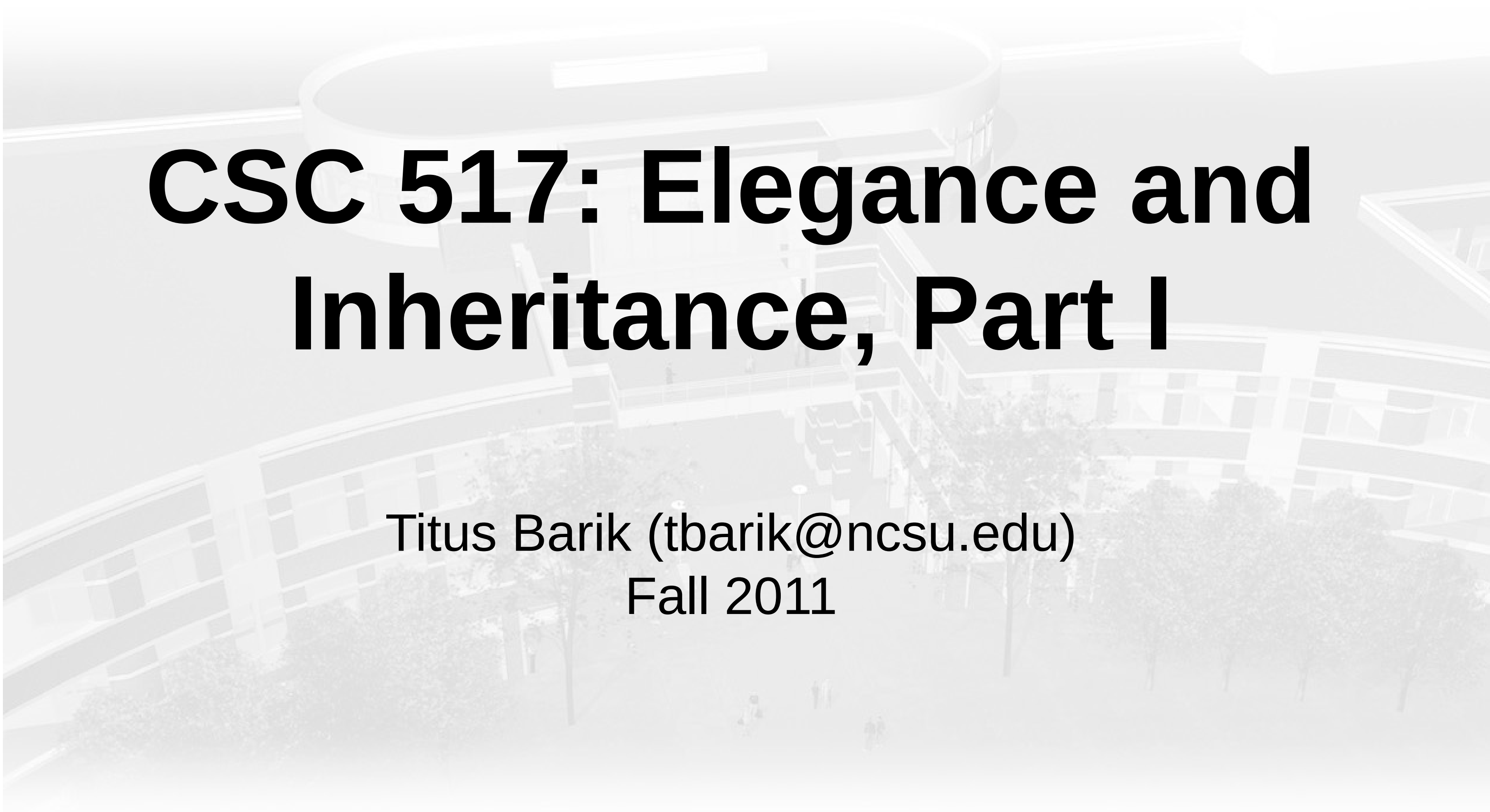


CSC 517: Elegance and Inheritance, Part I

Titus Barik (tbarik@ncsu.edu)

