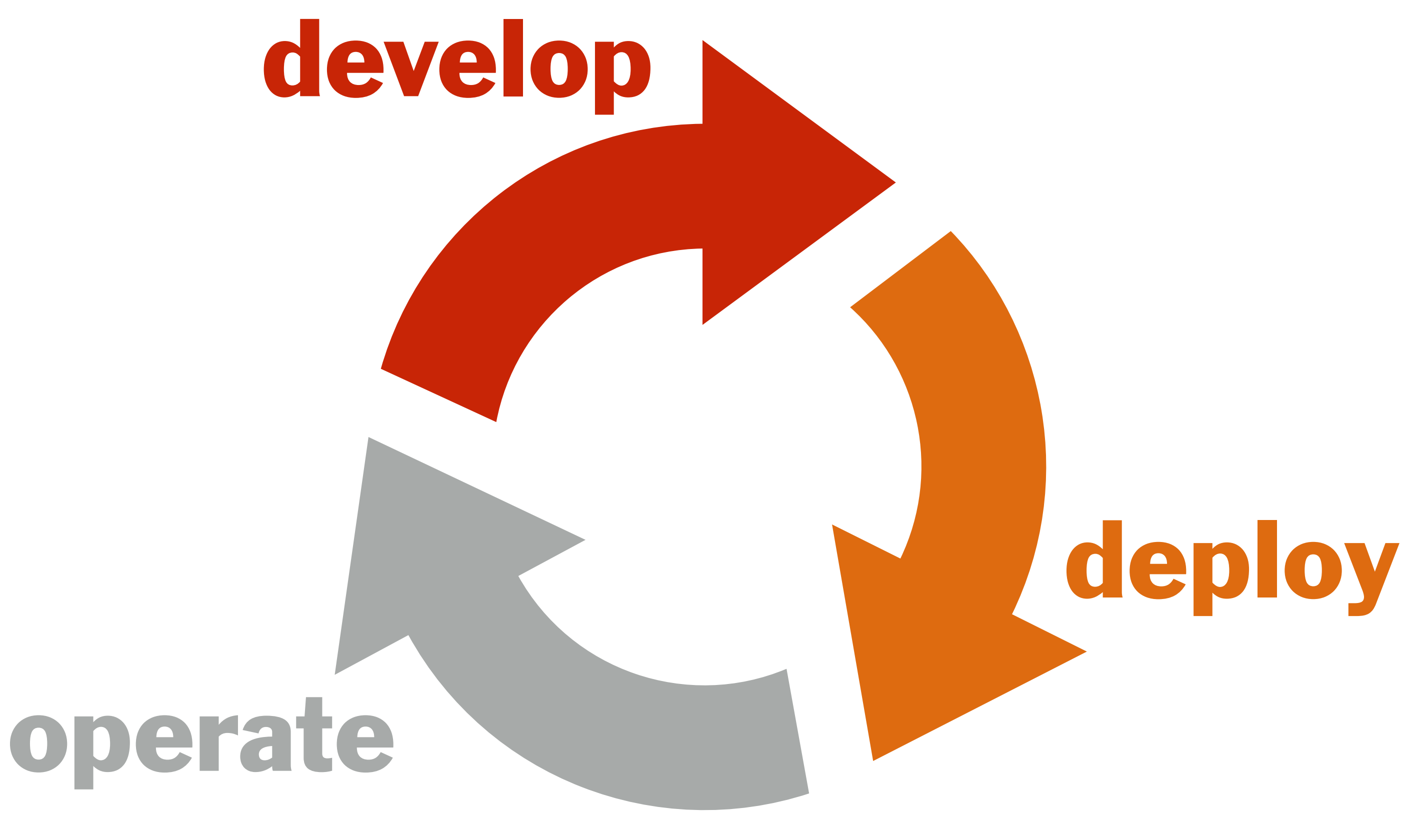
modularity

&

separation of concerns

learning objectives



- learn about modularity and unit testing
- learn about separation of concerns
- learn about code annotations

abstraction & modularization

abstraction is the ability to ignore details of parts to focus attention on a higher level of a problem

modularization consists in dividing a complex object into elemental objects that can be developed independently

once elemental objects have been developed and tested, they can be assembled into a more complex object

this is known as code reuse

but

to be reusable, a module* needs to be reliable

and

to be reliable, a module must be thoroughly tested

*usually a class

unit testing tools

- unit testing consist in writing a set of independent tests for each individual module (unit)
- unit testing frameworks make it easy to write clear and systematic tests and to automate test execution

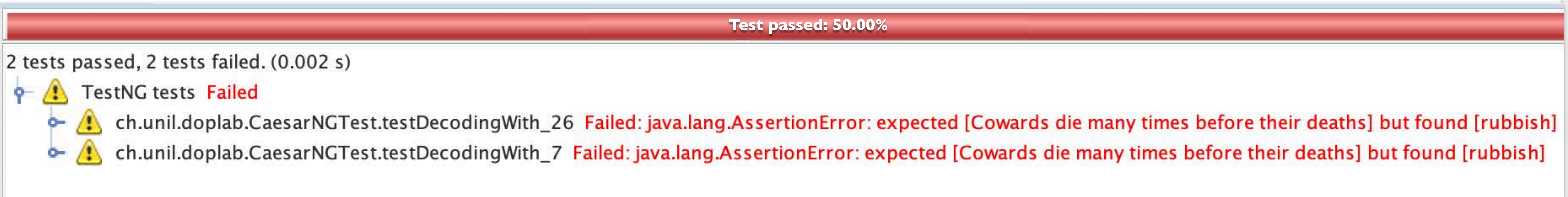
JUnit 5 TestNG pytest unittest

- test coverage is the ratio of coverage items* being tested

$$\text{coverage} = \frac{\text{number of tested code items}}{\text{total number of code items}} \times 100\,\%$$

- unit tests can be seen as the specification of the item to be tested
- unit tests are often developed before the actual item is implemented
- unit tests act as a safety net whenever refactoring the code

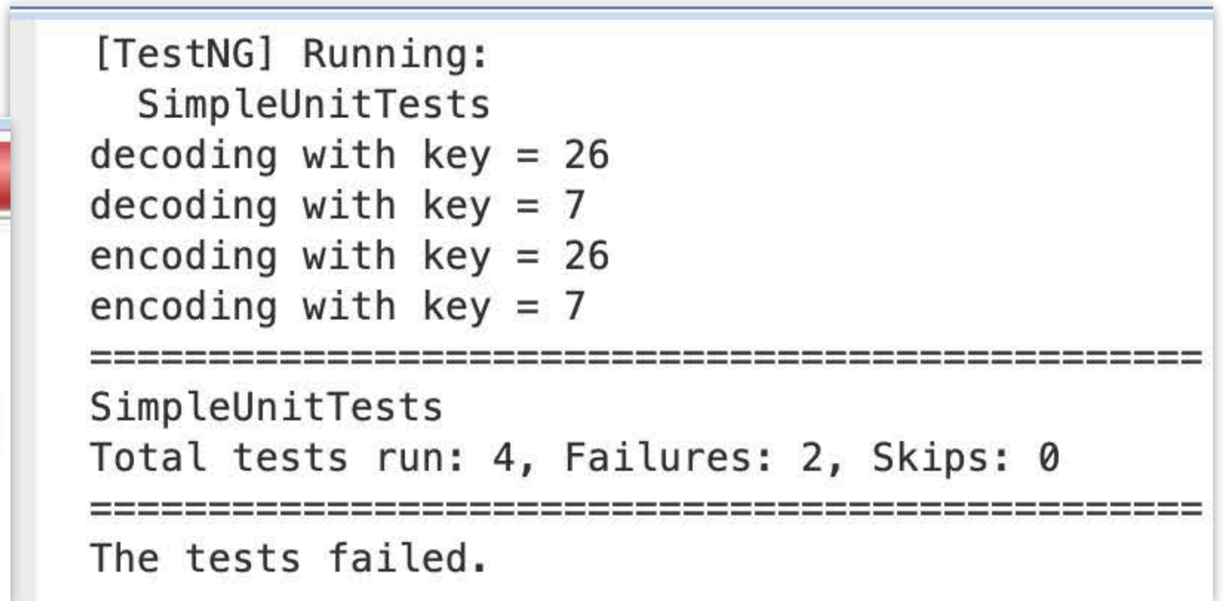
* whatever countable and identifiable code element that can be tested, e.g., a method, a function or a class



Test passed: 50.00%

2 tests passed, 2 tests failed. (0.002 s)

```
TestNG tests Failed
    ch.unil.doplab.CaesarNGTest.testDecodingWith_26 Failed: java.lang.AssertionError: expected [Cowards die many times before their deaths] but found [rubbish]
    ch.unil.doplab.CaesarNGTest.testDecodingWith_7 Failed: java.lang.AssertionError: expected [Cowards die many times before their deaths] but found [rubbish]
```



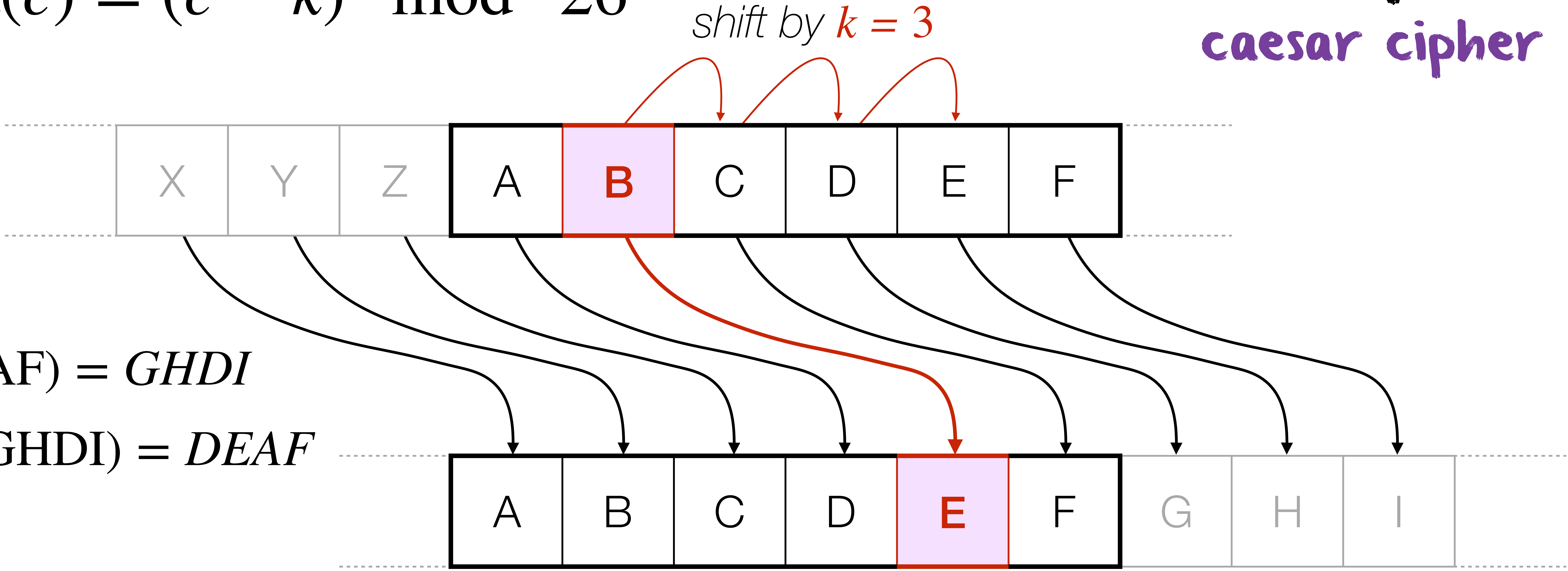
```
[TestNG] Running:
  SimpleUnitTests
decoding with key = 26
decoding with key = 7
encoding with key = 26
encoding with key = 7
===============================================
SimpleUnitTests
Total tests run: 4, Failures: 2, Skips: 0
===============================================
The tests failed.
```

