CS61B Lecture #8: Object-Oriented Mechanisms

• Project #1 was released Wednesday night.

Readings covered today:
- Chapters 7 and 8 in Head First Java.

Today:
- New in this lecture: the bare mechanics of “object-oriented programming.”
- The general topic is: Writing software that operates on many kinds of data.
Overloading

Problem: How to get System.out.print(x) to print x, regardless of type of x?

• In Scheme, one function can take an argument of any type, and then test the type.
• In Java, methods specify a single type of argument.
• Partial solution: overloading—multiple method definitions with the same name and different numbers or types of arguments.
• E.g., System.out has type java.io.PrintStream, which defines
  
  void println() *Prints new line.*
  void println(String s) *Prints S.*
  void println(boolean b) *Prints "true" or "false"*
  void println(char c) *Prints single character*
  void println(int i) *Prints I in decimal*
  etc.

• Each of these is a different function. Compiler decides which to call on the basis of arguments' types.
**Generic Data Structures**

**Problem:** How to get a “list of anything” or “array of anything”?

- Again, no problem in Scheme.
- But in Java, lists (such as IntList) and arrays have a single type of element.
- First, the short answer: any reference value can be converted to type `java.lang.Object` and back, so can use `Object` as the “generic (reference) type”:

```
Object[] things = new Object[2];
things[0] = new IntList (3, null);
things[1] = "Stuff";
// Now ((IntList) things[0]).head == 3;
// and ((String) things[1]).startsWith("St") is true
// things[0].head Illegal
// things[1].startsWith ("St") Illegal
```
Dynamic vs. Static Types

- Every value has a type—its dynamic type.
- Every container (variable, component, parameter), literal, function call, and operator expression (e.g. x+y) has a type—its static type.
- Therefore, every expression has a static type.

    Object[] things = new Object[2];
    things[0] = new IntList (3, null);
    things[1] = "Stuff";
Type Hierarchies

- A container with (static) type T may contain a certain value only if that value “is a” T—that is, if the (dynamic) type of the value is a subtype of T. Likewise, a function with return type T may return only values that are subtypes of T.

- All types are subtypes of themselves (& that’s all for primitive types)

- Reference types form a type hierarchy; some are subtypes of others. null’s type is a subtype of all reference types.

- All reference types are subtypes of Object.
The Basic Static Type Rule

• Java is designed so that any expression of (static) type T always yields a value that “is a” T.

• Static types are “known to the compiler,” because you declare them, as in

```java
String x;   // Static type of field
int f (Object s) { // Static type of call to f, and of parameter
    int y;       // Static type of local variable
}
```

or they are pre-declared by the language (like 3).

• Compiler insists that in an assignment, \( L = E \), or function call, \( f(E) \), where

```java
void f (SomeType L) { ... },
```

E’s static type must be subtype of L’s static type.

• Similar rules apply to \( E[i] \) (static type of E must be an array) and other built-in operations.

• Slight fudge: compiler will coerce “smaller” integer types to larger ones, float to double, and (from last lecture) between primitive types and their wrapper types.
Consequences of Compiler's “Sanity Checks”

• This is a conservative rule. The last line of the following, which you might think is perfectly sensible, is illegal:

```java
int[] A = new int[2];
Object x = A; // All references are Objects
A[i] = 0;    // Static type of A is array...
x[i+1] = 1;  // But not of x: ERROR
```

Compiler figures that not every Object is an array.

• Q: Don’t we know that x contains array value!?

• A: Yes, but still must tell the compiler, like this:

```java
((int[])) x)[i+1] = 1;
```

• Defn: Static type of cast (T) E is T.

• Q: What if x isn’t an array value, or is null?

• A: For that we have runtime errors—exceptions.
**Overriding and Extension**

- Notation so far is clumsy.

- **Q:** If I know `Object` variable `x` contains a `String`, why can’t I write, `x.startsWith("this")`?

- **A:** `startsWith` is only defined on `Strings`, not on all `Objects`, so the compiler isn’t sure it makes sense, unless you cast.

- But, if an operation *were* defined on all `Objects`, then you *wouldn’t* need clumsy casting.

- **Example:** `.toString()` is defined on all `Objects`. You can always say `x.toString()` if `x` has a reference type.

- The default `.toString()` function is not very useful; on an `IntList`, would produce string like "IntList@2f6684"

- But for any subtype of `Object`, you may *override* the default definition.
Overriding toString

• For example, if \( s \) is a String, \( s.toString() \) is the identity function (fortunately).

• For any type you define, you may supply your own definition, as we did in class IntList:

```java
public String toString () {
    StringBuffer b = new StringBuffer ();
    b.append ("[");
    for (IntList L = this; L != null; L = L.tail)
        b.append (" " + L.head);
    b.append ("]");
    return b.toString ();
}
```

• If \( x = \text{new IntList}(3, \text{new IntList}(4, \text{null})) \), then \( x.toString() \) is "[3 4]."

• Conveniently, the "+" operator on Strings calls .toString when asked to append an Object, and so does the "%s" formatter for printf.

• With this trick, you can supply an output function for any type you define.
Extending a Class

• To say that class $B$ is a direct subtype of class $A$ (or $A$ is a direct superclass of $B$), write

   ```java
class B extends A { ... }
```

• By default, class ... extends java.lang.Object.

• The subtype inherits all fields and methods of its superclass (and passes them along to any of its subtypes).

• In class $B$, you may override an instance method (not a static method), by providing a new definition with same signature (name, return type, argument types).

• I'll say that a method and all its overridings form a dynamic method set.

• The Point: If $f(...)$ is an instance method, then the call $x.f(...)$ calls whatever overriding of $f$ applies to the dynamic type of $x$, regardless of the static type of $x.$
Illustration

class Worker {
    void work () {
        collectPay ();
    }
}

class Prof extends Worker {
    // Inherits work ()
}

class TA extends Worker {
    void work () {
        while (true) {
            doLab(); discuss(); officeHour();
        }
    }
}

Prof paul = new Prof ();  |
| paul.work() ==> collectPay();
TA mike = new TA ();      |
| mike.work() ==> doLab(); discuss(); ...
Worker wPaul = paul,      |
| wPaul.work() ==> collectPay();
wMike = mike;              |
| wMike.work() ==> doLab(); discuss(); ...

Lesson: For instance methods (only), select method based on dynamic type. Simple to state, but we'll see it has profound consequences.
What About Fields and Static Methods?

```java
class Parent {
    int x = 0;
    static int y = 1;
    static void f() {
        System.out.printf("Ahem!%n");
    }
    static int f(int x) {
        return x+1;
    }
}

class Child extends Parent {
    String x = "no";
    static String y = "way";
    static void f() {
        System.out.printf("I wanna!%n");
    }
}
```

```java
Child tom = new Child();  // tom.x ==> no  pTom.x ==> 0
Parent pTom = tom;        // tom.y ==> way  pTom.y ==> 1
                          // tom.f() ==> I wanna!  pTom.f() ==> Ahem!
                          // tom.f(1) ==> 2  pTom.f(1) ==> 2
```

**Lesson:** Fields *hide* inherited fields of same name; static methods *hide* methods of the same signature.

**Real Lesson:** Hiding causes confusion; so understand it, but don't do it!
What’s the Point?

- The mechanism described here allows us to define a kind of *generic*
  method.

- A superclass can define a set of operations (methods) that are com-
  mon to many different classes.

- Subclasses can then provide different implementations of these
  common methods, each specialized in some way.

- All subclasses will have at least the methods listed by the super-
  class.

- So when we write methods that operate on the superclass, they will
  automatically work for all subclasses with no extra work.