Software Engineering

Lecture 1(a,b):
Using Annotation in
Object Oriented Programming

Outline

(A) Basic class design with annotation

Lect 1

(B) Collection class design with annotation

(C) Design validation & Coding

(D) Type hierarchy

Lect 2

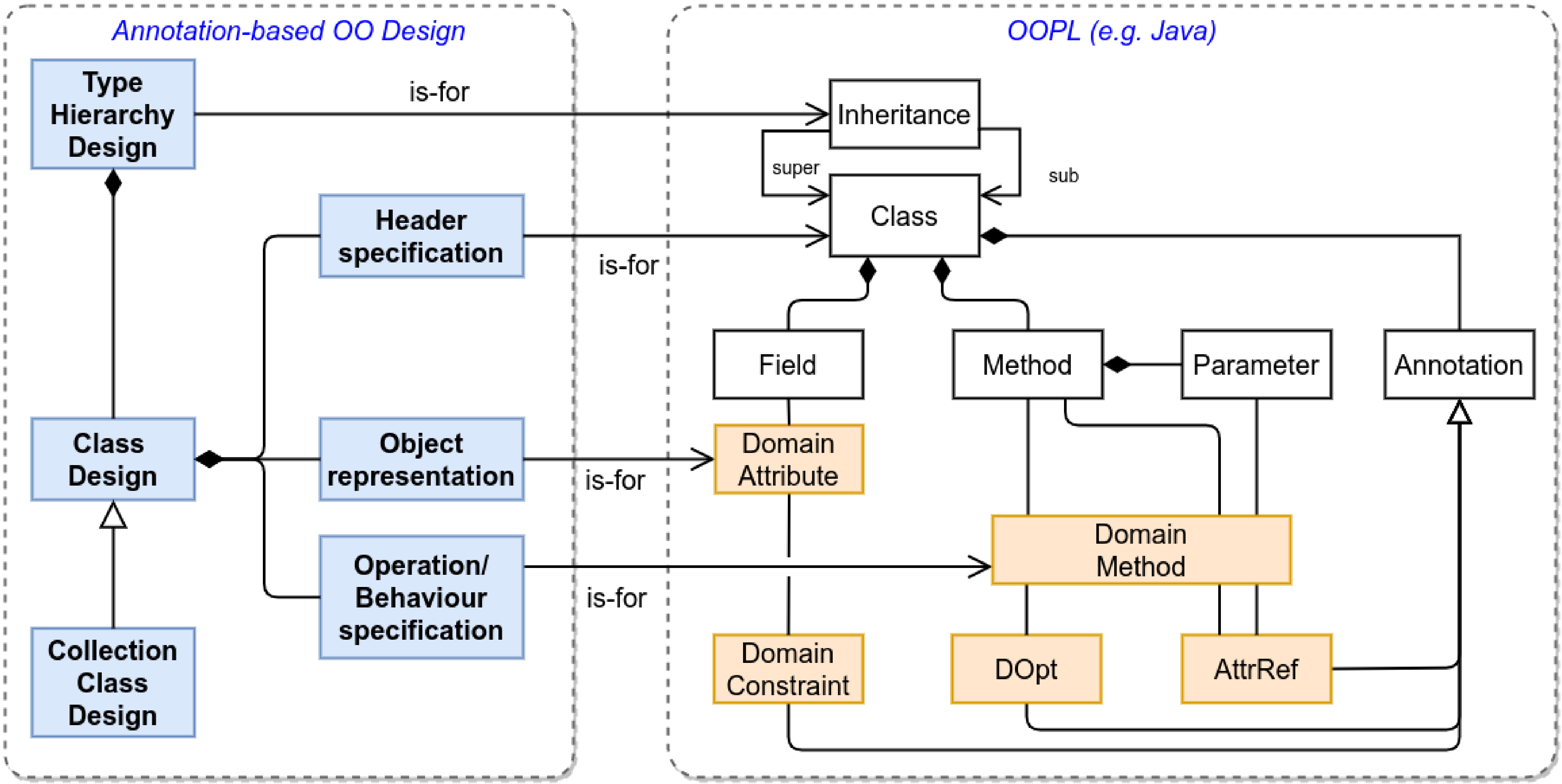
Pre-requisites

- Basic object oriented programming:
  - class, object
  - encapsulation
  - inheritance
  - polymorphism
  - abstraction
  - interface
  - exception handling
  - input/output streams
- Java programming language: 8 or above

References

- Course book: **Chapters 4-6**
- Liskov and Guttag (2000), Chapters 2-3,5,9
- Java language specification:
  - esp. the annotation feature

Design Method Overview



(UML class diagram: https://www.uml-diagrams.org/class-diagrams-overview.html)

(A) Basic Class Design with Annotation

1) Motivation: why detailed class design?
   - focus on the essential design rules
2) Using annotation to express class design rules

Motivation

Why using annotation

in OOP design?

Is this good enough to code?

```
CustomerSimple
-----------------------------
- id : int
- name : String
-----------------------------
+ setName(String)
+ getId(): int
+ getName(): String
```

Conventional code

```
public class CustomerSimple {
  private int id;
  private String name;
  public CustomerSimple(int id, String name) {
    this.id = id;
    this.name = name;
  }

  public int getId() {
    return id;
  }
  public String getName() {
    return name;
  }
  public void setName(String name) {
    this.name = name;
  }
}
```

More design details are needed...

- Can `name` be typed `char[]` instead of `String`?
- Can `id` be negative?
- Can `name` be uninitialised (i.e. takes `null`)?
- How do we create a `Customer` object with a given `id` and `name`?
- Should there be an operation to change value of `id`?
- etc.

How to express those design rules?

- Using validation methods:
  - rules are implicit (implied by the method behaviour)
  - rules **can not be** applied at compile time (because method execution is required)
- Using annotation:
  - Java: annotation; C#: attribute
  - rules are explicit in the design
  - rules **can be** applied at compile time
    - to validate the design
  - define behaviour of validation methods

Example: validate id method

- id value is checked in the constructor

```java
public class CustomerSimple {
  private int id;
  private String name;

  public CustomerSimple(int id, String name) {
    if (validateId(id)) {
      this.id = id;
    } else {
      throw new NotPossibleException("CustomerSimple.init:
                invalid id " + id);
    }
    this.name = name;
  }

  private boolean validateId(int id) {
    return id > 0;
  }
}
```

Example: annotate & validate id

- id field is annotated to make clear its min constraint
- id value is checked in the constructor

```
public class CustomerSimple {
  @DomainConstraint(min=1)
  private int id;
  private String name;
  public CustomerSimple(int id, String name) {
    if (validateId(id)) {
      this.id = id;
    } else {
      throw new NotPossibleException("CustomerSimple.init: invalid id " + id);
    }
    this.name = name;
  }
  // code omitted
}
```

State-of-the-art: annotation usage in the software industry

- Java makes extensive uses of annotations
- Back-end Java-based software tools:
  - data management:
    - Java persistence API (JPA)
    - Hibernate, etc.
  - web-based software development:
    - Spring
    - OpenXava, etc.
- Front-end (non-Java) software tools:
  - Angular, etc.

Example: Java annotations

- Override a supertype’s method
- Informs compiler to supress warnings
- Provide documentation

```
@Override
public boolean equals(Object o) { return this == o; }

@SuppressWarnings("unchecked")
void myMethod() { }

@Author(name = "Jane Doe")
class MyClassA { }

@Author(name = "Jane Doe")
@Author(name = "John Smith")
class MyClassB { }
```

Provided by Java

User-defined

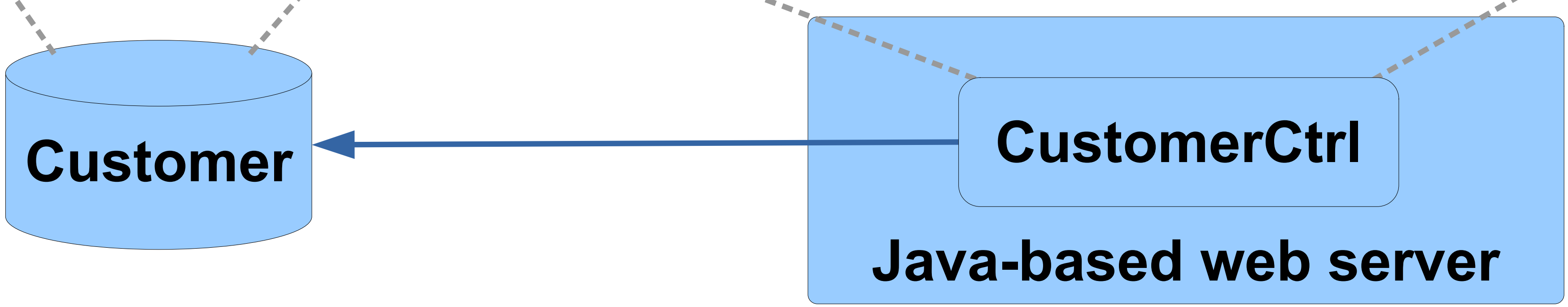
Source: https://docs.oracle.com/javase/tutorial/java/annotations/basics.html

Back-end: Data management & Spring (Java framework)

```java
@Entity
@Table(name="Customer")
public class CustomerEntity {
  @Id
  @Column
  private int id;

  @Column(length=50)
  private String name;
}
```

```java
@Controller
@RequestMapping("/display")
public class CustomerCtrl {
  @RequestMapping(method = GET)
  public String displayCustomer(...) {
    // code omitted
    return "customer";
  }
}
```



Front-end: Angular (Javascript-based)

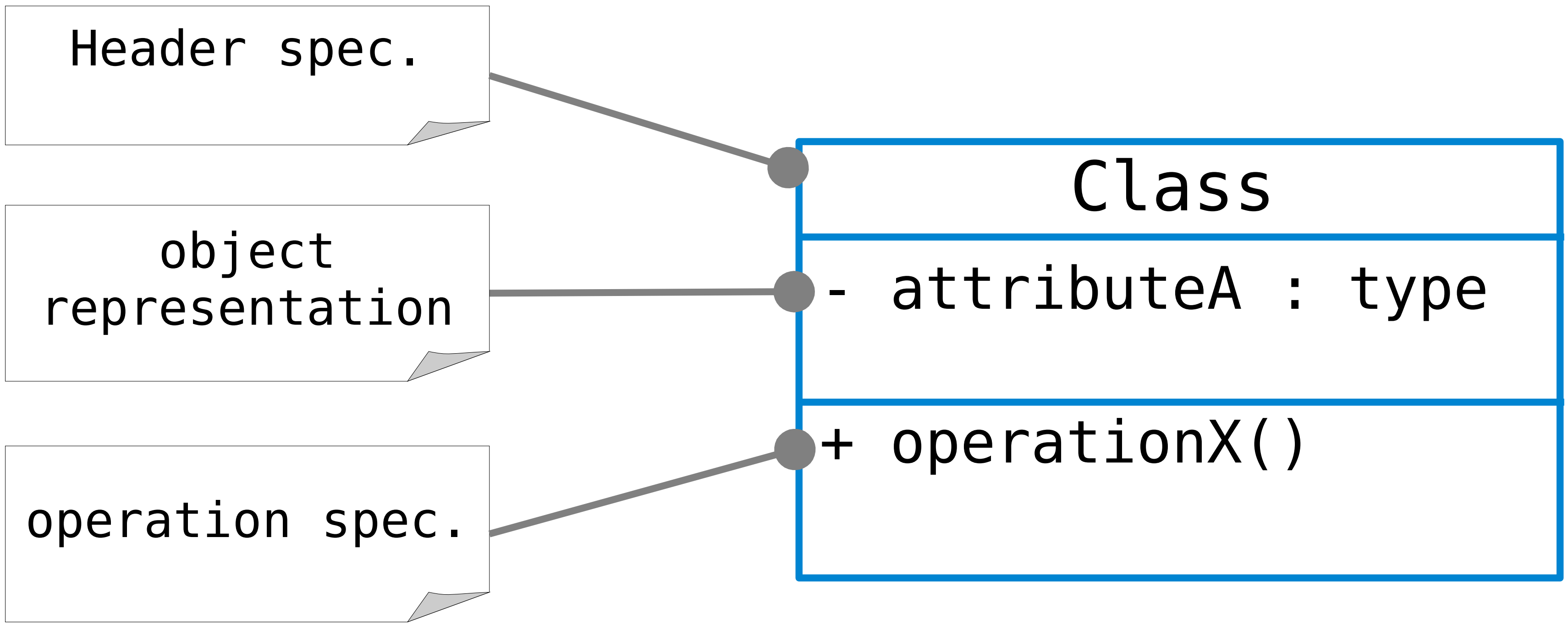
- **@Component**: defines a web user interface component