unit testing example

caesar cipher

$$E_k(c) = (c + k) \mod 26$$

$$D_k(c) = (c - k) \mod 26$$



$$E_3(\text{DEAF}) = \mathit{GHDI}$$

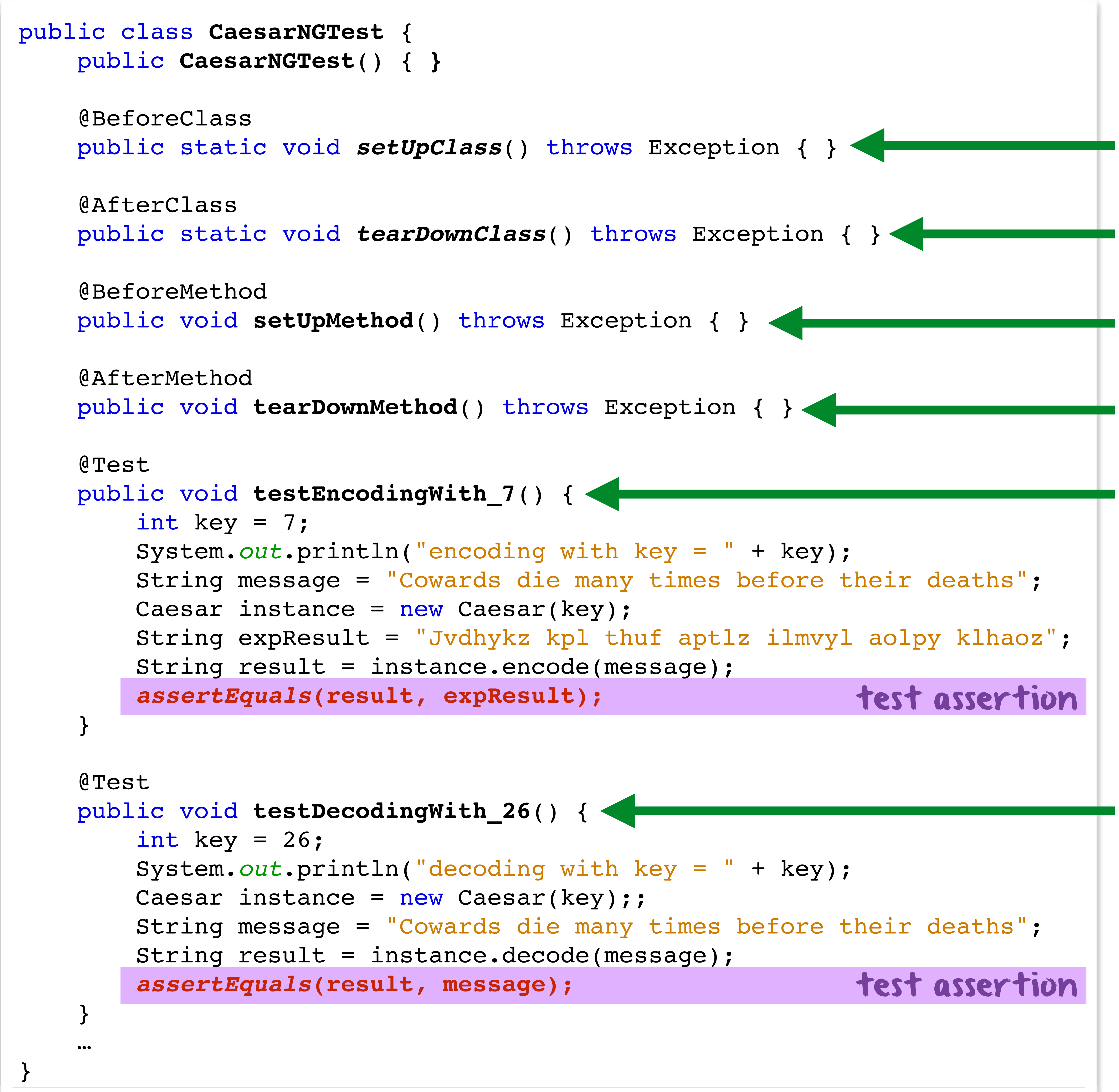
$$D_3(\text{GHDI}) = \mathit{DEAF}$$

$$E_3(\text{DEAF}) = E_{29}(\text{DEAF}) \qquad D_3(\text{GHDI}) = D_{29}(\text{GHDI})$$

$$E_{26}(\text{DEAF}) = E_0(\text{DEAF}) = D_{26}(\text{DEAF}) = D_0(\text{DEAF}) = \mathit{DEAF}$$

unit testing example

caesar cipher



```
public class CaesarNGTest {
    public CaesarNGTest() { }

    @BeforeClass
    public static void setUpClass() throws Exception { }

    @AfterClass
    public static void tearDownClass() throws Exception { }

    @BeforeMethod
    public void setUpMethod() throws Exception { }

    @AfterMethod
    public void tearDownMethod() throws Exception { }

    @Test
    public void testEncodingWith_7() {
        int key = 7;
        System.out.println("encoding with key = " + key);
        String message = "Cowards die many times before their deaths";
        Caesar instance = new Caesar(key);
        String expResult = "Jvdhykz kpl thuf aptlz ilmvyl aolpy klhaoz";
        String result = instance.encode(message);
        assertEquals(result, expResult);
    }

    @Test
    public void testDecodingWith_26() {
        int key = 26;
        System.out.println("decoding with key = " + key);
        Caesar instance = new Caesar(key);;
        String message = "Cowards die many times before their deaths";
        String result = instance.decode(message);
        assertEquals(result, message);
    }
    …
}
```

test assertion

test assertion

executed before the class is tested

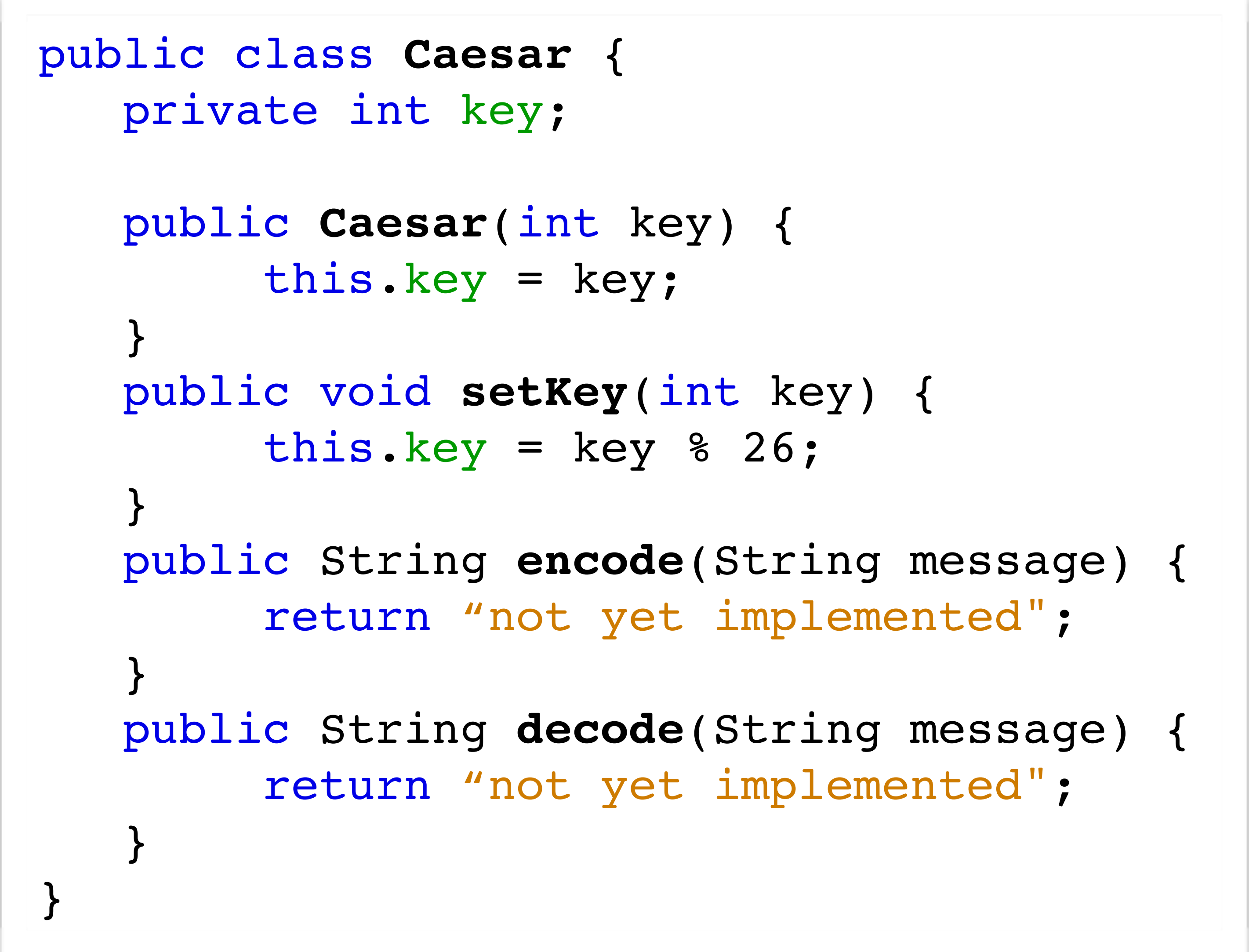
executed after the class was tested

executed before each test method is executed

executed after each test method is executed

unit test of a method

unit test of a method



```
public class Caesar {
    private int key;

    public Caesar(int key) {
        this.key = key;
    }
    public void setKey(int key) {
        this.key = key % 26;
    }
    public String encode(String message) {
        return “not yet implemented";
    }
    public String decode(String message) {
        return “not yet implemented";
    }
}
```

assertions are the mechanism through which unit tests are automatically assessed

but unit testing is not enough

- in addition to their business functionalities, applications have critical technical requirements, such as reliability, security, scalability, etc.
    - these requirements are orthogonal to the business domain, i.e., they can be found in many other applications
        - achieving code reuse is difficult when business concerns and technical concerns are tightly interwoven in the same code

solution

a flexible software architecture supporting separation of concerns, which allows for the reuse of both business code and technical code

separation of concerns

general principle

Let me try to explain to you, what to my taste is characteristic for all intelligent thinking. It is, that one is willing to study in depth an aspect of one's subject matter in isolation for the sake of its own consistency, [...] occupying oneself only with one of the aspects.

We know that a program must be correct and we can study it from that viewpoint only; we also know that is should be efficient and we can study its efficiency on another day [...] But nothing is gained - on the contrary - by tackling these various aspects simultaneously. It is what I sometimes have called "the separation of concerns" [...]

