CS61B, Spring 2003 **Discussion #6** **Amir Kamil**
UC Berkeley **03/03/03**

Topics: Inheritance, Static vs. Dynamic, Modifiers, Nested and Inner Classes

1 Inheritance

- Every class has a parent class, except the built-in `Object` class. For classes that you define without specifying a parent, its parent is assumed to be the `Object` class.
- Java only allows a single parent for each class.
- The class heirarchy forms a tree, with `Object` at its root.

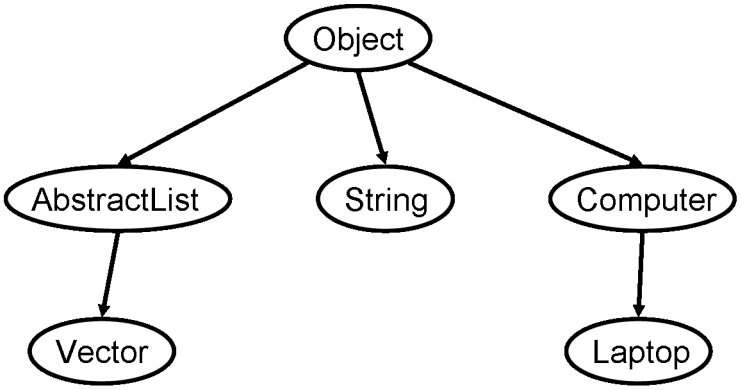

Figure 1: Part of a Java class heirarchy.

- All methods and fields in the parent class that the child has access to are inherited by the child (we will do access modifiers later). So calling a method defined in a parent class from a child class is perfectly fine.
- Children can `override` parents' methods by redefining them. Then calling the overridden method in the child will use the child's version of the method. Overridden methods must retain the same return type as in the original.
- Inheritance in Java:
  `public class Laptop extends Computer {..}`
  The `extends` keyword is used to declare a parent. In this case, we are defining a `Laptop` class with `Computer` as its parent.
- Example:

```
public class Laptop extends Computer {

  int battery; // charge remaining on battery, in minutes

  /* Override turnOn() method to check battery charge */
  public void turnOn() {
    if (battery <= 5) {
      System.out.println("Battery Low");
    } else {
      state = 1;
      boot();
```

```
    }
  }

  /* Add charge method */
  public void charge(int time) {
    battery += time;
  }

}
```

- An `interface` is a collection of methods prototypes that must be provided by implementing classes. No code is provided by the interface, only signatures. There are two reasons to use interfaces: any class implementing an interface is guaranteed to have the methods declared in that interface, and instances of implementing classes can be assigned to variables of type of that interface. Interfaces also allow multiple inheritance in a sense. A class can only extend one other class, but it may implement arbitrary number of interfaces.

```
public interface Laptop {

  public void charge(int min);

  public void discharge(int min);

  public void getStolen();

}

public class PCLaptop extends PC implements Laptop {

  private int remainingTime;

  public void charge(int min) {
    remainingTime += min;
  }

  public void discharge(int min) {
    remainingTime += min;
  }

  public void getStolen() {
    throw new Exception("Finders keepers, loser weepers.");
  }

  ... // other methods, fields

}
```

- An `abstract class` is halfway between an interface and a normal class; it implements some methods and only declares others. Unimplemented methods must be declared abstract. Abstract classes cannot be instantiated. Inheritance with abstract classes works the same way as with regular classes, except inheriting classes must implement the abstract methods.

```
public abstract class Computer {
```

```
  protected int state;

  protected String os;

  protected Vector programs;

  public void turnOn() {
    state = 1;
    boot();
  }

  protected abstract void boot();

  ... // other methods, fields

}

public class PC extends Computer {

  protected void boot() {
    if (os.equals("Windows 95")) {
      throw new Exception("CRASH!");
    } else {
      programs.addElement(os);
    }
  }

}
```

2 Static vs. Dynamic

- Static = compile-time
  Dynamic = run-time
- Static vs. Dynamic Types:
  Define some terminology: Let class `A` be a subtype of `B` if either `A` is a descendent of, or is `B`, or if `B` is an interface, if `A` implements or extends a class that implements `B`. Let `A <= B` mean `A` is a subtype of `B`.

  Then:

  a. A variable of type `A` can be assigned a reference to an object of type `B` if `B <= A`.
  b. An object of type `A` must be cast in order to assign it to a variable of type `B`, if `A` is not `<= B`. Note that if the actual run-time object is not of type `<= B`, a `ClassCastException` will result.

