INF226 – Software Security

Håkon Robbestad Gylterud

2019–10–16

Warm-up exercise

Is it possible that the following code outputs `Bye!`?

```
char[] message = { 'h', 'e', 'l', 'l', 'o'};

someFunction(message);

if (message[0] == 'h')
   System.out.println("Hello!");
else
   System.out.println("Bye!");
```

Warm-up exercise

What about the following code?

```
String message = "hello";

someOtherFunction(message);

if (message.equals("hello")
   System.out.println("Hello!");
else
   System.out.println("Bye!");
```

Immutability

An object is *immutable* if it cannot be changed after creation.

Example: String is an immutable class in Java.

Strings in Java

Why are strings immutable in Java?

- Because of string interning:
  - Every copy of a string is stored only once.
- Allows memoization of hashcodes (for say `HashMap`):
  - Since the string doesn't change, we never have to recompute the hashcode.
- Thread safety
- Security

State and immutability

Program state

The state of the program consists of:

- Variables
- File descriptors:
    - Files
    - Network connections
- Cookies
- Client storage

Data state vs flow control state

Some state **controls the flow** of the program:

Example: A variable `boolean authenticated` controls the flow in the statement `if(authenticated) {} else {}`

Some state is **just data** being passed aroud:

Example: A variable `String message` is usually inert, unless `null`.

Reasoning about the state of the program

Controlling and reasoning about the state of the program is essential to security.

- Security bugs often happen when a program reaches an **unanticipated state**.

Reasoning about the state of the program

Controlling and reasoning about the state of the program is essential to security.

- Security bugs often happen when a program reaches an **unanticipated state**.

Combinatorial explosion: n boolean values have 2^n possible states.

Reasoning about the state of the program

Controlling and reasoning about the state of the program is essential to security.

- Security bugs often happen when a program reaches an **unanticipated state**.

Combinatorial explosion: n boolean values have 2^n possible states.

(and in Java, n Boolean references have 3^n possible states.)

Object orientation and abstraction

An object is the combination of a (hidden) representation and (visible) interface.

Preservation of invariants: The methods of an object ensure that the internal state is a valid representation.

Object orientation and abstraction

```
class TimeIntervall {
  private Date start;
  private Date stop;
  public TimeIntervall(Date start,Date stop) {
     if(start.compareTo(stop) >= 0)
       throw new IllegalArgumentExcecption(
              "stop must be after start");
     this.start = start;
     this.stop = stop;
  }

  void setStop(Date stop) {
     if(start.compareTo(stop) >= 0)
       ...
  }
  ...
```

The problem of mutation

If you pass reference to a mutable object, you give permission to mutate the object.

The problem of mutation

If you pass reference to a mutable object, you give permission to mutate the object.

If you receive a reference to a mutable object, you must accept that it mutates beyond your control:

```
void outputHTML(Message msg) {
    if(isHTMLsafe(msg)) {
        response.print(msg);
    }
}
```

The problem of mutation

If you pass reference to a mutable object, you give permission to mutate the object.

If you receive a reference to a mutable object, you must accept that it mutates beyond your control:

```
void outputHTML(Message msg) {
    if(isHTMLsafe(msg)) {
        response.print(msg);
    }
}
```

If `Message` is mutable, we cannot know that `msg` is `HTMLsafe`! Thus, we could have a XSS if msg is changed by another thread.

Immutable objects

- Passing a reference only gives “read access”.
- When receiving a reference, you can safely test for invariants
- Thread safety for free!

Object orientation and abstraction

```
class TimeIntervall {
  private Date start;
  private Date stop;
  public TimeIntervall(Date start,Date stop) {
     if(start.compareTo(stop) >= 0)
       throw new IllegalArgumentExcecption(
              "stop must be after start");
     this.start = start;
     this.stop = stop;
  }

  void setStop(Date stop) {
     if(start.compareTo(stop) >= 0)
       ...
  }
  ...
```

Never `Date` in Java

The `Date` class is mostly deprecated and should never be used.