A scientific discipline separates a fraction of human knowledge from the rest: we have to do so, because, compared with what could be known, we have very, very small heads.

E.W. Dijkstra, On the role of scientific thought
EWD 477, 30th August 1974, Neuen, The Netherlands

separation of concerns

general principle - an example

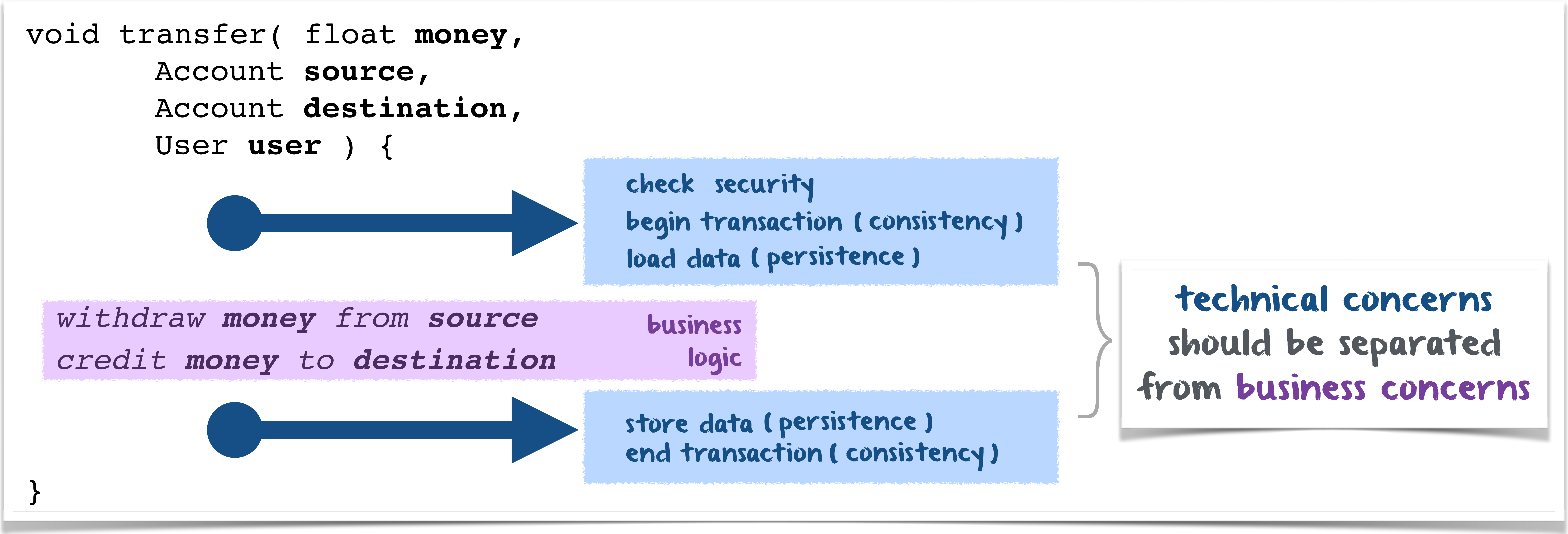
```
void transfer( float money,
       Account source,
       Account destination,
       User user ) {
  check whether this user is allowed to perform the transfer          security
  begin transaction                                                   consistency
  load source & destination accounts from database(s)                 persistence

  withdraw money from source
                                                                      business logic
  credit money to destination

  store source & destination accounts to database(s)                  persistence
  end transaction                                                     consistency
}
```

separation of concerns

general principle - an example



```
void transfer( float money,
       Account source,
       Account destination,
       User user ) {
```

check security
begin transaction (consistency)
load data (persistence)

```
withdraw money from source
credit money to destination
```

business logic

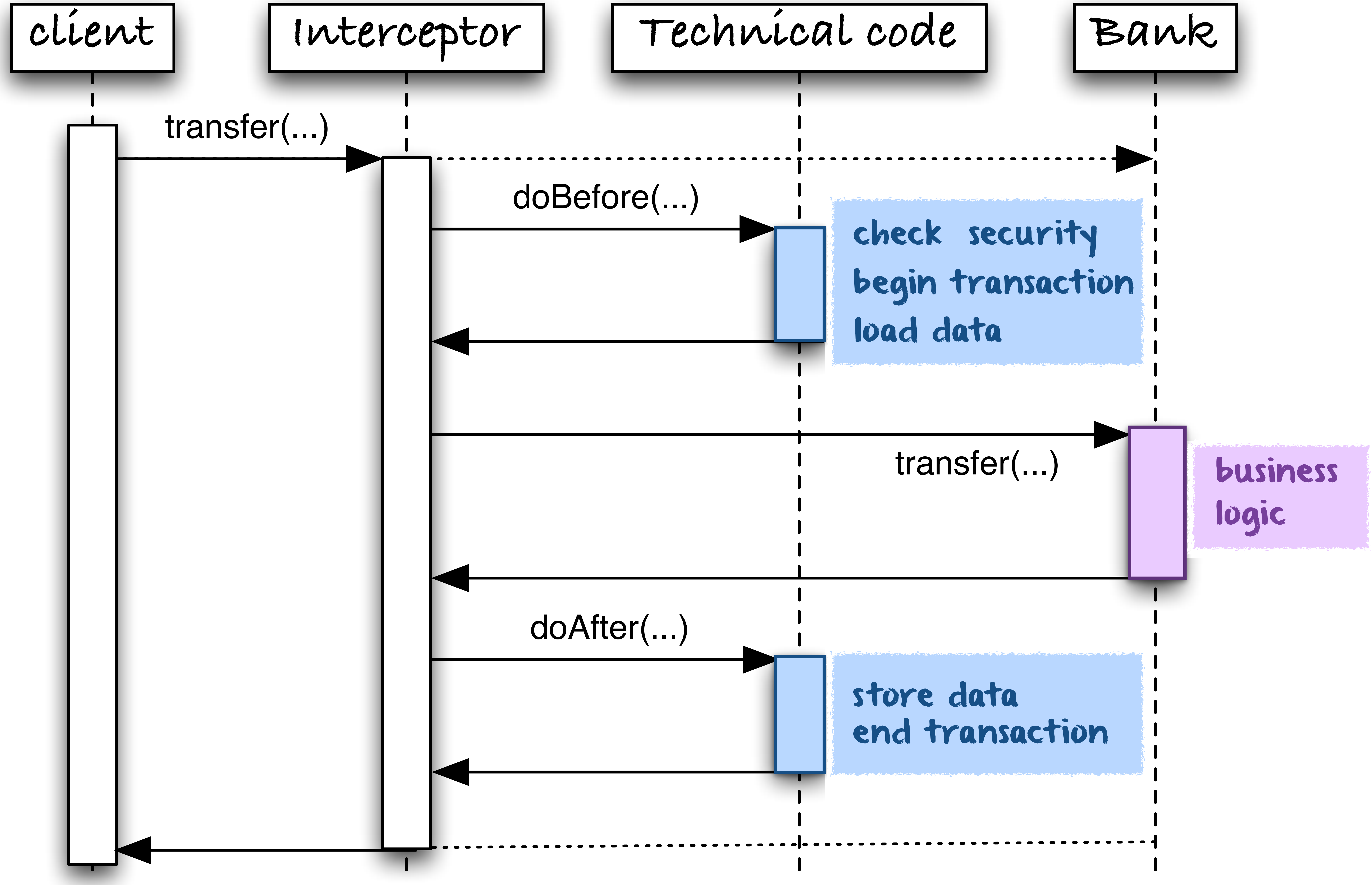
store data (persistence)
end transaction (consistency)

```
}
```

technical concerns should be separated from business concerns

separation of concerns

invocation interception as basic mechanism



separation of concerns

different approaches

- when does interception occur?
  - ✓ at compile-time (static interception)
  - ✓ at run-time (dynamic interception)

- how do we deal with technical concerns?
  - ✓ by coding and assembling technical objects
  - ✓ by declaring technical requirements

- the aspect-j programming model
  - ✓ when? at compile-time
  - ✓ how? by coding and assembling

- the GARF programming model
  - ✓ when? at run-time
  - ✓ how? by coding and assembling

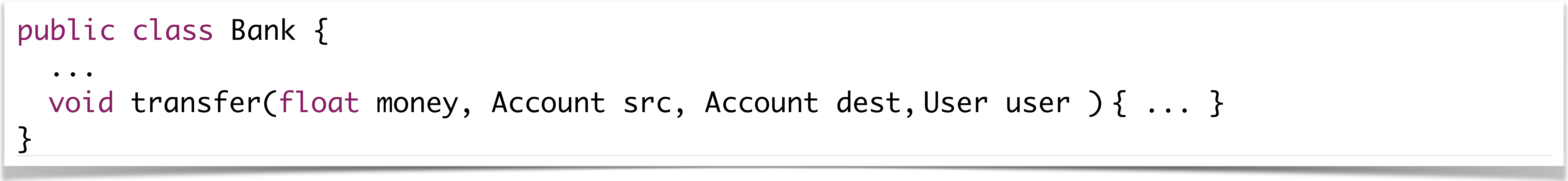
- the enterprise java beans component model
  - ✓ when? at compile-time
  - ✓ how? by declaring via annotations

Benoît Garbinato, Rachid Guerraoui, and Karim R. Mazouni. **Distributed programming in GARF.** In Rachid Guerraoui, Oscar Nierstrasz, and Michel Riveill, editors, *Object-Based Distributed Programming*, pages 225–239, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.

separation of concerns

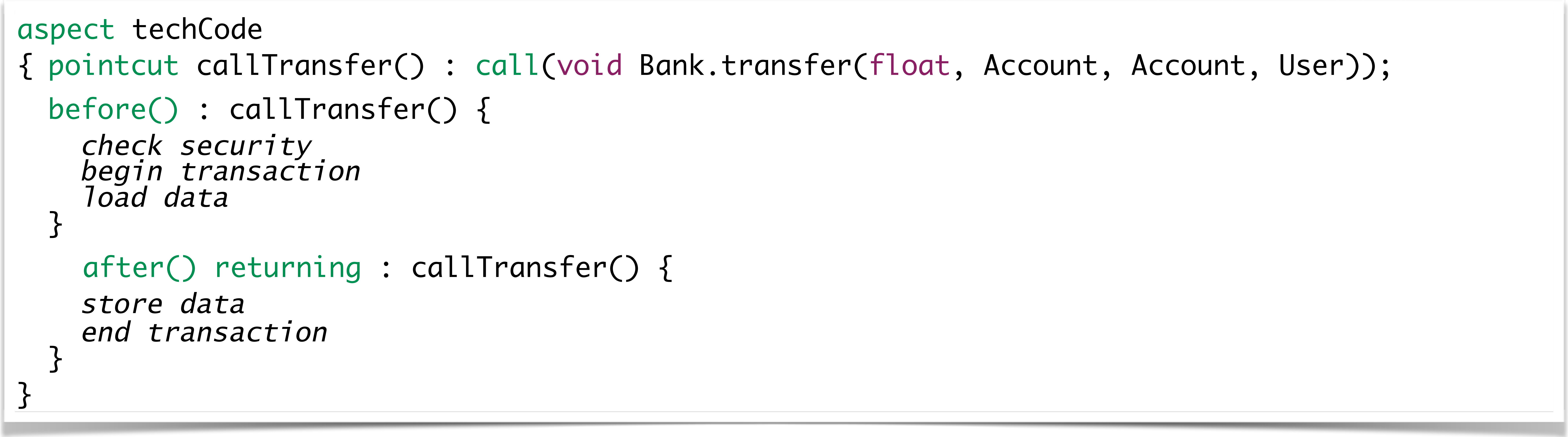
the aspect-j programming model → aspect-oriented programming

