CSE 331
Software Design & Implementation
Topic: Subtyping; Ethics

💬 Discussion: How long would you last in a Zombie apocalypse?
Reminders

- Think of HW5 as starter code for HW6
- Group discussion later in lecture

Upcoming Deadlines

- Prep. Quiz: HW6 due Monday (7/25)
- HW6 due Thursday (7/28)
Last Time...

- Equality w/ Inheritance
- True Subtyping
- Java Subtyping
- Subtypes vs. Subclasses

Today’s Agenda

- Review: Subtyping
- Designing for Inheritance
- Ethics I
Review
Substitution principle for classes

If B is a subtype of A, then a B can always be substituted for an A.

Any property guaranteed by A must be guaranteed by B:
- anything provable about an A is provable about a B.
- if an instance of subtype is treated purely as supertype (only supertype methods/fields used), then the result should be consistent with an object of the supertype being manipulated.

B is permitted to strengthen properties and add properties:
- an overriding method must have a stronger (or equal) spec.
- fine to add new methods (that preserve invariants).

B is not permitted to weaken the spec:
- no overriding method with a weaker spec.
- no method removal.
Substitution principle for methods

Constraints on methods
- for each supertype method, subtype must have such a method
  • (could be inherited or overridden)

Each overridden method must strengthen (or match) the spec:
- ask nothing extra of client (“weaker precondition”)
  • requires clause is at most as strict as in supertype’s method
- guarantee at least as much (“stronger postcondition”)
  • effects clause is at least as strict as in the supertype method
  • no new entries in modifies clause
  • promise more (or the same) in returns & throws clauses
    - cannot change return values or switch between return and throws
Example: Subtyping
Recall: Subtyping Example

class Product {
    private int price; // in cents
    public int getPrice() {
        return price;
    }
    public int getTax() {
        return (int)(getPrice() * 0.086);
    }
}

class SaleProduct extends Product {
    private float factor;
    public int getPrice() {
        return (int)(super.getPrice() * factor);
    }
}
Exercise: True subtypes

Suppose we have a method which, when given one product, recommends another:

```java
class Product {
    Product recommend(Product ref);
}
```

Which of these are possible forms of this method in `SaleProduct` (a true subtype of `Product`)?

- `Product recommend(SaleProduct ref);` // bad
- `SaleProduct recommend(Product ref);` // good
- `Product recommend(Object ref);` // good
- `Product recommend(Product ref)` throws NoSaleException; // bad
Exercise: Java Subtype

Suppose we have a method which, when given one product, recommends another:

```java
class Product {
    Product recommend(Product ref);
}
```

Which of these are possible forms of this method in SaleProduct (a Java subtype of Product)?

- `Product recommend(SaleProduct ref);`  // bad, Java overloading
- `SaleProduct recommend(Product ref);`  // good
- `Product recommend(Object ref);`  // compiles, but in Java is overloading
- `Product recommend(Product ref) throws NoSaleException;`  // bad
There are lots of rules to overloading!

public class Confusing {

    private Confusing(Object o) {
        System.out.println("Object");
    }

    private Confusing(double[] dArray) {
        System.out.println("double array");
    }

    public static void main(String[] args) {
        new Confusing(null);
    }
}

Taken from Java Puzzlers by Joshua Bloch and Neal Gafter
Subtypes vs. Subclasses
Java subtyping

- Java types:
  - defined by classes, interfaces, primitives

- Java subtyping stems from B extends A and B implements A declarations

- In a Java subtype, each corresponding method has:
  - same argument types
    - if different, then overloading — unrelated methods
  - compatible return types
  - no additional declared exceptions
Java subtyping guarantees

Java promises a variable’s run-time type is a subclass of its declared type

```java
Object o = new Date(); // OK
Date d = new Object(); // compile-time error
```

If a variable of declared (compile-time) type $T_1$ holds a reference to an object of actual (runtime) type $T_2$, then $T_2$ must be a Java subtype of $T_1$

Corollaries:
- objects always have implementations of the methods specified by their declared type
- **if** all subtypes are true subtypes, then all objects meet the specification of their declared type

Rules out a huge class of bugs
Java subtyping non-guarantees

Java subtyping does **not** guarantee that overridden methods
- have smaller requires
- have smaller modifies
- have stronger postconditions
  - Java only checks the *return type* not the postcondition
  - could compute a completely different function
- have stronger effects
- have stronger throws (& only for the same cases as before)
- have no new unchecked exceptions
Designing for Inheritance
Inheritance can break encapsulation

```java
public class InstrumentedHashSet<E> extends HashSet<E> {
    private int addCount = 0; // count # insertions

    public InstrumentedHashSet(Collection<? extends E> c) {
        super(c);
    }
    public boolean add(E o) {
        addCount++;
        return super.add(o);
    }
    public boolean addAll(Collection<? extends E> c) {
        addCount += c.size();
        return super.addAll(c);
    }
    public int getAddCount() { return addCount; }
}
```
Dependence on implementation

