTrace();
    }
  }
}
```

```java
public class ConnectionPool implements Serializable{
```

```java
public class Connection implements Serializable{
```

```
instance1 hashCode=865113938
instance2 hashCode=1831932724
```

Two instances of
the ConnectionPool

Serialization solution

```
public class ConnectionPool {
  private static ConnectionPool pool;
  private Connection connection = new Connection();

  private ConnectionPool() {
    // Prevent form the reflection api.
    if (pool != null) {
      throw new RuntimeException("Use getInstance() method to get the single instance
                                  of this class.");
    }
  }

  public static ConnectionPool getPool() {
    // Double check locking pattern
    if (pool == null) { // Check for the first time
      synchronized (ConnectionPool.class) { // Check for the second time.
        if (pool == null)  pool = new ConnectionPool();
      }
    }
    return pool;
  }

  public Connection getConnection() {
    return connection;
  }

  // This method is called immediately after an object of this class is deserialized.
  protected Object readResolve() {
    return getPool();
  }
}
```

Implement the readResolve() method

```
instance1 hashCode=865113938
instance2 hashCode=865113938
```

Main point

- A singleton is a class that can have only one instance.

- Pure consciousness is the unified basis of all of creation.

Connecting the parts of knowledge with the wholeness of knowledge

1. The decorator class decorates a target class.
2. The factory pattern, builder pattern and singleton pattern are patterns that help in constructing objects

3. **Transcendental consciousness** is the field of all knowledge.
4. **Wholeness moving within itself:** In unity consciousness, one appreciates the inherent underlying unity that underlies all the diversity of creation.