Fall 2011

Ice Cream and Cake and Cake

- What is the object-oriented way of getting rich?
- **Inheritance.**

Interfaces

- Java (and C#) don't have multiple inheritance, but how can we simulate it?
- "Program to an interface, not an implementation." (GoF)
- Which is preferred:
  - *ArrayList* list = new ArrayList();
  - *List* list = new ArrayList();

Generics

- Even better:
  - *ArrayList* list = new ArrayList<*Shape*>();
- How do generics help us?
  - Shape s = new Shape();
  - list.add(s);
  - Shape q = List.get(0)
- Generics allow us to statically communicate **type** information to the compiler; use them everywhere.

Inheritance is tricky.

- We have done examples with inheritance, but when should it be used?
- Is Y an X? What does it mean for Y to be an X? (is-a relationship).
- Temptations:
  - Code reuse.
  - Subclassing for specializing.
  - Public interface.
  - Polymorphism (collections).
- In theory, this is great. *But is it sufficient? Is it even a good reason?*

Inheritance is overused.

- When all you have is a hammer, everything you see is a nail.
- *What about the code reuse example?*
- Inheritance breaks encapsulation. *Why?*
- Favor composition over class inheritance.

Is a Square a Rectangle?

Is a Square a Rectangle?

```
Class Rectangle {
  private int width, height;
  public
     Rectangle(int w, int h);
  public int getWidth();
  public setWidth(int width);
  ...
  public setSize(int w, int h);
}
```

http://www.objectmentor.com/resources/articles/lsp.pdf

Uh-oh Spaghettios

- `Constructor: Square(int s);`
- `setSize(int w, int h)`
  - Let's remove it, but derived classes requiring changes to base classes already indicates a bad design.
- `setHeight(5) => changes width.`
  - Are we now okay?
- `r.getWidth() * r.getHeight()`

Validity is not instrinsic and cannot be viewed in isolation.

A square is a rectangle, but a *square object* is not a *rectangle object*.

Inheritance Formalization

- Liskov substitution principle, easy in theory, difficult in practice to get right.
- *Let q(x) be a property provable about objects x of type T. Then q(y) should be true for objects y of type S where S is a subtype of T.*
- You're probably thinking: *"I hope this isn't on the exam."*
- But you would be wrong.

Direct From the Java API

- Remember code reuse? Inheritance is all or nothing.
- "Because `Properties` **inherits** from `Hashtable`, the `put` and `putAll` methods can be applied to a `Properties` object. **Their use is strongly discouraged** as they allow the caller to insert entries whose keys or values are not `Strings`. The `setProperty` method should be used instead. 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."
- Didn't you get the memo?
- Yeah. I got the memo. And I understand the policy. And the problem is just that I forgot the one time. And I've already taken care of it so it's not even really a problem anymore.

Real-World Violations of LSP

- In practice*:
  - Java violates LSP (*unmodifiableList*).
  - Ruby violates LSP (*dup* method).
  - C# probably violates LSP.