assume we have some Bank class :



```
public class Bank {
  ...
  void transfer(float money, Account src, Account dest,User user ){ ... }
}
```

we add the technical code as follows :



```
aspect techCode
{ pointcut callTransfer() : call(void Bank.transfer(float, Account, Account, User));
  before() : callTransfer() {
    check security
    begin transaction
    load data
  }
    after() returning : callTransfer() {
    store data
    end transaction
  }
}
```

separation of concerns

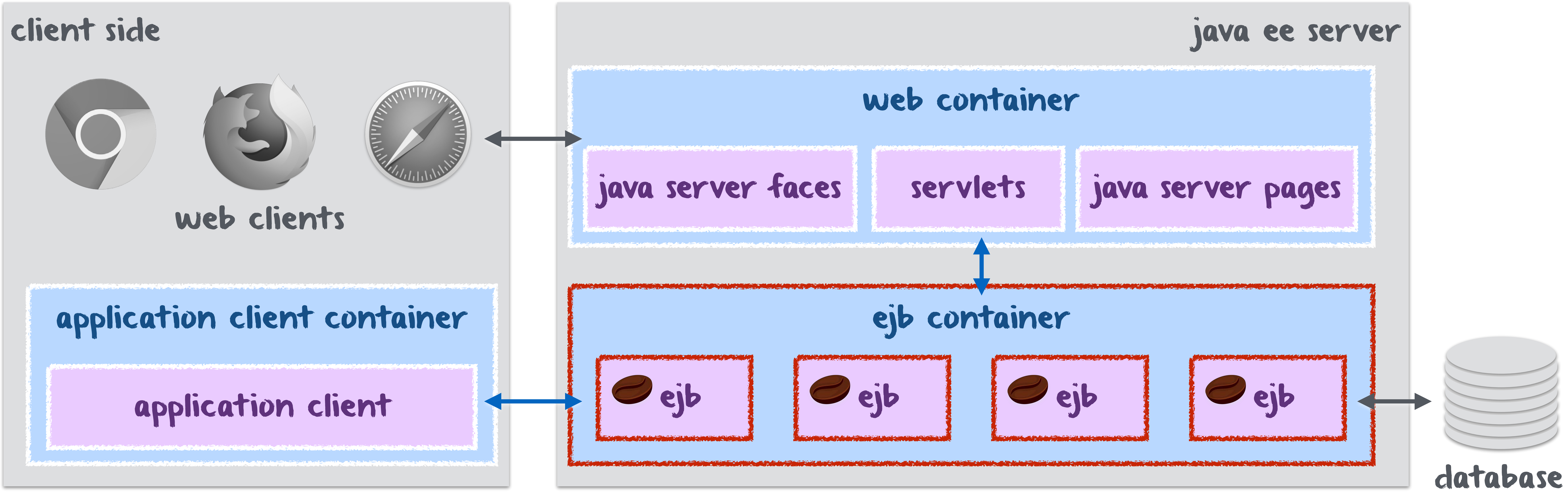
the enterprise java beans model
is based on two key notions

the component is a server-side software unit encapsulating
some business logic and deployed into a dedicated container
this is the actual enterprise java bean (ejb)

the container is the hosting environment interfacing the ejb
with its clients and with the low-level platform services,
and ultimately managing all technical aspects for the ejb
it is also known as the ejb container

separation of concerns

the enterprise java beans model is just one part of java ee*
which heavily relies on the component/containers dichotomy



*java enterprise edition

the enterprise java beans model

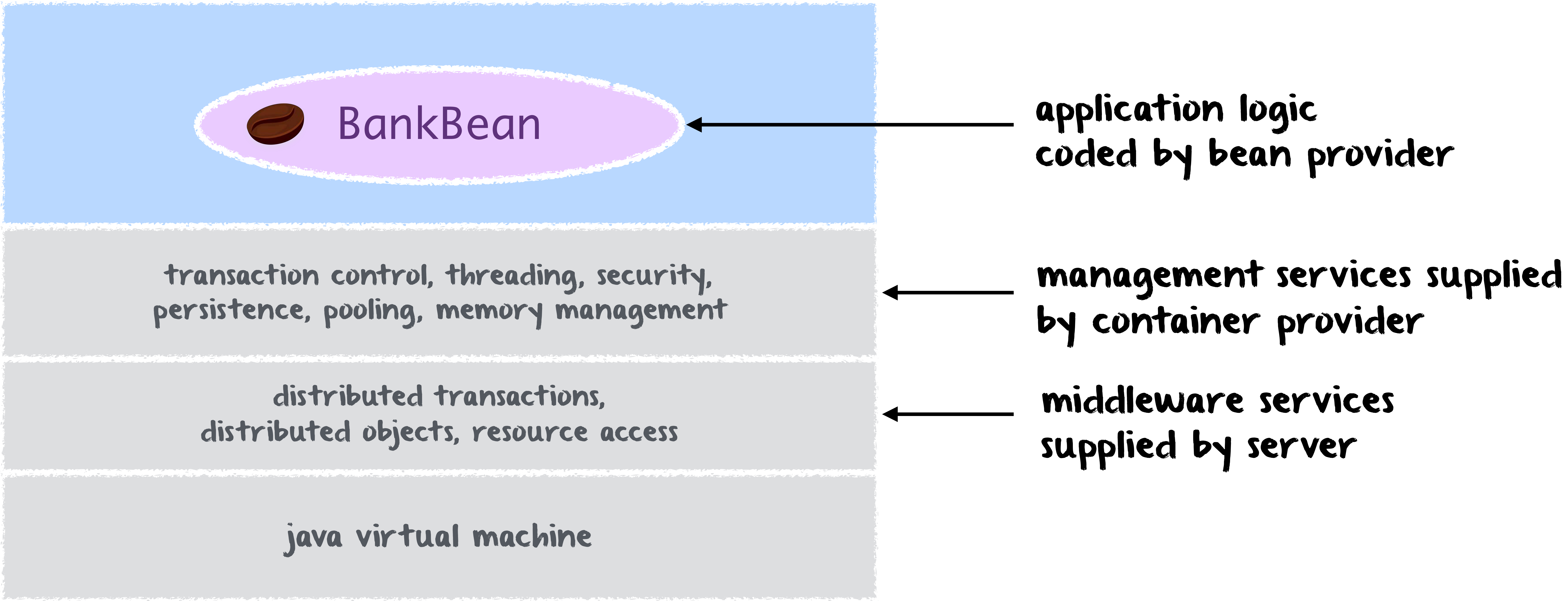
types of enterprise beans

- a session bean represents a session with a client application and can be either:
  - stateless: it belongs to a client only during a method call
  - stateful: it belongs to a client during the whole session
- a singleton is an object which class can have only one instance
  - any reference to a bean of that class point to the same single instance
  - a singleton is stateful by definition (otherwise use a stateless session bean)
- a message-driven bean is an object that can receive asynchronous messages
  - we will come back these beans when discussing asynchronous interactions

the enterprise java beans model

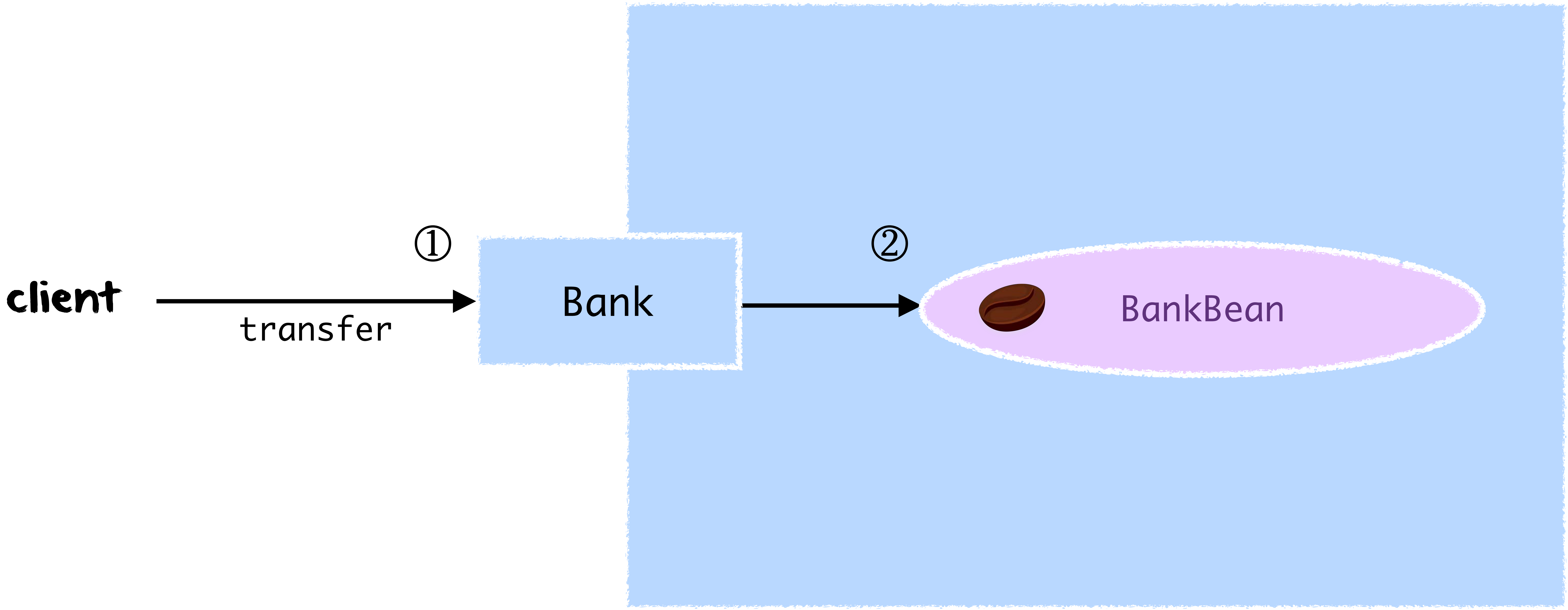
container responsibilities

the container intercepts client calls to manage the ejb lifecycle and its technical aspects


BankBean
transaction control, threading, security, persistence, pooling, memory management
distributed transactions, distributed objects, resource access
java virtual machine
application logic coded by bean provider
management services supplied by container provider
middleware services supplied by server

