ESOF 422:

Advanced Software Engineering: Cyber Practices

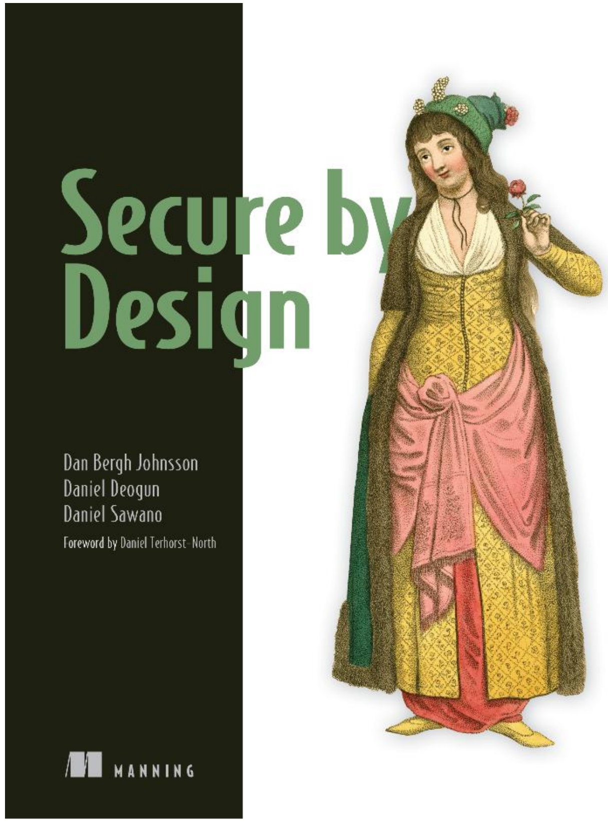
Secure by Design (Part 1)

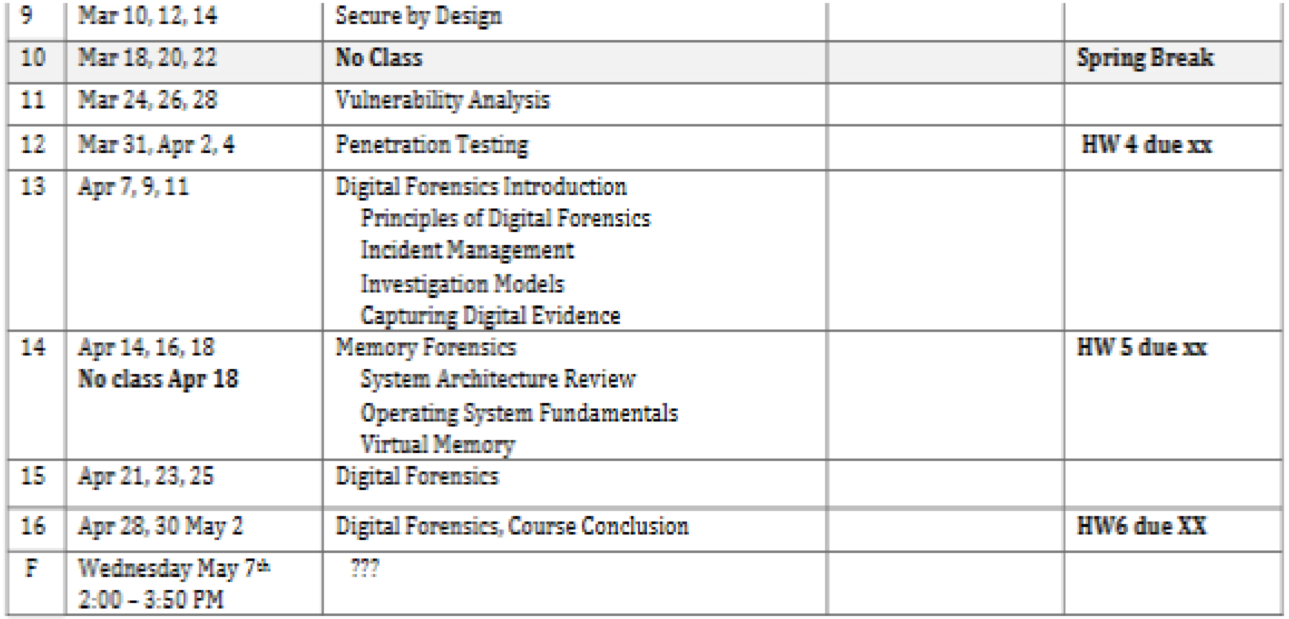
Domain-Driven Development, Domain Primitives, Input Validation

Reese Pearsall

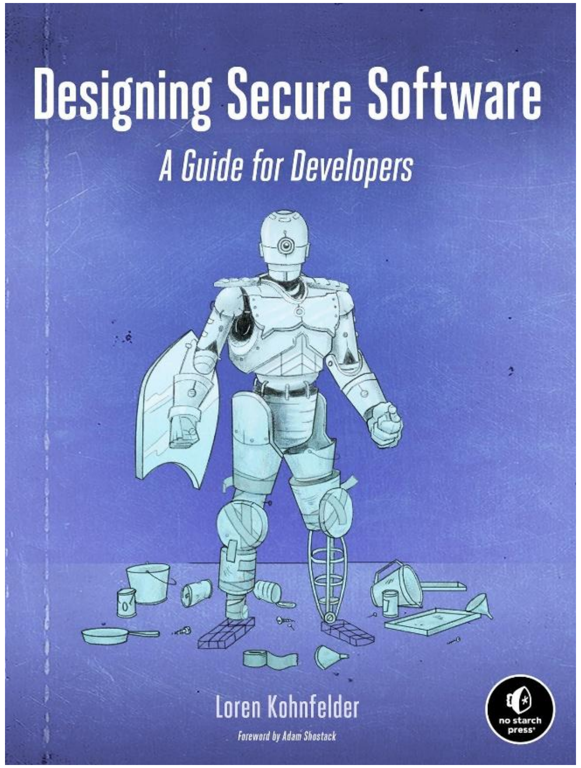
Spring 2025

Upcoming Schedule





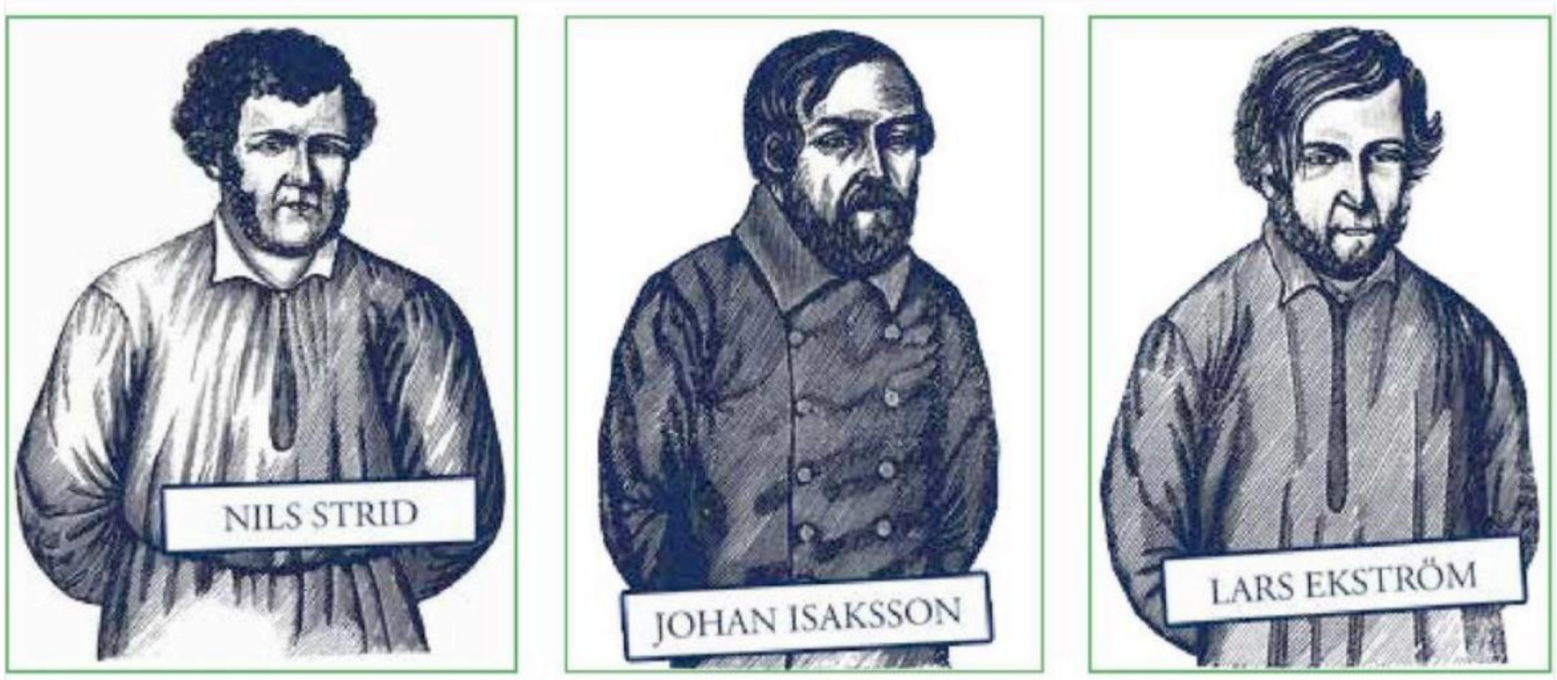
9	Mar 10, 12, 14	Secure by Design		
10	Mar 18, 20, 22	**No Class**		**Spring Break**
11	Mar 24, 26, 28	Vulnerability Analysis		
12	Mar 31, Apr 2, 4	Penetration Testing		**HW 4 due xx**
13	Apr 7, 9, 11	Digital Forensics Introduction Principles of Digital Forensics Incident Management Investigation Models Capturing Digital Evidence		
14	Apr 14, 16, 18 **No class Apr 18**	Memory Forensics System Architecture Review Operating System Fundamentals Virtual Memory		**HW 5 due xx**
15	Apr 21, 23, 25	Digital Forensics		
16	Apr 28, 30 May 2	Digital Forensics, Course Conclusion		**HW6 due XX**
F	Wednesday May 7th 2:00 – 3:50 PM	???		