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'My first Angular App!';
}
```

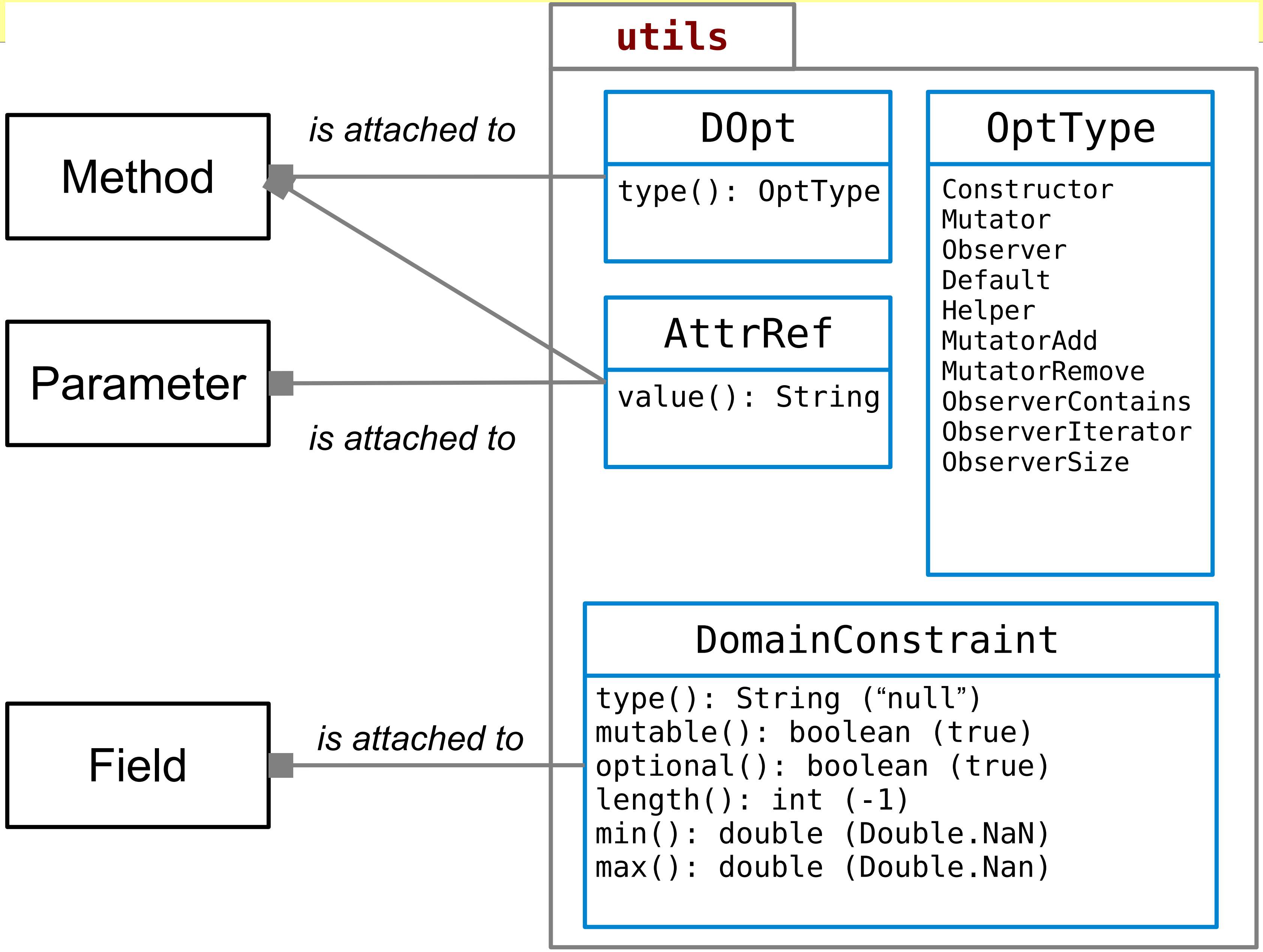
Our study: Enhancing design with annotation

- Attribute rules:
  - attributes have restrictions on values that they can take (called *domain constraints*)
- Operations (methods) must preserve the attribute rules
  - at pre and post conditions
  - not necessarily during behaviour invocation

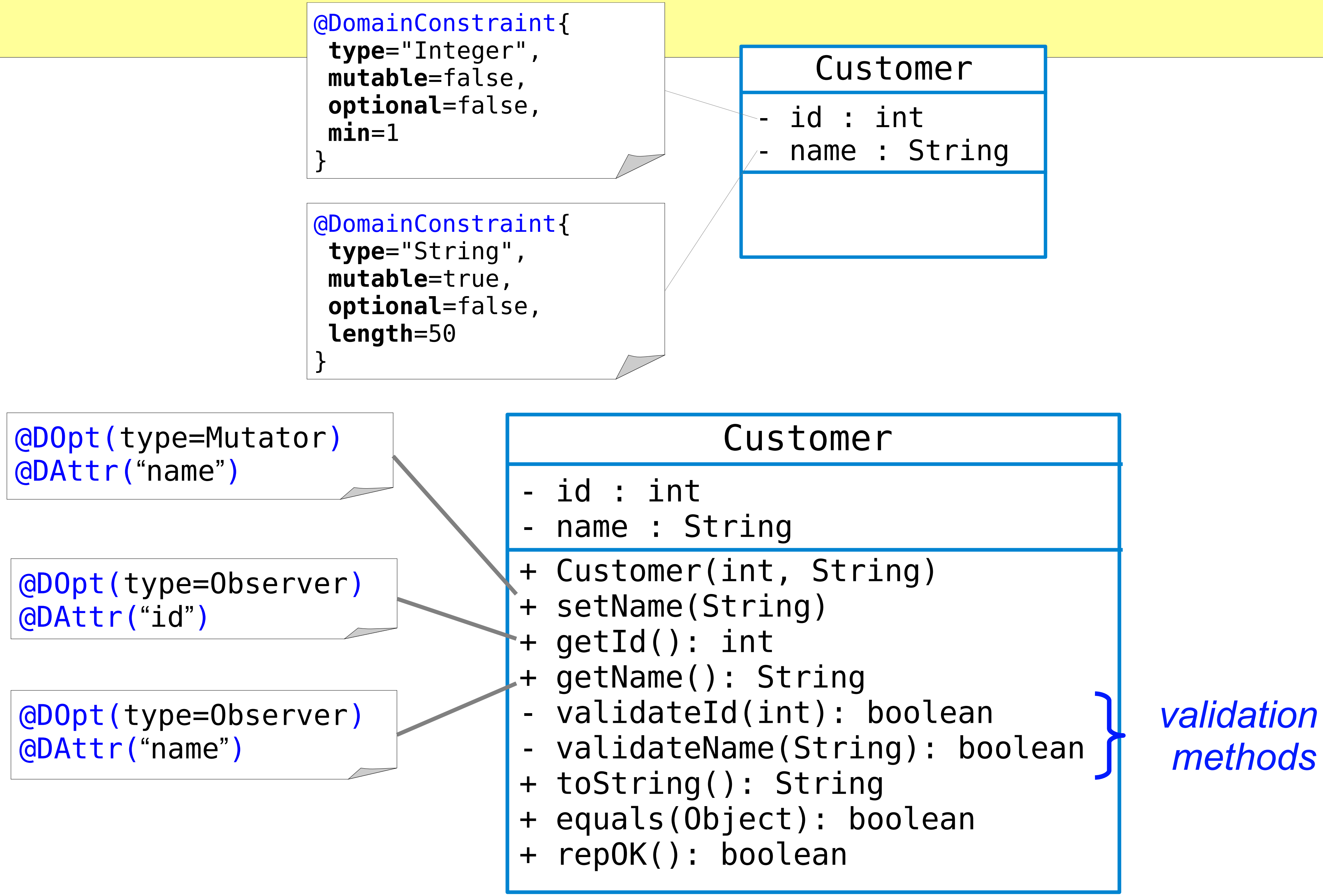
Class design overview



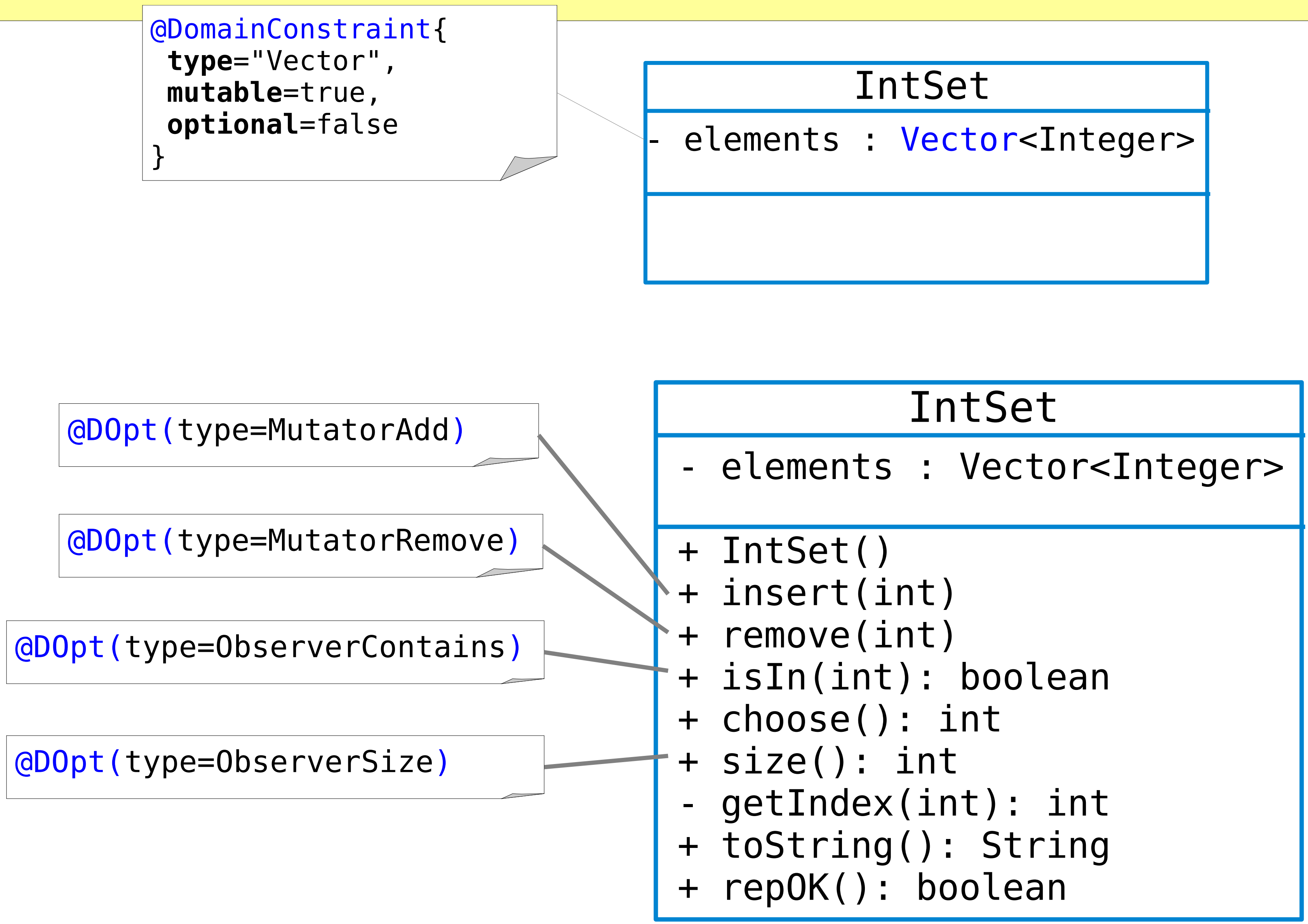
Essential design annotations



Example: Annotated Customer



Example: Annotated IntSet



What if an OOPL does not support annotations?

- Popular OOPLs (e.g. Java, C#) support annotation, but other OOPLs may not support it
- For those OOPLs:
  - if there is an alternative representation of the constraints then transform annotations to that representation
  - otherwise, leave the annotations in the code but comment them out
    - the commented annotations will serve as valuable documentation for the code

Annotated class design (Using annotation to express class design rules)

1) Class header specification (abstract concept)
2) Concrete attribute types
3) Object representation
4) Object operations (a.k.a methods)

Class header specification

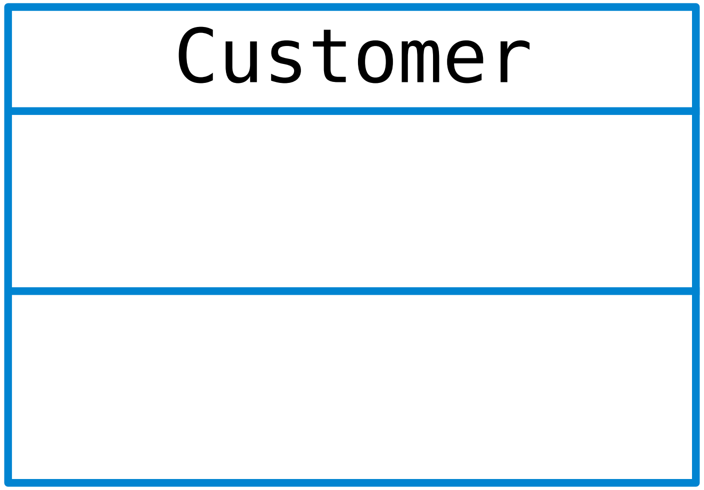
- Define the abstract concept that a class represents:
  - e.g. concept CUSTOMER is represented by class Customer
- Class **header specification** includes:
  - Concept name
  - @overview: brief description of the class (its purpose)
  - @attributes: the concept's attributes
  - @object: how to write the object state
  - @abstract_properties: domain constraints, other rules

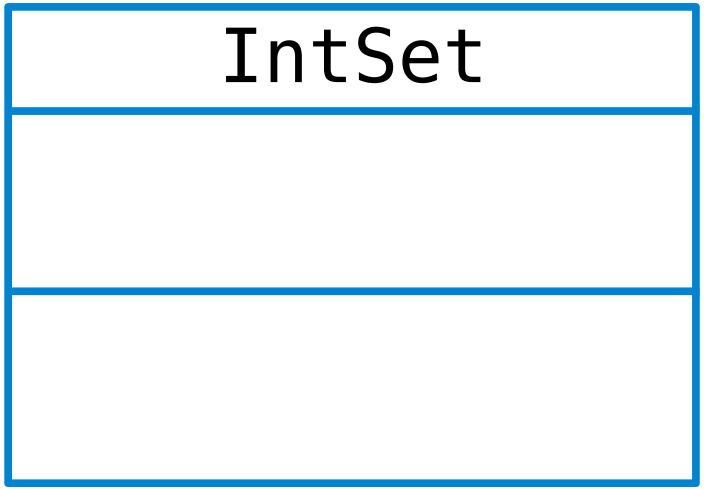
Header specification format