  Note: The compiler can only use the static type of a variable in determining the above relationships.

```
PC pc = new PC();
Laptop lt = new PCLaptop();
Computer comp;
comp = pc; // OK since PC <= Computer
pc = comp; // not OK since Computer is not <= PC
pc = (PC) comp; // OK since run-time object is <= PC
pc = lt; // not OK since Laptop is not <= PC
pc = (PC) lt; // OK since the run-time object is <= PC
```

	definer	child	package	world
`private`	**X**			
`<default>`	**X**		**X**	
`protected`	**X**	**X**	**X**	
`public`	**X**	**X**	**X**	**X**

Table 1: Privileges granted by Java access modifiers.

- Static vs. Dynamic Method Calls:
  Static methods may be overridden. The method used will be based on the static type of the variable.

  ```
  PC pc = new PCLaptop();
  pc.foo(); // uses foo() defined in PC class if foo() is static
  ```

  It's preferable to call static methods using the type name rather than an instance of the class.

  ```
  PC.foo();
  ```

  Calls to non-static methods use the method defined in the dynamic type of the variable.

  ```
  PC pc = new PCLaptop();
  pc.bar(); // uses bar() defined in PCLaptop class if bar() isn't static
  ```

- `super` can be used to call methods or access fields of the parent class.

  ```
  public class PCLaptop extends PC implements Laptop {
    .. // fields and methods
    public void turnOn() {
      if (remainingTime < 5) {
        return;
      } else {
        super.turnOn(); // calls turnOn() method in PC class
      }
    }
  }
  ```

3 Modifiers

- `static` makes the field or method a class field or method.
- A field declared `final` cannot be changed after the object is instantiated. A method declared `final` cannot be overridden by a child class.
- Java also has four access modifiers: `public`, `protected`, `private`, and package protected, the default when no other access modifier is specified. These modifiers restrict access to the fields or methods the modify according to the relationship between the accessing class and the defining class. Table 1 summarizes the privileges these modifiers grant.

4 Nested and Inner Classes

In Java, it is possible to define classes within other classes. Two such examples are nested and inner classes.

4.1 Nested Classes

Nested classes are defined almost like normal classes, but with two exceptions. They are defined within other classes, and they are modified with the `static` keyword. In the following example, `Bar` is a nested class:

```
class Foo {
  static class Bar {
    // code ...
  }
  // more code ...
}
```

Nested classes are externally accessed as *<outer class>.<nested class>*. For example, `Bar` would be accessed as `Foo.Bar` from outside `Bar`. Otherwise, nested classes are pretty much treated like normal classes externally.

So if nested classes are almost the same as normal classes, what is the benefit of a nested class? The first is organization. Sometimes a class is used only as an auxiliary to another class, in which case it is better to define it as a nested class. The second is protection. A nested class and its outer class can access each other's private variables.

4.2 Inner Classes

Inner classes are defined like nested classes, but without the `static` keyword. For example, `Baz` is an inner class:

```
class Foo {
  class Baz {
    // code ...
  }
  // more code ...
}
```

Inner classes are similar in some ways to nested classes, but quite different in other ways. Just like nested classes, they are accessed as *<outer class>.<inner class>*. And just like nested classes, an inner class and its outer class can access each other's private variables. However, unlike nested classes, an instance of an inner class must have an associated instance of the outer class. For example, an instance of `Baz` must be associated with an instance of `Foo`. As a result, an instance of an inner class can access instance variables of its associated outer instance as if they were its own fields.

This leads to some complications when using an inner class. An inner class cannot me directly created from a `static` context (`static` methods, `static` initializers, and initialization of `static` variables), nor can it be directly created by an external class. However, it can be created in either case through an instance of the outer class. The syntax for creation is *<outer instance>.new <inner class>(<params>)*. For example, an instance of `Baz` can be created by `new Foo().new Baz()`.

There are also a few other obscure rules concerning inner classes, such as the fact that they are not allowed to define `static` members, except compile-time constants (e.g. `static final int MAX_NUM = 10;`). You really don't have to worry too much about such details, the compiler will tell you if you do something illegal.

4.3 Usage

Nested and inner classes have somewhat different usage. Inner classes are used when it makes sense and is beneficial for instances of the class to be associated with instances of the outer. Typical examples include `Enumeration`s on containers. An `Enumeration` instance is associated with only one container instance and needs access to that instance's fields, so it makes sense to use an inner class.

Nested classes are generally used where inner classes either aren't allowed or don't make sense to use. For example, if the Java `Collections` class is composed entirely of `static` methods, so any internal classes it needs must be nested. If you look at the source code for `Collections`, you'll see many nested classes

defined. It is also preferable to use nested classes when an association between instances of the internal class and instances of the outer class are not necessary, since that allows the class to be used in more contexts.

There are also other types of internal classes, such as internal interfaces (always implicitly nested), anonymous classes (always inner), and local classes. There are very few cases in which such classes are used, so we will ignore them.