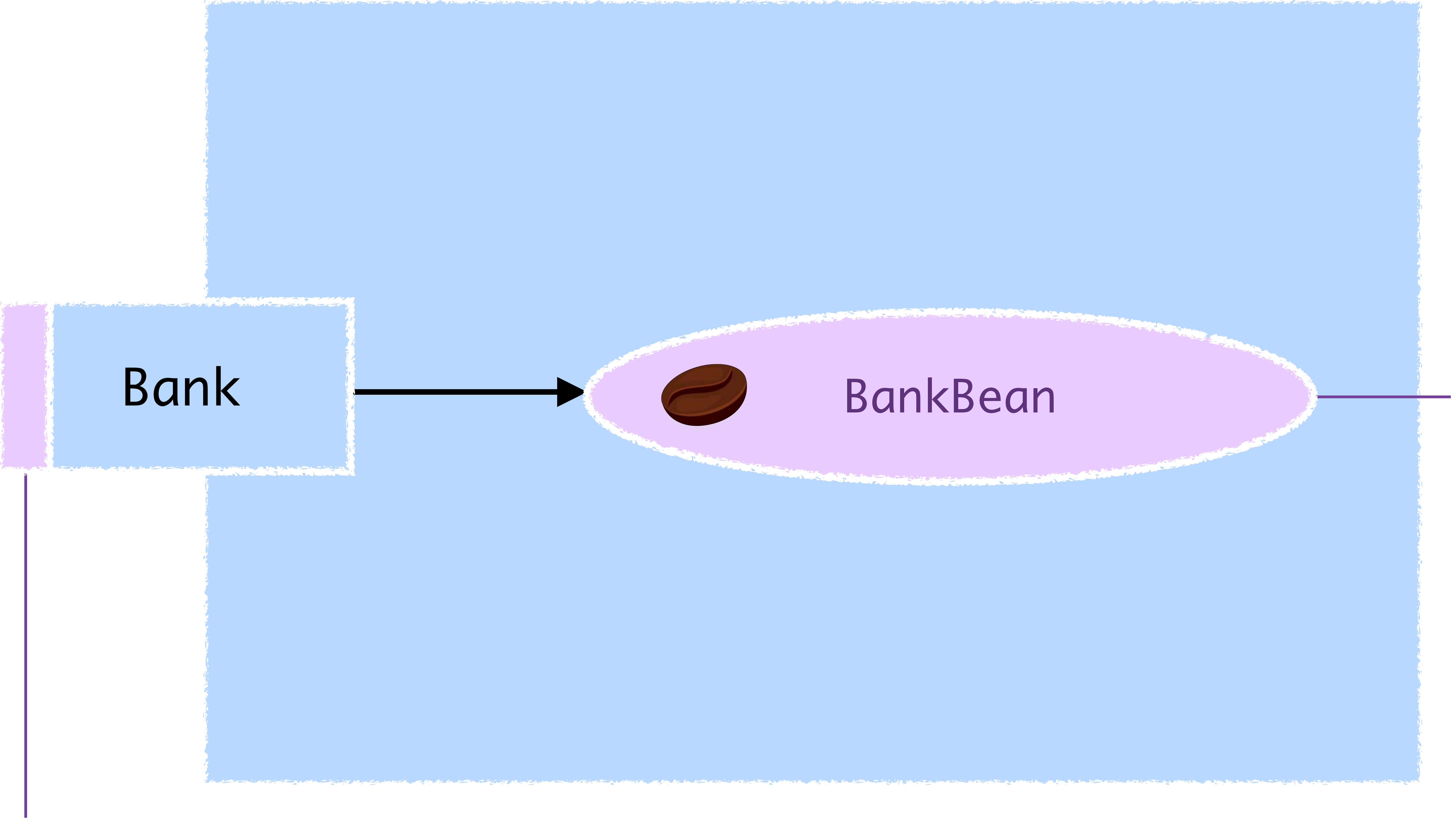
the enterprise java beans model

container as interceptor
of business methods



the enterprise java beans model

bean provider tasks

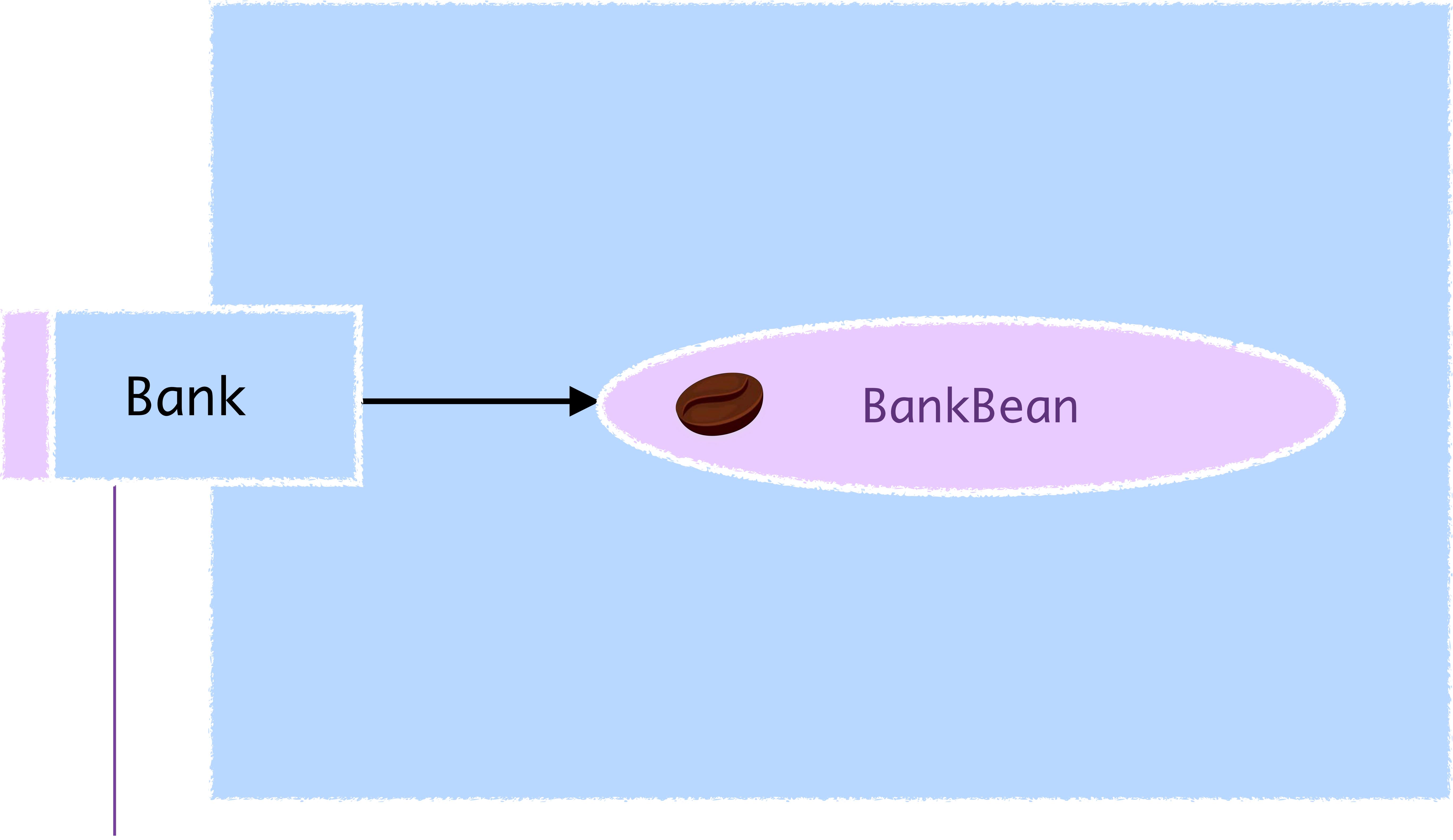


creates the bean class,
coding business methods

creates an interface for the bean

the enterprise java beans model

container provider tasks



provide an ejb-compliant container

implements the remote interface,
i.e., provides the interceptor object

the enterprise java beans model

a typical session bean

```
@Local
public interface Bank {
    public void transfer( Account source, Account destination,double amount )
    throws BankingException;
    void initialize();
}
```

dependency injection

```
@Stateful
public class BankBean implements Bank {
    @Resource
    SessionContext ctx;

    public void transfer( Account source, Account destination,double amount )
    throws BankingException { ... }

    public void initialize() { ... }
}
```

the enterprise java beans model

dependency injection

with dependency injection, an object does not set its dependencies to other objects itself

with dependency injection, an object's field can be set by an external actor, in our case the container

dependency injection allows us to decouple various components at the code level

dependency injection is expressed by the programmer via annotations

the enterprise java beans model

annotations

an annotation is a portion of text that expresses information about the code directly in the code

an annotation does not directly modify the semantics of your code but the way it is treated by tools

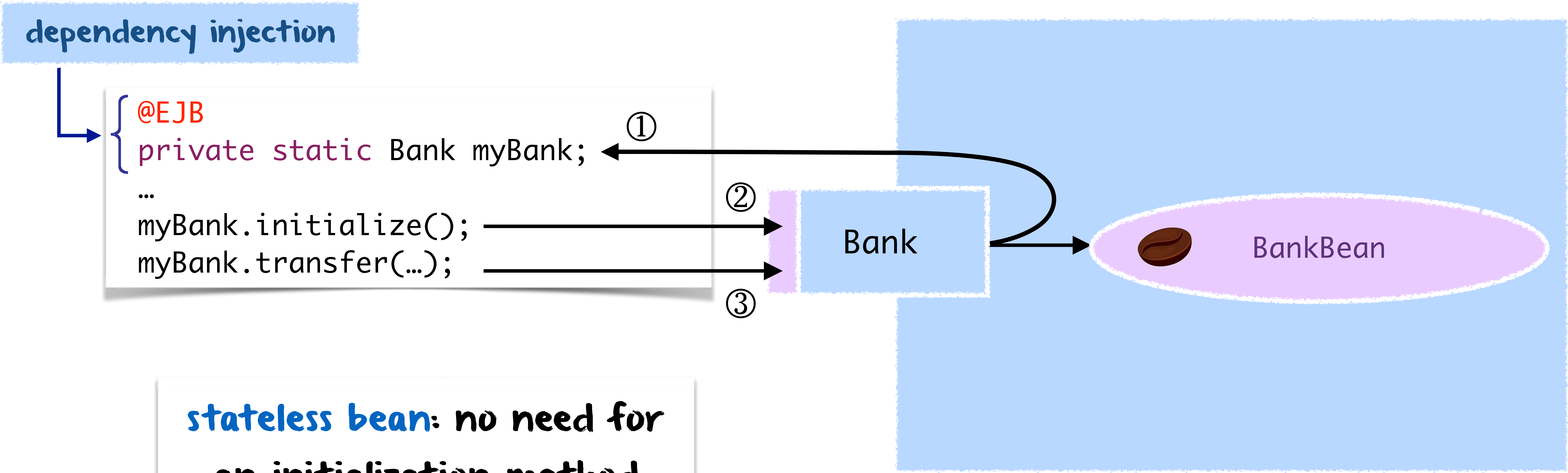
java always had ad hoc annotation, e.g., java comments, the `transient` keyword, etc.

