As motivating examples, let’s consider two programs.
Example 1: method alwaysTrue()

```java
// should always return true
public static boolean alwaysTrue(int i) {

    // Just 'return true;' is all too boring.
    // Instead:
    return ( Math.abs(i) >= 0 );
}
```
Example 1: Testing alwaysTrue()

Scanner sc = new Scanner(System.in);

while (true) {

    // read an integer from System.in
    int i = sc.nextInt();

    // this will print "true"
    System.out.println(alwaysTrue(i));
}
Example 1: Testing alwaysTrue()
Example 1: Testing alwaysTrue()

Scanner sc = new Scanner(System.in);

while (true) {

    // read an integer from System.in
    int i = sc.nextInt();

    // this will print "true"
    System.out.println(alwaysTrue(i));
}

Demo: TestAlwaysTrue.java

Surprise: with input -2147483648, the program prints false!
We want to understand the problem

Another test:
System.out.println(Math.abs(-2147483648))
prints
-2147483648

We cannot come any closer to the problem by testing/debugging.

So how can we?
We want to understand the problem

- Another test:
  System.out.println(Math.abs(-2147483648))
  prints
  -2147483648

- We cannot come any closer to the problem by testing/debugging.

- So how can we?
We want to understand the problem

Another test:
System.out.println(Math.abs(-2147483648))
prints
-2147483648

We cannot come any closer to the problem by testing/debugging.

So how can we?
From the Java API Specification, class Math:

```java
public static int abs(int a)
```

Returns the absolute value of an int value. If the argument is not negative, the argument is returned. If the argument is negative, the negation of the argument is returned.

Note that if the argument is equal to the value of `Integer.MIN_VALUE`, the most negative representable int value, the result is that same value, which is negative.
From the Java API Specification, class Math:

```java
public static int abs(int a)
```

Returns the absolute value of an int value. If the argument is not negative, the argument is returned. If the argument is negative, the negation of the argument is returned.

Note that if the argument is equal to the value of `Integer.MIN_VALUE`, the most negative representable int value, the result is that same value, which is negative.
From the Java API Specification, class Math:

```java
public static int abs(int a)
```

Returns the absolute value of an int value. If the argument is not negative, the argument is returned. If the argument is negative, the negation of the argument is returned.

Note that if the argument is equal to the value of `Integer.MIN_VALUE`, the most negative representable int value, the result is that same value, which is negative.
The problem was:

**Caller** (here `alwaysTrue()`)

had **unfulfilled expectations** about

**Callee** (here `Math.abs()`).
public class Book {

    private String title;
    private String author;
    private long isbn;

    public Book(...) { ... }
}

public boolean equals(Object other) {
    Book otherBook = (Book) other;
    return (isbn == otherBook.isbn);
}

public String toString() { ... }

(From W. Ahrendt's first-year course in OO Programming.)
Example 2: equal Objects in Sets?

*From the Java API Specification, Interface Set:*

```java
public interface Set
extends Collection

Sets contain no pair of elements e1, e2 such that e1.equals(e2) ...
...

boolean add(E e)

Adds e to this set if the set contains no element e2 such that e.equals(e2) ...
```
Example 2: equal Objects in Sets?

*From the Java API Specification, Interface Set:*

```java
public interface Set
extends Collection

Sets contain no pair of elements e1, e2 such that e1.equals(e2) ...
...

boolean add(E e)

Adds e to this set if the set contains no element e2 such that e.equals(e2) ...
```
Adding two equal books to a set:

Set catalogue = new HashSet();

Book b1 = new Book("Effective Java", "Joshua Bloch", 201310058);

Book b2 = new Book("Effective Java", "J. Bloch", 201310058);

catalogue.add(b1);
catalogue.add(b2);

How many elements has catalogue now?
Example 2: equal Objects in Sets?

Adding two equal books to a set:

Set catalogue = new HashSet();

Book b1 = new Book("Effective Java", "Joshua Bloch", 201310058);

Book b2 = new Book("Effective Java", "J. Bloch", 201310058);

catalogue.add(b1);
catalogue.add(b2);

How many elements has catalogue now?

Demo: AddTwoBooks.java
Example 2: equal Objects in Sets?

Adding two equal books to a set:

Set catalogue = new HashSet();

Book b1 = new Book("Effective Java", "Joshua Bloch", 201310058);

Book b2 = new Book("Effective Java", "J. Bloch", 201310058);

catalogue.add(b1);
catalogue.add(b2);

How many elements has catalogue now?

two! (?)
We want to understand the problem also

But again:

- We cannot come any closer to the problem by testing/debugging.
- So how can we?
We want to understand the problem also

But again:

1. **We cannot come any closer to the problem by testing/debugging.**
2. **So how can we?**
We want to understand the problem also

But again:

- We cannot come any closer to the problem by testing/debugging.
- So how can we?
Again: Specification is the Answer!

- here, specification of Set or HashSet does not reveal problem
- Instead: check the specification of Book!
- Is there any?
- Yes, because Book extends Object, and inherits the specifications from there!
Again: Specification is the Answer!

- here, specification of Set or HashSet does not reveal problem
- Instead: check the specification of Book!
  - Is there any?
  - Yes, because Book extends Object, and inherits the specifications from there!
Again: Specification is the Answer!

