

Classes, Objects and Invariants

School of Computing
Clemson University
Clemson, SC 29634 USA

CPSC 2150: Wednesday, Jan. 25, 2023

Slide Sources:

- 1 Previous/Current CPSC 2150 Instructors
- 2 The lecture slides have also benefitted from instructors at Ohio State: Paolo Bucci, Wayne Heym, Paul Sivilotti, and Bruce Weide.

Last Class

- 1 Design by Contract (DBC)
- 2 Preconditions and Postconditions
- 3 Javadocs and APIs
- 4 Introduction to Tracing Tables
- 5 Running Javadoc on School of Computing Unix Machines

Homework Assignment

- 1 Watch the various videos posted on your lab Canvas page.
- 2 Project 1 will be posted, so please start as soon as possible!

Introduction

- ▶ Why classes and objects?
 - ▶ Encapsulation and Information Hiding
- ▶ How to develop classes and objects in Java
 - ▶ See the “Classes in Java” video on lab canvas page.
- ▶ We will focus more on how to effectively design our systems to use classes
 - ▶ How to use Design-by-Contract with classes

Invariants

- ▶ Third type of contract: **Invariants**
- ▶ Preconditions and Postconditions were about things that are *true* before and after some method call or event
- ▶ Invariants are “always” *true* and must always be enforced.
- ▶ Invariants usually refer to some property of a variable in a class

Always...

- ▶ Often, we will say that invariants are something that is *always* true.
 - ▶ Should be kept at a high level
- ▶ In the code there will be times where the invariant is not true...
 - ▶ ...because it can take several lines of code to enforce it.
 - ▶ Change to one data member could trigger changes to another data member to maintain the invariant
 - ▶ but these are only transient conditions

Always...

- ▶ The invariant should be *true*:
 - ▶ After the constructor has been called
 - ▶ Before any `public` methods are invoked
 - ▶ After any `public` methods have finished
- ▶ The invariant may not be true:
 - ▶ During the execution of a method
 - ▶ Before the call of a `private` method
 - ▶ `private` method can be called by another method
 - ▶ Can use preconditions and postconditions to add the invariant
 - ▶ After the call of a `private` method
 - ▶ May be difficult to enforce if the invariant was not true before the call

Who is responsible for the invariant?

- ▶ The implementer of the class defines the invariant
- ▶ The implementer of any `public` class method is responsible for making sure that the invariant is always held to be true

One Simple Example

- ▶ Suppose Java didn't have a `Date` class
- ▶ How might we implement one?
 - ▶ The `Date` class example in the next slide has nothing to do with the Java `Date` class.

One Simple Example

Date class

- ▶ One possible Date class representation
 - ▶ `int day`
 - ▶ `int month`
 - ▶ `int year`
- ▶ Representation invariant for valid Date
 - ▶ day, month, and year should be in “sync” and valid
 - ▶ 02/31/2017 is not valid
- ▶ If day, month, and year can be modified arbitrarily, invalid dates could be represented
 - ▶ **Example:** 13/35/2017

One Simple Example

Date class

- ▶ Through **public** methods provided by a class, access to the **private** representation can be controlled
- ▶ **Date** class methods either
 - ▶ disallow setting the day, month, and year independently
 - ▶ put preconditions
- ▶ Unless there is related data to be “hidden” so that they can be kept in “sync”, grouped, and managed, classes may not be needed

If There Were No Invariants...

- ▶ All fields can be `public`. No methods are needed! Users can directly access and modify what they want.

```
public class Pencil {  
    public boolean hasEraser;  
    public String color;  
    public int length;  
}
```

- ▶ Note that user can set `length` to `-10`.
- ▶ If you don't want that, then you need invariants!

