UNIVERSITÀ DELLA CALABRIA
il Campus per eccellenza

Code constructs promoting security

Mario Alviano

This lesson content

- Strategies to solve security issues in code
  - **Immutability**, to solve integrity and availability issues
  - **Fail fast**, to solve irregular input and state issues
  - **Validation**, to ensure that input conform to some specification
- It’s time to use these stategies in our code!
- We must also recognize security issues in legacy code

Immutability

- An object is **mutable** if it allows for state changes
- An object is **immutable** if it inhibits state changes
  - Safely share them among threads
  - High data availability => prevent DoS attacks
- Unless required, do not make an object mutable!
  - Dangerous
  - Expensive

Example. Online shop

- Each customer is associated with a score based on the previous purchases
- Customers with high score can benefit delayed payment (as an alternative to immediate payment)
- Scores are computed while orders are added to the system
- Suddenly, there are problems during an advertising campaign
  - Many long waiting lines and many orders timeout
  - Payment methods are not consistent with the shop policy

The financial department reports several delayed payments for many customers with low score

```
public class Customer {
   private static final int MIN_INVOICE_SCORE = 500;  ①
   private Id id;     ②
   private Name name;     ③
   private Order order;     ④
   private CreditScore creditScore;     ⑤
public synchronized Id getId() {
   return id;
}

public synchronized void setId(final Id id) {
   this.id = id;
}
```

Several problems!

Attributes have to be initialized with setter methods: the state can change and it isn't clear when the object is actually initialized

All methods are synchronized: threads compete the access to the object, and stuck!

```
   public synchronized Name getName() {
      return name;
   }

   public synchronized void setName(Name name) {
      this.name = name;
   }

   public synchronized Order getOrder() {
      this.order = OrderService.fetchLatestOrder(id);
      return order;
   }

   public synchronized void setOrder(Order order) {
      this.order = order;
   }

   public synchronized CreditScore getCreditScore() {
      return creditScore;
   }

   public synchronized void setCreditScore(CreditScore
creditScore){
      this.creditScore = creditScore;     ⑥
   }

   public synchronized boolean
isAcceptedForInvoicePayment() {
      return creditScore.compute() >
                                 MIN_INVOICE_SCORE;     ⑦
   }
   ...
}
```

Categorizing the observed issues helps to understand how the issues actually depend on the implementation of class Customer

Observed issue	Category	Probable cause
Long waiting lines and poor efficiency	Availability	The system cannot relaiably access to customer data and timeouts
Orders timeout at checkout	Availability	The system cannot timely obtain the data required to process orders
Inconsistent payment methods	Integrity	Customer scores are irregularly changed

```
public class Customer {
   private static final int MIN_INVOICE_SCORE = 500;  ①
   private Id id;     ②
   private Name name;     ③
   private Order order;     ④
   private CreditScore creditScore;     ⑤
public synchronized Id getId() {
   return id;
}

public synchronized void setId(final Id id) {
   this.id = id;
}
```

Availability issues

Due to synchronization

This app does many reads and few writes

To remove synchronized is OK for reads, but not for writes

Nontrivial solution: ReadWriteLock

```
   public synchronized Name getName() {
      return name;
   }

   public synchronized void setName(Name name) {
      this.name = name;
   }

   public synchronized Order getOrder() {
      this.order = OrderService.fetchLatestOrder(id);
      return order;
   }

   public synchronized void setOrder(Order order) {
      this.order = order;
   }

   public synchronized CreditScore getCreditScore() {
      return creditScore;
   }

   public synchronized void setCreditScore(CreditScore
creditScore){
      this.creditScore = creditScore;     ⑥
   }

   public synchronized boolean
isAcceptedForInvoicePayment() {
      return creditScore.compute() >
                              MIN_INVOICE_SCORE;     ⑦
   }
   ...
}
```

```java
import static org.apache.commons.lang3.Validate.notNull;

public final class Customer {
   private final Id id;    ①
   private final Name name;    ②
   private final CreditScore creditScore;    ③

   public Customer(final Id id, final Name name,
                   final CreditScore creditScore) {
      this.id = notNull(id);
      this.name = notNull(name);
      this.creditScore = notNull(creditScore);
   }

   public Id id() {
      return id;
   }

   public Name name() {
      return name;
   }

   public Order order() {
      return OrderService.fetchLatestOrder(id);
   }

   public boolean isAcceptedForInvoicePayment() {
      return creditScore.isAcceptedForInvoicePayment();
   }
```