since version 5, Java supports general and extensible annotations mechanism, using the @ character

```
@Stateless
@Stateful
@LocalBean
@Remote
@Resource
@EJB
@Remove
@PostConstruct
@PreDestroy
@PrePassivate
@PostActivate
...
```

the enterprise java beans model

client developer tasks



```
@EJB
private static Bank myBank;
…
myBank.initialize();
myBank.transfer(…);
```

stateless bean: no need for an initialization method

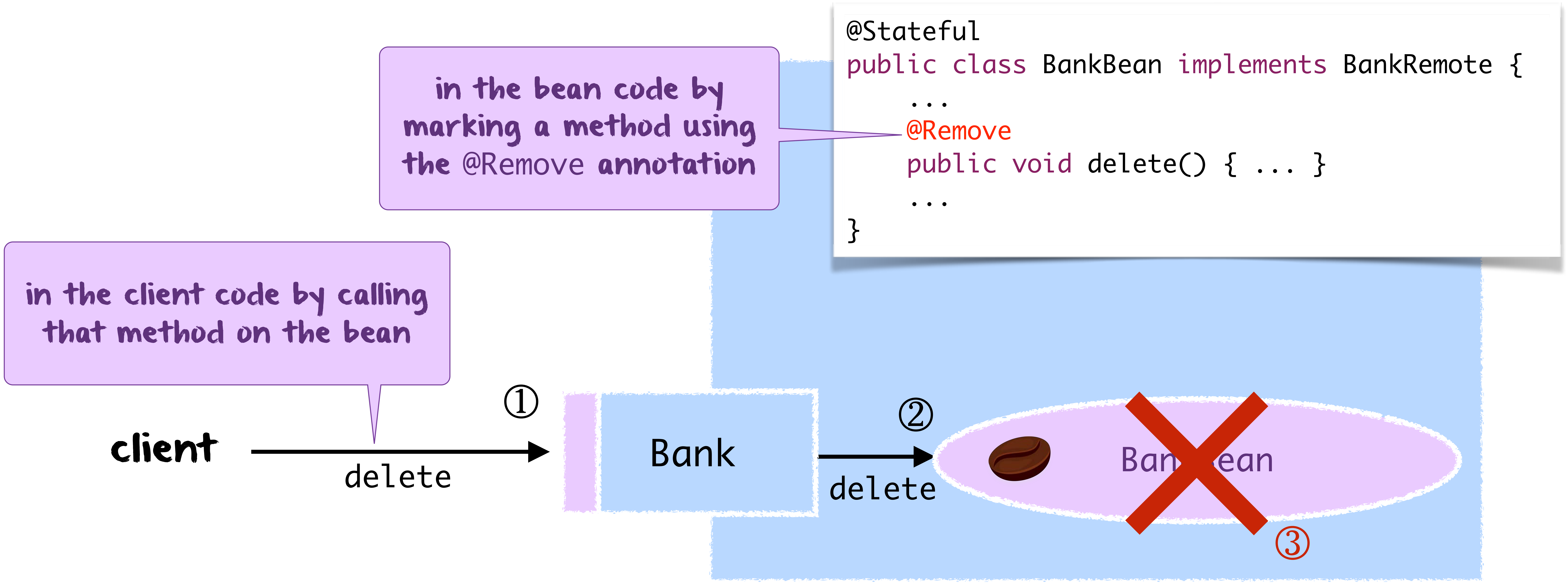
stateful bean: one or more initialization methods (business method)

the enterprise java beans model

removing a session bean

to perform some house cleaning before stopping to use that bean

to indicate to the container that we no longer need that bean



```
@Stateful
public class BankBean implements BankRemote {
    ...
    @Remove
    public void delete() { ... }
    ...
}
```

in the bean code by marking a method using the @Remove annotation

in the client code by calling that method on the bean

the enterprise java beans model

to ensure availability & scalability, the container uses pooling strategies to manage enterprise beans

resource pooling

client$_1$
client$_2$
client$_3$
client$_4$
client$_5$
Bank
BankBean
BankBean
BankBean
pool of ejb instances
stateless session bean
stateful session bean
client$_1$
Bank
client$_2$
Bank
client$_3$
Bank
BankBean
BankBean
BankBean
BankBean
BankBean
pool of ejb instances

the enterprise java beans model

activation/passivation

the container can only host a limited number of session beans in memory

when more beans are needed, it uses passivation/activation strategy

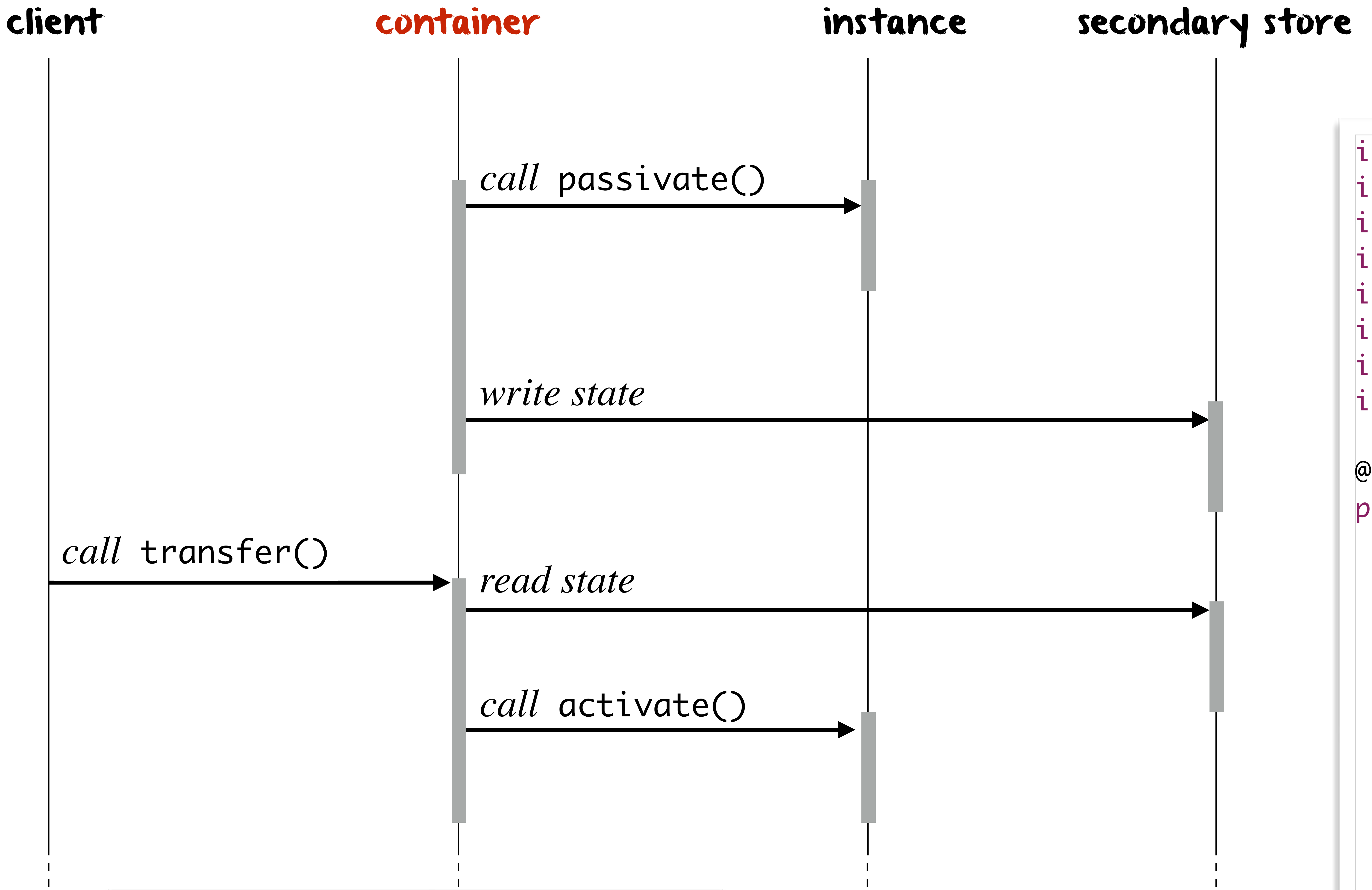
- passivation: write a bean to disk and remove it from volatile memory (swap out)
- activation: read a bean from disk and recreate it in volatile memory (swap in)
- usually follows a least recently used policy

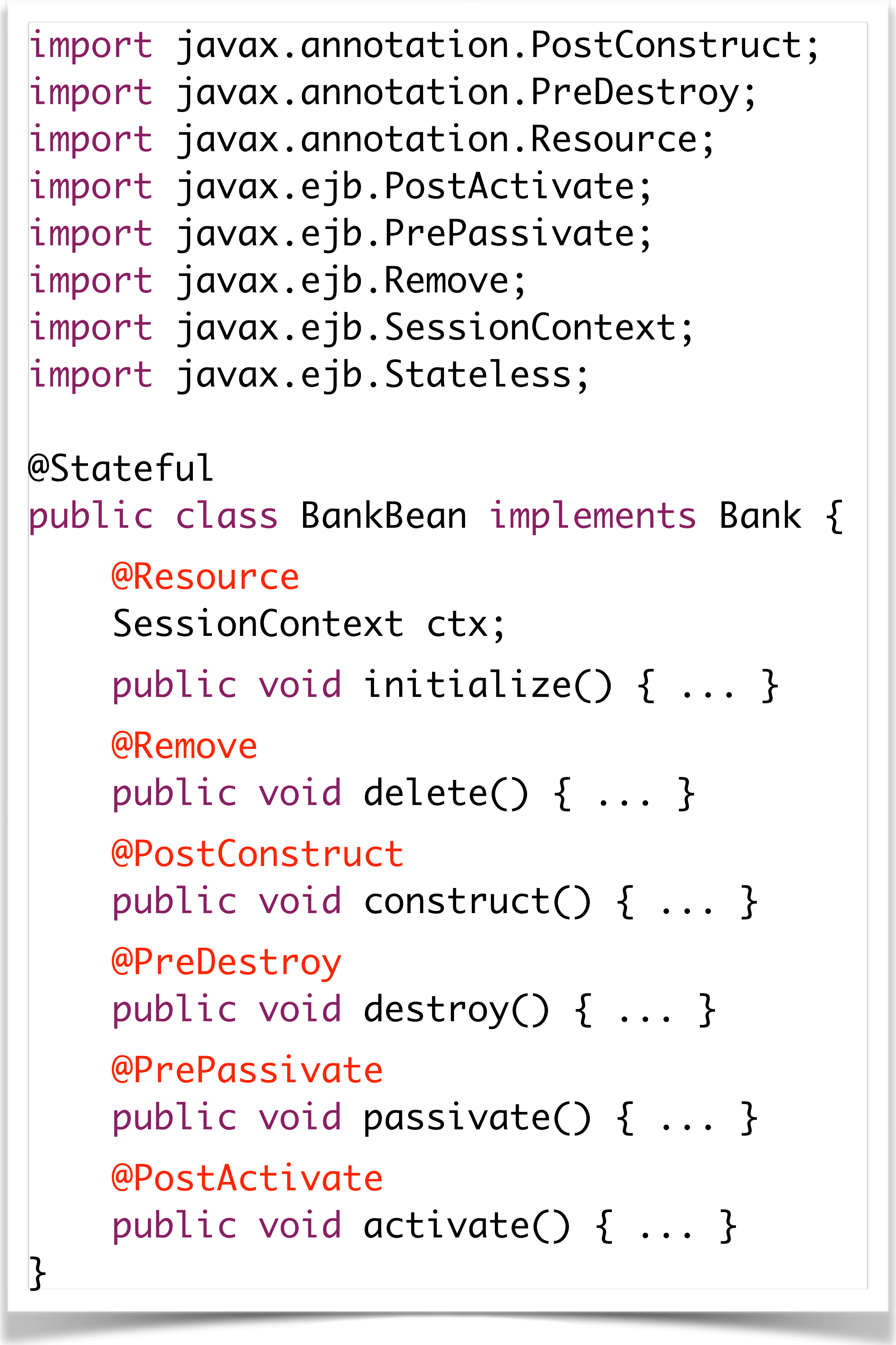
the container can only manage part of the state of a passivated/activated session bean, i.e., primitive types, serializable objects, context objects, etc.

