

6.170 Lecture 9 Subtyping



MIT EECS

Michael Ernst
Saman Amarasinghe

1

Outline

- Theory and Practice
- Why we like subclasses?
- True subtypes vs. Java subtypes
- Inheritance vs. Composition
- Substitution Principle
- Interfaces and Abstract Classes

2

Two important concepts

Substitution (Subtype)

S is a subtype of T iff an object of S can masquerade as an object of T in any context

Inheritance (Subclass)

Enables incremental changes to classes and abstract out repeated code

Subclasses can lead to subtype relationships.

3

reminder: why we like subclasses

Suppose we run a web store with a class for Products...

```
class Product {  
    private String title, description;  
    private float price;  
    public float getPrice() { return price; }  
    public float getTax() { return getPrice() * 0.05f; }  
    // ...  
}
```

... and we decide we want another class for Products that are on sale

4

reminder: why we like subclasses

Suppose we run a web store with a class for Products...

```
class SaleProduct {  
    private String title, description;  
    private float price;  
    private float factor;  
    public float getPrice() { return price*factor; }  
    public float getTax() { return getPrice() * 0.05f; }  
    //...  
}
```

... and we decide we want another class for Products that are on sale

We would never dream of cutting and pasting like this:

5

reminder: why we like subclasses

Much better to do this:

```
class SaleProduct extends Product {  
    private float factor;  
    public float getPrice() {  
        return super.getPrice()*factor;  
    }  
    //...  
}
```

6

potential benefits of subclassing

Implementation reuse

Implementations need not repeat unchanged fields and methods – instead reuse from the superclass

This simplifies maintenance too – just need to fix bugs once

Specification reuse

Clients who understand the superclass specification need only study novel parts of subclass

Differences are not buried under mass of similarities

Ability to substitute new implementations

Clients may not need to change their code when new subclasses are developed

7

potential downsides of subclassing

Often misused

Relationships are not simple or clear-cut

- Easy to make an awkward, muddled inheritance hierarchy
- Needs very careful planning

Subtyping is source of most benefits of subclassing

Just because you want to inherit an implementation does not mean you want to inherit a type

Subclass often ends up tightly coupled with superclass

Changes in superclass can break subclass (“fragile base class”)

Can get alarming dependencies on implementation details of superclass

8

What is subtyping?

Sometimes **every A is a B**

library database: every book and CD is a library holding

Subtyping expresses this in the program

programmer declares **A is a subtype of B**

meaning: “every object that satisfies interface A also satisfies interface B”

Goal: code written using B's specification operates correctly even if given an A

9

what are subtypes?

Subtypes are *substitutable* for supertypes

Instances of subtype won't surprise client by failing to meet guarantees made in supertype's specification

Instances of subtype won't surprise client by having expectations not mentioned in supertype's specification

We say that A is a **true subtype** of B if A has a stronger specification than B

This is not exactly the same notion as **Java subtypes**

Java subtypes that are not true subtypes are dangerous

Subclasses in particular have some pitfalls

10

is a square a rectangle?

```
class Rectangle {
    // effects: fits shape to given size
    void setSize(int w, int h);
}

class Square extends Rectangle {
    // requires: w = h
    // effects: fits shape to given size
    void setSize(int w, int h);

    // effects: sets all edges to given size
    void setSize(int edgeLength);

    // effects: fits shape to given size
    // throws BadSizeException if w != h
    void setSize(int w, int h) throws BadSizeException;
}
```

A

B

C

11

is a square a rectangle?

Square is not a (true subtype of) Rectangle:

Rectangles are expected to have a width and height that can be changed independently

Squares violate that expectation, could surprise client



Rectangle is not a (true subtype of) Square:

Squares are expected to have equal widths and heights

Rectangles violate that expectation, could surprise client



Hard to work with this inheritance hierarchy

Could give both a common ancestor, eliminate setSize(), give more careful specification in base class, etc.