Simple solution

Make the object immutable

Attributes are initialized on construction and cannot change

This way the object can be shared among several threads

We still have the issue on writes: we need the **identity snapshots**, a concept we will see later

```
public class Customer {
  private static final int MIN_INVOICE_SCORE = 500;
  private CreditScore creditScore;
  ...
  public synchronized void setCreditScore(
                          CreditScore creditScore) {  ①
     this.creditScore = creditScore;     ②
  }

  public synchronized CreditScore getCreditScore() {
     return creditScore;     ③
  }

  public synchronized boolean isAcceptedForInvoicePayment()
{
     return creditScore.compute() > MIN_INVOICE_SCORE;
  }
  ...
}
```

synchronized is useless if the object can be externally modified!

Integrity issues

1) CreditScore is initializied by setter: we cannot control changes
2) Method setCreditScore() doesn't copy its argument:
   the object may be shared and externally modified
3) Method getCreditScore() inadvertently releases a reference to CreditScore:
   its value can be externally modified

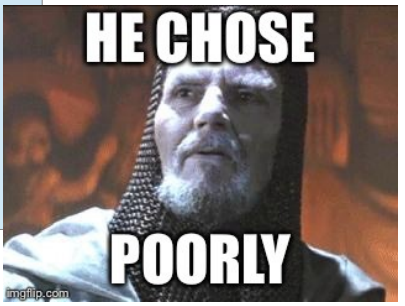
```
public class Customer {
  private static final int MIN_INVOICE_SCORE = 500;
  private CreditScore creditScore;
  ...
  public synchronized void setCreditScore(
                        CreditScore creditScore) {  ①
    this.creditScore = creditScore;    ②
  }

  public synchronized CreditScore getCreditScore() {
    return creditScore;    ③
  }

  public synchronized boolean isAcceptedForInvoicePayment()
{
    return creditScore.compute() > MIN_INVOICE_SCORE;
  }
  ...
}
```

CreditScore computes customer scores:
it needs access to customer data

The logic of Customer and CreditScore
is mutually dependent:
very bad!




```
import static org.apache.commons.lang3.Validate.isTrue;

public class CreditScore {
   private static final int MIN_INVOICE_SCORE = 500;
   private final int score;

   public CreditScore(final int computedCreditScore) {
      isTrue(computedCreditScore > -1, "Credit score must
be > -1");
      this.score = computedCreditScore;    ①
   }

   public boolean isAcceptedForInvoicePayment() {
      return score > MIN_INVOICE_SCORE;
   }
   ...
}
```

CreditScore should be immutable:
the calculation must be done by Customer
or by another higher entity

If CreditScore is immutable, it can be
shared among several threads

The method assigning CreditScore
to Customer must be synchronized
(or use another mechanism like
entity snapshots)

Fail fast by contracts

```
public Customer(final Id id, final Name name,
                final CreditScore creditScore) {
   this.id = notNull(id);
   this.name = notNull(name);
   this.creditScore = notNull(creditScore);
}
```

```
public CreditScore(final int computedCreditScore) {
   isTrue(computedCreditScore > -1, "Credit score must
be > -1");
   this.score = computedCreditScore;    ①
}
```

Invalid data are blocked before they can create an invalid object

This is another strategy promoting secure software coding

The idea is that every class defines a contract: it guarantees some invariants

Other classes can assume that all invariants hold

Contract example

- You call a plumber to fix a sink in your bathroom
- The plumber asks you to not lock the door of your bathroom and to close the water stop valve
  - These are the prerequisite of the job (or of the contract)
  - If they are not satisfied, better to not do the job (fail fast)
- The plumber guarantees that after the job the sink will properly work
  - This is the postcondition of the contract
- Classes should define similar contracts

Class in a system for cat breeders

Cat names are put in a queue as soon as they come to mind

When a cat is born, a name from the queue is extracted

```
public class CatNameList {
    private final List<String> catNames = new
ArrayList<String>();

    public void queueCatName(String name) {     ①
        catNames.add(name);
    }

    public String nextCatName() {     ②
        return catNames.get(0);
    }

    public void dequeueCatName() {     ③
        catNames.remove(0);
    }

    public int size() {     ④
        return catNames.size();
    }
}
```