for state (fields) outside this category, the bean provider must manage activation/passivation programmatically

the enterprise java beans model

activation/passivation

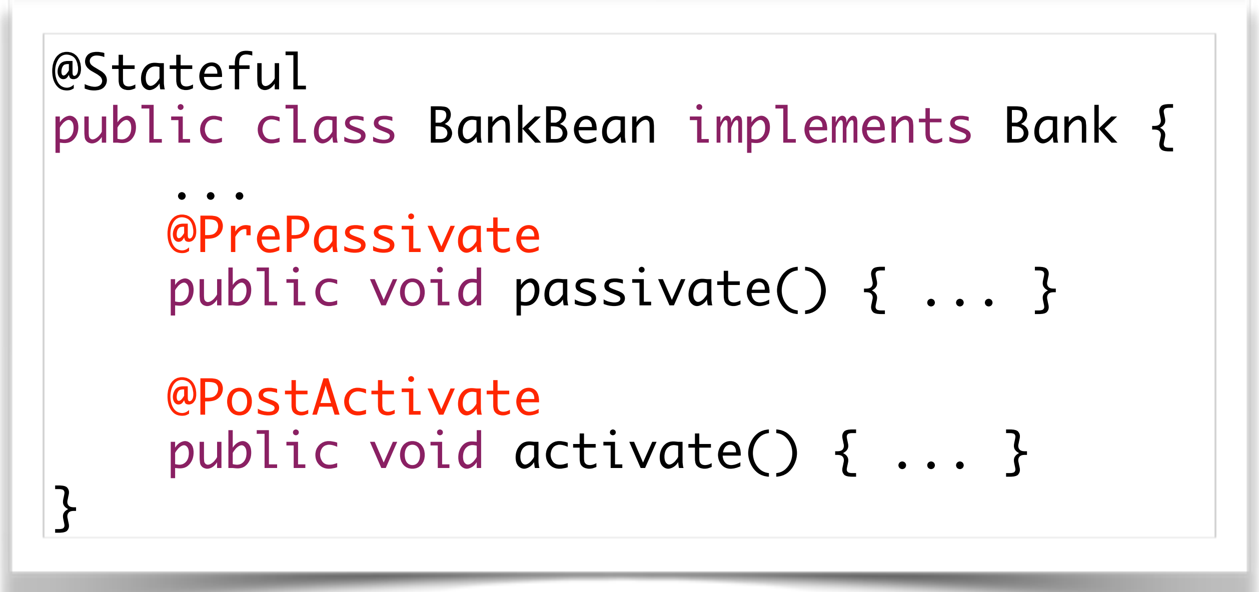




```
import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
import javax.annotation.Resource;
import javax.ejb.PostActivate;
import javax.ejb.PrePassivate;
import javax.ejb.Remove;
import javax.ejb.SessionContext;
import javax.ejb.Stateless;

@Stateful
public class BankBean implements Bank {
    @Resource
    SessionContext ctx;
    public void initialize() { ... }
    @Remove
    public void delete() { ... }
    @PostConstruct
    public void construct() { ... }
    @PreDestroy
    public void destroy() { ... }
    @PrePassivate
    public void passivate() { ... }
    @PostActivate
    public void activate() { ... }
}
```

called by the container only if they exist (optional)

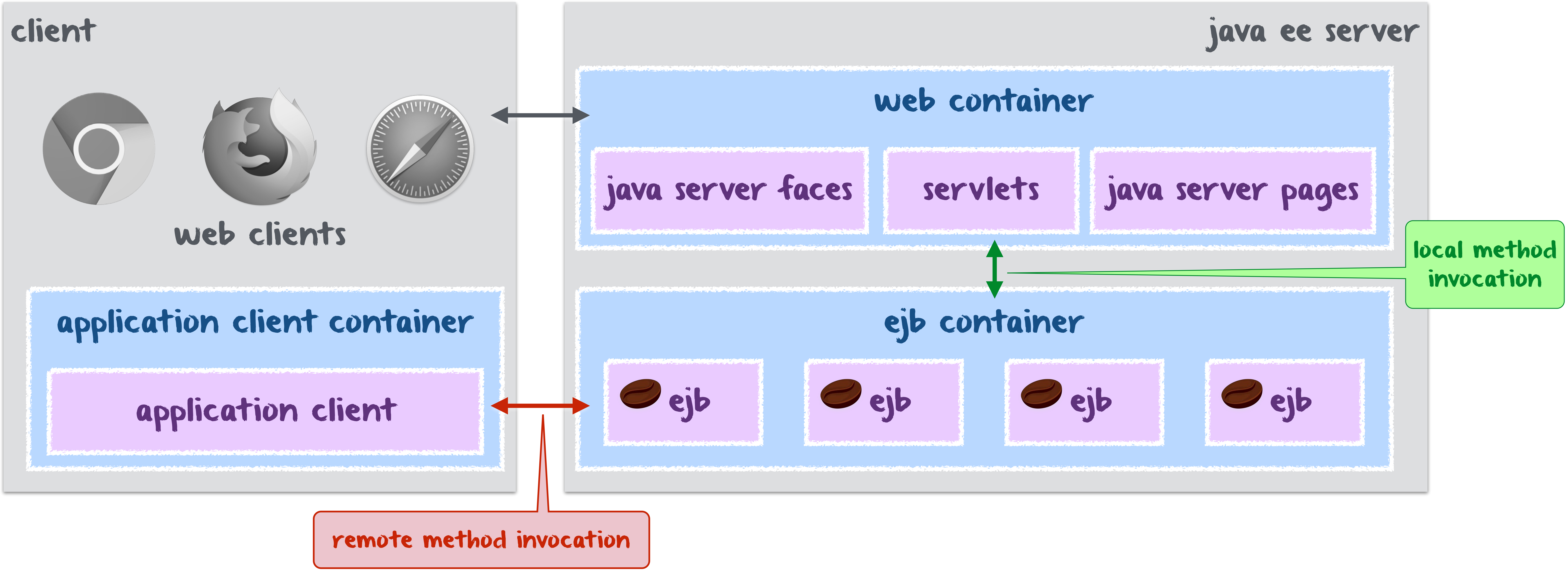


```
@Stateful
public class BankBean implements Bank {
    ...
    @PrePassivate
    public void passivate() { ... }

    @PostActivate
    public void activate() { ... }
}
```

