CS 211: Final/Abstract to Stop/Force Inheritance

Chris Kauffman

Week 6

Logistics

Goals Today

- Stopping inheritance: `final`
- Forcing inheritance: `abstract`

Lab Quiz this Week

P3 Due Sunday

- Tests released
- Note on `AutomaticSC` and upper/lower case
- Questions?

Reading: Inheritance

- Building Java Programs Ch 9
- Lab Manual Ch 7

Exam 1 Schedule

Mon 2/27	Equals, Dispatch
Wed 3/1	Abstract, Final
	Lab Quiz
Sun 3/5	Project 3 Due
Mon 3/6	Review
Wed 3/8	Exam 1
Mon 3/13	Spring Break

Exercise: Override vs Overload

Find examples of overriding a method and overloading a method.

```
class P{
  public void print(String s){
    System.out.println(s);
  }
  public void print(int i){
    System.out.println(i);
  }
  public void print(String s, int i){
    this.print(s);
    this.print(i);
  }
}
class C extends P{
  public void print(String s){
    System.out.println("Different: "+s);
  }
  public void print(double x){
    System.out.println(x);
  }
```

Exercise: Dispatch Across Classes

```
class Combiner {
  protected int result;
  public Combiner(){
    this.result = 0;
  }
  public int getResult(){
    return result;
  }
  public void combine(int i){
    this.result += i;
  }
  public void combineAll(int [] a){
    for(int x : a){
      this.combine(x);
    }
  }
}
class Productizer extends Combiner{
  public Productizer(){
    this.result = 1;
  }
  @Override
  public void combine(int i){
    this.result *= i;
  }
}
```

What gets printed...

When `main()` gets run and why do the numbers differ?

```
public class Dispatcher{
  public static void main(String args[])
  {
    int arr [] = {1, 2, 3, 4, 5};

    Combiner sum =
      new Combiner();
    sum.combineAll(arr);
    System.out.printf("Sum: %d\n",
                      sum.getResult());

    Combiner prod =
      new Productizer();
    prod.combineAll(arr);
    System.out.printf("Product: %d\n",
                      prod.getResult());
  }
}
```

Preventing Inheritance

- Occasionally want to prevent inheritance of a class
- Keyword `final` prevents changes

Examples of `final`

```
public final int x; // assign variable/field x

public final class C {..} // cannot extend C

// Can extend P but ....
public class P {
  // Cannot override doIt
  public final int doIt(){...}
  public int fakeIt(){...}
}
```

Class `P` can have children, children can override `fakeIt()` but cannot override `doIt()`. Examine `PreventInheritance.java`

Why Make a Class/Method final?

- Somewhat beyond the scope of this course
- Canonical example: `String` is `final` to keep it immutable
- Prevents any crazy, change-able child strings from being used in place of immutable version
- Enables compiler/runtime optimizations and potentially some security
- `final` methods may enable somewhat better performance to avoid dynamic dispatch

Forcing Inheritance

- Sometimes want to set up a hierarchy but don't have a good default behavior
- Force implementation of certain methods
- Example: `Combiner` from early was suspicious: *added* in parent class which was arbitrary
- Every combiner must be able to `combine(..)`
- But no default way to proclaim: make it abstract

```
abstract public class Combiner{ // Abstract
  abstract public void combine(int i);
  public void combineAll(int [] a){ ... }
  ...;
}
public class Summer extends Combiner { // Concrete
  public void combine(int s){ result+=i; }
}
public class Productizer extends Combiner { // Concrete
  public void combine(int s){ result*=i; }
}
```

Why `abstract class`?

Interchangeable parts

Interchangeable components can be set up via 3 mechanisms

- Inheritance (normal and abstract classes)
- Interfaces (soon)
- Generics (later in the course)