Method	Required preconditions	Guaranteed postconditions
nextCatName	list is nonempty	size doesn't change The returned name contains 's'
dequeueCatName	list is nonempty	size decreases by 1
queueCatName	name is not null and contains at least an 's' name must not be already in the list	size increases by 1

The contract clarifies that the called must provide a name not already in the list

Method	Required preconditions	Guaranteed postconditions
nextCatName	list is nonempty	size doesn't change The returned name contains 's'
dequeueCatName	list is nonempty	size decreases by 1
queueCatName	name is not null and contains at least an 's' name must not be already in the list	size increases by 1

```
public void queueCatName(String name) {
    if (name == null)    ①
        throw new NullPointerException();
    if (!name.matches(".*s.*"))    ②
        throw new IllegalArgumentException("Must contain
s");
    if (catNames.contains(name))    ③
        throw new IllegalArgumentException("Already
queued");
    catNames.add(name);
}
```

The contract is represented in code

Methods start with validity checks:
if prerequisites are not satisfied,
fail fast by raising exceptions

Use NullPointerException if null
is used where it is not expected

Use IllegalArgumentException
for other checks (unless there is a more specific Java exception)

```
import org.apache.commons.lang3.Validate.*;
...
public void queueCatName(String name) {
    notNull(name);    ①
    matchesPattern(name,".*s.*",
                   "Cat name must contain s");
    isTrue(!catNames.contains(name),
           "Cat name already queued");    ③
    catNames.add(name);
}
```

Validate of Apache's Commons Lang

Several methods useful for validation: notNull, isTrue, matchesPattern, exclusiveBetween, inclusiveBetween

Use import static to directly access all static methods of Validate (note that the book has a few typos in code, it misses static)

https://commons.apache.org/proper/commons-lang/apidocs/org/apache/commons/lang3/Validate.html

Caution! Do not use invalid data in exception messages.
It opens doors to security issues: injections and sensitive data leakage, to start!

When to use contracts

- It is advisable to define contracts and to verify prerequisites of public methods
- For methods with package visibility it depends
  - yes if the package is big and the methods widely used
  - no if it is part of a small utility class
- Private methods do not need contracts (and usually use assentions)
- Protected methods, well, you should not use them

Invariants in constructor

```
import org.apache.commons.lang3.Validate.*;

enum Sex {MALE, FEMALE;}

public class Cat {

    private String name;
    private final Sex sex;

    public Cat(String name, Sex sex) {
        this.name = notNull(name);     ①
        this.sex = notNull(sex);       ①
        matchesPattern(name,".*s.*",
                       "Cat name must contain s");
    }
    ...
}
```

The same validation of name is done in Cat() and CatNameList(): it would be better to define a CatName class

Avoid empty constructors
(unless there are clear default values)

Name and gender of the cat are required: we give them in the constructor and we verify that they are not null

Note that notNull returns a reference to its argument
(if it doesn't raise an exception)

matchesPattern() is void

Fail on invalid state

```
public String nextCatName() {
    validState(!catNames.isEmpty());
    return catNames.get(0);
}
```

The method of CatNameList
to access the next name

- validState() raises an IllegalStateException if the expression is false
- We don’t use isTrue() because it’s dedicated to arguments: isTrue() raises IllegalArgumentException if the expression is false

```java
import org.apache.commons.lang3.Validate.*;

public class CatNameList {
    private final List<String> catNames = new
ArrayList<String>();

    public void queueCatName(String name) {    ①
        notNull(name);    ②
        matchesPattern(name,".*s.*",
                       "Cat name must contain s");
        isTrue(!catNames.contains(name),
               "Cat name already queued");    ④
        catNames.add(name);
    }

    public String nextCatName() {
        validState(!catNames.isEmpty());    ⑤
        return catNames.get(0);
    }

    public void dequeueCatName() {
        validState(!catNames.isEmpty());    ⑤
        catNames.remove(0);
    }

    public int size() {    ①
        return catNames.size();
    }
}
```

The full class CatNameList

A few more lines of code,
but a clear contract

The code is safer

Invalid data are immediately stopped

The list cannot be in an invalid state

Validation

- OWASP stresses on the importance of input validation
- Input validation is contextual
  - For the quantity of books in an order, 42 is valid, -1 is not valid
  - For a temperature, -1 is valid
  - <script>alert(42)</script> is usually invalid, but it is valid for a website to report security issues
- “Validate your input” is an advise helpful as “when driving, avoid accidents”
- We have to clarify the several kinds of validation and the order in which they have to be done
  - From the cheapest check, like the length
  - To the most expensive, those involving the database

Types of validation (to be done in this order)

- Origin
  - Are data coming from a legitimate sender?
- Size
  - Is the size reasonable?
- Lexical content
  - Does it contain only admissible characters?
- Syntax
  - Is the format correct?
- Semantic
  - Do data make sense?