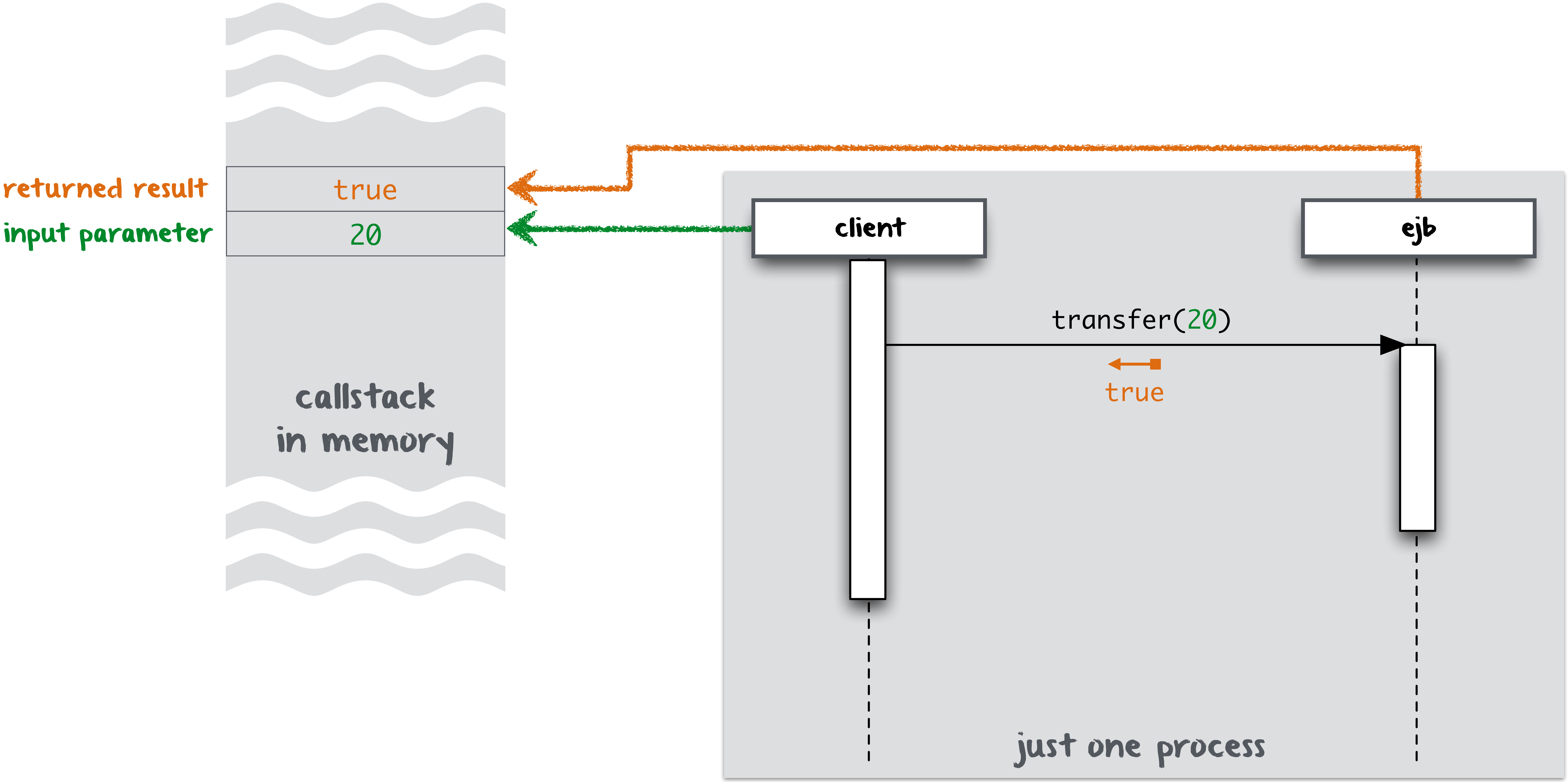
the enterprise java beans model

local & remote method invocations



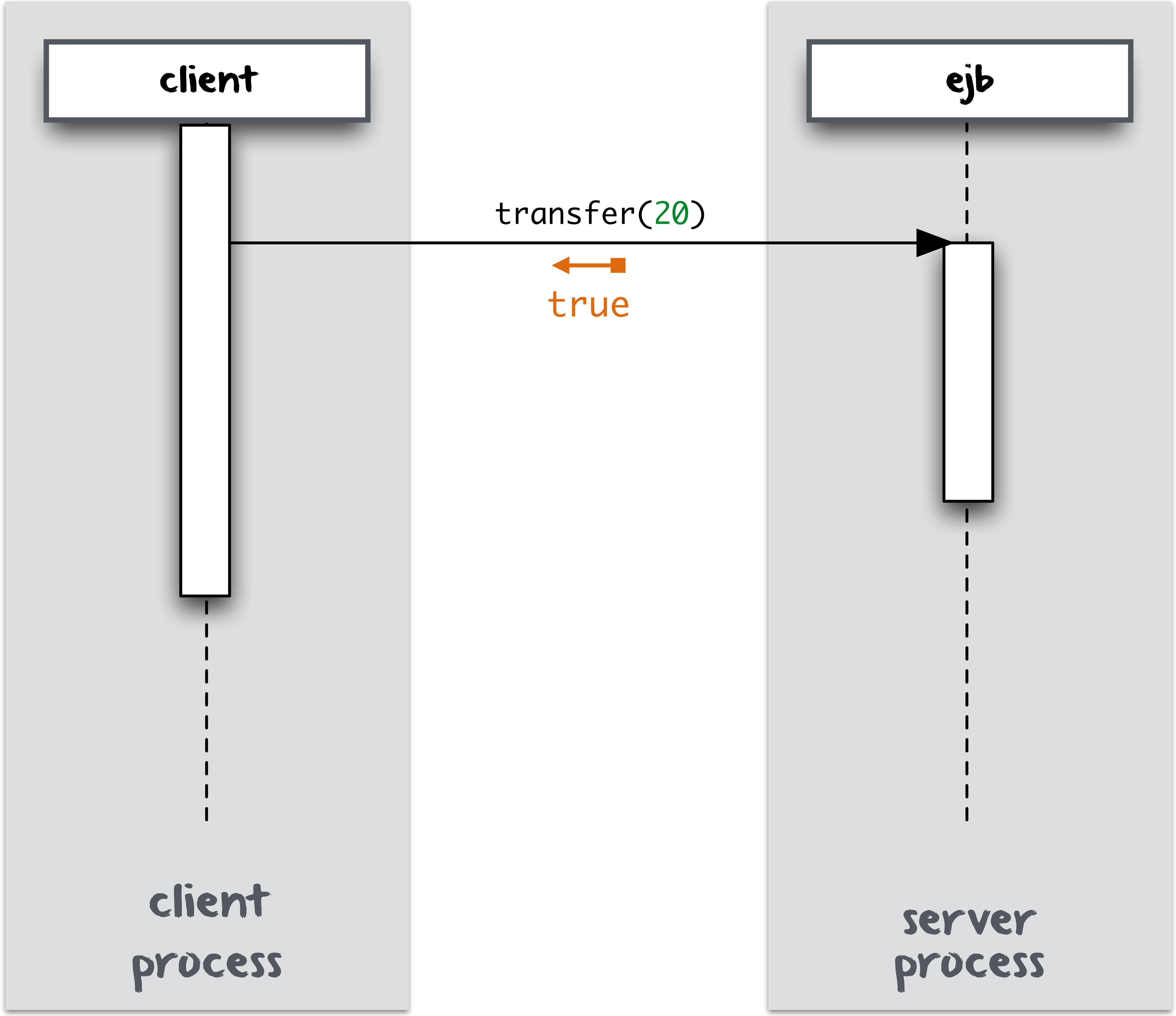
the enterprise java beans model

local method invocations



the enterprise java beans model

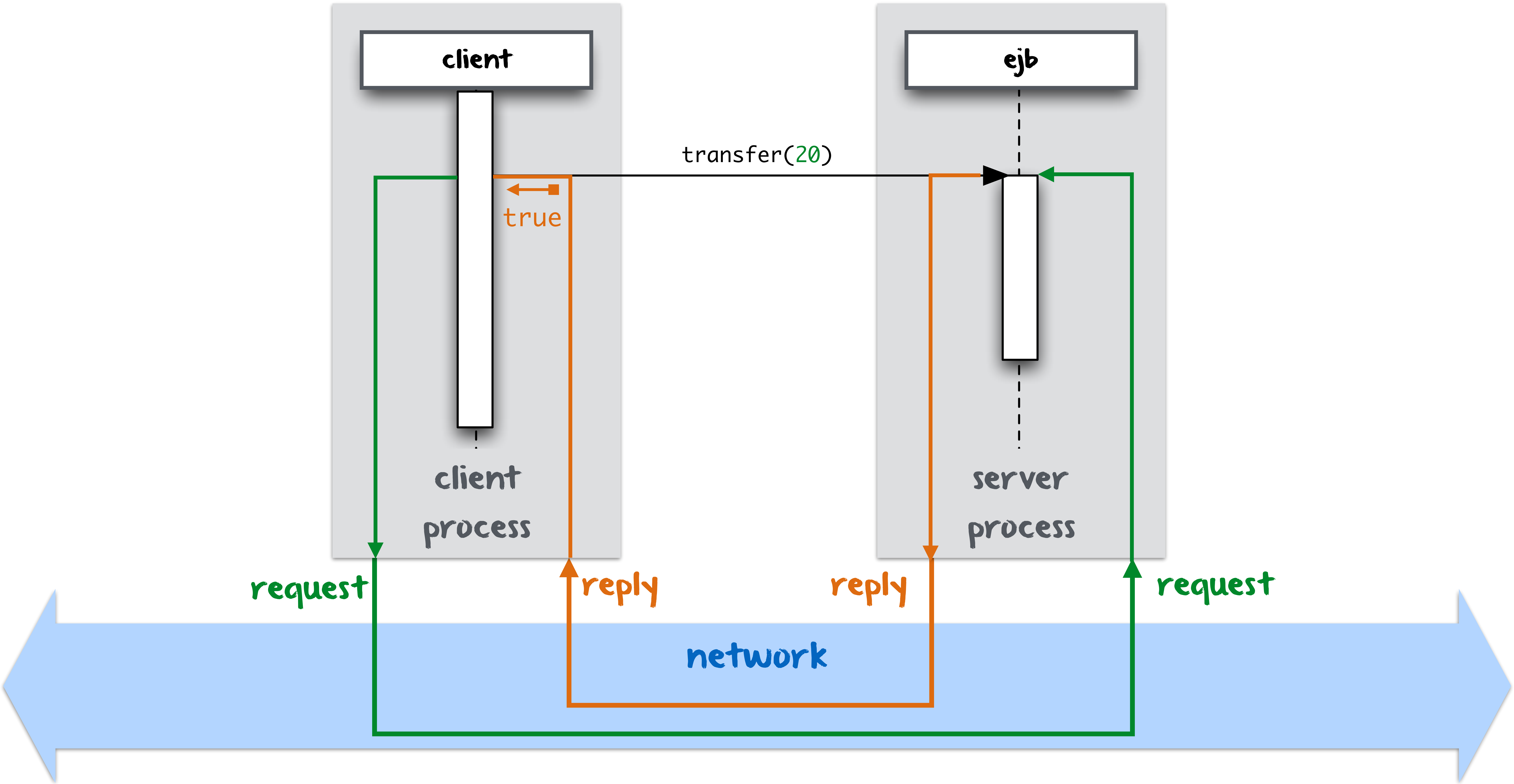
remote method invocations



a **remote method** is transparently invoked across the network, **as if it was local**

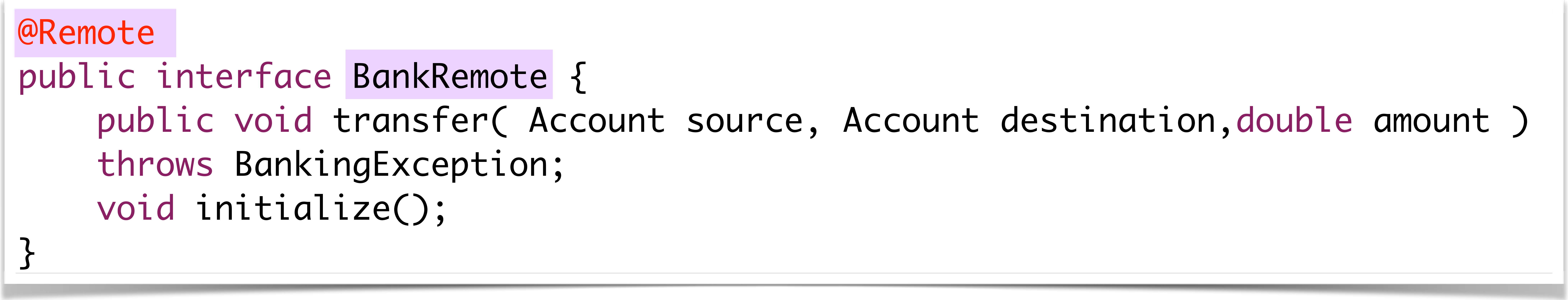
the enterprise java beans model

remote method invocations

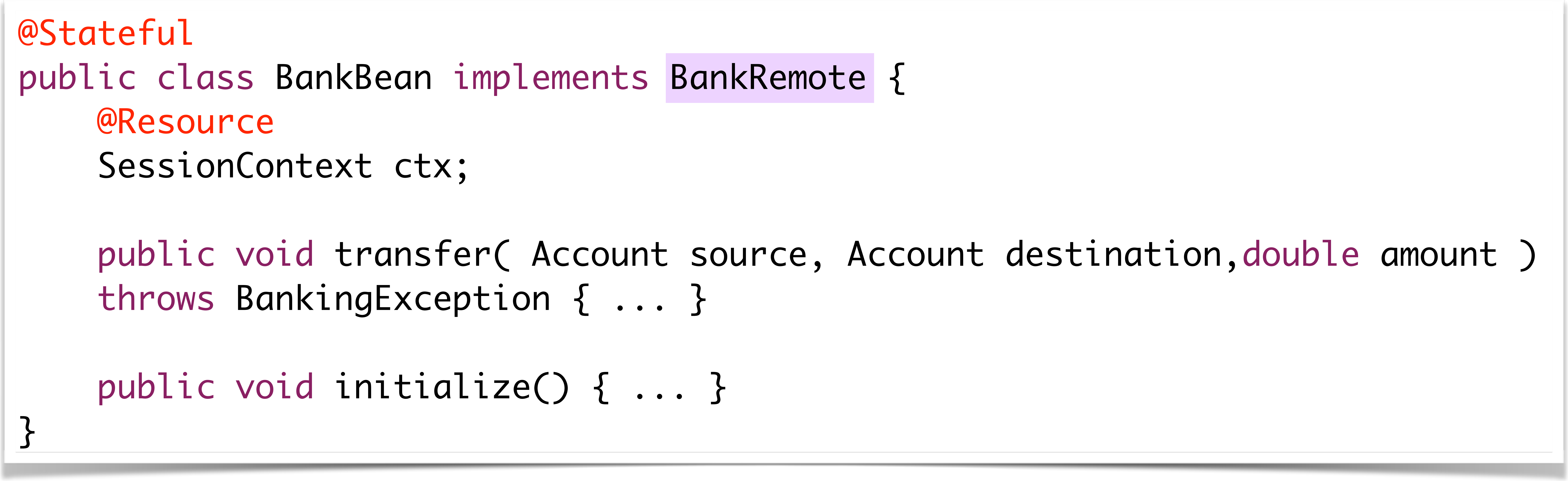


the enterprise java beans model

remote method invocations



```
@Remote
public interface BankRemote {
    public void transfer( Account source, Account destination,double amount )
    throws BankingException;
    void initialize();
}
```



```
@Stateful
public class BankBean implements BankRemote {
    @Resource
    SessionContext ctx;

    public void transfer( Account source, Account destination,double amount )
    throws BankingException { ... }

    public void initialize() { ... }
}
```