Inheritance isn't always intuitive

Benefit: it forces clear thinking

12

inappropriate use of subtyping

```
class Hashtable<K,V> {
    // modifies: this
    // effects: associates the specified value with the specified key
    public void put (K key, V value);

    // returns: value with which the
    // specified key is associated
    public V get (K key);
}

class Properties extends Hashtable<Object,Object> { // simplified
    // modifies: this
    // effects: associates the specified value with the specified key
    public void setProperty(String key, String val) { put(key,val); }

    // returns: the string with which the key is associated
    public String getProperty(String key) { return(String)get(key); }
}
```

Properties class stores string key-value pairs. It borrows Hashtable functionality to do this. What's the problem?

13

the problem

Properties class has a simple rep invariant

keys, values are Strings.

But client can treat Properties as a Hashtable

Can put in arbitrary content, break rep invariant

From Javadoc:

Because Properties inherits from Hashtable, the put and putAll methods can be applied to a Properties object. ... If the store or save method is called on a "compromised" Properties object that contains a non-String key or value, the call will fail.

Also, semantics are more confusing than I've shown

getProperty("prop") works very differently to get("prop")

14

a solution: composition

```
class Properties {
    private Hashtable hashtable;

    // requires: key and value are not null
    // modifies: this
    // effects: associates specified value with specified key
    public void setProperty (String key, String value) {
        hashtable.put(key,value);
    }

    // effects: returns string with which key is associated
    public String getProperty (String key) {
        return (String) hashtable.get(key);
    }
    ...
}
```

15

substitution principle

Constraints on methods

For each method in supertype, subtype must have a corresponding method (may introduce new ones as well)

For each corresponding method, the subtype version must:

Ask nothing extra of client ("weaker precondition")

- Requires clause is at most as strict as in the supertype method

Guarantee at least as much ("stronger postcondition")

- Effects clause is at least as strict as in the supertype method
- No new entries in modified clause

16

substitution principle

For parameter types:

Argument types may be replaced with supertypes ("contravariance").

This doesn't place any extra demand on the client.

- careful, Java has its own opinions about this

For result type:

Result type may be replaced with a subtype ("covariance").

This doesn't violate any expectation of the client.

For exceptions:

There can be no new exceptions. Existing exceptions can be replaced with subtypes. This doesn't violate any expectation of the client.

17

Example

```
class Hashtable { // class is somewhat simplified (generics omitted)
    // modifies: this
    // effects: associates the specified value with the specified key
    public void put (Object key, Object value);

    // returns: value with which the
    // specified key is associated
    public Object get (Object key);
}

class Properties extends Hashtable {
    // modifies: this
    // effects: associates the specified value with the specified key
    public void put(String key, String val) { super.put(key,val); }

    // returns: the string with which the key is associated
    public String get (String key) { return(String)super.get(key); }
}
```

Arguments are subtypes
Violates
Contravariance Argument

Result type is a subtype
Satisfies Covariance Result

Can throw an exception
Violates no new exceptions

18

substitution principle: continued

Constraints on properties

The subtype is permitted to strengthen & add properties
But any properties guaranteed by supertype must be guaranteed by subtype

If instance of subtype is treated purely as supertype – only supertype methods and fields queried – then result should be consistent with an object of the supertype being manipulated

19

Example

```
Hashtable tbl = new Properties();  
  
Integer one = new Integer(1);  
  
tbl.put("One", one);
```

put method in Properties
expect a String

This is no good!

It breaks a put specification of Properties, and could surprise a client.

20

Is this a good Inheritance?



- Depends on the members, methods and the specifications

21

Arranging the World

How would you arrange

Particles
Atoms
Electrons
Protons
Neutrons

22

Java subtypes

Java types:

classes, interfaces, primitives

A is Java subtype of B if:

declared relationship (A extends B, A implements B)
for each method in A, B has corresponding method

Automatic inheritance with extends

For each corresponding method

same argument types
compatible result types (“covariant return”)
no additional declared exceptions