An input invalid due to its length is identified with few resources

To not delegate to the database simple and cheap checks

Check for data origin

- First thing to do
- Many attacks are asymmetric in favor of the attacker
  - Sending malicious data is not so expensive for the attacker
- Prevent DoS and DDoS
  - Check IP addresses
    - For internal services, limits to a few known IPs
    - Be aware of spooffing
  - Ask for an access key to your API
    - Assign unique keys to legitim users
    - Asks to send back some token

Check for data size

```
import org.apache.commons.lang3.Validate.*;

public class ISBN {
    private final String isbn;

    public ISBN(final String isbn) {
        notNull(isbn);     ①
        inclusiveBetween(10, 10,isbn.length());
        this.isbn = isbn;     ③
    }
}
```

- Try to understand what is a reasonable size
  - It depends from the context
- Avoid to process too huge data
- An ISBN is 10 characters
  - Discard any other length
  - Do not rely only on the (subsequent) regular expression
  - What happen if you receive 1 billion characters?

Check lexical content

```
import org.apache.commons.lang3.Validate.*;

public class ISBN {
    private final String isbn;

    public ISBN(final String isbn) {
        notNull(isbn);
        inclusiveBetween(10, 10,isbn.length());
        isTrue(isbn.matches("[0-9X]*"));    ②
        this.isbn = isbn;
    }
}
```

- Verify that received characters are those permitted
- Verify that the encoding is the correct one
- An ISBN-10 contains only digits and the letter X
- Usually also dashes and spaces are accepted, but let's simplify
- More complex input require a lexical analyzer (lexer)

Check data syntax

```
import org.apache.commons.lang3.Validate.*;

public class ISBN {
    private final String isbn;

    public ISBN(final String isbn) {
        notNull(isbn);
        inclusiveBetween(10, 10,isbn.length());
        isTrue(isbn.matches("[0-9X]*"));     ①
        isTrue(isbn.matches("[0-9]{9}[0-9X]"));     ②
        isTrue(checksumValid(isbn));     ③
        this.isbn = isbn;
    }

    private boolean checksumValid(String isbn) { /.../ }
}
```

```
import org.apache.commons.lang3.Validate.*;

...

    public ISBN(final String isbn) {
        notNull(isbn);
        inclusiveBetween(10, 10,isbn.length());
        isTrue(isbn.matches("[0-9]{9}[0-9X]"));
        isTrue(checksumValid(isbn));     ②
        this.isbn = isbn;
    }
```

- Check that the several characters are in the correct place
- Usually with a regex
  - If the regex is unreadable, prefer code
- In a ISBN-10 letter X is permitted only in the last position

Lexical content and syntax are often checked together

Check data semantics

- Determine if data are consistent with respect to the state of the system
  - Do the product in the basket exist?
  - Is the payment method permitted?
- These checks are part of the model
  - If data are invalid, raise an IllegalStateException

Secure by design

```
import org.apache.commons.lang3.Validate.*;

...

    public ISBN(final String isbn) {
        notNull(isbn);
        inclusiveBetween(10, 10,isbn.length());
        isTrue(isbn.matches("[0-9]{9}[0-9X]"));
        isTrue(checksumValid(isbn));    ②
        this.isbn = isbn;
    }
```

An ISBN immutable class, validating data on construction

We can use this class without further worries on the validity of the ISBN code

Define such small bricks to promote security

We call them **domain primitives**

Reading exercises

- Understand the notion of immutability

  https://codesjava.com/immutable-class-in-java

  http://web.mit.edu/6.031/www/fa18/classes/08-immutability/
- Understand the notions of specification and exception

  http://web.mit.edu/6.031/www/fa18/classes/06-specifications/

Fine della lezione