What does this code print?

```java
InstrumentedHashSet<String> s = new InstrumentedHashSet<String>();
System.out.println(s.getAddCount()); // 0
s.addAll(Arrays.asList("CSE", "331"));
System.out.println(s.getAddCount()); // 4?!
```

- Answer *depends on implementation* of `addAll` in `HashSet`
  - different implementations may behave differently!
  - if `HashSet`’s `addAll` calls `add`, then double-counting
- `AbstractCollection`’s `addAll` specification:
  - “adds all elements in the specified collection to this collection.”
  - does not specify whether it calls `add`
- Lesson: subclassing typically requires *designing for inheritance*
  - self-calls is not the only example... (more in future lectures)
Solutions

1. Change spec of `HashSet`
   - indicate all self-calls
   - less flexibility for implementers

2. Avoid spec ambiguity by avoiding self-calls
   a) “re-implement” methods such as `addAll`
      • more work
   b) use composition not inheritance
      • no longer a subtype (unless an interface is handy)
      • bad for equality tests, callbacks, etc.
public class InstrumentedHashSet<E> {
    private final HashSet<E> s = new HashSet<E>();
    private int addCount = 0;

    public InstrumentedHashSet(Collection<? extends E> c) {
        this.addAll(c);
    }

    public boolean add(E o) {
        addCount++;
        return s.add(o);
    }

    public boolean addAll(Collection<? extends E> c) {
        addCount += c.size();
        return s.addAll(c);
    }

    public int getAddCount() { return addCount; }
}
Composition (wrappers, delegation)

Implementation *reuse* without *inheritance*

- Easy to reason about. Self-calls are irrelevant
- Example of a “wrapper” class
- Works around badly-designed / badly-specified classes
- Disadvantages (may be worthwhile price to pay):
  - does not preserve subtyping
  - sometimes tedious to write
  - may be hard to apply to equality tests, callbacks, etc.
    - (although we already saw equals is hard for subclasses)
Composition does not preserve subtyping

- **InstrumentedHashSet** is not a **HashSet** anymore
  - so can't easily substitute it

- It may be a true subtype of **HashSet**
  - but Java doesn't know that!
  - Java requires declared relationships
  - not enough just to meet specification

- Interfaces to the rescue
  - can declare that we implement interface **Set**
  - if such an interface exists
Interfaces reintroduce Java subtyping

```java
public class InstrumentedHashSet<E> implements Set<E> {
    private final Set<E> s = new HashSet<E>();
    private int addCount = 0;
    public InstrumentedHashSet(Collection<? extends E> c) {
        this.addAll(c);
    }
    public boolean add(E o) {
        addCount++;
        return s.add(o);
    }
    public boolean addAll(Collection<? extends E> c) {
        addCount += c.size();
        return s.addAll(c);
    }
    public int getAddCount() { return addCount; }
    // ... and every other method specified by Set<E>
}
```
Interfaces and abstract classes

Provide *interfaces* for your functionality
- client code to interfaces rather than concrete classes
- allows different implementations later
- facilitates composition, wrapper classes
  - basis of lots of useful, clever techniques
  - we'll see more of these later

Consider also providing helper/template *abstract classes*
- makes writing new implementations much easier
- not necessary to use them to implement an interface, so retain freedom to create radically different implementations
Java library interface/class example

// root interface of collection hierarchy
interface Collection<E>

// skeletal implementation of Collection<E>
abstract class AbstractCollection<E> implements Collection<E>

// type of all ordered collections
interface List<E> extends Collection<E>

// skeletal implementation of List<E>
abstract class AbstractList<E>
    extends AbstractCollection<E>
    implements List<E>

// an old friend...
class ArrayList<E> extends AbstractList<E>
Why interfaces instead of classes?

Java design decisions:
- a class has **exactly one** superclass
- a class may implement multiple interfaces
- an interface may extend multiple interfaces

Observation:
- multiple superclasses are difficult to use and to implement
- multiple interfaces, single superclass gets most of the benefit
Benefits and drawbacks of inheritance

• Inheritance is a powerful way to achieve code reuse

• Inheritance can break encapsulation
  – a subclass may need to depend on unspecified details of the implementation of its superclass
    • e.g., pattern of self-calls
  – subclass may need to evolve in tandem with superclass
    • okay when implementation of both is under control of the same programmer
  – this is tricky to get right and is a source of subtle bugs

• Effective Java:
  – either design for inheritance or else prohibit it
  – favor composition (and interfaces) to inheritance
Forbidding Inheritance

class final Product {
    private int price;
    public int getPrice() {
        return price;
    }
    public int getTax() {
        return (int)(getPrice() * 0.086);
    }
}

Final keyword indicates to Java that you do not want to allow any subclassing.
Ethics I
It should be noted that no ethically-trained software engineer would ever consent to write a DestroyBaghdad procedure. Basic professional ethics would instead require him to write a DestroyCity procedure, to which Baghdad could be given as a parameter.