…things may change

Exam 3?

Bank Robbery in Sweden



1854- Three Swedish men rob the *Öst-Götha Bank* and stole $5,000,000 -$10,000,000

The bank had secure locks, high-quality hinges, but the key to open the lock was hidden above the door frame

1855- new laws mandating a certain level of bank security

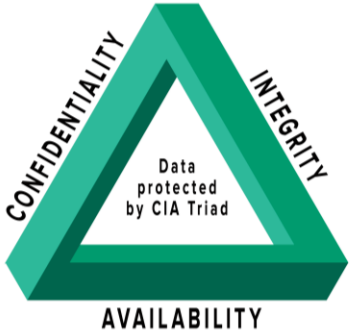
Bank Robbery in Sweden

Security **Features**:

- Locks
- Strong Hinges

Features may give the perception that security is happening, but they will fail if implemented incorrectly (**design**)

Security is a concern, not a feature



A secure design ensures features are implemented correctly

Design as a list of features

These are often things adding during the software development process, after the design is decided

- Have a log in page with a password
- Use SHA-512 Hashing
- Passwords must be more than 6 characters

… this could be a very long list

Business logic usually never addresses security

Design as a concern

"Only allow authorized users to access their files"

Yes, achieving this will involve many features, but we are at least addressing security during the design step

Concerns fall under three types of attributes

1. **Confidentiality** – Preventing data from being disclosed to wrong people
2. **Integrity-** Preventing data from being tampered with or corrupted
3. **Availability-** Preventing services from being taken offline

Secure by Design

Software engineers are not security experts, and engineers are often not thinking about security while writing code

- If they are, security is usually a second priority

Secure by design is a **mindset** that emphasizes strong principles and practices for writing secure code, that result in security being a natural outcome

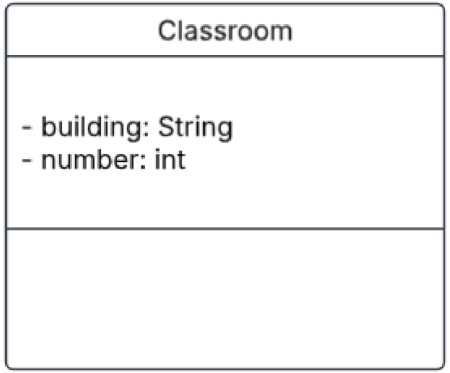
- Good **design** can yield a lot of security benefits

It is a **concern** that must be addresses at every level of the system (Code, OS, Network, Physical)

Disclaimer: This is not the 100% totally correct way to create software, but rather this is a specific lens to view the software development process through

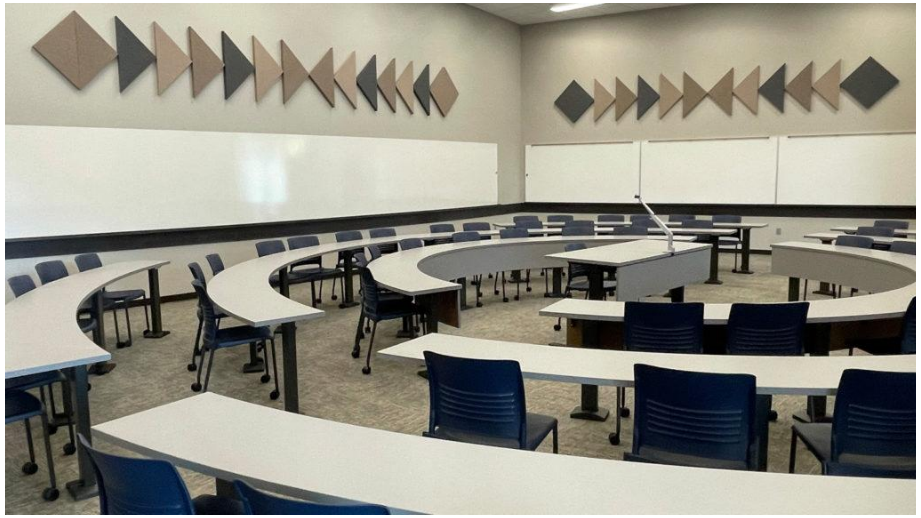
Modeling a Classroom in Code

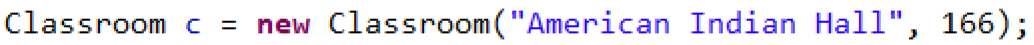
```
public class Classroom {

    private String building;
    private int number;

}
```



Classroom

- building: String
- number: int

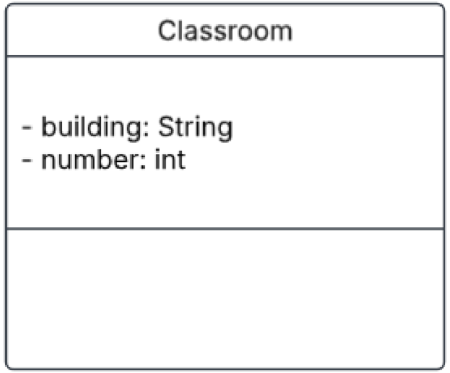




```
Classroom c = new Classroom("American Indian Hall", 166);
```

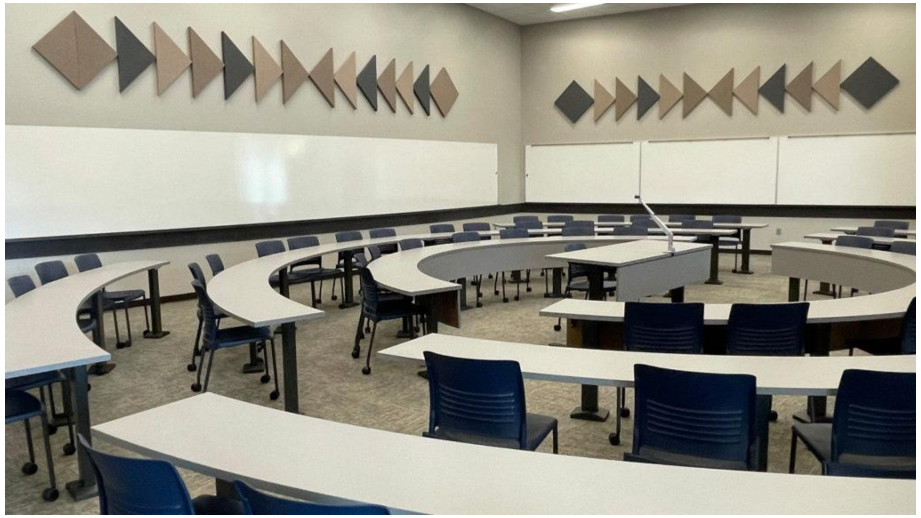
Modeling a Classroom in Code

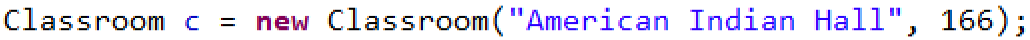
```java
public class Classroom {

    private String building;
    private int number;

}
```



Classroom

- building: String
- number: int





```java
Classroom c = new Classroom("American Indian Hall", 166);
```

number is an integer. According to math, integers have the following properties:

- Closure: a + b = integer
- Associativity: a + (b + c) = (a + b) + c
- Commutativity: a + b = b + a
- Identity: a + 0 = a
- Inverse: a + (−a) = 0

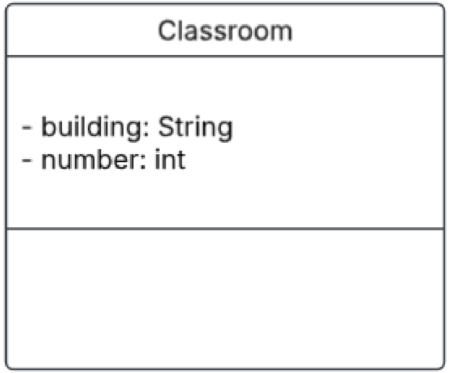
-20, 0, 20 are all integers

Peano's Axioms (Natural Numbers)

1. Zero is a number
2. If n is a number, the successor of n is a number
3. Zero isn’t the successor of a number
4. Two numbers of which the successors are equal are themselves equal
5. If a set S of numbers contains zero and also the successor of every number in S, then every number is in S

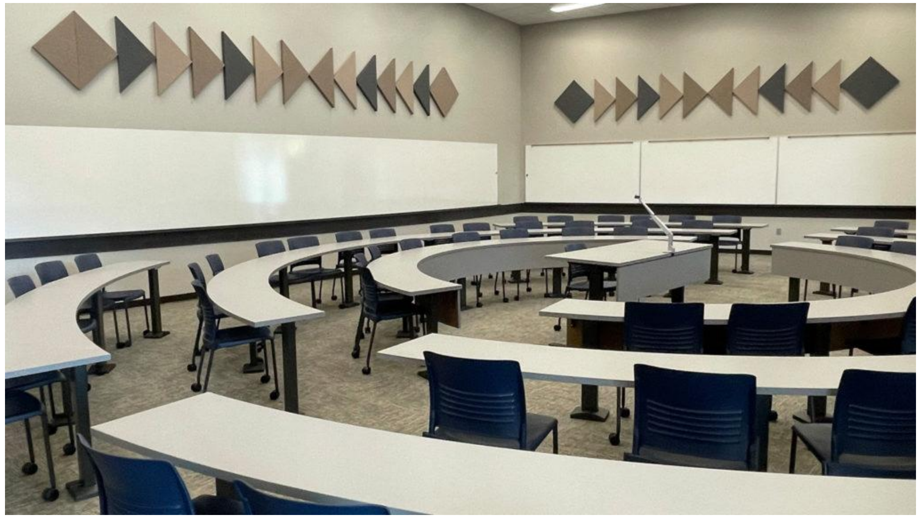
Modeling a Classroom in Code

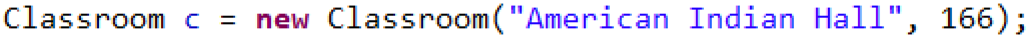
```
public class Classroom {

    private String building;
    private int number;

}
```