```
/**
 * @overview ...
 * @attributes ...
 * @object ...
 * @abstract_properties ...
 */
class C
```

Choose concept name

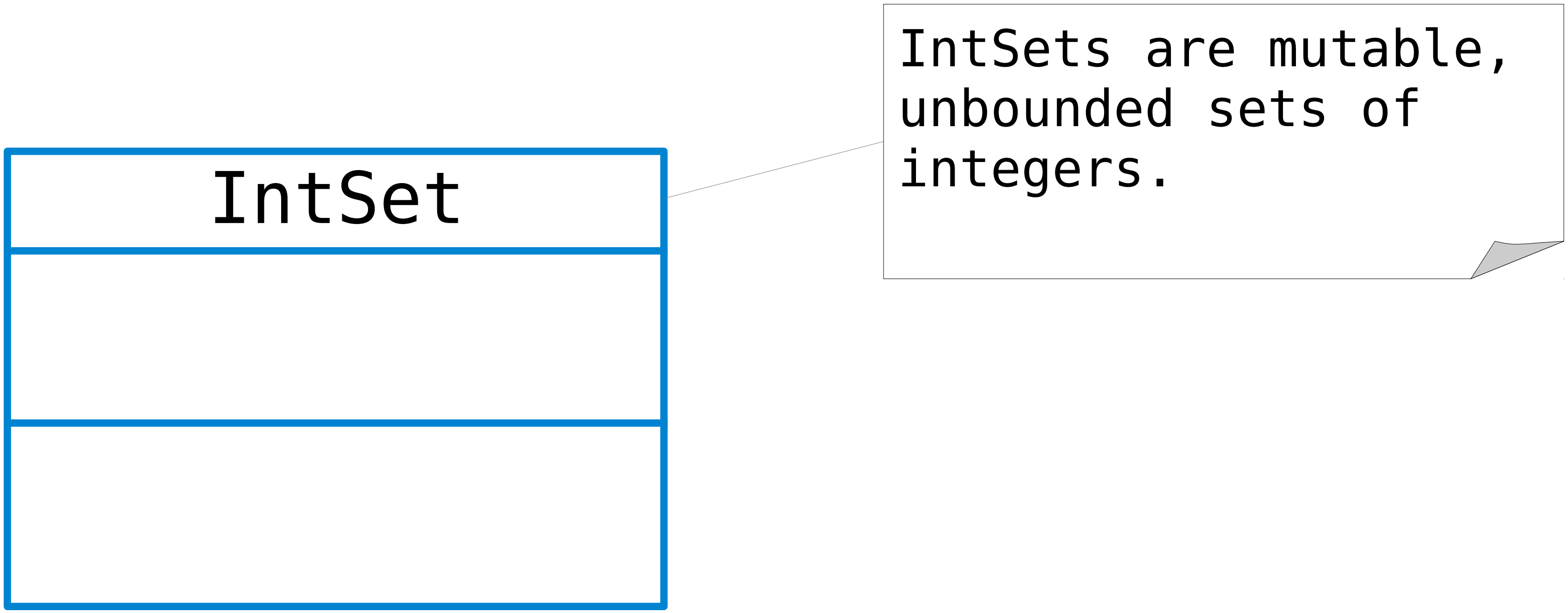
- Name of the concept that we want to model as class





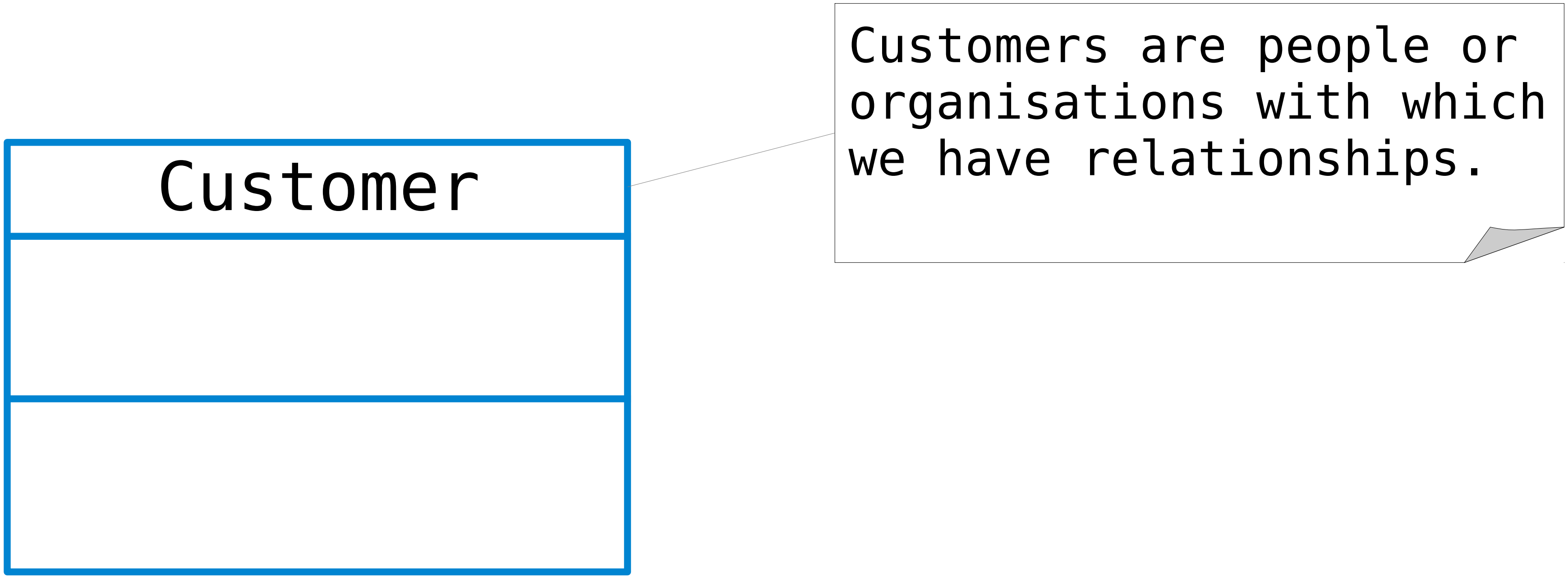
Write an overview

- Describes the meaning of the abstract concept



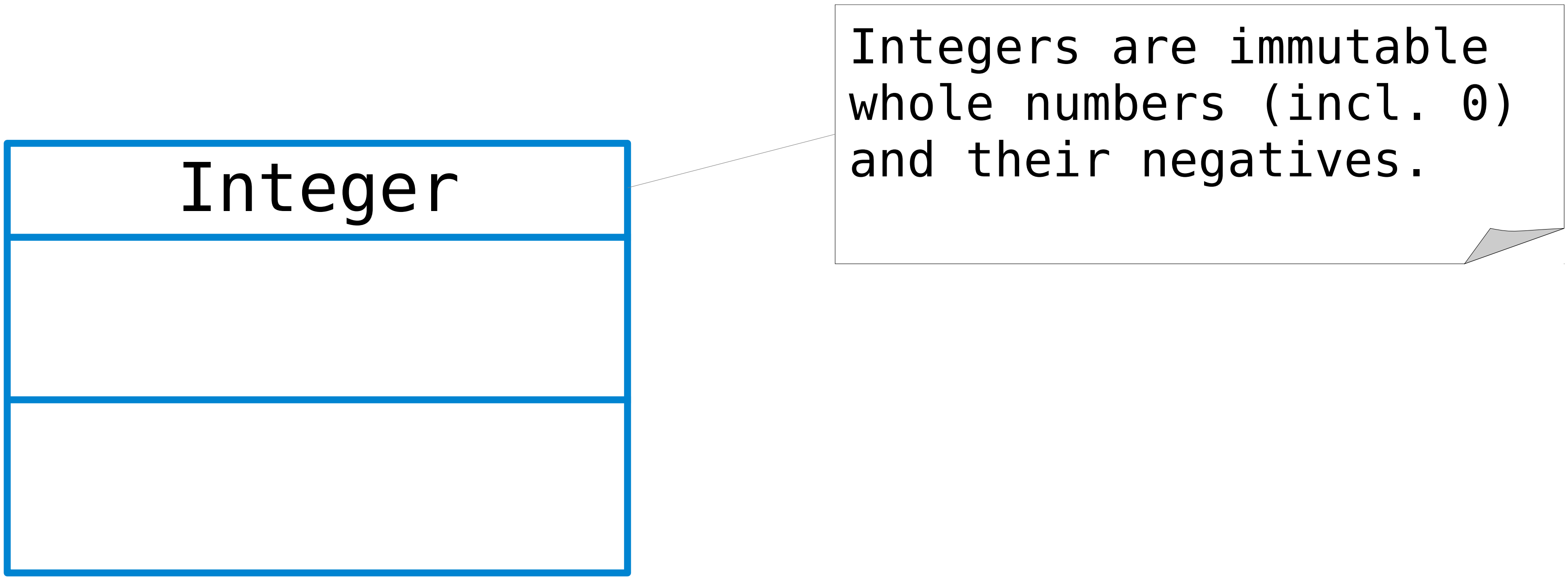

