Effective CS32

John Jannotti

course/cs0320/www/lectures/

Jan 31, 2017
Announcements

- Testing lab is out, and lab hours have begun.
- Stars “Gear-Up” session **tonight**, 6:30pm (BERT 130).
- Next week
  - Try to have a functional Stars repl/cli by next Tuesday’s class.
  - Lab 2 is on HTML. Then you can work on UI and details.
- **Jane Street Estimathon**¹ **today** CIT 368 5pm
- CS Department “Town Hall”, Thursday 4:30pm

¹Original title includes exclamation point here.
Help on getting help

- Never say “I tried to do X, but it doesn’t work.”
- Report exactly how it didn’t work. For example:
  - It didn’t compile.
  - It compiled and ran, but threw an exception.
  - It ran, but I got a different answer than I expected.
- In all cases, include details of the most pertinent error. The compile error, the exception message and stacktrace, or the results.
- But read them first!
- Report on things you tried to fix it. That prevents fruitless speculation.

A clear problem report is halfway to a solution. (And easier to search for, from other students.)
This lecture may be review, or it may not.

It is not intended to be about Java, *per se*.

But you should speak up if I show *anything* you don’t understand.

Today we’ll talk about good simple class and package design.

Thursday, we’ll talk more about interfaces and inheritance.

“Effective Java” by Joshua Bloch goes into all of this and more.

- I don’t require a textbook, but you could certainly treat this as one.
  It’s great.
Class Design

- **Principles to consider**
  - **Simplicity**
    - Ease of use.
    - Ease of understanding. (Principle of Least Astonishment)
    - Ease of coding.
  - **Abstraction**
    - Hide as much as possible.
    - Yet retain flexibility.
    - Common interfaces to disparate functionality (multiple implementations)
  - **Correctness**
    - Prevent misuse (e.g. don’t set a negative age).
    - Think ahead about testing. You can even “test ahead.”

- **Design Methodology**
  - Consider alternatives. Mutable? Inheritance? Primitives?
  - Evaluate possibilities. Try using them (write tests first).
Package Design

- Similar choices
- But at a higher-level.
- And focusing more on visibility than implementation.
- Fewer public classes and interfaces means easier understanding.
- Don’t write “public class Something” out of habit.
Which can legally be put in a single Java source file?

A) A top-level\(^2\) class, declared as `private class Person ...`

B) Several top-level constants.

C) Exactly two top-level classes, one public, one not.

D) Exactly two top-level classes, both public.

E) Exactly one top-level public function.

---

\(^2\)If the word “top-level” confuses you, think of it as “global”. It’s only necessary to make the question technically accurate.
Case study: A geography library

- You need to represent a location on the Earth.
- You’ll care about distances between them.
- Finding nearby locations.
- Some notion of “inside” an area.

Discuss ideas.
More concrete: A LatLng and a BoundingBox

- We’ll create two classes.
  - We’ve already made our first choice.
  - We don’t want to pass latitude and longitude as primitives.
  - Ick: distance(double lat1, double lng1, double lat2, double lng2);

- A location on the Earth — LatLng
- An square(ish) area — BoundingBox

Think about how you would like to represent them.

Think about how you would like client code to use them.

- Consider construction from simpler things.
- Consider useful “questions”.
- Consider creation from “algebra” on types.
- Consider (if you allow) mutation.
More concrete: A (too) simple implementation

```java
1 class LatLng {
2   double lat;
3   double lng;
4 }
```

- That’s probably as few keywords as you could possibly use...
- But what have we ended up with?
- Which callers can declare (and construct) LatLng objects?

- Which can access (and modify) the members? (lat and lng)
More concrete: A (too) simple implementation

```java
class LatLng {
    double lat;
    double lng;
}
```

- That’s probably as few keywords as you could possibly use...
- But what have we ended up with?
  - Which callers can declare (and construct) LatLng objects?
    - Any other class in the same package.
    - In this case, it’s the “default” package.
    - You should never do that.
- Which can access (and modify) the members? (lat and lng)
More concrete: A (too) simple implementation

```java
1 class LatLng {
2   double lat;
3   double lng;
4 }
```

- That’s probably as few keywords as you could possibly use...
- But what have we ended up with?
- Which callers can declare (and construct) LatLng objects?
  - Any other class in the same package.
  - In this case, it’s the “default” package.
  - You should never do that.
- Which can access (and modify) the members? (lat and lng)
  - Any code that can get one, can modify it.
  - Even outside package!
  - `Earth.getNorthPole().lat = 45;`
Consider visibility of classes and fields

```java
package edu.brown.cs.jj.geo;

public class LatLng {
  private double lat;
  private double lng;
}
```