Classroom

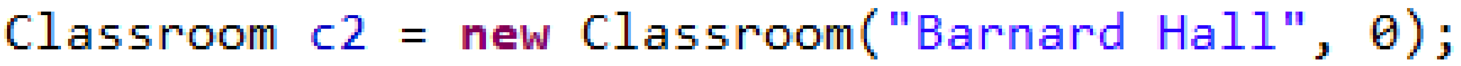
- building: String
- number: int





```
Classroom c = new Classroom("American Indian Hall", 166);
```

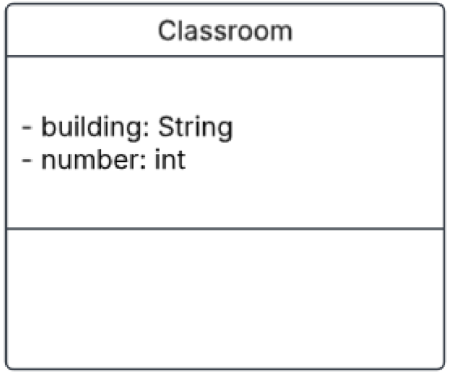
Is this a “valid” classroom ?



```
Classroom c2 = new Classroom("Barnard Hall", 0);
```

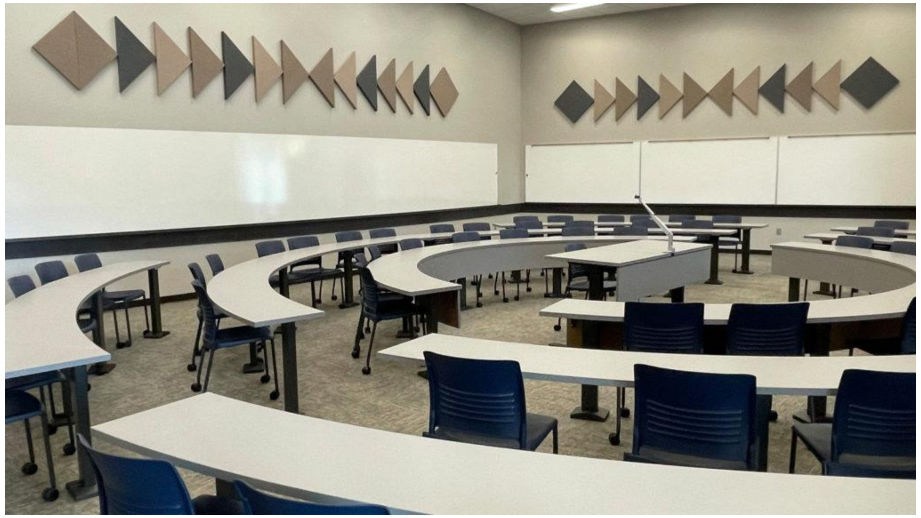
Modeling a Classroom in Code

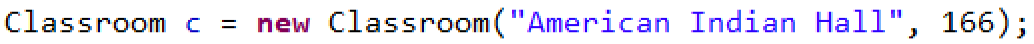
```java
public class Classroom {

    private String building;
    private int number;

}
```



Classroom

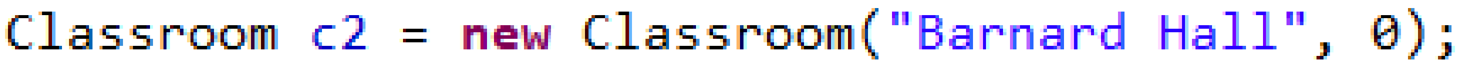
- building: String
- number: int





```java
Classroom c = new Classroom("American Indian Hall", 166);
```

Is this a “valid” classroom ?

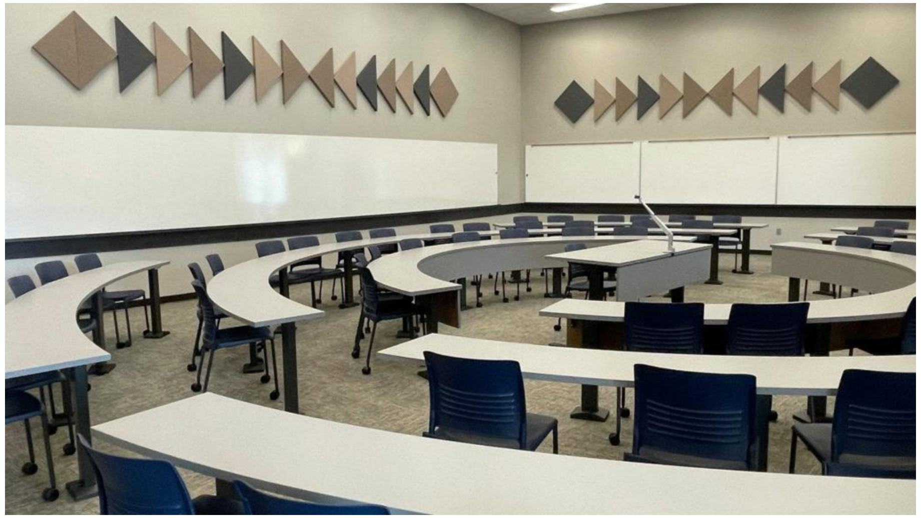


```java
Classroom c2 = new Classroom("Barnard Hall", 0);
```

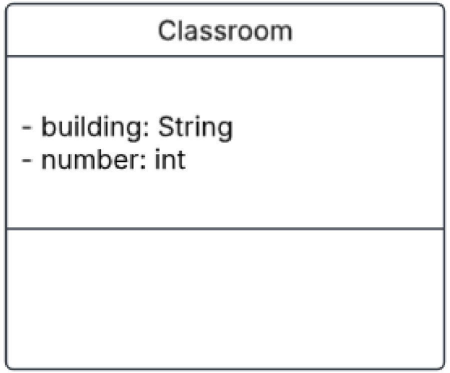
In the context of **Java Integers**, this is valid !

In the context of a **Building/Classroom**, this is not valid !

Modeling a Classroom in Code



```
public class Classroom {

    private String building;
    private int number;

}
```



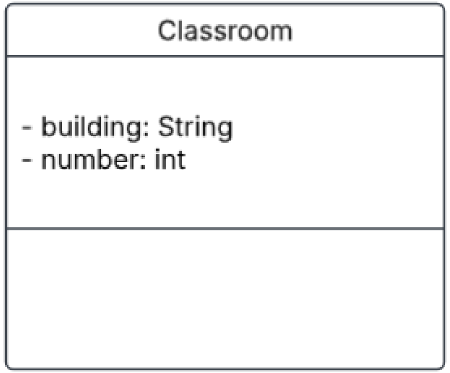
```
Classroom c = new Classroom("American Indian Hall", 166);
```

Is this a “valid” classroom ?

```
Classroom c3 = new Classroom("Reid Hall", -202);
```

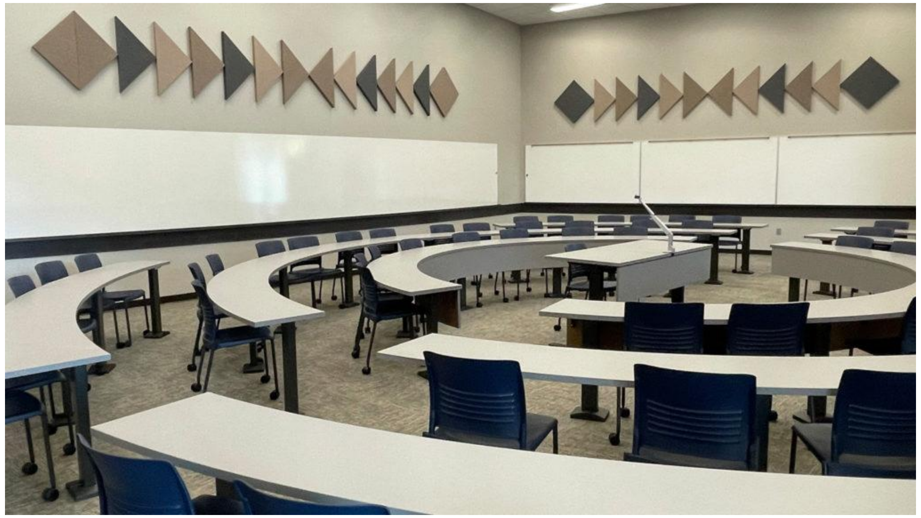
Modeling a Classroom in Code

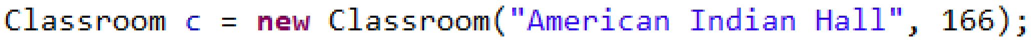
```
public class Classroom {

    private String building;
    private int number;

}
```



Classroom

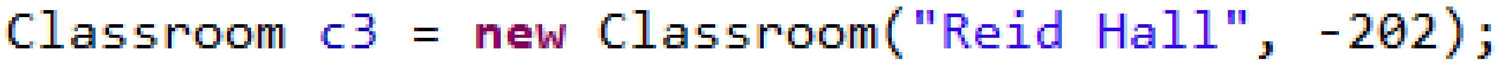
- building: String
- number: int





```
Classroom c = new Classroom("American Indian Hall", 166);
```

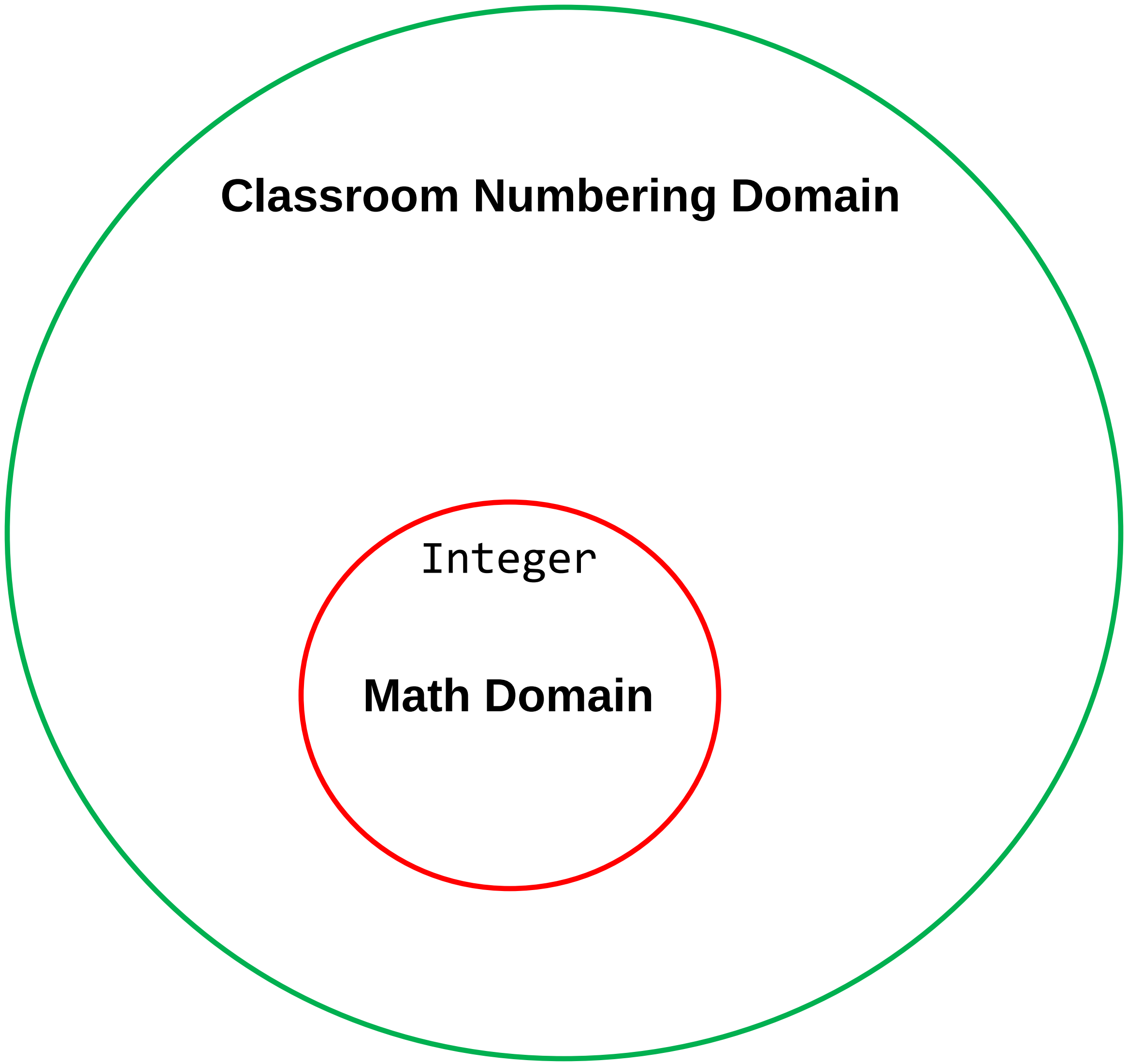
Is this a “valid” classroom ?



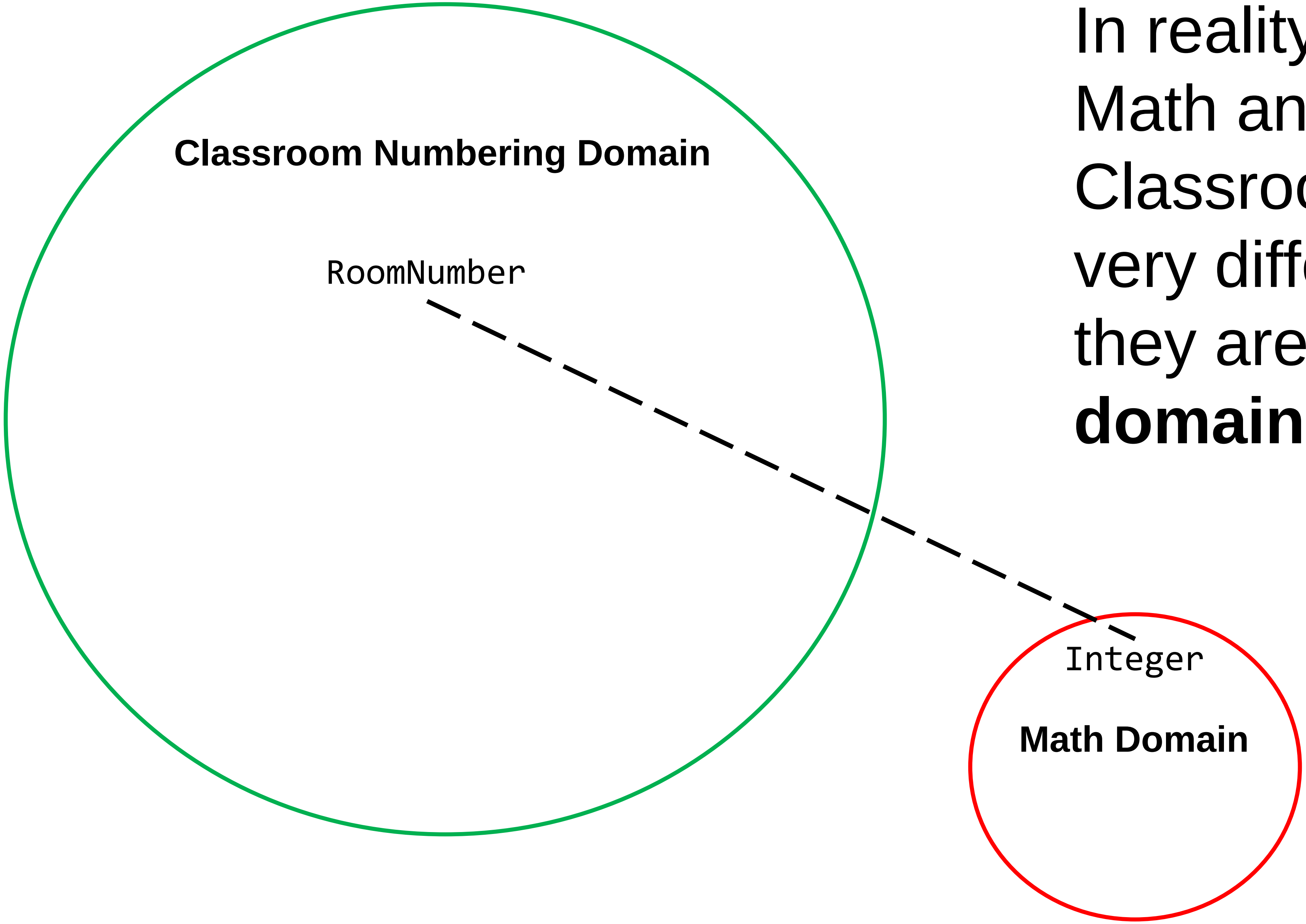
```
Classroom c3 = new Classroom("Reid Hall", -202);
```

In the context of **Java Integers**, this is valid !

In the context of a **Building/Classroom**, this is not valid !



Our current design assumes that the **mathematical** definition for an integer suffices for the context of **Classroom Numbering**



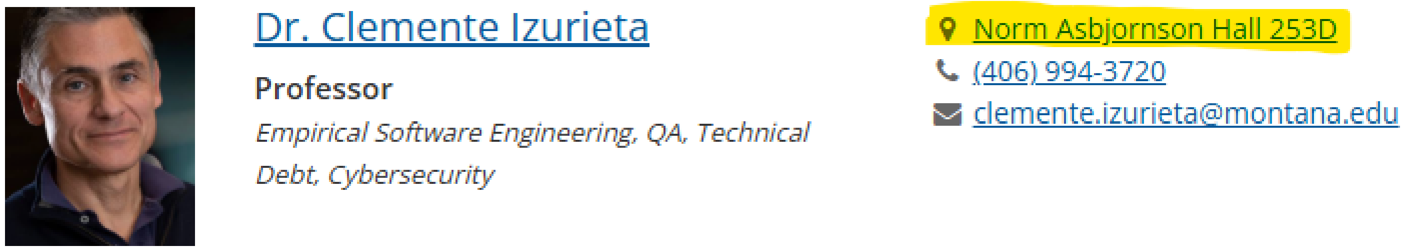
In reality, an integer in Math and an integer in Classroom numbering are very different, because they are two very different **domain**

Modeling a Classroom in Code

```java
public class Classroom {

    private String building;
    private ??? number;

}
```

Modeling a Classroom in Code

```
public class Classroom {

    private String building;
    private String number;

}
```

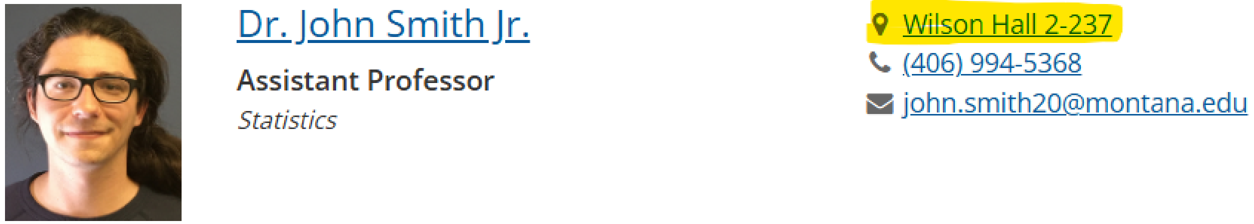


Dr. Clemente Izurieta

Professor

Empirical Software Engineering, QA, Technical Debt, Cybersecurity

📍 Norm Asbjornson Hall 253D
☎ (406) 994-3720
✉ clemente.izurieta@montana.edu



Dr. John Smith Jr.

Assistant Professor

Statistics

📍 Wilson Hall 2-237
☎ (406) 994-5368
✉ john.smith20@montana.edu

(these aren't "classrooms", but their may be a room number that includes a letter or a "-" !)

How about a `String` ?

A String can represent almost anything

Modeling a Classroom in Code

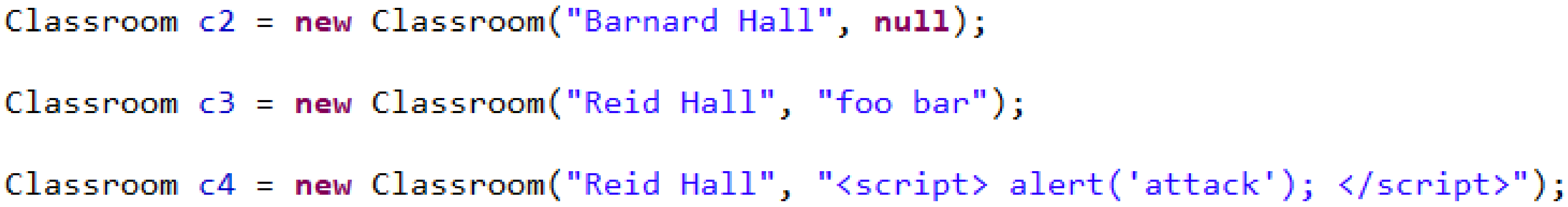
```
public class Classroom {

    private String building;
    private String number;

}
```

How about a `String` ?

A String can represent almost anything

- What characters are valid ?
- What operations are allowed ?
- Does it make sense ?

`String` encompasses a lot more possible inputs, which could lead to some strange, invalid objects

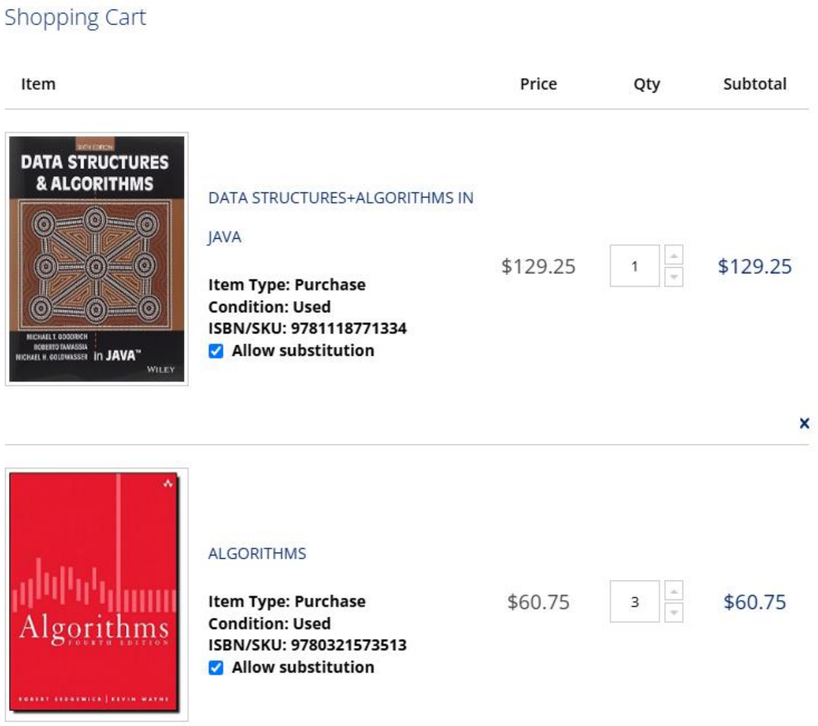


```
Classroom c2 = new Classroom("Barnard Hall", null);

Classroom c3 = new Classroom("Reid Hall", "foo bar");

Classroom c4 = new Classroom("Reid Hall", "<script> alert('attack'); </script>");
```

Web Shop

```
public class BookOrder {

    private String title;
    private String isbn;
    private int quantity;

}
```

```
public class Cart {

    private List<BookOrder> cart;
    private String customer_email;

}
```



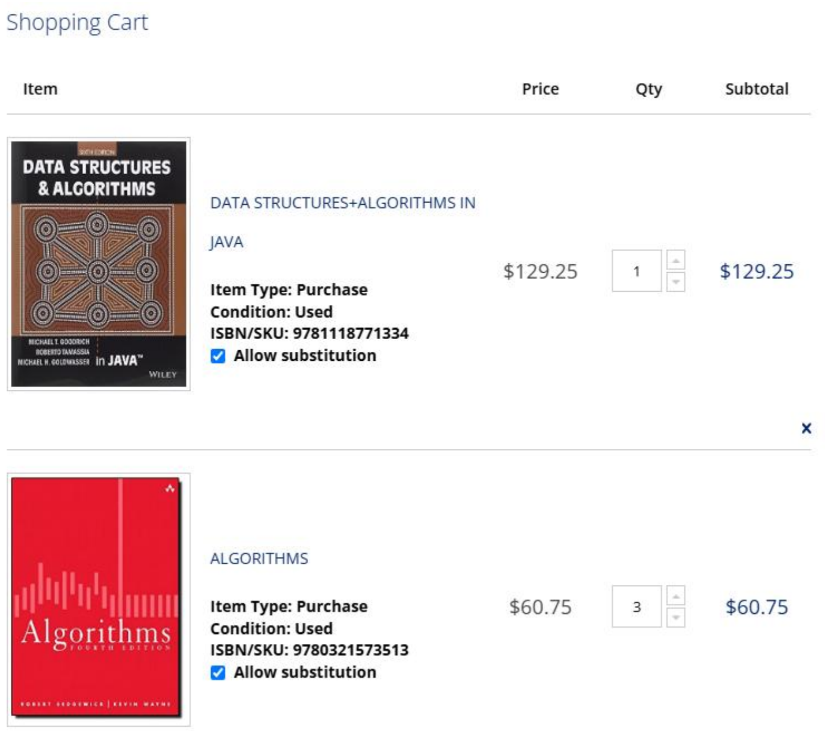
Shopping Cart

Item	Price	Qty	Subtotal
DATA STRUCTURES+ALGORITHMS IN JAVA Item Type: Purchase Condition: Used ISBN/SKU: 9781118771334 ☑ Allow substitution	$129.25	1	$129.25
ALGORITHMS Item Type: Purchase Condition: Used ISBN/SKU: 9780321573513 ☑ Allow substitution	$60.75	3	$60.75

Web Shop

```
public class BookOrder {

   private String title;
   private String isbn;
   private int quantity;

}
```

```
public class Cart {

   private List<BookOrder> cart;
   private String customer_email;

}
```



Shopping Cart

Item	Price	Qty	Subtotal
DATA STRUCTURES+ALGORITHMS IN JAVA **Item Type: Purchase** **Condition: Used** **ISBN/SKU: 9781118771334** ☑ **Allow substitution**	$129.25	1	$129.25
ALGORITHMS **Item Type: Purchase** **Condition: Used** **ISBN/SKU: 9780321573513** ☑ **Allow substitution**	$60.75	3	$60.75

`quantity` is represented by a Java `int`

Possible int values: -2 billion to 2 billion

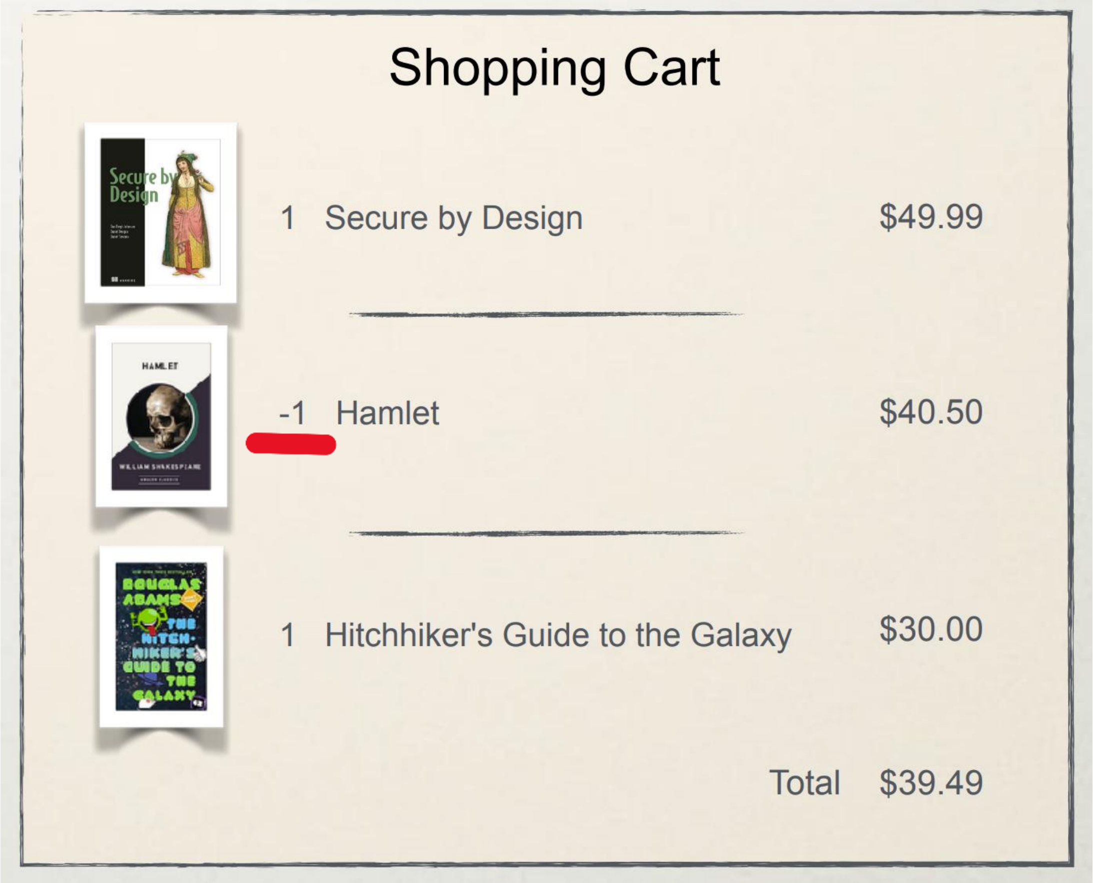
(This is rarely a good representation of anything in the real world)

Web Shop

```java
public class BookOrder {

    private String title;
    private String isbn;
    private int quantity;

}
```

-1 is a valid integer

We expect users to enter a positive integer, but there is nothing that is stopping a user from setting a quantity of one

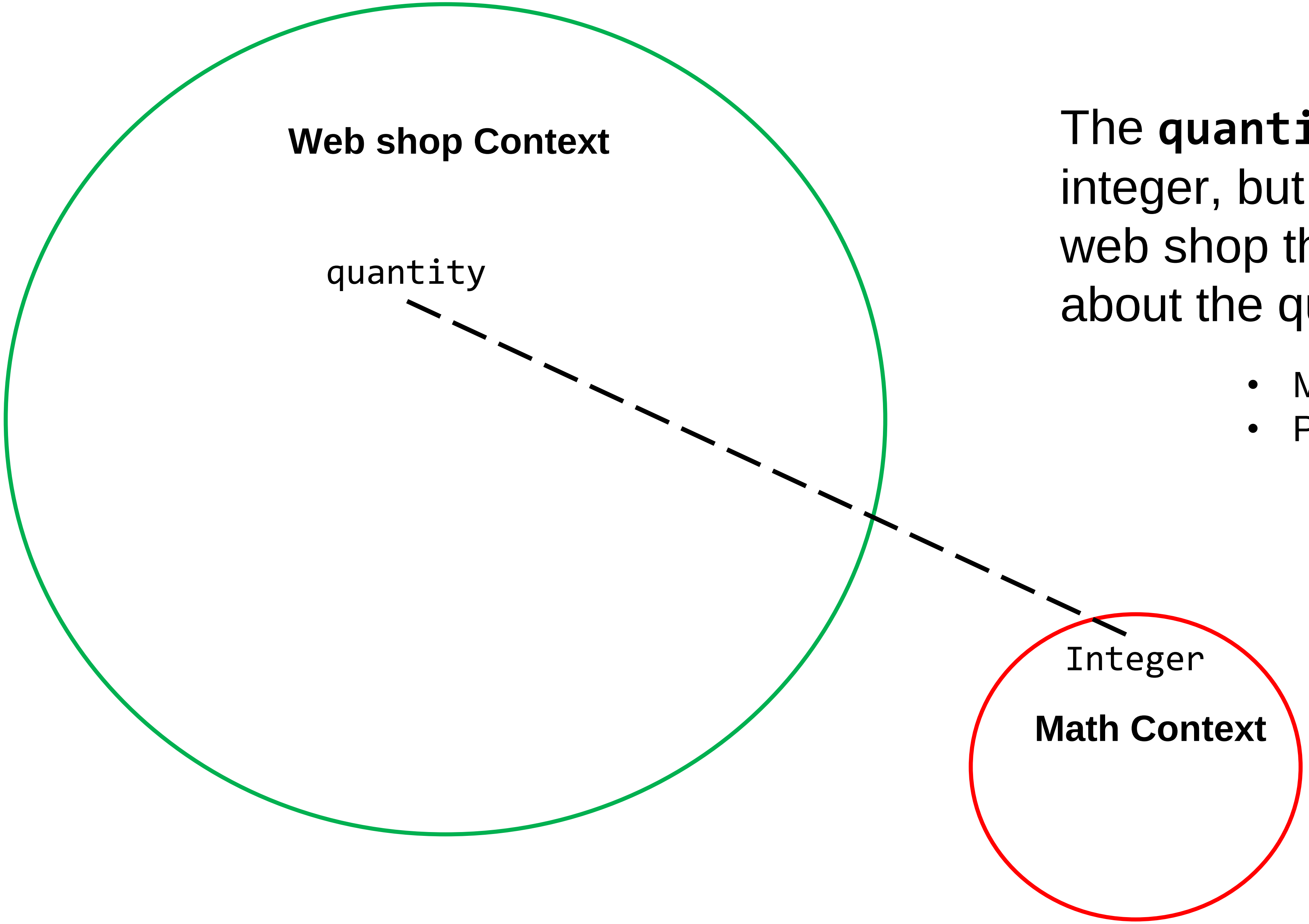


Shopping Cart

1	Secure by Design	$49.99
-1	Hamlet	$40.50
1	Hitchhiker's Guide to the Galaxy	$30.00
	Total	$39.49

```java
BookOrder my_order = new BookOrder("Java Programming", "1260440230", -1);
```

This could lead to the web shop paying someone the price instead of the customer paying them the price



The **`quantity`** of an order is an integer, but within the context of a web shop there are implicit rules about the quantity of an order

- Must be greater than 0
- Probably less than 99

```java
public class BookOrder {

   private String title;
   private String isbn;
   private int quantity;

}
```

Refactor

```java
public class BookOrder {

   private String title;
   private String isbn;
   private Quantity quantity;

}
```

```java
public class BookOrder {

   private String title;
   private String isbn;
   private int quantity;

}
```

Refactor

```java
public class BookOrder {

   private String title;
   private String isbn;
   private Quantity quantity;

}
```

We create our own implementation of what Quantity means within our domain

Our **implicit** rules for what makes a valid quantity is now **explicit**

```java
public class Quantity {
   private final int value;
   public Quantity(final int value) throws Exception {
      if(!inclusiveBetween(1,99)){
         throw new Exception("Invalid Quantity");
      }
      this.value = value;
   }
}
```

```java
public class BookOrder {

   private String title;
   private String isbn;
   private int quantity;

}
```

Refactor

```java
public class BookOrder {

   private String title;
   private String isbn;
   private Quantity quantity;

}
```

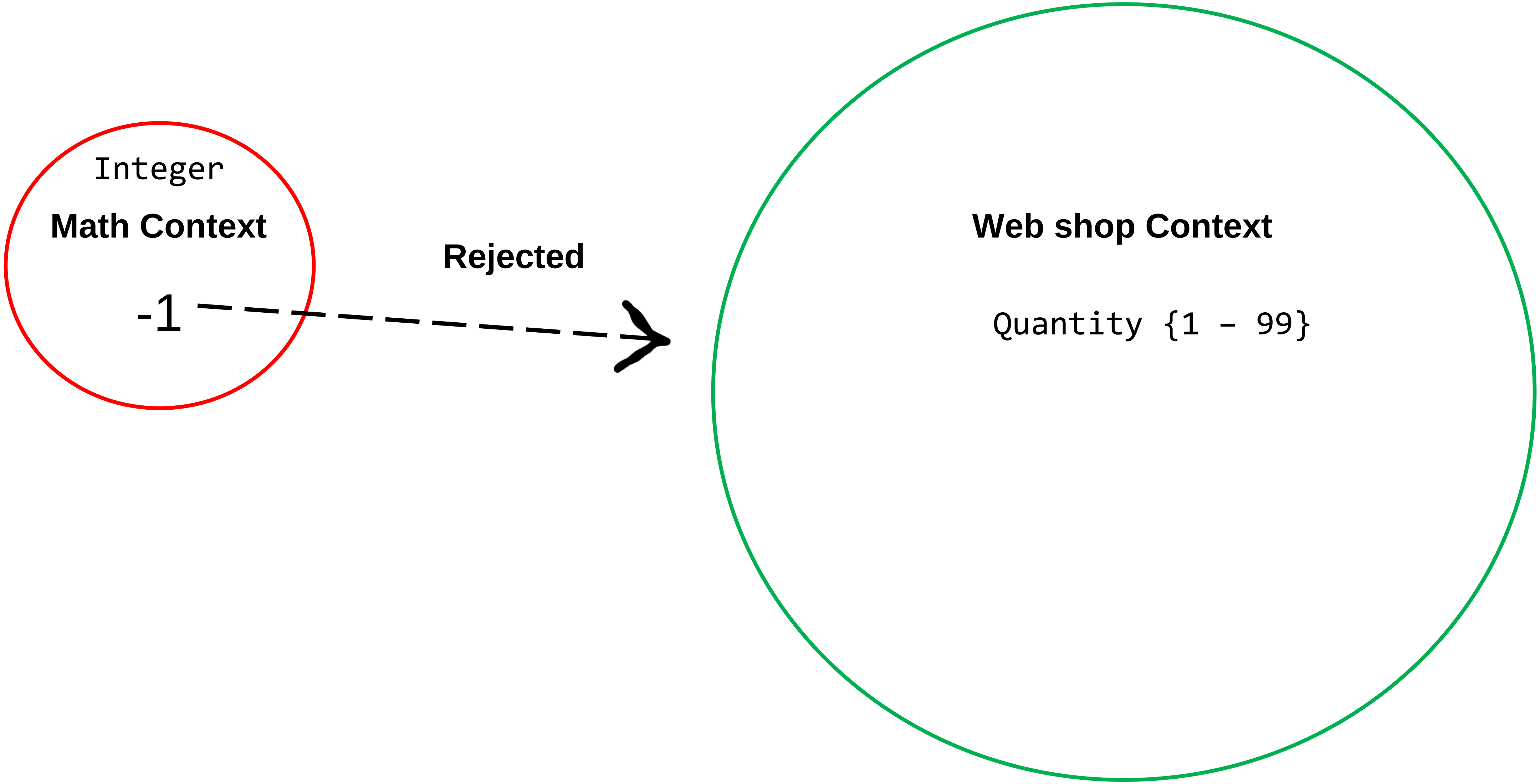
We create our own implementation of what Quantity means within our domain

```java
public class Quantity {
   private final int value;
   public Quantity(final int value) throws Exception {
      if(!inclusiveBetween(1,99)){
         throw new Exception("Invalid Quantity");
      }
      this.value = value;
   }
}
```

Our **implicit** rules for what makes a valid quantity is now **explicit**

```java
BookOrder my_order = new BookOrder("Java Programming", "1260440230", -1);
```

Now Rejected !



`Integer`

Math Context

-1

Rejected

Web shop Context

`Quantity {1 - 99}`

```java
public class BookOrder {

   private String title;
   private String isbn;
   private Quantity quantity;

}
```

`Quantity` is now a **domain primitive**

Domain Primitive: A value object so precise in its definition that it, by its mere existence, manifests its validity is called a Domain primitive

- Can only exist if its value is valid
- Building block that’s native to your domain
- Valid in the current context
- Immutable

```java
public class Quantity {
   private final int value;
   public Quantity(final int value) throws Exception {
      if(!inclusiveBetween(1,99)){
         throw new Exception("Invalid Quantity");
      }
      this.value = value;
   }
}
```

`Quantity` is now a **domain primitive**

```java
public class BookOrder {

   private String title;
   private String isbn;
   private Quantity quantity;

}
```

Domain Primitive: A value object so precise in its definition that it, by its mere existence, manifests its validity is called a Domain primitive

- Can only exist if its value is valid
- Building block that’s native to your domain
- Valid in the current context
- Immutable

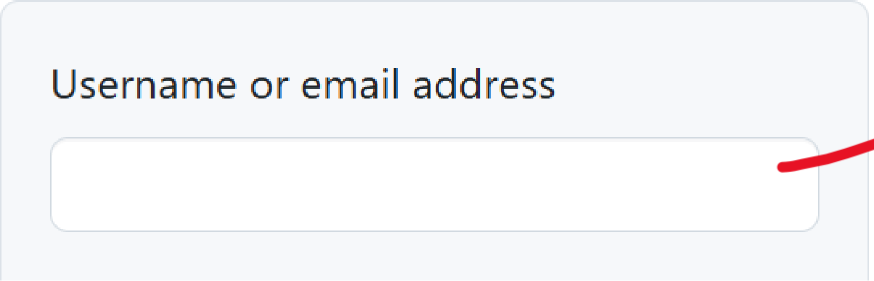
```java
public class Quantity {
   private final int value;
   public Quantity(final int value) throws Exception {
      if(!inclusiveBetween(1,99)){
         throw new Exception("Invalid Quantity");
      }
      this.value = value;
   }
}
```

Domain primitive enforce domain rule validation at creation time

Tightens our design by explicitly stating requirements and assumptions

Deeper modeling

```
public class Cart {

   private List<BookOrder> cart;
   private String customer_email;

}
```



Username or email address

Let’s think about email…

It makes sense for an email to be a String

There are a lot of *funky* strings though…

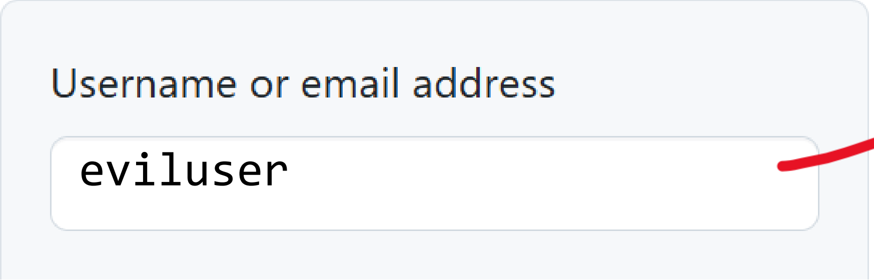
```java
public class Cart {

   private List<BookOrder> cart;
   private String customer_email;

}
```

Let’s think about email…

It makes sense for an email to be a String

There are a lot of *funky* strings though…

This is untrusted user input



Username or email address

eviluser

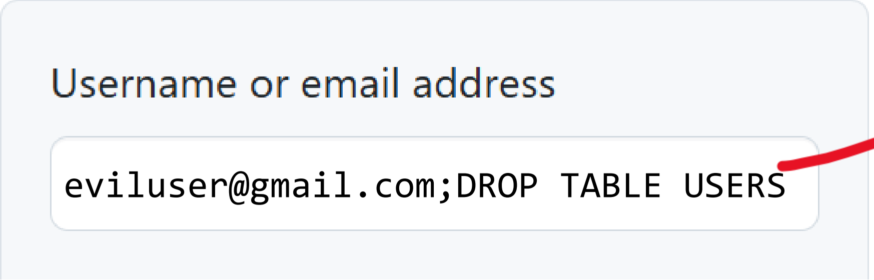
```
public class Cart {

   private List<BookOrder> cart;
   private String customer_email;

}
```

Let’s think about email…

It makes sense for an email to be a String

There are a lot of *funky* strings though…

This is untrusted user input



Username or email address

eviluser@gmail.com;DROP TABLE USERS

A user could provide a rather unexpected user input

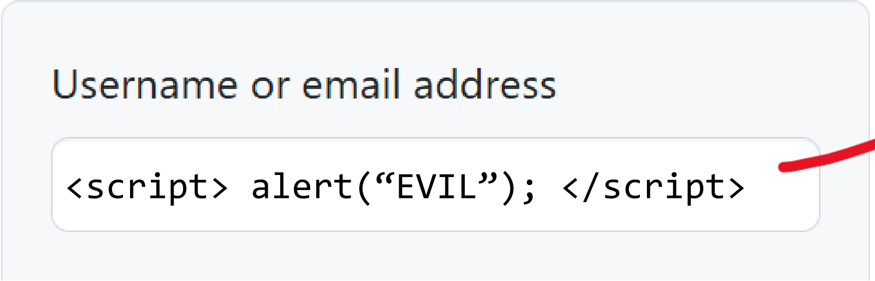
```java
public class Cart {

   private List<BookOrder> cart;
   private String customer_email;

}
```

Let’s think about email…

It makes sense for an email to be a String

There are a lot of *funky* strings though…

This is untrusted user input



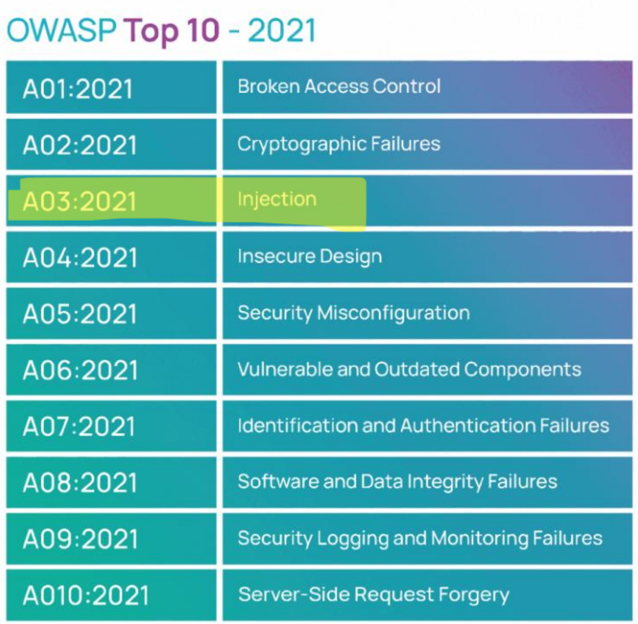
Username or email address

```
<script> alert(“EVIL”); </script>
```

A user could provide a rather unexpected user input

Cross-Site-Scripting (XSS) Attack:

Code injection attack where victim executes JavaScript that was injected by attacker



OWASP Top 10 - 2021

A01:2021	Broken Access Control
A02:2021	Cryptographic Failures
A03:2021	Injection
A04:2021	Insecure Design
A05:2021	Security Misconfiguration
A06:2021	Vulnerable and Outdated Components
A07:2021	Identification and Authentication Failures
A08:2021	Software and Data Integrity Failures
A09:2021	Security Logging and Monitoring Failures
A010:2021	Server-Side Request Forgery