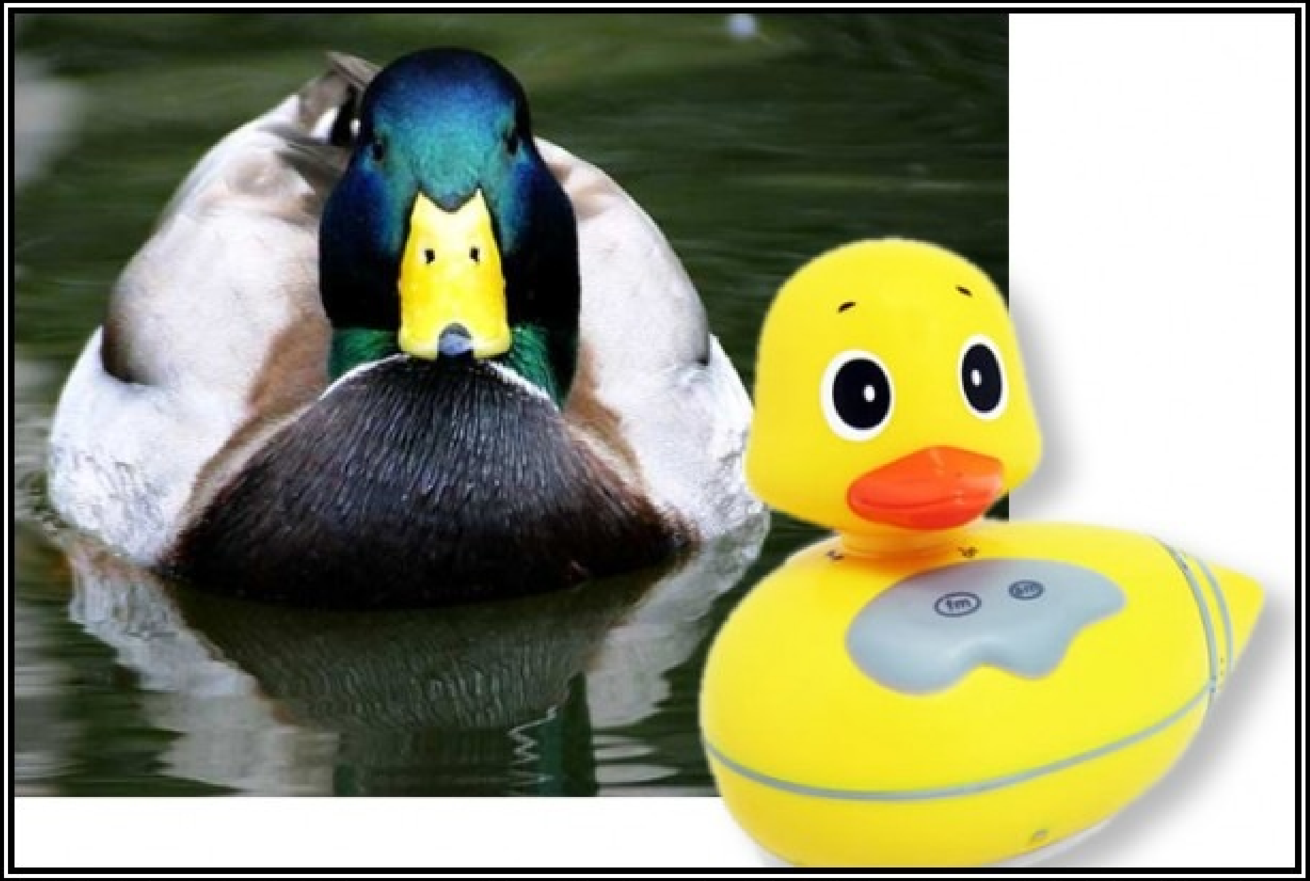
Ruby Example

```
irb> 5.respond_to? :dup
=> true
irb> 5.dup
TypeError: can't dup Fixnum
     from (irb):1:in `dup'
     from (irb):1
```

- **And I would have gotten away with it too, if it weren't for you meddling kids…**

Real-World Violations of LSP?

- Workarounds:
  - Java fine print: “`Returns an unmodifiable view of the specified list. This method allows modules to provide users with "read-only" access to internal lists. Query operations on the returned list "read through" to the specified list, and attempts to modify the returned list, whether direct or via its iterator, result in an` **`UnsupportedOperationException`**`.`“
  - Ruby **dup** method (found in Object). It’s dynamic, get over it. We don’t have a contract.
  - C# workaround is to **seal** everything, but this breaks the **open-closed principle**.



LISKOV SUBSTITUTION PRINCIPLE

If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction