CSE 331

Subtyping

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

Subtyping

- **subtype**: A datatype that is related to another datatype (supertype) by some notion of *substitutability*, such that program constructs written to operate on elements of the supertype can also operate on elements of the subtype.
  - If S is a subtype of T, any term of type S can be safely used in a context where a term of type T is expected.

- Subtyping expresses the following:
  - "B is a subtype of A if every object that satisfies the specification and interface for B also satisfies the specification and interface for A."

- Goal: code using A's specification operates correctly if given a B.

Substitution

- Subtypes must be substitutable for supertypes.
  Instances of a subtype must not surprise a client by:
  - failing to satisfy the supertype's specification
  - having more expectations than the supertype's specification.

- B is a *true subtype* of A if B has a stronger specification than A.
  - This is **not the same as a Java subclass**.

- Java subclasses that are not true subtypes are dangerous.
  - **OO Design Heuristic #55:** Whenever there is inheritance in an OO design, ask yourself two questions:
    - (a) Am I a special type of the thing from which I am inheriting?
    - (b) Is the thing from which I am inheriting part of me?

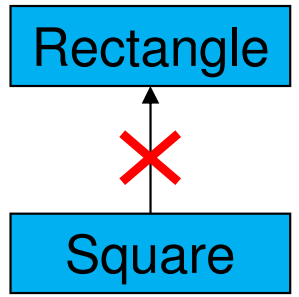
Subtyping example

```
public class Rectangle {
    public int getArea()
    public int getHeight()
    public int getPerimeter()
    public int getWidth()
    public void setHeight(int height)
    public void setSize(int width, int height)
    public void setWidth(int width)
}
```

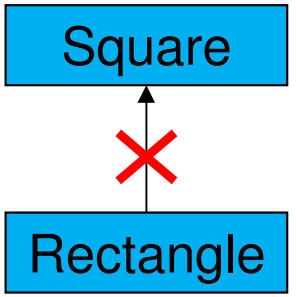
- From basic geometry, we know that every square is a rectangle.
  - If we make a `Square` class, should it extend `Rectangle` ?

Square/Rect relationship

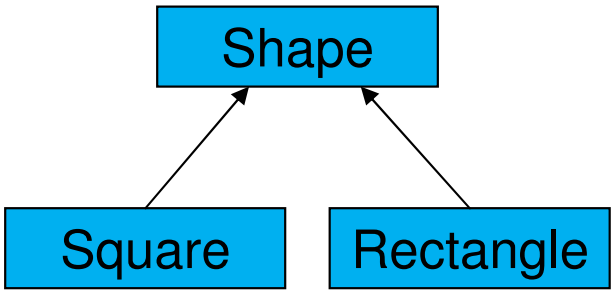
- `Square` is not a (true subtype of) `Rectangle`:
  - Rectangles are expected to have a width and height that can be changed independently
  - Squares violate that expectation; surprises client



- `Rectangle` is not a (true subtype of) `Square`:
  - Squares are expected to have equal widths and heights
  - Rectangles violate that expectation; surprises client



- Solutions:
  - Make them unrelated
  - Make them siblings under a common parent
  - Make them immutable



Bad subtypes in JDK

```
public class Hashtable<K, V> {   // basically a Map
    public V get(K key)
    public void put(K key, V value)
}

// A class for saving/loading string key/value settings.
public class Properties extends Hashtable<Object, Object> {
    public void setProperty(String key, String val) {
        this.put(key, val);
    }

    public String get(String key) {
        return (String) super.get(key);
    }

    public String getProperty(String key) {
        return (String) this.get(key);
    }
}
```

- What is wrong with this design?

Breaking Properties

```
Hashtable tbl = new Properties();
tbl.put("oops", new Integer(1));
tbl.getProperty("oops");          // ClassCastException
```

- The `Properties` object is a `Hashtable` and can be used as one.
- But it does not behave properly when it is used as a `Hashtable` if you perform some `Hashtable` operations on it.

- From `Properties` Javadoc (they seem to know it's bad!):
  - "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."