```
/**
 * @overview IntSet are mutable, unbounded sets
 *   of integers.
 */
public class IntSet
```

Example: Customer




```
/**
 * @overview Customers are people or
 *   organisations with which we have
 *   relationships.
 */
public class Customer
```

Example: Integer




```java
/**
 * @overview Integers are immutable whole
 *   numbers (incl. 0) and their negatives.
 */
public class Integer
```

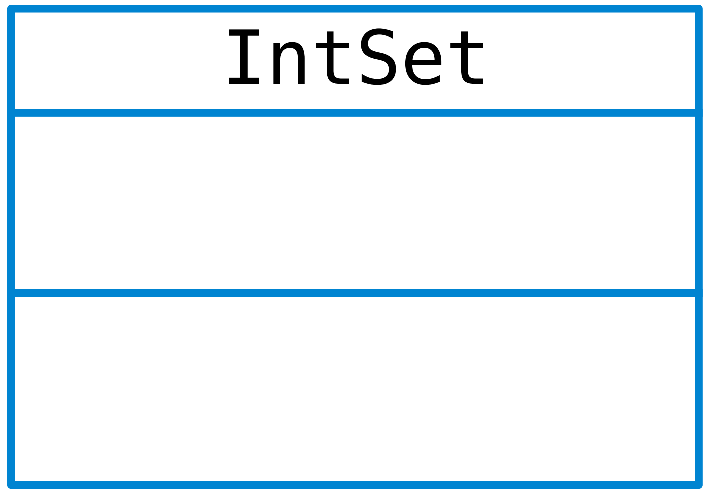
Specify the attributes

- Written using tag @attributes
- Each attribute entry has three parts:
  - **name**: the attribute name
  - **(formal) type**: the abstract data type of the attribute
  - **concrete type**: the actual data type
    - left blank for now (added later)
- Drawn in the second compartment of the UML diagram
  - visibility (+/-) is not yet determined (added later)

Common formal attribute types

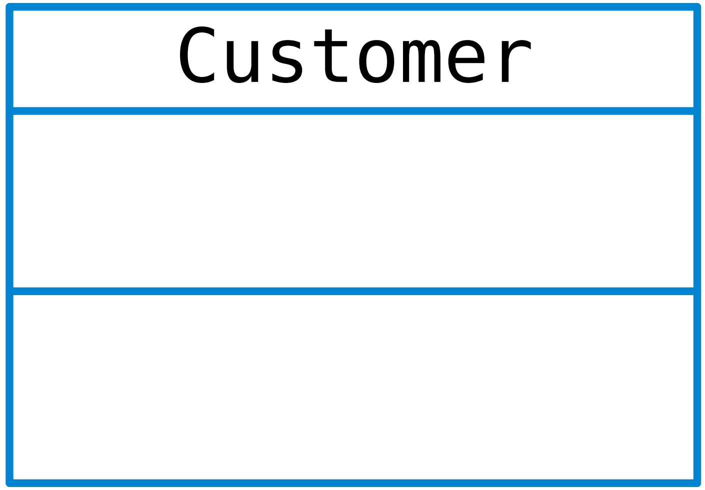
- **Integer**: integral values
- **String**: text values
- **Real**: real values (e.g. 1.5, 2.0)
- **Char**: character
- **Boolean**: true, false
- **Sequence** (i.e. array) of the above: e.g. Integer[] is a sequence of integers
- **Set** of the above: e.g. Set<Integer> is a set of integer

Example: IntSet




```java
/**
 * @overview ... as before ...
 * @attributes
 *   elements   Set<Integer>
 */
public class IntSet
```

Example: Customer



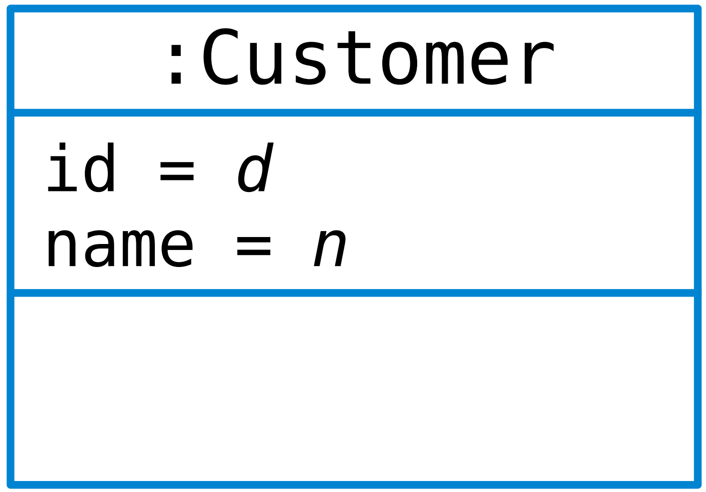

```
/**
 * @overview ... as before ...
 * @attributes
 *   id     Integer
 *   name   String
 */
public class Customer
```

Specify the abstract object

- An object created from typical values of the attributes
- If has only one attribute:
  - use a typical value of the attribute, e.g.:
    - set-based type: {x1,...,xn}
    - others: e.g. -2, -1, …
- If has more than one attributes:
  - use the tuple notation, e.g.:

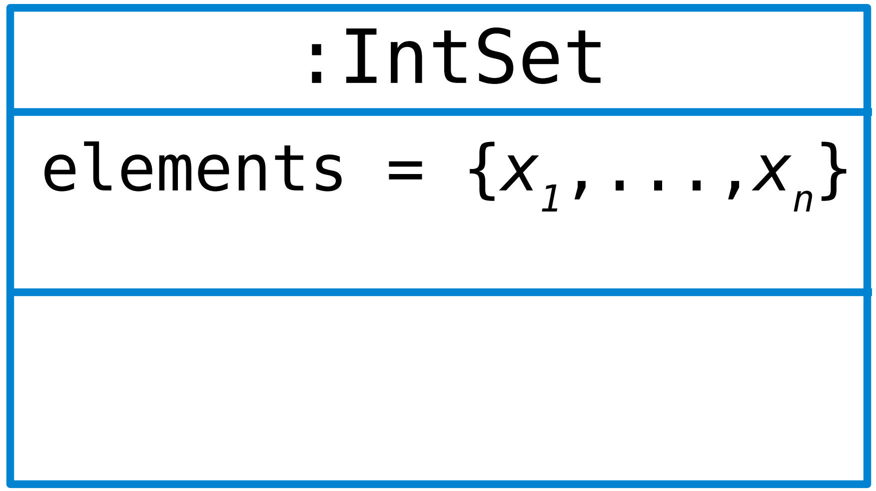
    A typical Customer is <d, n> where id(d), name(n)

Example: Customer



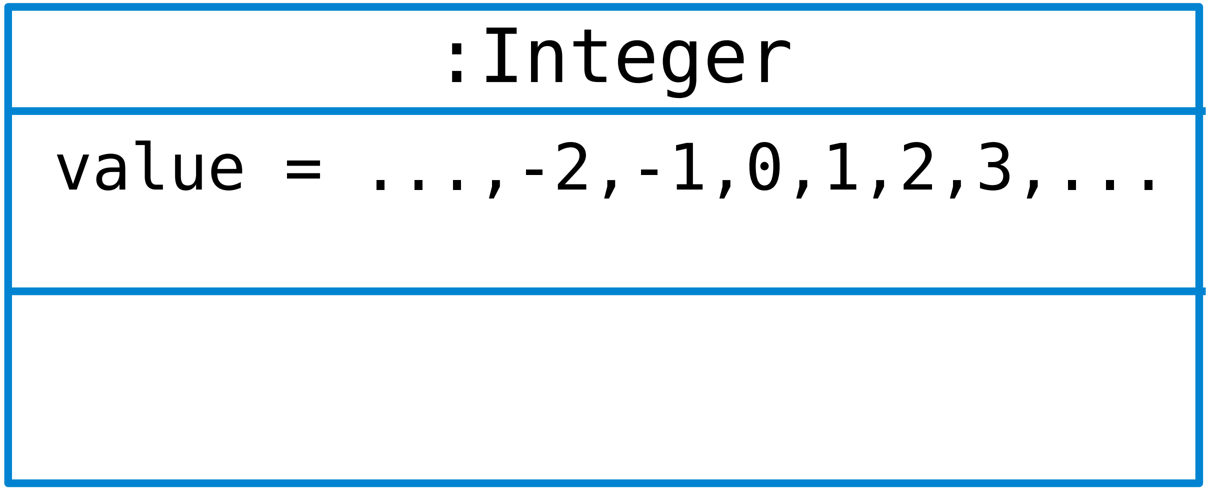

```
/**
 * @overview ... as before ...
 * @attributes ... as before ...
 * @object A typical Customer is c=<d,n>, where
 *     id(d), name(n).
 */
public class Customer
```

Example: IntSet




```
/**
 * @overview ... as before ...
 * @attributes ... as before ...
 * @object A typical IntSet object is
 *   c={x1,...,xn}, where x1,...,xn are elements
 */
public class IntSet
```

Example: Integer




```
/**
 * @overview ... as before ...
 * @attributes
 *   value   Integer
 * @object Typical integers are ...,-2,-1,0,1,...
 */
public class Integer
```

Specify abstract properties

- Written using the tag @abstract_properties
- Two types:
  - domain constraint
  - others

Domain constraint

- A statement about what data values an attribute can take
- Properties to include:
  - **type**: the formal type
  - **mutable**: true | false
  - **optional**: true | false
  - **length** (for string type)): the max value length
  - **min** (for numeric type): min value
  - **max** (for numeric type): max value
- Omitted if no properties are specified

Domain constraint table: Customer

Attributes	type	mutable	optional	length	min	max
id	Integer	N	N	-	1	-
name	String	Y	N	50	-	-

```
/**
 * @overview ... as before ...
 * @attributes ... as before ...
 * @object ... as before ...
 * @abstract_properties
 *  mutable(id)=false /\ optional(id)=false /\
 *      min(id)=1 /\
 *  mutable(name)=true /\ optional(name)=false /\
 *      length(name)=50
 */
public class Customer
```

Domain constraint table: IntSet

Attributes	**type**	**mutable**	**optional**	**length**	**min**	**max**
elements	Set<Integer>	Y	N	-	-	-

```
/**
 * @overview ... as before ...
 * @attributes ... as before ...
 * @object ... as before ...
 * @abstract_properties
 *    mutable(elements)=true /\
 *    optional(elements)=false /\
 *    elements != {} →
 *       (for all x in elements. x is integer)
 */
public class IntSet
```

Other properties

- Properties other than those captured in the domain constraint
- Specific to each abstract concept
- Examples:
  - Set: elements are distinct
  - Array: elements form a sequence

Example: IntSet