All rely on interchangeable parts having similar/same methods

When to use `abstract class`

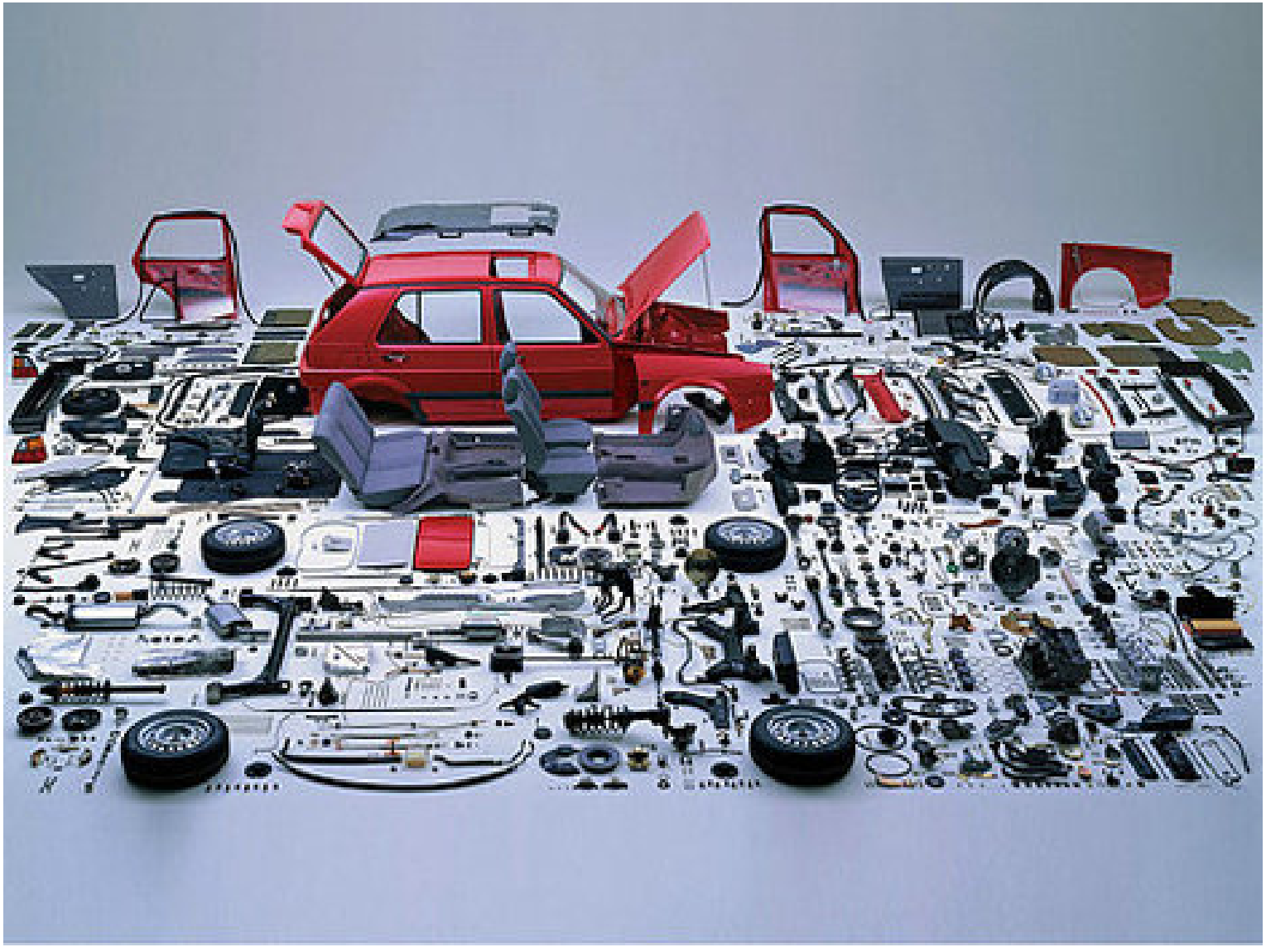
Following factors indicate `abstract class` is the correct mechanism

- Obvious hierarchy of objects
- No need to mix in methods: class `Z` does NOT need methods from both class `X` and `Y`
- Want to share implementation and fields between some classes
- No complete default implementation

Dremel: A tool with Interchangeable Parts

Source

Car: Many Interchangeable Parts



Source

Heels... Okay this is just ridiculous

Source

P3: Redesign?

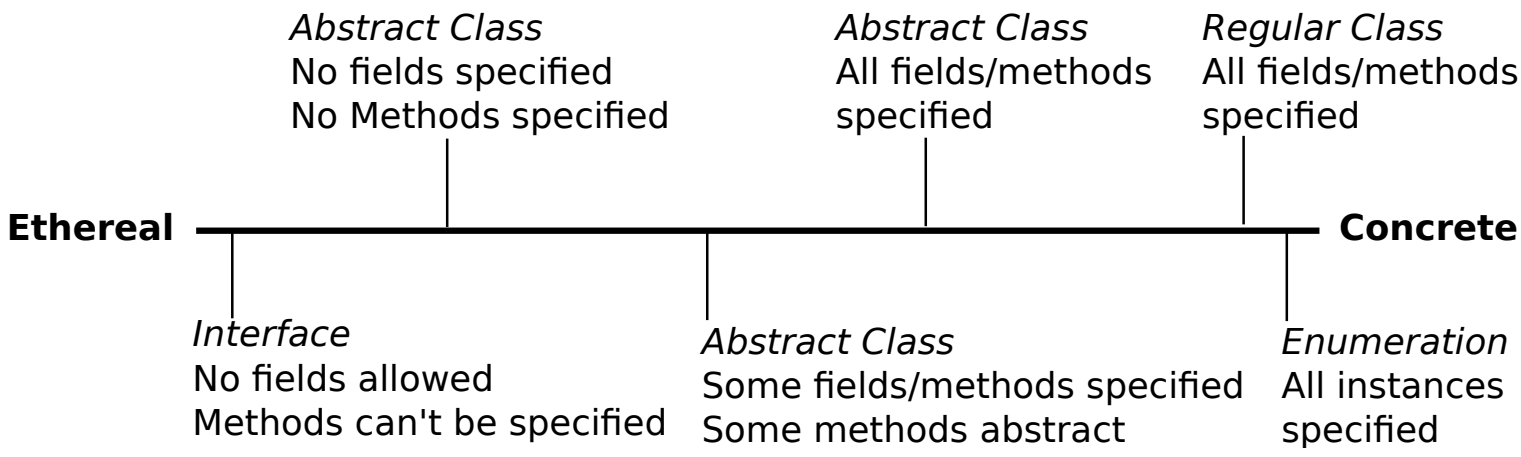
The `correctWord(word)` is a good candidate to be abstract

