CSE 331
Software Design & Implementation

Kevin Zatloukal
Spring 2022
Subtypes and Subclasses
Midterm

• Friday in class

• Covers lecture material through last Friday
  – required readings are fair game

• No notes or devices (shouldn’t be needed)

• 5 problems
  – Specifications
  – Reasoning x 2 (of the types mentioned before)
  – Testing
  – Multiple choice / short answer
SUBTYPES VS SUBCLASSES
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
Java subtyping

• Java types:
  – defined by classes, interfaces, primitives

• Java subtyping stems from \( B \text{ extends } A \) and \( B \text{ implements } A \) 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

A variable’s run-time type (i.e., the class of its run-time value) is a Java subtype of its declared type

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

If a variable of declared (compile-time) type T1 holds a reference to an object of actual (runtime) type T2, then T2 must be a Java subtype of T1

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 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
EQUALS WITH SUBCLASSES
equals specification

public boolean equals(Object obj) should be:

- **reflexive**: for any reference value \(x\), \(x\).equals(x) == true

- **symmetric**: for any reference values \(x\) and \(y\),
  \(x\).equals(y) == y.equals(x)

- **transitive**: for any reference values \(x\), \(y\), and \(z\), if \(x\).equals(y) and \(y\).equals(z) are true, then \(x\).equals(z) is true

- **consistent**: for any reference values \(x\) and \(y\), multiple invocations of \(x\).equals(y) consistently return true or consistently return false (provided neither is mutated)

- For any **non-null** reference value \(x\), \(x\).equals(null) should return false
Really fixed now

public class Duration {
  @Override
  public boolean equals(Object o) {
    if (!(o instanceof Duration))
      return false;
    Duration d = (Duration) o;
    return this.min == d.min && this.sec == d.sec;
  }
}

• Correct and idiomatic Java
• Gets null case right (null instanceof C always false)
• Cast cannot fail
Two subclasses

class \textit{CountedDuration} extends Duration {
    public static int \textit{numCountedDurations} = 0;
    public CountedDuration(int \textit{min}, int \textit{sec}) {
        super(\textit{min},\textit{sec});
        ++\textit{numCountedDurations};
    }
}

class \textit{NanoDuration} extends Duration {
    private final int \textit{nano};
    public NanoDuration(int \textit{min}, int \textit{sec}, int \textit{nano}){
        super(\textit{min},\textit{sec});
        this.\textit{nano} = \textit{nano};
    }
    public boolean \textit{equals}(Object \textit{o}) { \ldots }
    \ldots
}
**CountedDuration is (probably) fine**

- **CountedDuration** does not override `equals`
  - inherits `Duration.equals(Object)`

- Will (implicitly) treat any **CountedDuration** like a **Duration** when checking `equals`
  - `o instanceof Duration` is true if `o` is **CountedDuration**

- Any combination of **Duration** and **CountedDuration** objects can be compared
  - equal if same contents in `min` and `sec` fields
  - works because `o instanceof Duration` is true when `o` is an instance of **CountedDuration**
NanoDuration is (probably) not fine

- If we don’t override equals in NanoDuration, then objects with different nano fields will be equal

- Using what we have learned:

```java
@Override
public boolean equals(Object o) {
    if (!(o instanceof NanoDuration))
        return false;
    NanoDuration nd = (NanoDuration) o;
    return super.equals(nd) && nano == nd.nano;
}
```

- But we have violated the equals contract
  – Hint: Compare a Duration and a NanoDuration
The symmetry bug

```java
public boolean equals(Object o) {
    if (!(o instanceof NanoDuration))
        return false;
    NanoDuration nd = (NanoDuration) o;
    return super.equals(nd) && nano == nd.nano;
}
```

This is *not symmetric*!

```java
Duration d1 = new NanoDuration(5, 10, 15);
Duration d2 = new Duration(5, 10);
d1.equals(d2); // false
d2.equals(d1); // true
```
Fixing symmetry

This version restores symmetry by using `Duration's equals` if the argument is a `Duration` (and not a `NanoDuration`)

```java
public boolean equals(Object o) {
    if (!(o instanceof Duration))
        return false;
    // if o is a normal Duration, compare without nano
    if (!(o instanceof NanoDuration))
        return super.equals(o);
    NanoDuration nd = (NanoDuration) o;
    return super.equals(nd) && nano == nd.nano;
}
```

Alas, this still violates the `equals` contract

- Transitivity...
The transitivity bug

Duration \texttt{d1} = \texttt{new NanoDuration(1, 2, 3)};
Duration \texttt{d2} = \texttt{new Duration(1, 2)};
Duration \texttt{d3} = \texttt{new NanoDuration(1, 2, 4)};
\texttt{d1.equals(d2)}; \quad // \texttt{true}
\texttt{d2.equals(d3)}; \quad // \texttt{true}
\texttt{d1.equals(d3)}; \quad // \texttt{false}!
No perfect solution

- *Effective Java* says not to (re)override `equals` like this
  - generally good advice
  - but there is one way to satisfy `equals` contract (see below)

- Two less-than-perfect approaches on next two slides:
  1. Don’t make `NanoDuration` a subclass of `Duration`
     - fact that equals should be different is a hint it’s not a subtype
  2. Change `Duration`’s `equals` so only `Duration` objects that are not (proper) subclasses of `Duration` are equal
Option 1: avoid subclassing

Choose composition over subclassing (Effective Java)
- often good advice in general
- many programmers overuse subclassing

```java
public class NanoDuration {
    private final Duration duration;
    private final int nano;
    ...
}
```

Solves some problems:
- clients can choose which type of equality to use

Introduces others:
- can’t use NanoDurations where Durations are expected
  (since it is not a subtype)
Option 2: the `getClass` trick

Check if `o` is a `Duration` and not a subtype:

```java
@Overrides
public boolean equals(Object o) { // in Duration
    if (o == null)
        return false;
    if (!o.getClass().equals(getClass()))
        return false;
    Duration d = (Duration) o;
    return d.min == min && d.sec == sec;
}
```

But this breaks `CountedDuration`!

- subclasses do not “act like” instances of superclass because behavior of `equals` changes with subclasses
- generally considered wrong to “break” subtyping like this
Subclassing summary

• Subtypes *should* be useable wherever the type is used
  – Liskov substitution principle

• Unresolvable tension between
  – what we want for equality: *treat subclasses differently*
  – what we want for subtyping: *treat subclasses the same*

• No perfect solution for all cases...
• Choose whether you want subtyping or not
  – in former case, don’t override equals (make it final)
  – in latter case, can still use composition instead
    • this matches the advice in *Effective Java* and from us (later)
  – almost always best to avoid getClass trick
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.
Solution: composition

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; }
    // ... and every other method specified by HashSet<E>
}
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
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
Pluses and minuses 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