Never `Date` in Java

The `Date` class is mostly deprecated and should never be used.

Use `java.time.Instant` – which is better (and immutable).

Immutability in Java

The `final` keyword

The `final` keyword for variables mean:

- The reference cannot be changed after initialisation.
- Any constructor must initialise the field.

How to make immutable classes

Declaring all fields `final` is not enough:

```
final Date now = new Date();
now.setYear(2000);
```

Sufficient for:

- Strings
- primitive types
- Immutable classes

How to make immutable classes

An immutable class can hide a muteable object by:

- Keeping the only reference to this object.
- Not modify the object.
- Not providing setters.
- Declare your class final.

Important any getter for such a hidden object must make a copy of the object!

Example of an immutable class

```
final class TimeIntervall {
  public final Instant start;
  public final Instant stop;
  public TimeIntervall(Instant start, Instant stop) {
     if(start.compareTo(stop) >= 0)
       throw new IllegalArgumentExcecption(
              "stop must be after start");
     this.start = start;
     this.stop = stop;
  }
}
```

Example of an immutable class

```
final class TimeIntervall {
  public final Instant start;
  public final Instant stop;
  public TimeIntervall(Instant start, Instant stop) {
     if(start.compareTo(stop) >= 0)
       throw new IllegalArgumentExcecption(
              "stop must be after start");
     this.start = start;
     this.stop = stop;
  }
  public TimeIntervall newStop(Instant stop) {
     return new TimeIntervall(start, stop);
  }
}
```

Type algebra

Expressivity

Which types the language can express defines its **expressivity**.

- Different languages have different **expressivity**
- Rich expressivity allows:
    - more checks to be performed by type-checker
    - easier to read code
    - better code reuse

Expressivity

Common type formers:

- Parameterised types (generics)
- Record types/product types
- Sum types
- Function types
- Dependent types

An algebra of types

The types we have in programming languages can be seen as an algebra where:

- Multiplication is pairs, tuples, or structs.
- Addtion is for types where elements are from disjoint types.
- Numerals are represented by finite types. Example: `boolean` is 2.

Example: product type

```
class Message {
   public final String message;
   public final Instant timestamp;
   public Message(String message, Instant timestamp) {
      this.message = message;
      this.timestamp = timestamp;
   }
}
```

This type could be written

$\texttt{Message} = \texttt{String} \times \texttt{Instant}.$

Example: sum types

```
data Action = Say { message :: String }
            | Sleep { seconds :: Integer }
```

Here we have `Action = String + Integer`.

Example: sum type

```
data Weapon = Sword { name      :: String,
                      sharpness :: Float}
            | Bow   { name :: String
                      range :: Integer}
```

Expressed algebraically:

$\texttt{Weapon} = \texttt{String} \times \texttt{Float} + \texttt{String} \times \texttt{Integer}$

Algebraic laws

Associativity:

`A × (B × C) = (A × B) × C`

`A + (B + C) = (A + B) + C`

Commutativity:

`A × B = B × A` and `A + B = B + A`

Distributivity:

`A × (B + C) = A×B + A×C`

For types, these equalities represent refactorizations!

Example: Associativity

```
class A {
  public final String x;
  public final Float y;
}

class B {
  public final A a;
  public final Instant b;
}
```

Example: Associativity

```
class A {
  public final String x;
  public final Float y;
}

class B {
  public final A a;
  public final Instant b;
}
```

could be refactored to…

```
class C {
  public final Float a;
  public final Instant b;
}
class B {
  public final String x;
  public final C c;
}
```

$$\texttt{(String} \times \texttt{Float)} \times \texttt{Instant} = \texttt{String} \times \texttt{(Float} \times \texttt{Instant)}$$

Example: Commutativity

```
class A {
  public final String x;
  public final Float y;
}
```

Example: Commutativity

```
class A {
  public final String x;
  public final Float y;
}
```

Is the same as...

```
class A {
  public final Float y;
  public final String x;
}
```

$$\texttt{Float} \times \texttt{String} = \texttt{String} \times \texttt{Float}$$

Example: Distributivity

```
data Weapon = Sword { name      :: String,
                      sharpness :: Float}
            | Bow   { name :: String
                      range :: Integer}
```

```
Weapon = String × Float + String × Integer
       = String × (Float + Integer)
```

Sum types in Java

Any reference in Java can be `null`.