```
/**
 * @overview ... as before ...
 * @attributes ... as before ...
 * @object ... as before ...
 * @abstract_properties
 *    mutable(elements)=true /\
 *    optional(elements)=false /\
 *    elements != {} →
 *      (for all x in elements. x is integer) /\
 *    elements != {} →
 *      (for all x, y in elements. x != y)
 */
public class IntSet
```

Specify the concrete attribute type

- **Concrete type** is the actual data type used to implement an attribute
  - must be supported by the target OOPL
- Concrete type may differ from the formal one:
  - e.g. array is the concrete type for Vector's elements
- One formal type is typically mapped to one or more concrete types:
  - e.g. Set can be implemented by array or Vector
- Write in the third column of the attribute entry in @attributes (omit if same as the formal type)

Some useful Java data types

- Wrapper types
- Dynamic array

Wrapper type

- Wrapper class
- An object data type that 'wraps' the primitive types
- Suitable for used as formal attribute type
- Auto-boxing: automatically converts ('wraps) primitive values into wrapper objects
- Auto-unboxing: the reverse, i.e. wrapper object → primitive value

Wrapper classes

Primitive types	Wrapper classes (types)
`int`	**`Integer`**
`long`	**`Long`**
`float`	**`Float`**
`double`	**`Double`**
`char`	**`Character`**

E.g.: Integer

```java
/**
 * @overview A program that creates and
 *   manipulates Integer objects.
 */
public class IntegerWrapper {
  /**
   * The run method
   */
  public static void main(String[] args) {
    Integer i;
    int j, k;
    // create object using auto-boxing
    i = 5;                          /* i = Integer(5) */

    // auto-convert to primitive using unboxing
    k = i;                          /* k = 5 */

    // unboxing i back to primitive in expression
    j = i + 10;                     /* j = 15 */
    System.out.printf("i, j, k = %d, %d, %d %n", i, j, k);
  }
}
```

Dynamic array

- Class: `java.util.Vector`
- Elements are objects of any type:
  - can even be of different types
- Supports a parameterised syntax (generic):
  - if elements belong to a known type (e.g. Integer)

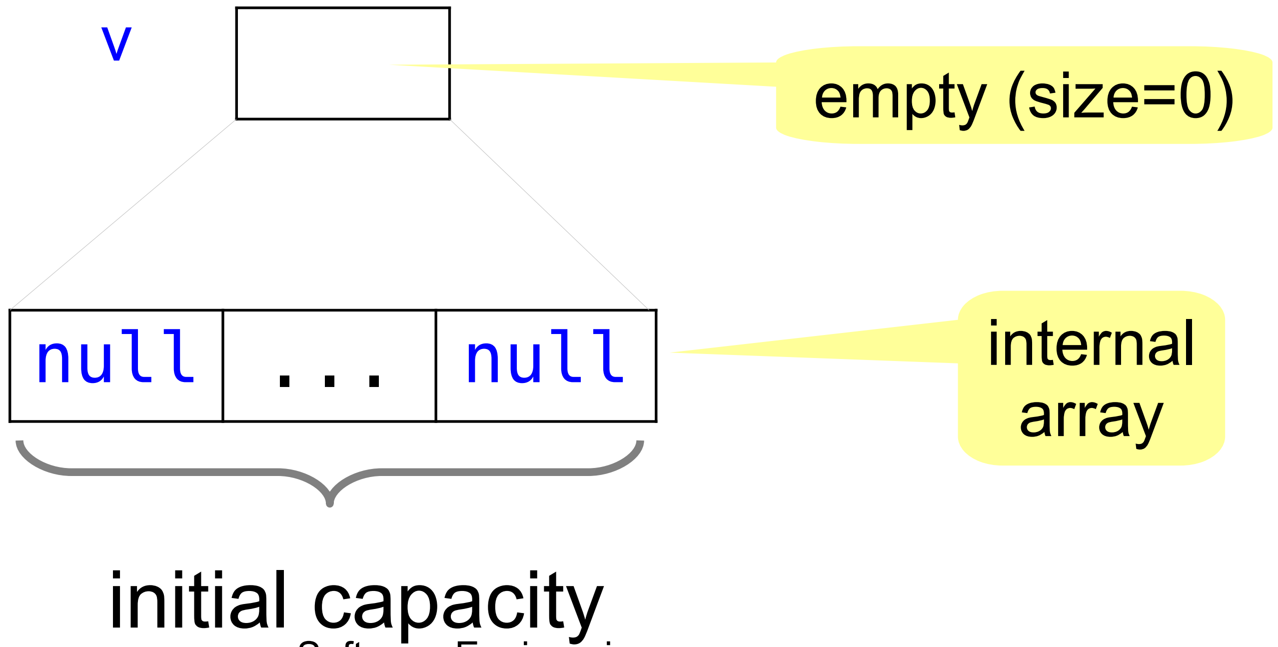
    `Vector<T>`: T is the element type
- Provide operations to operate on the elements:
  - elements are added/removed easily

Vector operations

- `add(T o)`:
  - add o to end of this vector
- `set(int i,T o)`:
  - replace the i^{th} element by o
- `get(int i)`:
  - return the i^{th} element
- `remove(int i)`:
  - remove element i^{th}
- `size()`:
  - return number of elements

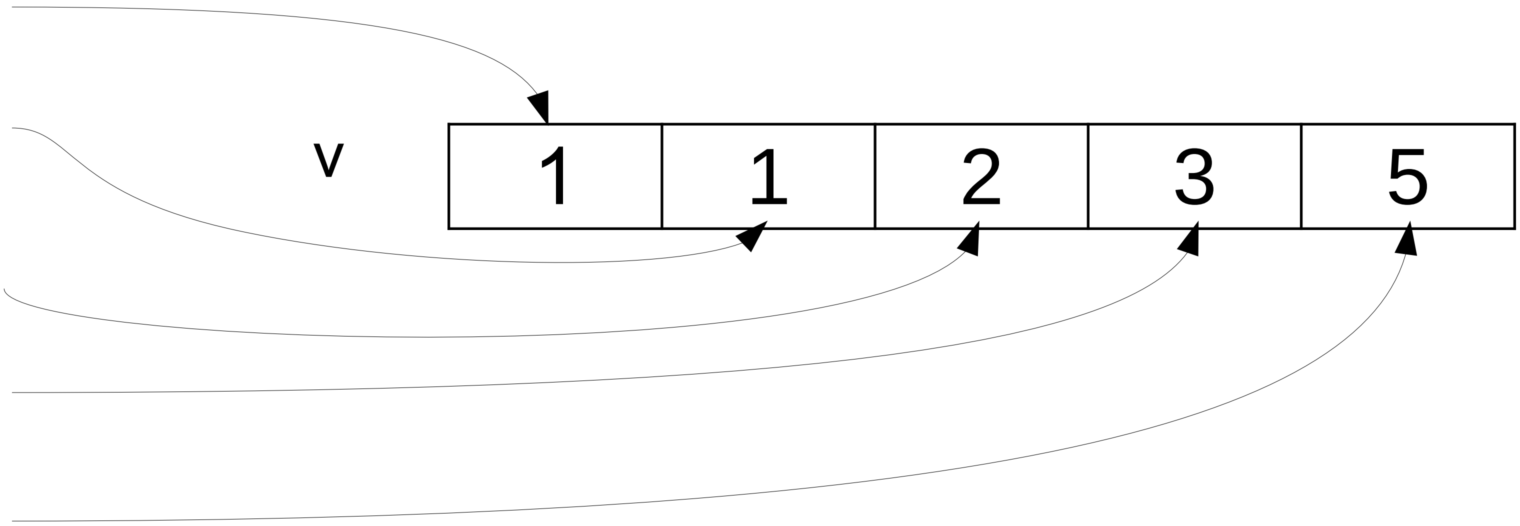
Create a Vector

```
import java.util.Vector;
...
Vector v1 = new Vector();
Vector<Integer> v = new Vector<>();
```



add()

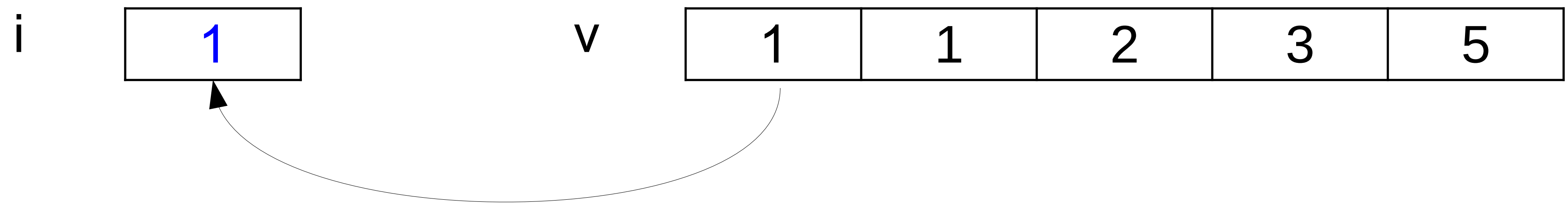
```
v.add(1);
v.add(1);
v.add(2);
v.add(3);
v.add(5);
```



get()

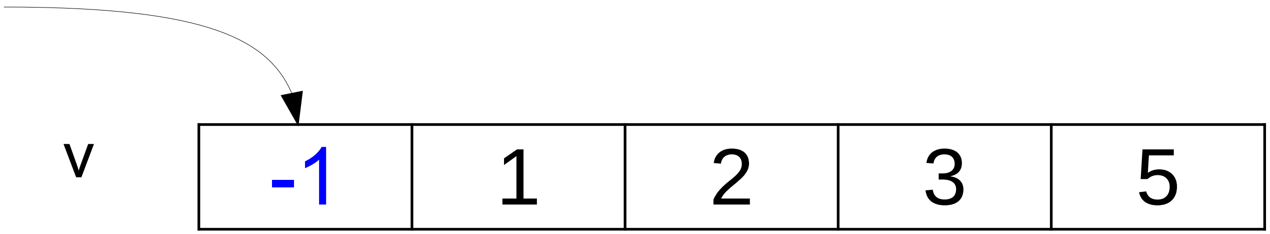
```java
int i = v.get(0);
```

auto-unboxing



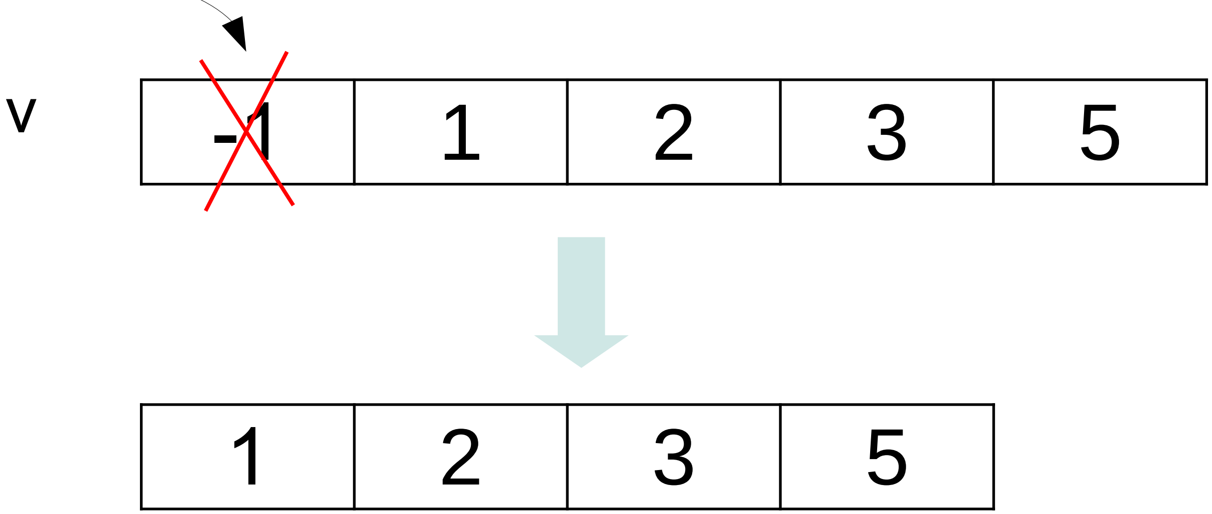
set()