Example Class Declaration

```
/**
 * @invariant (hasEraser iff length >= 10) AND
 *             [color is valid] AND length >= 0
 */
public class Pencil {

    private boolean hasEraser;
    private String color = "red";
    private int length;

    public int sharpen(int amount) {
        // Implementation of sharpen goes here
    }

    public String getDescription() {
        // Implementation of getDescription goes here
    }

    // Other things in this class...
}
```

Example Class Declaration

- ▶ Note that invariant is a Boolean expression. Here, `hasEraser` is `true` if and only if `length` ≥ 10 .

```
/**
 * @invariant (hasEraser iff length >= 10) AND
 *             [color is valid] AND length >= 0
 */
public class Pencil {

    private boolean hasEraser;
    private String color = "red";
    private int length;

    // Other things in this class...
}
```

Example Class Declaration

```
/**
 * @invariant (hasEraser iff length >= 10) AND
 *             [color is valid] AND length >= 0
 */
public class Pencil {

    private boolean hasEraser;
    private String color = "red";
    private int length;

    /**
     * @pre amount >= 0 AND amount <= length
     * @post length = #length - amount AND
     *       sharpen = length AND
     *       color = #color AND hasEraser = (length >= 10)
     */
    public int sharpen(int amount) {
        // Implementation of sharpen goes here
    }

    // Other things in this class...
}
```

Example Class Declaration

```
/**
 * @invariant (hasEraser iff length >= 10) AND
 *             [color is valid] AND length >= 0
 */
public class Pencil {

    private boolean hasEraser;
    private String color = "red";
    private int length;

    /**
     * @pre amount >= 0 AND amount <= length
     * @post length = #length - amount AND
     *        sharpen = length AND
     *        color = #color AND hasEraser = (length >= 10)
     */
    public int sharpen(int amount) {
        length = length - amount;
        hasEraser = (length >= 10);

        return length;
    }

    // Other things in this class...
}
```

Invariant Summary

- ▶ Why is this statement needed?
 - ▶ `hasEraser = (length >= 10);`

Invariant Summary

- ▶ Why is this statement needed?
 - ▶ `hasEraser = (length >= 10);`
- ▶ Because otherwise representation invariant will be violated!
 - ▶ For a brief period of time the invariant may not have been met
 - ▶ But only between two lines of code
- ▶ The constructor must guarantee invariant.
- ▶ After that, the invariant:
 - ▶ May be assumed at the beginning of every `public` method.
 - ▶ Must be guaranteed at the end of every `public` method.

Good Practice: Establish Invariant

- ▶ Class representations typically have a *convention* or *representation invariant*
 - ▶ What is true of the state for all instances?
 - ▶ **Eg:** All long pencils have erasers
`hasEraser iff length >= 10`
 - ▶ So the state: (false, "green", 14) is not valid
- ▶ Invariant must hold after constructor!
- ▶ **Caution:** Be careful when a constructor and other `public` methods call another `public` method
 - ▶ Danger! Convention (representation invariant) might not hold at call point
 - ▶ Use `private` helper methods.
 - ▶ If you do call `public` methods, YOU are the client and need to enforce the contracts!

Example Constructor

```
/**
 * @invariant (hasEraser iff length >= 10) AND
 *             [color is valid] AND length >= 0
 */
public class Pencil {

    private boolean hasEraser;
    private String color = "red";
    private int length = 14;

    /**
     * @pre [c is a valid color]
     * @post color = c
     */
    public Pencil (String c) {
        color = c;
        hasEraser = (length >= 10);
        //enforce the invariant!
    }

    // Other things in this class...
}
```

Members

- ▶ Two kinds of members in a class declaration

- ▶ Fields, i.e. data (determine the *state*)

```
private boolean hasEraser;  
private String color = "red";  
private int length;
```

- ▶ Methods, i.e. procedures (*access/modify* the state)

```
public int sharpen(int amount) {  
    length = length - amount;  
    hasEraser = (length >= 10);  
  
    return length;  
}
```

Visibility

- ▶ Members can be **private** or **public**

- ▶ Member-by-member declaration

```
private String color;  
public int length;  
public int sharpen(int amount) {  
    // Implementation of sharpen goes here  
}
```