- here, specification of Set or HashSet does not reveal problem
- Instead: check the specification of Book!
- Is there any?
  - Yes, because Book extends Object, and inherits the specifications from there!
here, specification of Set or HashSet does not reveal problem
Instead: check the specification of Book!
Is there any?
Yes, because Book extends Object, and inherits the specifications from there!
public int hashCode()

...  
If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result.

...

By overriding equals only, and not hashCode, we broke the specification of Book::hashCode().
public int hashCode()

... If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result.

... By overriding equals only, and not hashCode, we broke the specification of Book::hashCode().
Caller and Callee disagree

The problem was:

**Caller** (here `HashSet::add()`) had **unfulfilled expectations** about **Callee** (here `Book::hashCode()`).

Here:
The caller is library code, the callee is a method from our own class!

⇒ Call Back Mechanism in OO Programming
Caller and Callee disagree

The problem was:

**Caller** (here `HashSet::add()`) had **unfulfilled expectations** about **Callee** (here `Book::hashCode()`).

Here:
The caller is library code, the callee is a method from our own class!

⇒ Call Back Mechanism in OO Programming
Caller and Callee disagree

The problem was:

Caller (here HashSet::add())
had unfulfilled expectations about
Callee (here Book::hashCode()).

Here:
The caller is library code, the callee is a method from our own class!

⇒ Call Back Mechanism in OO Programming
Is this nasty?

- How could the implementer of Book foresee whether some class implementing Set would call Book::hashCode()?
- He/she cannot!

No alternative to fulfilling the inherited specification of Object, as potential callers might rely on it in unforeseeable ways!

Demo: fixing AddTwoBooks.java
Is this nasty?

- How could the implementer of Book foresee whether some class implementing Set would call Book::hashCode()?
- He/she cannot!

No alternative to fulfilling the inherited specification of Object, as potential callers might rely on it in unforeseeable ways!

Demo: fixing AddTwoBooks.java
Is this nasty?

- How could the implementer of Book foresee whether some class implementing Set would call Book::hashCode()?
- He/she cannot!

No alternative to fulfilling the inherited specification of Object, as potential callers might rely on it in unforeseeable ways!

Demo: fixing AddTwoBooks.java
In both cases: **Caller** had unfulfilled expectations about **callee**.

Difference: who is to blame?

**Example 1:** the **caller** (*alwaysTrue()*)

**Example 2:** the **callee** (*Book::hashCode()*)

We will focus on a crystal clear distinction
- of these **different roles**, and
- the **different obligations** attached to either of the roles.
Example 1/2: Similarities and Differences

In both cases:

**Caller** had unfulfilled expectations about **callee**.

Difference: who is to blame?

**Example 1:** the **caller** (alwaysTrue())

**Example 2:** the **callee** (Book::hashCode())

We will focus on a crystal clear distinction

- of these **different roles**, and
- the **different obligations** attached to either of the roles.
Example 1/2: Similarities and Differences

In both cases:
**Caller** had unfulfilled expectations about **callee**.

Difference: who is to blame?

**Example 1:** the **caller** \((\text{alwaysTrue}())\)

**Example 2:** the **callee** \((\text{Book::hashCode}())\)

We will focus on a crystal clear distinction
- of these **different roles**, and
- the **different obligations** attached to either of the roles.
Specifications as Contracts

to stress the different roles – obligations – responsibilities in a specification:

widely used analogy of the specification as a contract

“Design by Contract” methodology
What kind of Specifications

System level specifications (requirements analysis, GUI, use cases, performance) important, but *not subject of this course*.

instead:

*unit specification*—contracts among implementers on various levels:

- application level ↔ application level
- application level ↔ library level
- library level ↔ library level
What kind of Specifications

System level specifications (requirements analysis, GUI, use cases, performance) important, but *not subject of this course*.

instead:

unit specification—contracts among implementers on various levels:

- application level ↔ application level
- application level ↔ library level
- library level ↔ library level
What kind of Specifications

System level specifications (requirements analysis, GUI, use cases, performance) important, but *not subject of this course*.

instead:

**unit specification**—contracts among implementers on various levels:

- application level ↔ application level
- application level ↔ library level
- library level ↔ library level
What kind of Specifications

Natural language specs are very important (see the examples above).

Still:
we focus on

“formal” specifications:
Describing contracts of units in a mathematically precise language.

Motivation:

- higher degree of precision.
- eventually: automation of program analysis of various kinds:
  - static checking
  - program verification
What kind of Specifications

Natural language specs are very important (see the examples above).

Still: we focus on

“formal” specifications:

Describing contracts of units in a mathematically precise language.

Motivation:

- higher degree of precision.
- eventually: automation of program analysis of various kinds:
  - static checking
  - program verification
What kind of Specifications

Natural language specs are very important (see the examples above).

Still:
we focus on

“formal” specifications:
Describing contracts of units in a mathematically precise language.

Motivation:

- higher degree of precision.
- eventually: automation of program analysis of various kinds:
  - static checking
  - program verification
What kind of Specifications

Natural language specs are very important (see the examples above).

Still:
we focus on

“formal” specifications:
Describing contracts of units in a mathematically precise language.

Motivation:

- higher degree of precision.
- eventually: automation of program analysis of various kinds:
  - static checking
  - program verification