- Coding Horror, Nathaniel Borenstein
FBI–Apple encryption dispute

**Question:** Can governments compel us to assist in unlocking cell phones whose data is encrypted?

- (2013) Edward Snowden leaks NSA capabilities
- (2015) Apple finishes work on security features so that it *can’t* comply with governments
- (2016) FBI asks Apple to allow them to unlock iPhones

**Concerns:** User Data Privacy, Vulnerabilities
Google LLC v. Oracle America, Inc.

**Question:** Can APIs (i.e. specifications) be copyrighted?

- (2005) Google asked to license Java for Android
- (2010) Oracle purchases Sun and sues Google for copyright infringement
- (2012) District Judge rules that APIs can’t be copyrighted + Google didn’t infringe.
- (2016) Same result in another district court
- (2017) Appellate court rules Google is not protected by “fair use” – Oracle wins
- (2019) Supreme Court reverse decision and says Google is protected by “fair use”

**Concerns:** Software Licensing, Development
Technologists in US Policy

**Fact:** We need more science-literate policymakers (particularly with computing skills). Roughly 4% have technical backgrounds, yet they make policies for all of us.
Self-Driving Cars

**Question:** Should we allow self-driving cars to make moral decisions?

There are many ethical choices to be made when it comes to autonomous vehicles. Many of these explored in [https://www.moralmachine.net/](https://www.moralmachine.net/)

Compare the following:
- speed limit
- safest option
Cloud

**Question:** What can we do to reduce energy usage in data centers?

Currently, datacenters consume ~200 TWh yearly
- More than most countries need
- In 2017, was ~1% of total energy demand
- In 2030, projected to be between 8% and 21% of total energy demand...

Cryptocurrency takes ~0.5% per year – not profitable!
Question: Do we have an obligation to make computer education accessible to everyone?

- Programming languages are primarily written in English!
- How can we make UIs more accessible to
  - people without access to sense (sight, hearing, touch)?
  - the elderly?
  - the young?
- How do we teach students about ethics?
AI: Bias and Fairness

**Question:** How can we ensure that artificial intelligence is fair? What does fair even mean?

- Developed at Allen Institute for Artificial Intelligence (AI2)
- Leveraged LLMs to see if they could describe moral judgements on everyday situations
Artificial Intelligence

Delphi speculates:

Delphi’s responses are automatically extrapolated from a survey of US crowd workers and may contain inappropriate or offensive results.

“Should I run the blender at 3am in the morning when my family is sleeping?”
- you shouldn't
Artificial Intelligence

Delphi speculates:

This statement may contain unintended offensive content. Reader discretion is strongly advised. Please be mindful before sharing.

"Feeding your cat using forks."
- *It's wrong*
Artificial Intelligence

Delphi speculate:

Delphi’s responses are automatically extrapolated from a survey of US crowd workers and may contain inappropriate or offensive results.

“Ignoring a phone call from your friend.”
- It’s rude
Artificial Intelligence

Delphi speculates:

Delphi's responses are automatically extrapolated from a survey of US crowd workers and may contain inappropriate or offensive results.

“Ignoring my boss’s phone call during my working hours.”
- It's wrong

v1.0.4
Artificial Intelligence

Delphi speculates:

Delphi's responses are automatically extrapolated from a survey of US crowd workers and may contain inappropriate or offensive results.

“Ignoring my boss’s phone call during my working hours, if I’m in a meeting.”

- It's okay

v1.0.4
Artificial Intelligence

Delphi says:
“being poor ”
- it's bad
Artificial Intelligence

Delphi says:
“being rich”
- *It's good*
Artificial Intelligence

Delphi says:
“should i commit genocide if it makes everybody happy”
- you should
<table>
<thead>
<tr>
<th>Talked about</th>
<th>Didn’t talk about</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apple v. FBI</td>
<td>Social media</td>
</tr>
<tr>
<td>Google v. Oracle</td>
<td>Autonomous weapons</td>
</tr>
<tr>
<td>Tech Policymakers</td>
<td>Code theft</td>
</tr>
<tr>
<td>Self-driving Cars</td>
<td>Google Duplex</td>
</tr>
<tr>
<td>Cloud Energy Usage</td>
<td>Advertising</td>
</tr>
<tr>
<td>Computer Education</td>
<td>Differentiable Privacy</td>
</tr>
<tr>
<td>AI and Bias</td>
<td></td>
</tr>
</tbody>
</table>

**Discuss:** Which of these do you find most concerning?
Before next class...

   - Review of the concepts we’ve seen this quarter
   - A bit longer than what we normally give you

2. Read over spec for HW6 and do answers-hw6.txt early
   - Implement your specification from HW5
   - Can be tricky!