23

generics by example

Subtyping generalizes to generics with same type arguments

LinkedList<Long> is subtype of Collection<Long>

No relation between e.g. List<Number> and List<Integer>

Don't be misled by subtype relationship between arguments

Type with wildcard is a supertype of whatever it matches

List<Number> and List<Integer> are subtypes of List<?>

They are also subtypes of List<? extends Number>

But List<Object> is not

Also have subtyping relationship with raw types

24

true subtypes versus Java subtypes

- Java requires type equality for parameters
 - Different types are treated as different methods
 - More than needed but simplifies syntax and semantics.
- Java does permit covariant returns
 - Be careful – this is recent language feature
- Java has little notion of specification beyond method signatures
 - No check on precondition/postcondition
 - No check on promised properties (invariants)

25

what compiler gives us

- Objects are guaranteed to be (Java) subtypes of their declared type
 - If a variable of *declared (compile-time)* type T holds a reference to an object of actual (*runtime*) type T', then T' is a (Java) subtype of T
- 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

26

another problem

```
public class InstrumentedHashSet<E> extends HashSet<E> {
    private int addCount = 0; // count attempted 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; }
}
```

27

can we predict what this code does?

```
InstrumentedHashSet<String> s =
    new InstrumentedHashSet<String>();

System.out.println(s.getAddCount()); // 0

s.addAll(Arrays.asList("S", "Cr", "P"));
System.out.println(s.getAddCount()); // 6
```

Answer depends on implementation of addAll() in HashSet
This actually calls add(), so we get double counting

28

composition to the rescue (again)

```
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>
}
```

29

solved!

```
HashSet<String> hs = new HashSet<String>();
InstrumentedHashSet<String> s =
    new InstrumentedHashSet<String>(hs);

System.out.println(s.getAddCount()); // 0

s.addAll(Arrays.asList("S", "Cr", "P"));
System.out.println(s.getAddCount()); // 3
```

Problem solved
No longer care about whether addAll() in HashSet calls add()

30

composition

Composition is a very flexible solution

- Very easy to reason about
- No slippery self-calls
- Implementation inheritance is not exposed
- Example of a “wrapper” class

May be hard to apply in callback situations

Can be a bit tedious to write delegation methods

- Eclipse can help you
- Small price to pay for cleaner design

31

but we lost something

New problem

InstrumentedHashSet is not a HashSet anymore

- So can't easily substitute it
- It may in fact be a true subtype of HashSet
- But Java doesn't know that!

- Java requires declared relationships
- Not enough to just meet specification

Interfaces to the rescue

Can declare that we implement interface Set

32

interfaces to the rescue

```
public class InstrumentedHashSet<E> implements Set<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 Set<E>
}
```

33

might as well generalize a bit...

```
public class InstrumentedSet<E> implements Set<E> {
    private final Set<E> s;
    private int addCount = 0;
    public InstrumentedSet(Set<E> s) {
        this.s = s;
        addCount = s.size();
    }
    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>
}
```

34

interfaces and abstract classes

Provide interfaces for your functionality

- Lets client code to interfaces rather than concrete classes
- Allows different implementations later
- Facilitates composition, wrapper classes
 - Basis of lots of useful, clever tricks
 - We'll see more of these later

Consider providing helper/template abstract classes

- Can minimize number of methods that new implementation must provide
- Makes writing new implementations much easier
- Using them is entirely optional, so they don't limit freedom to create radically different implementations

35

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

36

dangers of inheritance

Inheritance is a powerful way to achieve code reuse

But inheritance breaks encapsulation

A subclass may need to depend on unspecified details of the implementation of its superclass

- e.g. pattern of self-calls

Subclass must evolve in tandem with superclass

Safe within a package where implementation of both is under control of same programmer

Otherwise, authors of superclass must design and document specifically for the purpose of being extended

Often better to avoid implementation inheritance and use composition instead

37

Concrete, Abstract or Interface?

Telephone

\$9.99 homephone, Speakerphone, cellphone, Skype, VOIP phone

TV

CRT, Plasma, DLP, LCD

Table

Dining table, Desk, Coffee table

Coffee

Espresso, Frappuccino, Decaf, Ice coffee

Computer

Laptop, Desktop, Server, Palmtop

CPU

Pentium, PowerPC, TigerSHARC

Professor

Amarasighe, Ernst

38