CSE 331

Mutation and immutability

slides created by Marty Stepp
based on materials by M. Ernst, S. Reges, D. Notkin, R. Mercer, Wikipedia
http://www.cs.washington.edu/331/

Mutation

- **mutation**: A modification to the state of an object.

```
Point p = new Point(3, 5);
System.out.println(p);         // (3, 5)
p.translate(1, 3);             // mutator
System.out.println(p);         // (4, 8)
```

- Mutation must be done with care.
  - Can the object's state be damaged?
  - Is the old state important? Is it okay to "lose" it?
  - Do any other clients depend on this object?
    - Do they expect that its state will not change?

Defensive copying

- **defensive copy**: To duplicate an object prior to making a mutation.

```
Point p = new Point(3, 5);
System.out.println(p);       // (3, 5)
Point copy = p.clone();
copy.translate(1, 3);        // mutator
System.out.println(copy);    // (4, 8)
System.out.println(p);       // (3, 5)
```

- Sometimes you then perform the mutation to the copy.
- Sometimes you perform the mutation to the original object.
  - The copy serves as a "backup" in case the mutation is undesirable.

- **EJ Tip #39**: Make defensive copies when needed.

A poor design

- Suppose we have a `BankAccount` class:
  - `getBalance` method returns the bank account's balance.
  - The method also charges a $1.50 fee if you ask for the balance too many times (calling the method more than 3x per day).

- Why is this a poor design?
  - Should not combine a crucial accessor with an unrelated mutation.
    - Impossible to access without (unintentionally?) mutating.
  - Client might call it many times without thinking, to print balance etc.
    - Another client might have already called the method before me.

  - **side effects**: Additional externally visible behavior beyond the core functionality of a method.

Mutator vs. accessor

- `public String next()    // Scanner`
- `public E next()         // Iterator<E>`
  - `Scanner`/`iterator`'s `next` method both accesses and mutates.

- **Horstmann Tip 3.4.3**: Whenever possible, keep accessors and mutators separate. Ideally, mutators return `void`.
  - One exception: Returning an old value (e.g. `HashMap get`)

- What would be a better design for `Scanner` and `iterator`?
  - a `getCurrent` method that returns current element
  - a `gotoNext` or next method that advances to the next element

Mutator vs. producer

- It is important to know whether a call performs its modifications "in place" on arguments passed to it, or creates new objects.
  - A *mutator* method modifies an existing object.
  - A *producer* method creates and returns a new object.

- Example: `Arrays.sort(int[])`
  - Does it sort the array passed? Or...
  - Does it leave that array untouched and return a new sorted array?
    - *(It sorts the array passed.)*

- Example: Maps have methods named `keySet` and `values` that return collections of all the keys or values in the map, respectively.
  - If I modify one of those returned collections, does it modify the map?
    - *(Yes.)*

"Modifying" strings

- What is the output of this code?

```
String name = "lil bow wow";
name.toUpperCase();
System.out.println(name);
```

- The code outputs `lil bow wow` in lowercase.
- To capitalize it, we must reassign the string:

```
name = name.toUpperCase();
```

  - The `toUpperCase` method is a producer, not a mutator.
  - Why must we do call the methods in this way?
  - What is going on with Strings to require this sort of usage?

Immutable classes

- **immutable**: Unable to be changed (mutated).
  - Basic idea: A class with no "set" methods (*mutators*).

- In Java, Strings are immutable.
  - Many methods appear to "modify" a string.
  - But actually, they create and return a new string (*producers*).

- Why was Java designed this way?
  - Why not make it possible to mutate strings?
  - Is there any way to mutate a string?
  - What is so bad about the idea of mutating a string object?

If Strings were mutable...

- What could go wrong if strings were mutable?

```
public Employee(String name, ...) {
    this.name = name;
    ...
}

public String getName() {
    return name;
}
```

  - A client could accidentally damage the Employee's name.

```
String s = myEmployee.getName();
s.substring(0, s.indexOf(" "));  // first name
s.toUpperCase();
```

Mutable StringBuilder

- A `StringBuilder` object holds a mutable array of characters:

method	description

`StringBuilder()`
`StringBuilder(`**capacity**`)`
`StringBuilder(`**text**`)`
	new mutable string, either empty or with given initial text

`append(`**value**`)`	appends text/value to string's end

`delete(`**start**, **end**`)`
`deleteCharAt(`**index**`)`
	removes character(s), sliding subsequent ones left to cover them

`replace(`**start**, **end**, **text**`)`	remove start-end and add text there

`charAt(`**index**`), indexOf(`**str**`),`
`lastIndexOf(`**str**`), length(),`
`substring(`**start**, **end**`)`
	mirror of methods of String class

`public String` **`toString`**`()`	returns an equivalent normal String

String + implementation

- The following code runs surprisingly slowly. Why?

```
String s = "";
for (int i = 0; i < 40000; i++) {
    s += "x";
}
System.out.println(s);
```

- Internally, Java converts the loop into the following:

```
for (int i = 0; i < 40000; i++) {
    StringBuilder sb = new StringBuilder(s);
    sb.append("x");
    s = sb.toString();  // why is the code slow?
}
```

- **EJ Tip #51**: Beware the performance of string concatenation.

Minimizing mutability

- **Effective Java Tip #15**: Minimize mutability.

- Why?
  - easier to design, implement, and use immutable objects
  - less prone to developer error
  - less prone to misuse by clients
  - more secure
  - can be optimized for better performance / memory use (sometimes)

  - from Effective Java: *"Classes should be immutable unless there is a very good reason to make them mutable."*
    - *"If a class cannot be immutable, limit its mutability as much as possible."*

FP and immutability

- **functional programming**: Views a program as a sequence of *functions* that call each other as *expressions*.
  - everything is immutable (almost)
  - variables' values cannot be changed (only re-defined)
  - functions' behavior depends only on their inputs (no side effects)

- Benefits of this programming style?
  - the compiler/interpreter can heavily *optimize* the code
  - much easier to understand/predict behavior of code;
    code can be more thoroughly *verified* for correctness
  - *robust*; hard for one chunk of code to damage another
  - lack of side effects reduces *dependency* between code
    - allows code to be more easily *parallelized*

Making a class immutable

- 1. Don't provide any methods that modify the object's state.
- 2. Ensure that the class cannot be extended.
- 3. Make all fields `final`.
- 4. Make all fields `private`. (ensure encapsulation)
- 5. Ensure exclusive access to any mutable object fields.
  - Don't let a client get a reference to a field that is a mutable object. (Don't allow any mutable representation exposure.)

The final keyword

- **final**: Unchangeable; unable to be redefined or overridden.
- Can be used with:
  - local variables (value can be set once, and can never be changed)
  - fields
  - static fields (they become "class constants")
  - classes (the class becomes unable to be subclassed)
  - methods (the method becomes unable to be overridden)
- **Effective Java Tip #17:** Design and document for inheritance or else prohibit it (by making your class `final`).

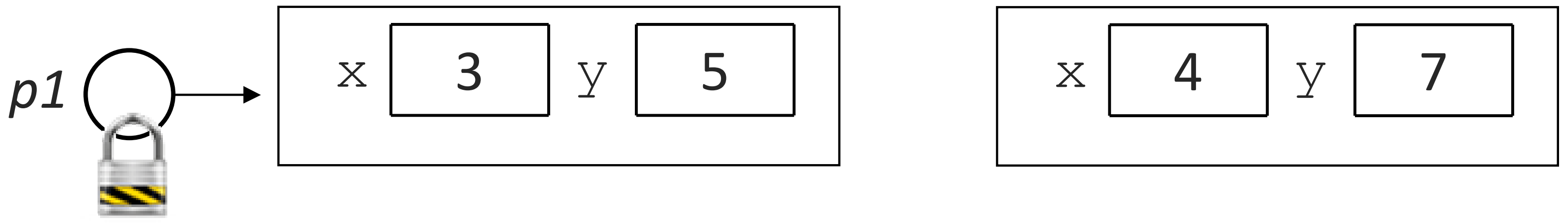
Examples of final

- on a local variable: `final int answer = 42;`
- on a field: `private final String name;`
  - set in constructor
- on a static constant: `public static final int DAYS = 7;`
- on a class: `public final class Point {`
  - no class can extend `Point`
- on a method: `public final int getX()`
  - no class can override `getX` (not necessary if class is already `final`)

Final references

- Setting a reference variable final means that it can never be reassigned to refer to a different object.
  - You can't set that reference to refer to another object later (=).
  - It does not mean that the object's state can never change!
  - Use caution when allowing clients to refer to your object's fields. Even if it is `final`, the client might be able to modify the object.

```
final Point p1 = new Point(3, 5);
p1 = new Point(4, 7);          // error; final
p1.translate(1, 2);            // allowed (oops?)
```



Mutable Fraction class

```
public class Fraction implements Cloneable, Comparable<Fraction> {
    private int numerator, denominator;

    public Fraction(int n)
    public Fraction(int n, int d)
    public int getNumerator(),  getDenominator()
    public void setNumerator(int n),  setDenominator(int d)

    public Fraction clone()
    public int compareTo(Fraction other)
    public boolean equals(Object o)
    public String toString()

    public void add(Fraction other)
    public void subtract(Fraction other)
    public void multiply(Fraction other)
    public void divide(Fraction other)
}
```

- How would we make this class immutable?

Immutable Fraction class

```
public final class Fraction implements Comparable<Fraction> {
    private final int numerator, denominator;

    public Fraction(int n)
    public Fraction(int n, int d)
    public int getNumerator(),  getDenominator()
    // no more setN/D methods

    // no clone method needed
    public int compareTo(Fraction other)
    public boolean equals(Object o)
    public String toString()

    public Fraction add(Fraction other)       // past mutators
    public Fraction subtract(Fraction other)  // are producers
    public Fraction multiply(Fraction other)  // (return a new
    public Fraction divide(Fraction other)    // (object)
}
```

Immutable methods

```
// mutable version
public void add(Fraction other) {
    numerator = numerator * other.denominator
              + other.numerator * denominator;
    denominator = denominator * other.denominator;
    reduce();   // private helper to reduce fraction
}

// immutable version
public Fraction add(Fraction other) {
    int n = numerator  * other.denominator
          + other.numerator * denominator;
    int d = denominator * other.denominator;
    return new Fraction(n, d);
}
```

- former mutators become *producers*
  - create/return a new immutable object rather than modifying this one
  - a "functional" style; maps input objects to output objects

Documenting mutability

In your Javadoc comments, you should always:

- document whether your class is mutable

- if class is mutable, then specifically document:
  - which methods modify/mutate the object
  - under what circumstances they mutate it
  - in what way they mutate it

  - Javadoc @modifies tag (include `this` as necessary)

Is it mutable?

```
Point p1 = new Point(5, 7);
Point p2 = new Point(2, 8);
Line line = new Line(p1, p2);
p1.translate(5, 10);          // move point p1
```

- Is the `Line` class mutable or immutable?
  - It depends on the implementation.
    - If it creates an internal copy of p1/p2: immutable
    - If it stores a reference to p1/p2: mutable
  - Lesson: Storing a mutable object in an immutable class can expose its internal representation. (That's bad.)

Is it mutable?

```
public class Course {
    private final Set<Weekday> days;

    public Course(String name, int credits,
                  Set<Weekday> days, ...)  {
        this.days = days;
        ...
    }

    public Set<Weekday> getDays() {
        return days;
    }
    ...
}
```

- Is the `Course` class mutable or immutable?
  - What if we removed or modified the `getDays` method?

Example client mutation

```
Set<Weekday> days = new HashSet<Weekday>();
days.add(Weekday.MONDAY);
days.add(Weekday.WEDNESDAY);
Course c = new Course("CSE 331", 4, days, ...);
...
days.add(Weekday.FRIDAY);  // c is modified!
```

- Since the course stores a direct copy of the set of days passed in, the object is in fact *mutable*.
  - What should the `Course` author do if he/she wants to provide a truly immutable class?

Immutable collections

- The `Collections` class in `java.util` has these static methods for wrapping up a given collection in an immutable casing:

Method	Description
`unmodifiableCollection(`**coll**`)`	Returns an immutable wrapping around the given collection. Any object that tries to mutate the wrapped collection receives an UnsupportedOperationException.
`unmodifiableList(`**list**`)`	
`unmodifiableMap(`**map**`)`	
`unmodifiableSet(`**set**`)`	

```
List<String> names = new ArrayList<String>();
names.add(......);
...
// pass the list to a method I don't trust
evilMethod(Collections.unmodifiableList(names));
```

Representation exposure

- **representation exposure:** When an object allows other code to examine or modify its internal data structures. (A bad thing.)

- **Law of Demeter**: An object should know as little as possible about the internal structure of other objects with which it interacts.
  - An object, especially an "immutable" one, should not expose its representation by returning a reference to its internal goodies.
    - sometimes called "shallow immutability" if not done properly

- If your object has an internal collection: don't return it!
  - or return a copy, or an immutable wrapper around it
- If your object has mutable objects as fields: don't let outside clients access them! Copy them if sent in from the outside.

Pros/cons of immutability

- Immutable objects are **simple**.
  - You know what state they're in (the state in which they were born).
- Immutable objects can be **freely shared** among code.
  - You can pass them, return them, etc. without fear of damage.
  - Immutable objects are also inherently *thread safe* (seen later).
- Con: Immutable objects can consume more **memory**.
  - Need a unique instance for each unique abstract value used.
  - Workaround: Factory methods and other techniques can help to consolidate equivalent objects (seen later).

Mutability and collections

- If you place a mutable object into a hash/tree collection, mutate it, then search for it, the object may be "lost":

```
Time dinner = new Time( 7, 00, true);

Set<Time> set = new TreeSet<Time>();
set.add(new time( 8, 00, false);     // breakfast
set.add(new Time(12, 00, true));     // lunch
set.add(dinner);                     // dinner

System.out.println(set.contains(
         new Time(7, 00, true)));    // true
breakfast.shift(30);
System.out.println(set.contains(
         new Time(7, 30, true)));    // false!
```

- Where did my `dinner` go? What is wrong?

Temporary mutability

- Another option when pure immutability is too restrictive:
  - Give the class an initial mutable state.
  - Have a method you can call to "lock" the object's state.
  - After it has been "locked", no more changes are permitted.
    - (For this to make sense, you shouldn't provide an easy way to "unlock" it.)

```
Course course = new Course("CSE 331", Quarter.SPRING);
course.addStudent(jim);
course.addStudent(sue);
...
course.lockRegistration();
course.addStudent(bob);  // exception
```