Sum types in Java

Any reference in Java can be `null`.

So every reference behaves like `A + 1` (where `1` represent the `null` value).

Sum types in Java

Any reference in Java can be `null`.

So every reference behaves like `A + 1` (where `1` represent the `null` value).

This means that if a class has two fields, say of type `A` and `B`, we get

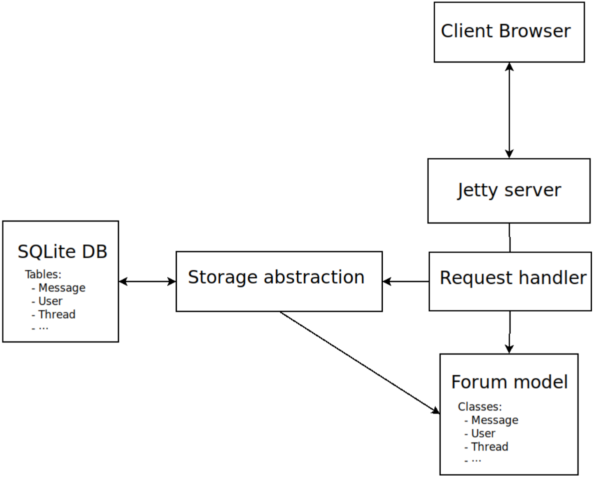
`(A + 1)`×`(B + 1) = A`×`B + A + B + 1`, where `A + B` occurs!

Sum types in java

```
public class Either<A,B> {
  private final A left;
  private final B right;

  private Either(A leftValue, B rightValue) {
     this.left = leftValue;
     this.right = rightValue;
     this.isLeft = isLeft;
  }

  public static<U,V> Either<U,V> left(U value) {
    return new Either<U,V>(value, null);
  }

  public static<U,V> Either<U,V> right(V value) {
    return new Either<U,V>(null, value);
  }

. . .
```

Inforum

Design overview



The Maybe type

`null` references

`null` is often given special meanings by functions or classes:

- No element was found (`lookup` in maps)
- No parameter was present (getting parameters from HTTP requests)
- This is a `left` value, when `right` is `null` in `Either`.

It is very easy to forget `null` checks.

`NullPointerExceptions`

`NullPointerExceptions` lead to unexpected control flows:

- When the exception is throws, execution races back up the stack.
- Can be caught by `catch (Exception ···)` clauses
- . . . or crash the thread / program.

If the program is not written carefully, these unexpected states could be insecure.

`null` gives a combinatorial explosion of states

$$\begin{aligned}\mathtt{(A + 1)}\times\mathtt{(B + 1)}\times\mathtt{(C + 1)} &= \mathtt{A}\times\mathtt{B}\times\mathtt{C} + \mathtt{A}\times\mathtt{B} + \mathtt{A}\times\mathtt{C} + \mathtt{B}\times\mathtt{C}\\ &\quad + \mathtt{A} + \mathtt{B} + \mathtt{C} + \mathtt{1}\end{aligned}$$

So, if your class has three fields, there are eight different ways the field references could be initialised with `null`!

The Maybe class

```
public class Maybe<T> {
   private final T value;

   public Maybe(T value) {
      this.value = value;
   }

   public T get() throws NothingException {
      if(value == null)
         throw new NothingException();
      else
         return value;
   }
. . .
```

The Storage and Stored types

Storage and threading

Requests are processed *concurrently*.

In order to avoid race conditions, we use version control on object going into storage. If you loose a race, an exception notifies you and you can redo with updated objects.

The stored class

The Storage interface