CS61B, Fall 2002 **Discussion #3** **Amir Kamil**
UC Berkeley **9/12/02**

Topics: Inheritance, Static vs. Dynamic, Modifiers, Exceptions

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.

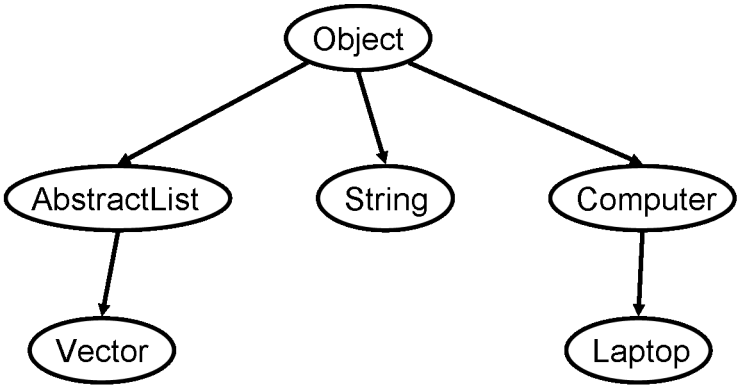

Figure 1: Part of a Java class heirarchy.

- 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
```

- 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.

	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.

4 Exceptions

- Exceptions are used to signify a run-time error in the program.
- They don't necessarily mean your code is bad. For example, an `IOException` will result if a file you are reading from is corrupt, or an `IllegalArgumentException` if an outside class calls one of your methods using invalid parameters.
- Exceptions are objects, so they are instantiated like objects and you can even define your own exceptions. The only requirement is that they inherit `Throwable`.

```
public class CrashException extends Exception {
  // We don't really need to write any code.
}
```

- Exceptions can be thrown using `throw E;` where `E` is an exception object (e.g. `throw new IOException();`).
- Exceptions are either handled using `try {..} catch (ExpType e) {..}` or passed on to the calling method by declaring it to be thrown.

```
public String readLine() throws IOException {..} // passes exception on

public String readLine() {
  try {
    char[] cbuf = new char[1000];
    str.read(cbuf, 0, 1000);
    return new String(cbuf);
  } catch (IOException ie) {
    return null;
  } catch (Exception e) {
    System.out.println("Error: unknown exception");
    return null;
  }
  return null;
}
```

- When an exception is thrown within the try block, the JVM tries to match the exception with the exception types in the catch blocks. The first catch block for which the type is `<=` the type of the exception is executed. For example, if the two catch blocks in the previous code were switched, an `IOException` would result in the Exception catch block to execute, since `IOException <= Exception`.
- Exceptions are `checked` or `unchecked`. Checked exceptions must be caught or declared, unchecked don't have to be. You don't need to memorize which exceptions are checked or unchecked, just let the compiler tell you.
- Exceptions that inherit `Error` or `RunTimeException` are unchecked.