- One could argue that it has no good default behavior
- Should be overridden by children that have a concrete idea

Redesign

```
public abstract class SpellChecker {
  ...;
  public boolean isCorrect(String word){...}
  public abstract String correctWord(String word);
  public String correctDocumnet(String word){...}
  ...
}
public class HighlightingSC {
  @Override
  public String correctWord(String word){
    return String.format("**%s**",word);
  }
}
```

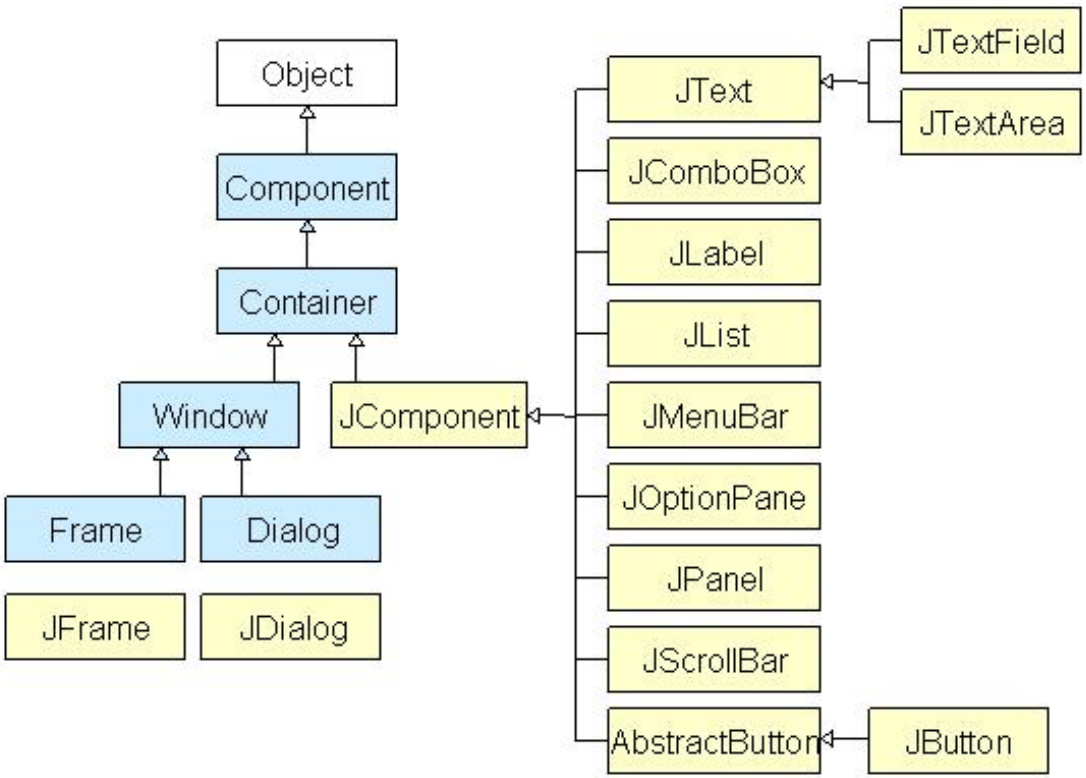
Preview: All of Java's Top-Level Entities



- Regular classes are more concrete
- Abstract classes are more ethereal

Exercise Swing: Java GUI Classes

- These set up a deep hierarchy, many abstract classes
- Examine the docs for `JButton` and find abstract classes from which it descends



Will trace down to `JButton`

JButton Java Doc

Top of the Hierarchy

```
public abstract class Component extends Object
implements ImageObserver, MenuContainer, Serializable
```

A component is an object having a graphical representation that can be displayed on the screen and that can interact with the user.

```
public class Container
extends Component
```

A generic Abstract Window Toolkit(AWT) container object is a component that can contain other AWT components.

```
public abstract class JComponent extends Container
implements Serializable
```

The base class for all Swing components except top-level containers. To use a component that inherits from JComponent, you must place the component in a containment hierarchy whose root is a top-level Swing container.

Buttony Things

```
public abstract class AbstractButton extends JComponent
implements ItemSelectable, SwingConstants
```

Defines common behaviors for buttons and menu items.

```
public class JButton extends AbstractButton
implements Accessible
```

An implementation of a "push" button.