- Always work in a package. Think about reuse.
- Many of your classes will be public, not not all!
- Why are the members private?
- So what else do we need?
First, a constructor

```java
package edu.brown.cs.jj.geo;

public class LatLng {
    public LatLng(double lat, double lng) {
        this.lat = lat;
        this.lng = lng;
    }

    private double lat;
    private double lng;
}
```

- Eschew default constructors
- But this constructor is not safe.
- (this should be rare. Just constructors and setters.)
First, a constructor

```java
package edu.brown.cs.jj.geo;

public class LatLng {

    public LatLng(double lat, double lng) {
        this.lat = lat;
        this.lng = lng;
    }

    private double lat;
    private double lng;
}
```

- Eschew default constructors
- But this constructor is not safe. new LatLng(91, 192);
- (this should be rare. Just constructors and setters.)
We use “private” for a reason, not ritual.

```java
public LatLng(double lat, double lng) {
  this.lat = checkLatitude(lat);
  this.lng = checkLongitude(lng);
}

private static double checkLatitude(double lat) {
  if (lat < -90.0 || lat > 90.0)
    throw new IllegalArgumentException(lat + " is out of range");
  return lat;
}
```

- LatLng makes no sense with lat or lng out of range.
- Don’t allow it.
- Use reasonable builtin Exception types when possible.
- Methods can and should be private, too.

What does static mean on these check method? How could it be useful?
Use public methods to export an “interface”

```java
import java.lang.Math;

public static double EARTH_RADIUS_M = 6.371e6;

public double distance(LatLng other) {
    if (equals(other))
        return 0.0;

    double a1 = Math.toRadians(lat);
    double b1 = Math.toRadians(lng);

    return EARTH_RADIUS_M * acos(c);
}
```

Avoid wildcard imports. Don’t import java.lang.*
Define toString()

```
@override
public String toString() {
    return String.format("(%1.3f,%1.3f)", lat, lng);
}
```

- Debugging and logging becomes much easier.
- String.format reads a lot nicer than a bunch of +’s.

What is @Override?
Define equals()

```java
@Override
public boolean equals(Object o) {
    if (o == this) return true;
    if (!(o instanceof LatLong)) return false;

    LatLong x = (LatLong) o;
    return (x.lat == lat && x.lng == lng);
}
```

- Without it, new LatLong(1,1).equals(new LatLong(1,1) is false.
- Why take an Object instead of a LatLong?
- @Override saves you if you forget.
- What happens if o is null?
- There are deep questions about equality... (that's a pun)
Define hashCode()

```java
import java.util.Objects;

@Override
public int hashCode() {
    return Objects.hash(lat, lng);
}
```

- Without it using LatLng as a key in a HashMap is badly broken.
- Use the Objects class to make this less tedious. (Note the “s”)
- Goal: few collisions.
- Required: a.hashCode() == b.hashCode() if a.equals(b);
Often (but not always!) member data should be available

- Do callers need access to the individual latitude or longitude?
- If so, how should we give it to them?
  - public double lat;

```
public double lat;
```

```
public double lat() { return lat; }
```

```
public final double lat;
```

- **final** means `lat` can't be reassigned, even by `LatLong`.
- It is one of the few ways that public can still be "safe."

- Beware of fields that are themselves mutable! Don't expose an array even if it's final.
- `final lat` must be assigned once, in a constructor.
- Our coding style requires private. Your future team probably does too.
Often (but not always!) member data should be available

- Do callers need access to the individual latitude or longitude?
- If so, how should we give it to them?
  - `public double lat;
  - `public double lat() { return lat; }
  - `public final double lat;

`final` means `lat` can't be reassigned, even by `LatLong`. It is one of the few ways that `public` can still be "safe." Beware of fields that are themselves mutable! Don't expose an array even if it's `final`. `final lat` must be assigned once, in a constructor. Our coding style requires `private`. Your future team probably does too.
Often (but not always!) member data should be available

- Do callers need access to the individual latitude or longitude?
- If so, how should we give it to them?
  - `public double lat;`
  - `public double lat() { return lat; }`
  - `public double getLat() { return lat; }`

```
public final double lat;
final
```
means `lat` can't be reassigned, even by `LatLong`. It is one of the few ways that public can still be "safe."

```
Beware of fields that are themselves mutable! Don't expose an array even if it's final.
```

```
final lat must be assigned once, in a constructor.
```

Our coding style requires `private`. Your future team probably does too.
Often (but not always!) member data should be available

Do callers need access to the individual latitude or longitude?
If so, how should we give it to them?
  - public double lat;
  - public double lat() { return lat; }
  - public double getLat() { return lat; }
  - public final double lat;

final means lat can’t be reassigned, even by LatLong.
It is one of the few ways that public can still be “safe.”
Beware of fields that are themselves mutable! Don’t expose an array even if it’s final.
final lat must be assigned once, in a constructor.
Our coding style requires private. Your future team probably does too.
Sometimes (less often!) member data should be modifiable

- Do callers need to modify the individual latitude or longitude?
- If so, how should we allow them?
  - `public double lat;`
Sometimes (less often!) member data should be modifiable

- Do callers need to modify the individual latitude or longitude?
- If so, how should we allow them?
  - public double lat;
  - public void setLat(double l) { lat = l; }
Sometimes (less often!) member data should be modifiable

- Do callers need to modify the individual latitude or longitude?
- If so, how should we allow them?
  - public double lat;
  - public void setLat(double l) { lat = l; }
  - " LatLong setLat(double l) { lat = l; return this;}"
Sometimes (less often!) member data should be modifiable

- Do callers need to modify the individual latitude or longitude?
- If so, how should we allow them?
  ```java
  public double lat;
  public void setLat(double l) { lat = l; }
  " LatLong setLat(double l) { lat = l; return this; }
  ```

Consider that immutable, but easily built, LatLng can be safer.

```java
  public LatLong withLat(double newLat) {
    return new LatLong(newLat, lng);
  }
  LatLong x = new LatLong(45, 100);
  LatLong y = x.withLat(50);
```
First question for a new type — does it have to be mutable?

- Prefer immutability.
- Simplifies reasoning, especially in threaded code.
- But not just there. Consider HashMap keys, shared objects...
- When to allow mutation?
  - Fundamental to the intended use: StringBuilder, Map as cache
  - Performance: Matrix library, Image manipulation
Which is likely to be the best first line for LatLng.setLat(double lat)?
A) this.lat = lat;
B) this.lat = checkLatitude(lat);
C) assert (lat <= 90.0 && lat >= -90.0);
D) assert (this.lat = checkLatitude(lat))
E) throw new UnsupportedOperationException("LatLng is immutable");
Assume checkLatitude is defined as before.
public class BoundingBox {

    public BoundingBox(LatLong... points) {
    
    }

} 

- Plenty of reasonable representations. “reps”
public class BoundingBox {
    public BoundingBox(LatLong... points) {
        .
        .
        .
        }
    }

• Plenty of reasonable representations. “reps”
• private final double[] limit = new double[4];
• private double north; private double south; ...
• private final LatLong[] corners = new LatLong[4];
• private LatLong ne; private LatLong sw;
public class BoundingBox {
  public BoundingBox(LatLong... points) {
    .
    .
    .
    .
  }
}

- Plenty of reasonable representations. “reps”
- private final double[] limit = new double[4];
- private double north; private double south; ...
- private final LatLong[] corners = new LatLong[4];
- private LatLong ne; private LatLong sw;
- Tradeoffs? Strive for clarity, avoid redundancy.
- Your choice of accessors should not depend on these decisions.
- Why would public LatLong[] get_corners() be tricky?
Consider writing `repOk()`

```java
private boolean repOk() {
    return limit.length == 4 && limit[N] > limit[S];
}

private boolean repOk()
{
    return north > south;
}

private boolean repOk() {
    return corners.length == 4 &&
    corners[NW].lat == corners[NE].lat &&
    corners[NW].lat > corners[SE].lat ... 
}

private boolean repOk()
{
    return ne.lat > sw.lat;
}
```

That may also help you decide on the best representation. Add assert statements to your methods that call `repOk`. (Where?)
Javadoc is source-embedded documentation

Use `mvn site` to generate HTML from Javadoc (and other reports).

```java
public class BoundingBox {
    . . .
    /**
     * Determines whether a point is located inside the BoundingBox. The point argument \{@link LatLong\}
     * must be non-null.
     *
     * @param pt The point to test for membership in the BoundingBox. Being ' ' on ' ' a line is considered in.
     * @return true, iff pt is inside the BoundingBox. Being ' ' on ' ' a line is considered in.
     */
    public boolean contains(LatLong pt) { . . . }
```
Review

- Keep the Code Simple
  - Well documented
  - Hide information as much as possible.
- Continually rethink the design.
  - Especially at the early stages
  - Can you simplify the abstraction?
    - Reduce the number of methods.
    - Reduce the number of parameters.
    - Make them easier to use.
  - Can you hide any more details? Should you?
  - How might a caller screw up? Can you prevent it?
- If you don’t see tradeoffs, you’re missing them.
“Boring” details

- Coding style is important
  - Spaces
  - Comments and documentation (use language tools)
  - Make the code readable
    - Indentation, spacing, blank lines
    - Variable names
    - Consistency often matters more than anything.
    - You are communicating with *humans*
  - Consistent organization
    - Ordering of elements (imports, methods)
    - Naming conventions

- We codified some of this in our Checkstyle configuration (mvn site)
- Use Checkstyle often and configure your editor, or you will be overwhelmed.