CSE 331
Software Design & Implementation

Hal Perkins
Winter 2013
Java Classes, Interfaces, and Types
The fundamental unit of programming in Java is a class
everything is defined in some class
But Java also provides interfaces…
Classes can extend other classes and implement interfaces…
Interfaces can extend other interfaces…
Some classes are abstract…
And somehow this is all related to types!

How does this work? How are these things connected? What is their intended use?
   More in the fullness of time, but let’s get started…
Classes, Objects, and Java

Ignoring static cruft for now...
Everything is an instance of a class (an object)
Every class defines data and methods
Every class extends exactly one other class
   Object if no superclass is explicitly named
A class inherits superclass fields and methods
Every class also defines a type – i.e., class Foo defines type Foo, and also has all inherited types, e.g., Object
   Not explored in depth today, but later...
So a class is both specification and implementation
But…

How do we express relationships between classes? Inheritance captures what we want if one class “is-a” specialization of another

```java
class Cat extends Mammal { … }
```

But that’s not really right if classes share a behavior or concept but don’t have an “is-a” relationship:

E.g., Strings, Sets, and Dates are “Comparable” (we can ask if \( x \) is “less than” \( y \)) but there are no “is-a” relationships involved

And what if we want a class with multiple properties? Can’t extend multiple classes, even if that would do what we want…
Java Interfaces

Pure type declaration. Example (without generics):

```java
public interface Comparable {
    int compareTo(Object other);
}
```

Defines a type (Comparable here). Can contain:
- Method specifications (no implementations)
- Named constants

Interface elements are implicitly public
- Constants are also implicitly final, static
- Methods are also implicitly abstract (means: specified only, no implementation provided…)

Cannot create instances of interfaces – they’re abstract and do not contain implementations of methods
- e.g., can’t do Comparable c = new Comparable();
Implementing Interfaces

A class can implement one or more interfaces:

```java
class Gadget implements Comparable{
    ...}
```

Semantics:
- The implementing class and its instances have the interface type(s) as well as the class type.
- The class must provide or inherit an implementation of all methods defined in the interface(s).
  - Approximately correct – need to fix for abstract classes (later)
Using Interface Types

An interface defines a type, so we can declare variables and parameters of that type

Key point: A variable with an interface type can refer to an object of *any* class implementing that type

Examples:

```java
List<String> x = new ArrayList<String>();
List<String> y = new LinkedList<String>();
```

Variables `x` and `y` both have type `List<String>`
This is not new. You’ve seen this with Java collections:

```java
class ArrayList implements List {...}
class LinkedList implements List {...}
```

(Generic types omitted for simplicity)

Client code:

```java
void mangle(List victim) { ... }
```

Method argument can be anything that has type `List` (like an `ArrayList` or `LinkedList`)}
Guidelines for Interfaces

Provide interfaces for significant types / abstractions

Write code using interface types like `Map` wherever possible; only use specific classes like `HashMap` or `TreeMap` when you need to (creating new objects is the most obvious example, or if particular behavior is required like sorted keys)

  Allows code to work with different implementations later

Consider providing classes with complete or partial interface implementation for direct use or subclassing

Both interfaces and classes are appropriate in various circumstances