XSS and code injections are still a very common and severe in today’s world

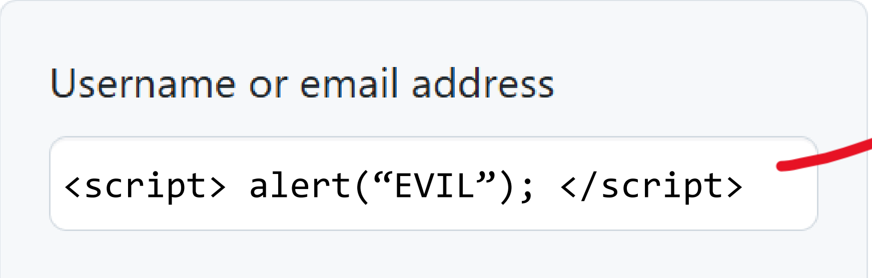
```
public class Cart {

   private List<BookOrder> cart;
   private String customer_email;

}
```

Let’s think about email…

It makes sense for an email to be a String

There are a lot of *funky* strings though…

This is untrusted user input



Username or email address

```
<script> alert(“EVIL”); </script>
```

A user could provide a rather unexpected user input

Unexpected Input can lead to unexpected behavior.
Unexpected behavior can lead to security vulnerabilities

```
public class Cart {

    private List<BookOrder> cart;
    private String customer_email;

}
```

A hacker looks at this

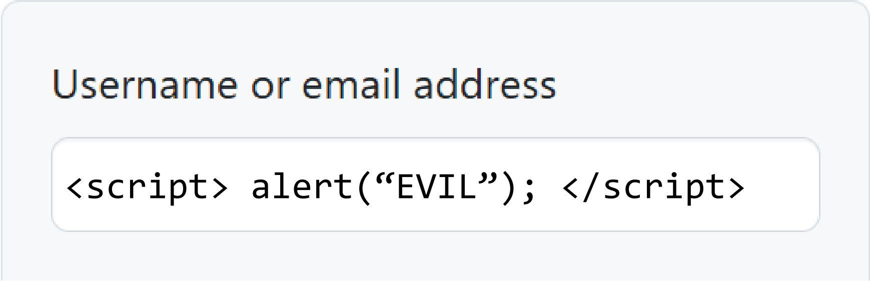
A programmer looks at this

Let’s think about email…

It makes sense for an email to be a String

There are a lot of *funky* strings though…

This is untrusted user input



Username or email address

```
<script> alert(“EVIL”); </script>
```

A user could provide a rather unexpected user input

Unexpected Input can lead to unexpected behavior.
Unexpected behavior can lead to security vulnerabilities

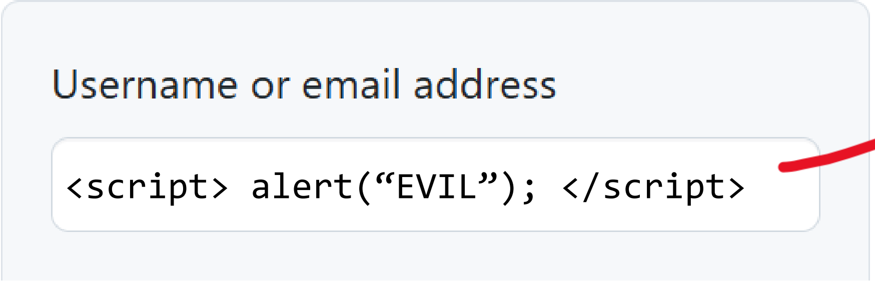
```java
public class Cart {

   private List<BookOrder> cart;
   private String customer_email;

}
```

Let’s think about email…

It makes sense for an email to be a String

There are a lot of *funky* strings though…

This is untrusted user input



Username or email address

<script> alert(“EVIL”); </script>

A user could provide a rather unexpected user input

Treat any area of untrusted user input as possible malicious input

```java
public class Cart {

   private List<BookOrder> cart;
   private String customer_email;

}
```

Refactor email to be a domain primitive

```java
public class Cart {

   private List<BookOrder> cart;
   private Email customer_email;

}
```

```java
public class Email {
   private final String value;
   public Email(final String value) throws Exception {
      if(!isValidEmail(value)) {
         throw new Exception("Invalid email");
      }
      this.value = value;
   }
   private boolean isValidEmail(String value) {
      String regex = "^[a-zA-Z0-9]+@[a-zA-Z0-9]+$";
      Pattern pattern = Pattern.compile(regex);
      Matcher matcher = pattern.matcher(value);
      return matcher.matches();
   }
}
```

Ensures that only valid objects are created

Logic for checking of valid email address for our web shop domain*

Secure Design and Domain Primitives

The `Email` primitive enforce domain rule validation at creation time.

This reduced the attack vector to data that meets the rules in the context where it’s used

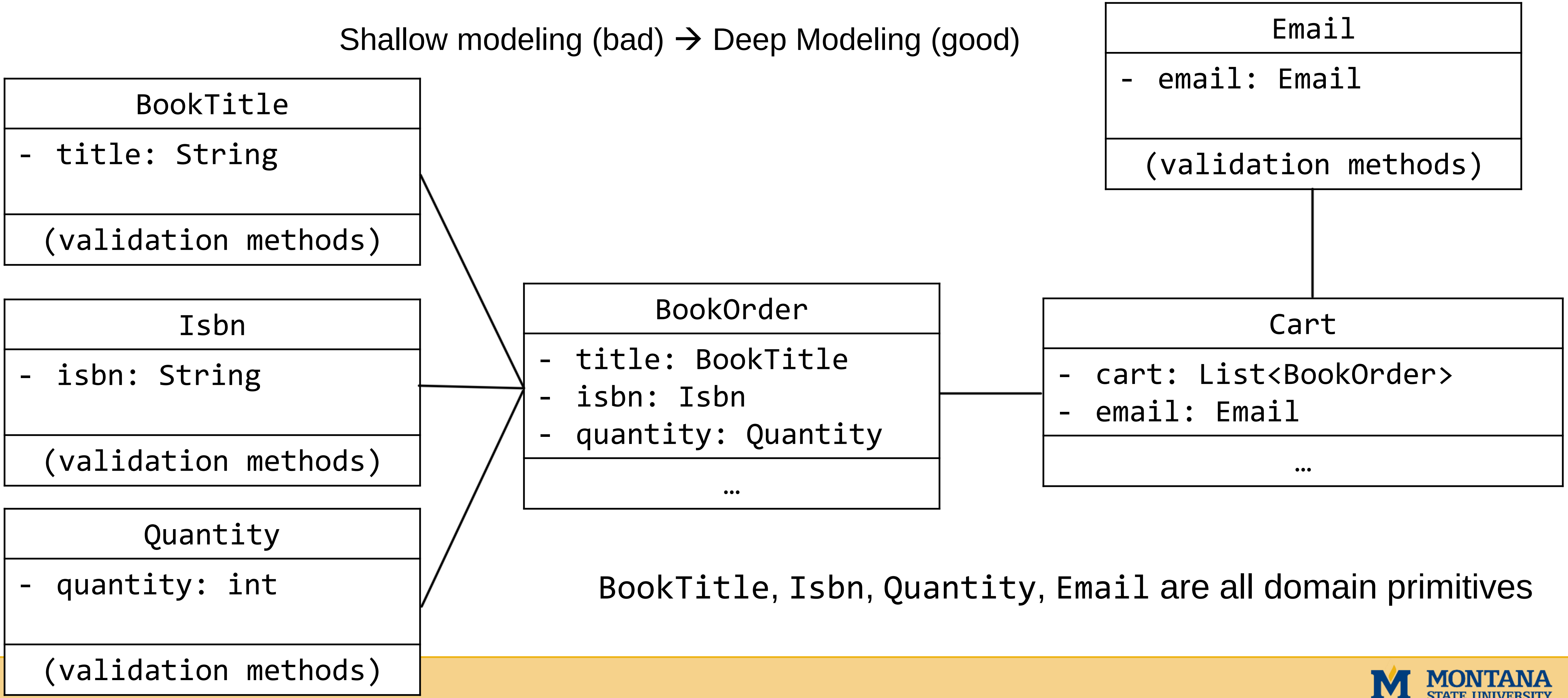
`<script> alert(“EVIL”) </script>` doesn’t meet the rules (fails the regex) and is rejected by design

Principle of Fail-Fast: detect and respond to errors as early of possible. If in invalid state occurs (such as invalid object creation), do not let program proceed with invalid state. Program deals with invalid state → Weird things happen → Potential Vulnerabilities

Fail-Fast is part of our design w/ Domain Primitives, which makes our application more secure

Domain-Driven Design: focus of modeling software to match a domain according to input from domain experts. Divide system into bounded contexts (domain primitives), each having their own model

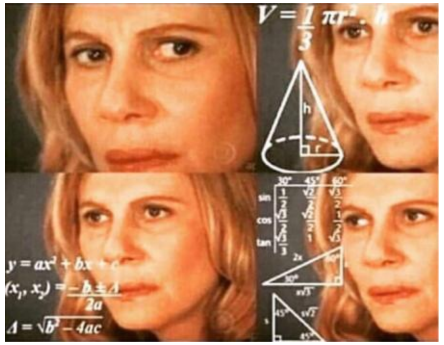
Shallow modeling (bad) → Deep Modeling (good)



BookTitle, Isbn, Quantity, Email are all domain primitives

Too many classes?

Overly Complex ?



Performance?

```
BookTitle
- title: String
(validation methods)
```

```
Isbn
- isbn: String
(validation methods)
```

```
Quantity
- quantity: int
(validation methods)
```

```
BookOrder
- title: BookTitle
- isbn: Isbn
- quantity: Quantity
…
```

```
Email
- email: Email
(validation methods)
```

```
Cart
- cart: List<BookOrder>
- email: Email
…
```

BookTitle, Isbn, Quantity, Email are all domain primitives