```
v.set (0,-1);
```



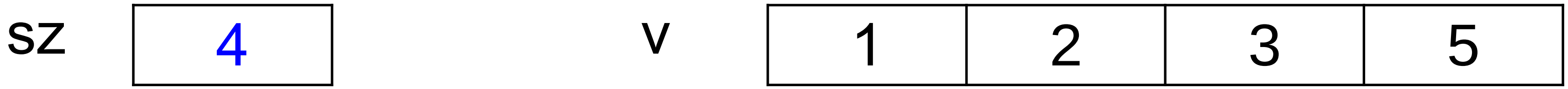
remove()

```
v.remove(0);
```



size()

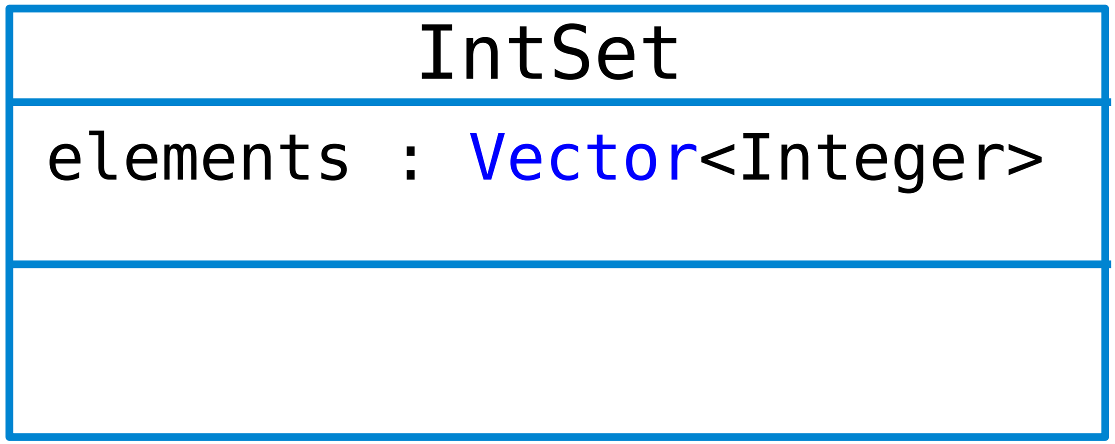
```
int sz = v.size();
```



Vector vs. Set

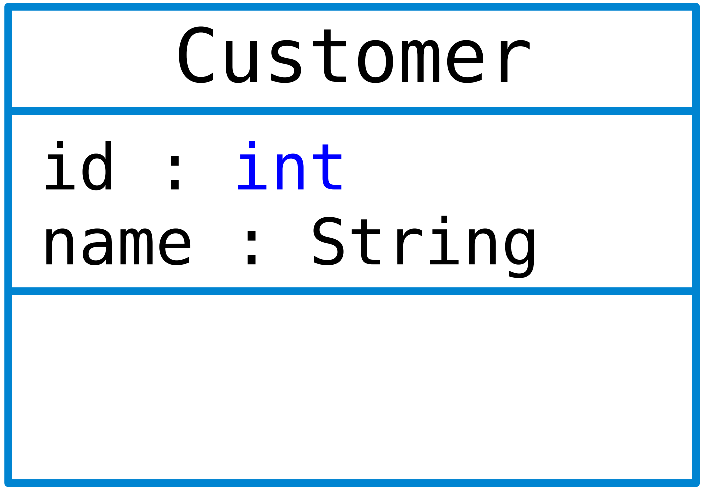
- Similarity:
  - a collection of items
  - items can be of different types
- Differences:
  - Vector allows duplicates
  - Vector's elements can be accessed by index

Example: IntSet




```
/**
 * @overview ...
 * @attributes
 *     elements    Set<Integer> Vector<Integer>
 * ...
 */
public class IntSet
```

Example: Customer




```
/**
 * @overview ...
 * @attributes
 *      id     Integer    int
 *      name   String
 * ...
 */
public class Customer
```

Guidelines for choosing concrete type

- Must be supported by the prog. language
- May be the same or different from the formal type
- To balance between productivity and efficiency:
  - **productivity**: ease coding with the type
  - **efficiency**: run-time efficiency of the using code (code that uses the type)

Example: Customer

- **Productivity**:
  - attributes are referred to directly as variables
  - make use of built-in `String` and integer operations
- **Efficiency**:
  - no type conversions are required

Example: IntSet
Vector or array?

- **Productivity**:
  - Vector is better for adding and removing elements
    - provides add(), remove() operations for these
  - Array is slightly better with retrieving element
    - the notation is really simple: a[index]
- **Efficiency**:
  - Compared to Vector, array is:
    - faster to retrieve and remove elements
    - slower to add an element (requires array copy)

Object representation (Rep)

- In class design:
  - an attribute is also called *instance* or *object* variable
- Attributes (together with their concrete types) form the **representation** (**rep**) of the object:
  - ‘represent’ the object state
- Specification steps:
  - define an instance variable for each attribute
  - annotate with domain constraint(s) (if any)

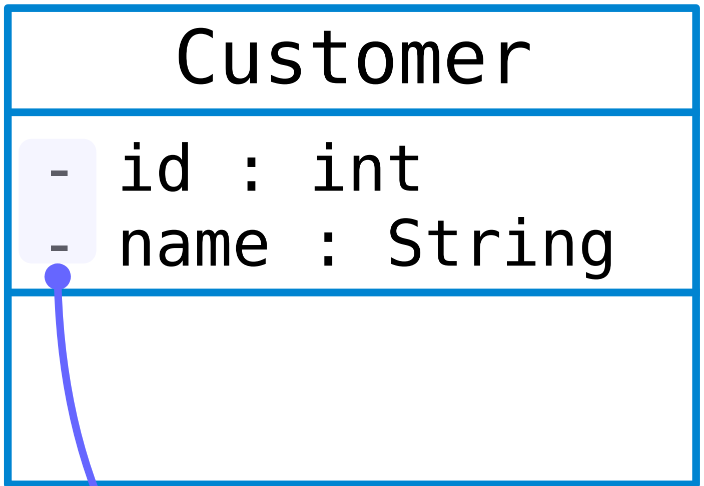
Examples

- Customer:
  - attributes: `id` (`int`) and `name` (`String`) **represent**
  - `Customer` objects <1,"Duc">, <2,"Thang">, ...
- IntSet:
  - attribute: `elements` (`Vector`) **represents**
  - `IntSet` objects {-11,2,3}, {10,-12,15}, ...

Define instance variable

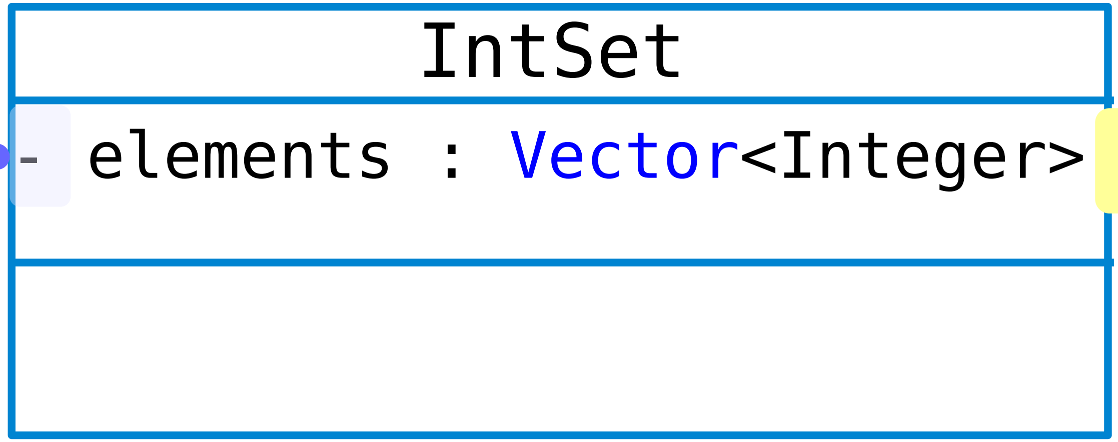
- For each attribute, define an instance variable:
  - identifier = attribute name
  - data type = concrete type
  - access modifier: `private`
- Modifier `private` is to protect attributes from direct outside access:
  - recall: information hiding

Example: Customer




```
/**
 * @overview ...
 * @attributes
 *    id     Integer int
 *    name   String
 * ...
 */
public class Customer {
  private int id;
  private String name;
}
```

Example: IntSet




```
/**
 * @overview ...
 * @attributes
 *    elements  Set<Integer>  Vector<Integer>
 * ...
 */
public class IntSet {
  private Vector<Integer> elements;
}
```

Annotate instance variables with domain constraints

- Annotate each instance variable with annotation `@DomainConstraint`
  - realises the essential domain constraints discussed earlier
- All @DomainConstraint's properties are given default values:
  - can be omitted if not specified
- All annotations are located in the package `utils`:
  - We will discuss other annotations later

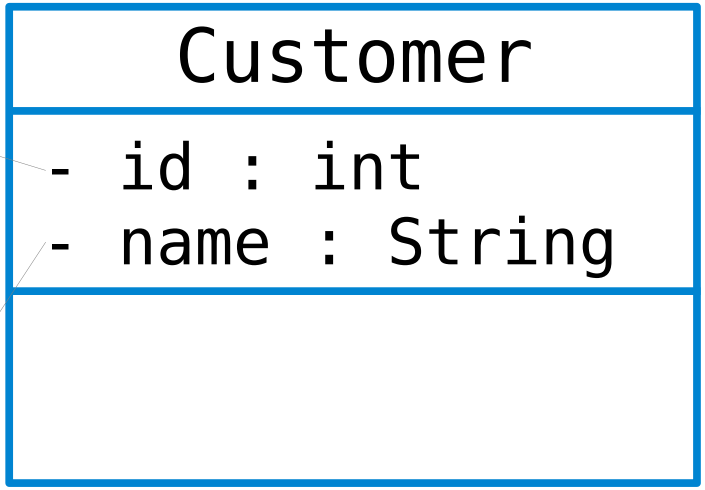
Annotation @DomainConstraint

```
public @interface DomainConstraint {
  public String type() default "null";
  public boolean mutable() default true;
  public boolean optional() default true;
  public int length() default -1;
  public double min() default Double.NaN;
  public double max() default Double.NaN;
}
```

Example: Customer

```
@DomainConstraint{
 type="Integer",
 mutable=false,
 optional=false,
 min=1
}
```

```
@DomainConstraint{
 type="String",
 mutable=true,
 optional=false,
 length=50
}
```



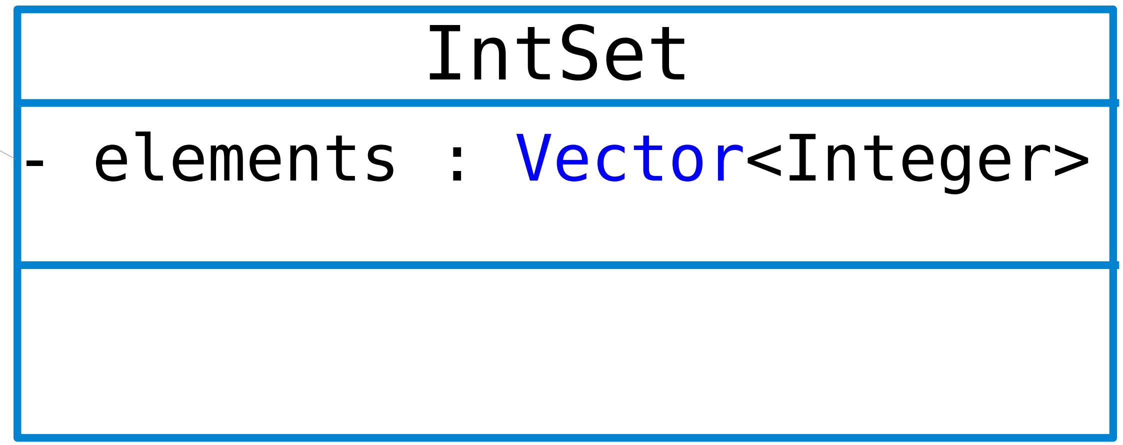
... in Java's textual form