- ▶ **private** members:

- ▶ Can be accessed only by instances of same class
 - ▶ Provide concrete implementation / representation

- ▶ **public** members:

- ▶ Can be accessed by any object
 - ▶ Provide abstract view (client-side)

Good Practice: Member Declarations

- ▶ Group member declarations by visibility
 - ▶ Java's convention: `private` members at top
- ▶ No fields should be `public`
 - ▶ **Avoid:** Incorrect use of getters and setters

```
public class Pencil {  
    private int length;  
  
    public int getLength() {  
        return length;  
    }  
  
    public void setLength(int newLength) {  
        length = newLength;  
    }  
}
```

- ▶ **Note:** Tied to specific `private` variable names
- ▶ **Better:** Provide `public` member methods for observing and controlling *abstract* values of objects
 - ▶ **Eg:** 2 pencil classes, `PencilA` and `PencilB` classes should have *exactly the same accessors* (including signatures)

Contracts and Information Hiding

- ▶ Our contracts help us hide information
 - ▶ Clients can rely on our contracts instead of looking at code
- ▶ But our contracts often expose the names of `private` variables
 - ▶ In general it's ok to reveal obvious information in the contracts
 - ▶ A `Rectangle` class will have a *length* and a *width*
 - ▶ A `Bank` account class has a *balance*
 - ▶ We still try to hide what happens in the code
- ▶ When we discuss **Interfaces**, we'll have a new way to hide this information

The Audience of Contracts

- ▶ Remember who the audience of the contracts are
- ▶ Preconditions and Postconditions are meant for the client of the class (aka other software engineers)
 - ▶ Also help the implementer on a large project
 - ▶ The design team comes up with the contracts
 - ▶ Developers implement according to the contracts
- ▶ The invariants are meant for the implementer
- ▶ Client may learn about invariants through the preconditions
 - ▶ **Invariant:** `length >= 0`
 - ▶ `sharpen(int amount)` has precondition:
@pre `amount >= 0 AND amount <= length`

Types of Objects

- ▶ As our systems get larger, we start to classify the types of our objects and classes
 - ▶ Entity Objects
 - ▶ Boundary Objects
 - ▶ Control Objects
- ▶ In simpler programs, these objects tend to get mixed together

Types of Objects

Entity Objects

- ▶ Most classes you would have made in previous programming courses were **Entity Objects**
- ▶ Entity objects represent some real-world entity
 - ▶ Car
 - ▶ Employee
 - ▶ Pizza
- ▶ Can be more abstract concepts as well
- ▶ Entity objects are usually nouns

Types of Objects

Boundary Objects

- ▶ **Boundary Objects** represent the interactions between the system and it's users (or other systems)
- ▶ **Ex 1:** The user interface is in a boundary object
 - ▶ Complex systems and GUIs will have many boundary objects
- ▶ **Ex 2:** Our **Scanner** object is a boundary object
- ▶ They exist to create a connection, not to encapsulate data
 - ▶ Still encapsulating functionality
- ▶ These boundary objects are what needs to change if the UI changes
 - ▶ Or if the external system we interact with changes

Types of Objects

Control Objects

- ▶ **Control Objects** are responsible for the actual flow of the program
 - ▶ Handles interactions between the boundary and entity objects
- ▶ Makes sure events happen in a certain order
- ▶ Our `main` method is responsible for the flow of the program
 - ▶ In Java, our `main` method is in a class, creating a control object
 - ▶ In Java, every class can theoretically have a `main` method
 - ▶ We want to avoid this and try to keep our control objects separate

Homework Assignment

- 1 Watch the various videos posted on your lab Canvas page.
- 2 Project 1 will be posted, so please start as soon as possible!

Summary

- 1 Classes and Objects
- 2 Invariants
- 3 Good Practices
- 4 Types of Objects