Solution: Composition

- Instead of having `Properties` extend `Hashtable`, have it *use* a `Hashtable` internally.
  - **Effective Java Tip #16:** Favor composition over inheritance.

```
public class Properties {
    private Hashtable<Object, Object> hashtable;

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

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

Liskov Substitution Principle

- **Liskov Substitution Principle:** If B is a subtype of A, a B must *always* be able to be substituted for an A.
  - Any property guaranteed by A must be guaranteed by B as well.
    - The subtype is permitted to strengthen and add properties.
    - Anything provable about an A is provable about a B.
  - If an instance of the subtype is treated purely as the supertype -- only supertype methods and fields queried -- then the result should be consistent with an object of the supertype being manipulated.

- No specification weakening allowed:
  - No method removal
  - No overriding methods with stronger preconditions or weaker / incompatible postconditions

Substitution continued

- Each overriding method must:
  - Ask nothing extra of the client (weaker precondition).
  - Guarantee at least as much (stronger postcondition).
    - No new objects modified or new changes to "this".
- Method *parameters* (inputs):
  - May be replaced with *supertypes* ("contravariance").
- Method *returns* (outputs/results):
  - May be replaced with a *subtype* ("covariance").
- Method *exceptions*:
  - No new exceptions may be added to any overridden headers.
  - Existing exceptions can be replaced with subtypes.

Subtyping exercise

- Suppose a method connects couples on a dating site:

```
public class DatingSiteUser {
    public Couple date(DatingSiteUser u)
}
```

- Which of these are valid methods in subclass `PremiumUser` ?
  - a) `public Couple date(PremiumUser u)`
  - b) `public PremiumUser date(DatingSiteUser u)`
  - c) `public Couple date(Object u)`
  - d) `public Couple date(DatingSiteUser u) throws UndateableSlobException`

  - Answers: a NO; b YES; c OK but overloaded; d NO

Bad subtypes in Java

```
public class Hashtable<K, V> {    // basic
    public V get(K key)
    public void put(K key, V value)
}

// A class for easily save/loading string key/value settings.
public class Properties extends Hashtable<Object, Object> {
    public void setProperty(String key, String val) {
        this.put(key, val);
    }

    public String get(String key) {
        return (String) super.get(key);
    }

    public String getProperty(String key) {
        return (String) this.get(key);
    }
}
```

Arguments are subtypes
Stronger requirement = weaker specification!

Result type is a subtype
Stronger guarantee = OK

Can throw an exception
New exception = weaker spec!

Revealing implementation

- Consider the following subclass of `HashSet`:

```
public class CountingHashSet<E> extends HashSet<E> {
    private int addCount = 0;  // count (attempted) adds

    public CountingHashSet(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; }
}
```

Depending on implementation

- What does this code print?

```
Set<String> s = new CountingHashSet<String>();
s.addAll(Arrays.asList("CSE", "331"));
System.out.println(s.getAddCount());
```

- Answer depends on implementation of `addAll` in `HashSet`:
  - If `HashSet.addAll` calls `add`? Elements will be counted twice.

- `addAll` specification from Java API Specs:
  - "Adds all of the elements in the given collection to this collection."
  - (Does not specify whether it calls `add`.)

- **fragile base class problem**: When subclasses depend on the unspecified implementation details of their superclass.

Using composition

- This version of `CountingHashSet` keeps a proper count:

```
public class CountingHashSet<E> {
    private final HashSet<E> s;
    private int addCount = 0;

    public CountingHashSet(Collection<? extends E> c) {
        s = new HashSet<E>();
        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 in HashSet<E>
}
```

Regaining subtyping

- The composition version of `CountingHashSet` is suboptimal because it has lost its type relationship to `HashSet`.
  - Can't interchange `HashSet` and `CountingHashSet` in code.

- Solution: Use an *interface* .

```
public class CountingHashSet<E> implements Set<E> {
    private final HashSet<E> s;
    private int addCount = 0;

    public CountingHashSet(Collection<? extends E> c) {
        s = new HashSet<E>();
        addAll(c);
    }

    // ...
}
```

What about this constructor?

```
public CountingHashSet(Set<E> s) {
  this.s = s;
  addCount = s.size();
}
```

Class design question

- What's wrong with the design of this class?

```
public class DatingSiteUser {
    ...
    public double getSubscriptionPrice() {
        if (this instanceof PremiumUser) {
            return 2.00 * months;
        } else if (this instanceof TrialUser) {
            return 50.00;
        } else {
            return 4.00 * months;
}   }   }
```

  - **OO Design Heuristic #37:** Derived classes must have knowledge of their base class by definition, but base classes should not know anything about their derived classes.
  - **OO Design Heuristic #46**. Case analysis on the type of an object is usually an error. The designer should use polymorphism instead.