```
/**
 * @overview ...
 * @attributes
 *     id    Integer
 *     name  String
 * @object ...
 * @abstract_properties
 *   mutable(id)=false /\ optional(id)=false /\ min(id)=1 /\
 *   mutable(name)=true /\ optional(name)=false /\
 *                     length(name)=50
 */
public class Customer {
  @DomainConstraint(type="Integer",mutable=false,
                         optional=false,min=1)
  private int id;

  @DomainConstraint(type="String",mutable=true,
                     optional=false,length=50)
  private String name;
}
```

Example: IntSet

```
@DomainConstraint{
  type="Vector",
  mutable=true,
  optional=false
}
```



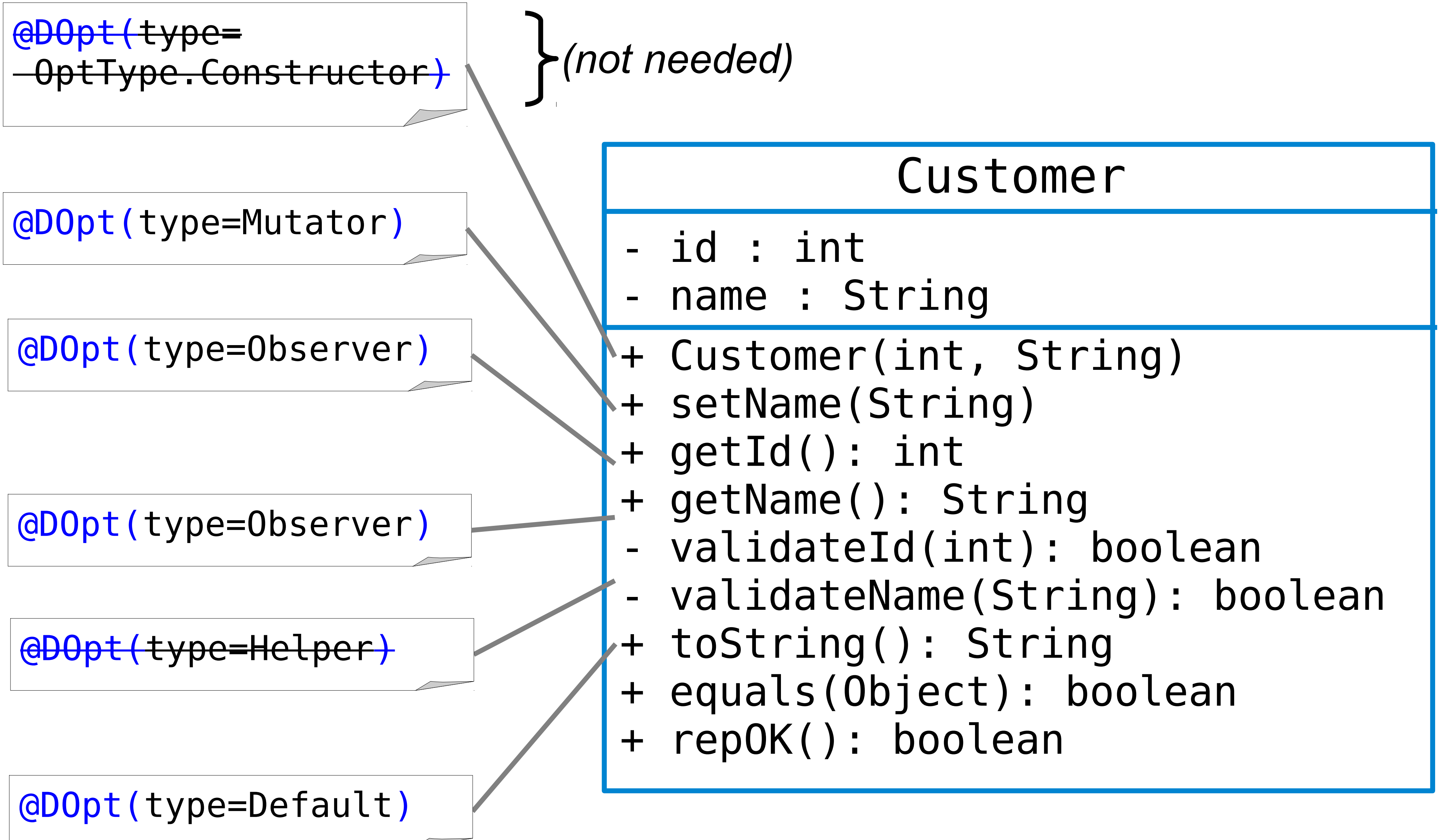
... in Java's textual form

```
/**
 * @overview ...
 * @attributes
 *     elements    Set<Integer> Vector<Integer>
 * @object ...
 * @abstract_properties
 *  mutable(elements)=true /\
 *                optional(elements)=false /\
 *   elements != {} →(for all x in elements. x is integer) /\
 *   ...
 *   ...
 */
public class IntSet {
  @DomainConstraint(type="Vector",mutable=true,
                                 optional=false)
  private Vector<Integer> elements;
}
```

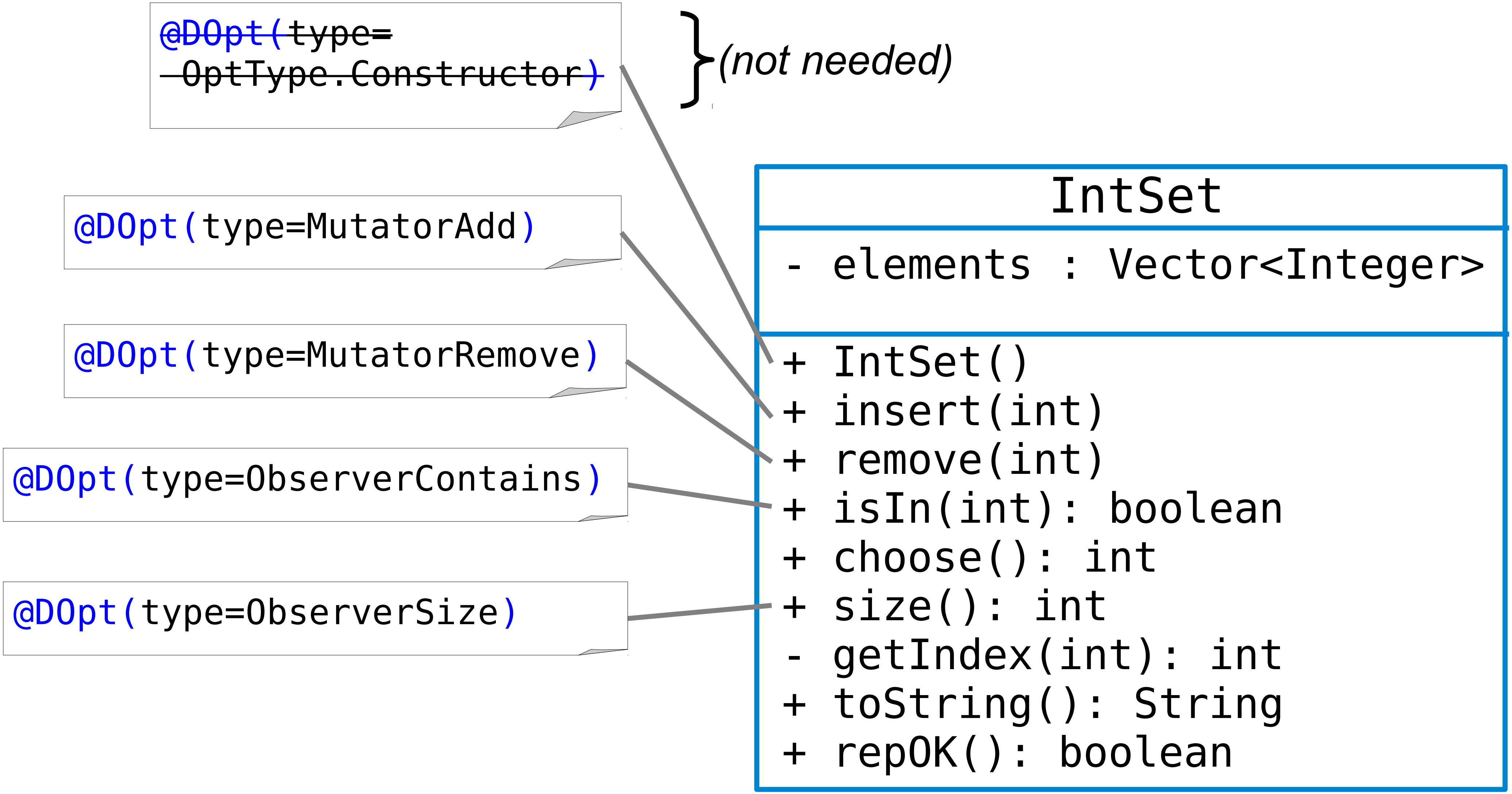
Operations (a.k.a methods)

- Operations are *object procedures*
  - must be invoked on an object of the class
  - differ from stand-alone procedures (*how?*)
- Operations typically observe and/or modify object state:
  - other types of behaviour also exist
- Two key design questions:
  - what are the essential operations?
  - how to specify them?

Example: Customer



Example: IntSet



What are the essential operations?

- Focus on the *essential* operations first
- Some general guidelines:
  - focus on significant functional requirements
  - at least include one constructor and one observer
    - [Java] default constructor may be omitted
  - define a mutator for an mutable attribute
  - define data validation operations for attributes with domain constraints
  - define operations that ease programming with objects
  - define helper operations (to help other operations)

Operation specification guidelines (1)

- Use a ***well-defined design specification***
- Scope is usually `public` (some are `private`)
- Annotate with `@DOpt`, `@AttrRef` (where needed)
- Must <u>not</u> use keyword `static`
- Must take into account the attributes & abstract properties
- May use the `_post` postfix to denote value of a variable in the post-condition
- Can use keyword `this` to refer to other members

Well-defined design specification of an operation

- A design specification that makes precise and clear the behaviour of an operation
- Consists of two parts:
  - a structured behaviour description
  - operation header
- Specification can be defined at two levels:
  - logical: language-neutral specification
  - physical: language dependent
- Our focus: ***physical specification, Java as the target language***

Example: swap two numbers

```
/**
 * Swap two numbers
 * @requires
 *   xy != null /\ xy.length=2
 * @modifies xy
 * @effects
 *    xy = [xy_0[1], xy_0[0]]
 */
void swap(int[] xy)
```

Javadoc format

- We use the Javadoc format to write specification
  - a type of comment format that is used to generate code documentation
- Block comment of the form: /**...*/
- Support use of tags:
  - document tags: prefixed with '@' (e.g. @effects)
  - HTML tags, e.g.:
    -
: line break
    - <p>: paragraph break
    - <pre>...</pre>: pre-formated text
    - <tt>...</tt>: code snippets

Specification structure

- **@requires**: pre-conditions
  - only required for partial procedures
- **@modifies**: side-effects (if any)
  - list the parameter(s)
- **@effects**: post-conditions
  - state the transformation of inputs into output
- **@pseudocode**: the pseudocode (if any)
  - typically low level pseudocode statements

Example: swap two numbers

```
/**
 * Swap two numbers
 * @requires <tt>xy != null /\
 *                     xy.length=2</tt>
 * @modifies xy
 * @effects <tt>xy = [xy_0[1],
 *                        xy_0[0]]</tt>
 */
void swap(int[] xy)
```

detailed description of behaviour

Specification language

- A **Java-like language** that supports:
  - logical notation
  - reserved names (@requires, etc.) for the specification components

A Java-like language

- **Comments**: single and block comments
- Procedure definition
- Java's primitive and array **types**
- Keyword `null`
- **no semi-colon** at end of statement
- **indentation**, no curly brackets

Statements & operators

- Basic **statements**:
  - Java's variable declaration and assignment
  - Java's conditional and loop
  - `read`: read some data from some input
  - `print`: display some data to the standard output
  - `return`: return some data as output
- **High-level** (natural language) statements are also allowed
- **Operators:**
  - `eq` (==), `not eq` (!=), `lt` (<), `gt` (>), etc.

Operations on array

- `add x to a`:
  - add x to the next index position in a
- `put x in a`:
  - put x in any index in array a
- `delete x in a`:
  - set the first item matching x in a to a pre-defined constant used to denote the discarded state

Logical notation

Logical symbols	Textual form
$\wedge$	/\
$\vee$	\/
$\rightarrow$	->
$\leftrightarrow$	<->
$\forall$	for all
$\exists$	exist

Criteria of a good specification

- **Restrictive**:
  - to rule out unsatisfactory implementations
  - include @requires when necessary
- **General**:
  - to cover a majority of satisfactory implementations
  - use definitional-style description when necessary
- **Clear**: balance between
  - *conciseness*: consolidate statements, use pseudocode language syntax
  - *redundancy*: use example when necessary

Example: swap

```
/**
 * Swap two numbers
 * @requires xy != null /\ xy.length=2
 * @modifies xy
 * @effects xy = [xy_0[1], xy_0[0]]
 */
void swap(int[] xy)
```

Restrictive

General (definitional style)

Concise

Example: swap

Redundancy
(use example)

```
/**
 * Swap two numbers
 * @requires xy != null /\ xy.length=2
 * @modifies xy
 * @effects xy = [xy_0[1], xy_0[0]]
 *    e.g. xy=[1,2] /\ swap(xy)=[2,1]
 */
void swap(int[] xy)
```

Operation specification guidelines (2)

- Use a ***well-defined design specification***
- Scope is usually `public` (some are `private`)
- Annotate with `@DOpt`, `@AttrRef` (where needed)
- Must <u>not</u> use keyword `static`
- Must take into account the attributes & abstract properties
- May use the `_post` postfix to denote value of a variable in the post-condition
- Can use keyword `this` to refer to other members

Design specification guidelines for specific operation types

- **Creator** (also called constructors):
  - create objects of a class from attribute values
- [0] **Producer**: creates a new object from an object of the same type
- **Mutator**: change object state
- **Observer**: obtain information from object state
- **Default**: default language-specific operations (for all objects)
- **Helper**: utility operations that help others

Annotation DOpt

- Annotate an opreation
- Describes a behaviour pattern
- Has one property named `type`, which specifies the operation type:
  - The data type is the enum `OptType`
- **Syntax**: written before the operation header as:

  `@DOpt(type=t)`, where $t \in$ `OptType`.

Annotation AttrRef

- Describes reference to an attribute that is being manipulated by an operation's behaviour
- Annotate either operation or operation parameter:
  - for ***operation***: only used for non-constructor operations
  - for ***parameter***: mainly used for parameters of constructor operations
- Has one property named `value`, which specifies the name of the referenced attribute:
  - the data type is `String`

(cont'd)

- ***Syntax:***

  `@AttrRef(value=`*n*`)` or simply `@AttrRef(`*n*`)`,

  where *n* is the referenced attribute's name

  - for ***operation***: the statement is written before the operation header
  - for ***parameter***: the statement is written immediately before the declaration of the parameter

Annotation usage guidelines

- `AttrRef` is often (but not always) used with `DOpt`
- `DOpt` is usually NOT required for constructor and helper operations
- `AttrRef` is usually NOT required for parameters of *non-constructor* operations

Constructor

- Create a new object from arguments
- **Name**: same as the class
- **Return type**: omitted
- Two design methods:
  - create objects with default state and update them later (using mutator operations, discussed later)
  - object state may not be valid at creation
  - **essential constructor**:
    - ensures object state is valid at creation

Essential constructor

- **Parameters**:
  - one parameter for each non-optional, non-collection-typed attribute
  - use @AttrRef to map each parameter to attribute
  - types must match the attributes' concrete types
- **@effects**: if domain constraints apply then states data validation for the arguments:
  - throws NotPossibleException if violation occurs

Example: Customer

why?

```
/**
 * @effects <pre>
 *  if custID, name are valid
 *    initialise this as <custID,name>
 *  else
 *    throws NotPossibleException
 *  </pre>
 */
public Customer(@AttrRef("id") int custID,
                @AttrRef("name") String name)
      throws NotPossibleException
```

Mutator

- Update value(s) of attribute(s)
  - required for mutable attributes, forbidden for immutable ones
- Annotated with:
  - @DOpt.type=OptType.Mutator
  - @AttrRef.value=<*attribute-name*>
- Setter: a common mutator that directly sets the value of an attribute:
  - name: set*X* where X is attribute name (first letter capitalised)
  - return type: boolean
  - parameter: matches the attribute
- Return type: `false` if an error occurs (e.g. invalid input)

Example: Customer

why?

```
/**
 * @effects  <pre>
 *   if name is valid
 *     set this.name to name
 *     return true
 *   else
 *     return false</pre>
 */
@DOpt(type=OptType.Mutator) @AttrRef(”name”)
public boolean setName(String name)
```

Observer

- Obtain information about the object state
- Annotated with:
  - @DOpt.type=OptType.Observer
  - @AttrRef.value=<*attribute-name*>
- Getter: a common type of observer that directly gets value of an attribute
  - **name**: get*X*, where *X* is the attribute name (first letter capitalised
  - **parameters**: empty
  - **return type**: matches the attribute's type

Example: Customer

why?

```
/**
 * @effects return <tt>id</tt>
 */
@DOpt(type=OptType.Observer) @AttrRef(”id”)
public int getId()

/**
 * @effects  return <tt>name</tt>
 */
@DOpt(type=OptType.Observer) @AttrRef(”name”)
public String getName()
```

Default

- Operations that are common to all Java classes:
  - defined in the class `java.lang.Object` (from which all classes are derived)
- Three common operations:
  - `toString`, `equals`, `hashCode`
- Annotated with `@Override`
- Specification need not be defined, but can be added to explain the behaviour

toString()

```
@Override
public String toString()
```

- No arguments
- Returns a string representation of an object
  - similar to the abstract object definition

equals()

```
@Override
public boolean equals(Object o)
```

- Takes an `Object` argument and returns `boolean`
  - `true` if the argument is equal to the current object, `false` if otherwise
- Also means the two objects are behaviourally equivalent

[0] hashCode()

- Generates a hash value from object state
- For use as the storage key of an object in a hash-based collection
  - e.g. Hashtable, HashMap
- Not discussed further

Helpers

- Operations that perform tasks needed by other operations
- Three common types of helper:
  - repOK: short for “representation OK”
  - Data validation
  - Utility

repOK

```
/**
 * @effects <pre>
 *  if this satisfies abstract properties
 *    return true
 *  else
 *    return false</pre>
 */
public boolean repOK()
```

- Check if the object state satisfies the abstract properties
  - for testing the object and the overall implementation
- Specified using the above:
  - scope: usually `public` (but can also be `private`)

Data validation

- Validates input data against the domain constraints
  - invoked by constructor, setter, and repOK operations
- **Name**: validate*X*, where *X* is an attribute name (first letter capitalised)
- **Access modifier**: `private`
- **Parameters**: match the attribute
- **Return type**: boolean
- May also invoke other validate operations

E.g: Customer

```java
/**
 * @effects <pre>
 *   if id is valid
 *     return true
 *   else
 *     return false
 *   </pre>
 */
private boolean validateId(int id)

/**
 * @effects <pre>
 *   if name is valid
 *     return true
 *   else
 *     return false
 *   </pre>
 */
private boolean validateName(String name)
```

Utility

- Other helper operations:
  - determined based on the specifications of the existing operations
- Examples:
  - `IntSet.getIndex`: needed by `insert` and `remove`
  - `Rat.reduce`: performs a key operation
- Scope: (usually) `private`
  - made `public` if useful for outside access (e.g. `IntSet.isIn`)

Example: IntSet

```java
/**
 * @effects <pre>
 *  if x is in this
 *    return the index where x appears
 *  else
 *    return -1</pre>
 */
private int getIndex(int x)
```

(B) Collection class design

1) **What is collection class?**
2) **Design approach with annotation**

What is collection class?

- Collection classes differ from non-collection ones:
  - operations do not usually follow the usual set-get pairings
  - mutator and observer operations are designed to add/remove/observe one or some element(s) at a time, not all elements at once
- Example: interface `java.util.Collection<E>`
  - `add(E e)`
  - `remove(E e)`
  - `contains(E e)`

Design approach

- Collection class implements a marker interface
- Essential constructor has empty parameter list
- Essential mutators to maintain the collection
- Essential observers to obtain information about elements in the collection

Collection class marker

- The collection class must implement the marker interface `utils.collections.Collection`
  - this marker signifies that the class is a collection
- Unlike Java, it does not force the implementation of any operations:
  - although essential operations (specified through the OptTypes) are recommended
- Example:

```
import utils.collections.Collection;

public class IntSet implements Collection {

  //

}
```

Using DOpt with specific OptTypes

- Each operation of a collection class is marked with the operational annotation `DOpt`
- There are `OptTypes` specifically designed for the ***key*** mutator and observer operations:
  - AttrRef is not required for these operations
- Other `OptTypes` are still applicable to other operations of the collection, if needed

The specific OptTypes

- **MutatorAdd**: for operation that adds an element to the collection
- **MutatorRemove**: for the operation that remove an element from the collection
- **ObserverContains**: for the operation that checks if an element is in the collection
- **ObserverSize**: for the operation that returns the number of elements in the collection
- **ObserverIterator**: for the iteration abstraction of the collection (for future use)

Example: IntSet Constructor

why?

```
/**
 * @effects initialise <tt>this</tt> to be
 *    empty
 */
public IntSet()
```

Mutator

why?

```
/**
 * @modifies  <tt>this</tt>
 * @effects   <pre>
 *  if x already in this
 *    do nothing
 *  else
 *    add x to this, i.e., this_post=this+{x}</pre>
 */
@DOpt(type=OptType.MutatorAdd)
public void insert(int x)
```

(cont'd)

why?

```
/**
 * @modifies  <tt>this</tt>
 * @effects   <pre>
 *   if x is not in this
 *     do nothing
 *   else
 *     remove x from this, i.e. this_post=this-{x}
 *   </pre>
 */
@DOpt(type=OptType.MutatorRemove)
public void remove(int x)
```

Observer

why?

```
/**
 * @effects <pre>
 *    if x is in this
 *      return true
 *    else
 *      return false</pre>
 */
@DOpt(type=OptType.ObserverContains)
public boolean isIn(int x)
```

(cont'd)

why?

```
/**
 * @effects return the cardinality of <prtt>this</tt>
 */
@DOpt(type=OptType.ObserverSize)
public int size()
```

(cont'd)

why?

```
/**
 * @effects
 *  if this is not empty
 *    return Integer[] array of elements of this
 *  else
 *    return null
 */
@DOpt(type=OptType.Observer)
public Integer[] getElements()
```

Summary

- Annotation is a feature of high-level OOPL (Java, C#) that provides metadata for the code
  - State-of-the-art Java tools make extensive use of annotation
- Conventional OOP design lacks support for explicit design rules
- Three annotations are introduced to define essential design rules: @DOpt, @AttrRef, @DomainConstraint
- Collection classes need to additionally implement a marker interface

Q & A