CS61B Lecture 8

Friday, February 9th, 2020

Object-Oriented Mechanisms

The general idea of object-oriented programming is to write software that operates on many kinds of data.

Overloading

So far, we have designed functions that only take in and return a specific kind of data. In Python and Scheme, a single function can take an argument of any type, then test the type if needed. A partial solution for this is a method called **overloading** -- multiple method definitions with the same name and different numbers of or types of arguments.

For example, the function `System.out.println` has type `java.io.PrintStream`, which defines:

```
void println(); // Prints a new line
void println(String s); // Prints S
void println(boolean b); // Prints "true" or "false"
void println(char c); // Prints a single character
void println(int i); // Prints I in decimal
```

Each of these is a different function, and the compiler decides which to call on the basis of the types of argument.

Magic Arrays

How can we build arrays that can contain data of any type? In Python and Scheme, this was no problem, but Java requires you to declare a type, e.g. `int[]`.

The short answer is that any **reference** value can be converted to the type `java.lang.Object` and back, so we can use the generic reference type `Object` as the type it takes in.

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

To access these `Object`s' natural properties, we must convert them back to what they were:

```
/** This is OK */
((IntList) things[0].head == 3);
((String) things[1].startsWith("St"));

/** This is not */
things[0].head == 3;
things[1].startsWith("St");
```

> **Primitive and Reference Types**

> See my notes for CSJ61B (Josh Hug's version of CS61B) to know more about Java's primitive and reference types.

When we try to build a list containing primitive types, this presents a problem because they are not readily convertible to `Object`. Java introduces a set of **wrapper types**, one for each primitive type.

Primitive Type	Wrapper
`byte`	`Byte`
`long`	`Long`
`float`	`Float`
`short`	`Short`
`char`	`Character`
`double`	`Double`
`int`	`Integer`
`boolean`	`Boolean`

One creates new wrapper objects for any value in a process called boxing:

```
Integer Three = new Integer(3);
Object ThreeObj = Three;
```

and vice-versa, called unboxing:

```
int three = Three.intValue();
```

Autoboxing

Now this is very clumsy, so Java introduced a slightly more elegant solution to automate the boxing process:

```
Integer Three = 3;
int three = Three;
int six = Three + 3;

Integer[] someInts = {1,2,3};
for (int x : someInts) {
    System.out.println(x);
}

System.out.println(someInts[0]);
    // This prints Integer 1, not int 1; the println
    // for object is called instead of for int.
```

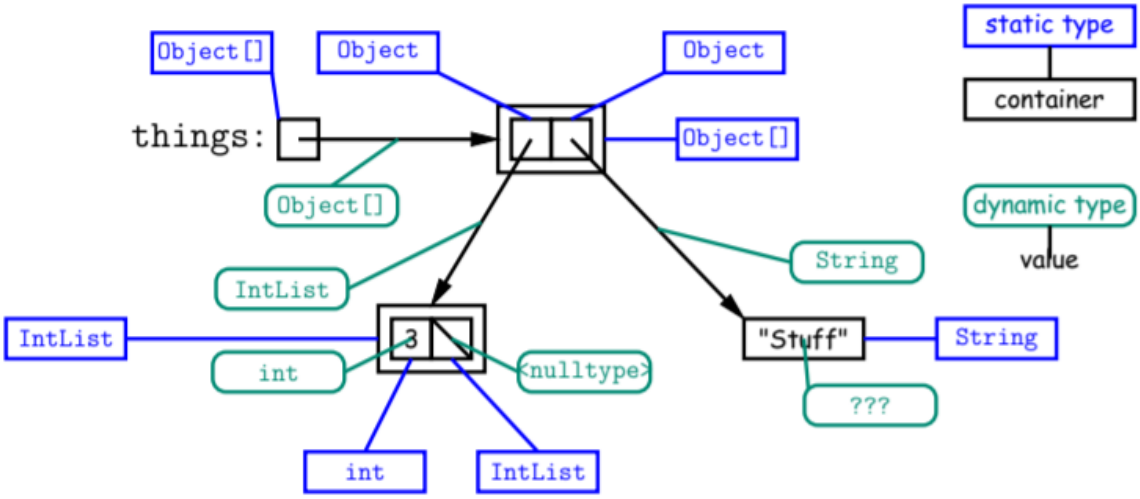
Dynamic vs. Static Types

Every value has a type -- its **dynamic type**.

Every container, a variable, component or parameter, literal, function call and oeprator 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";
```



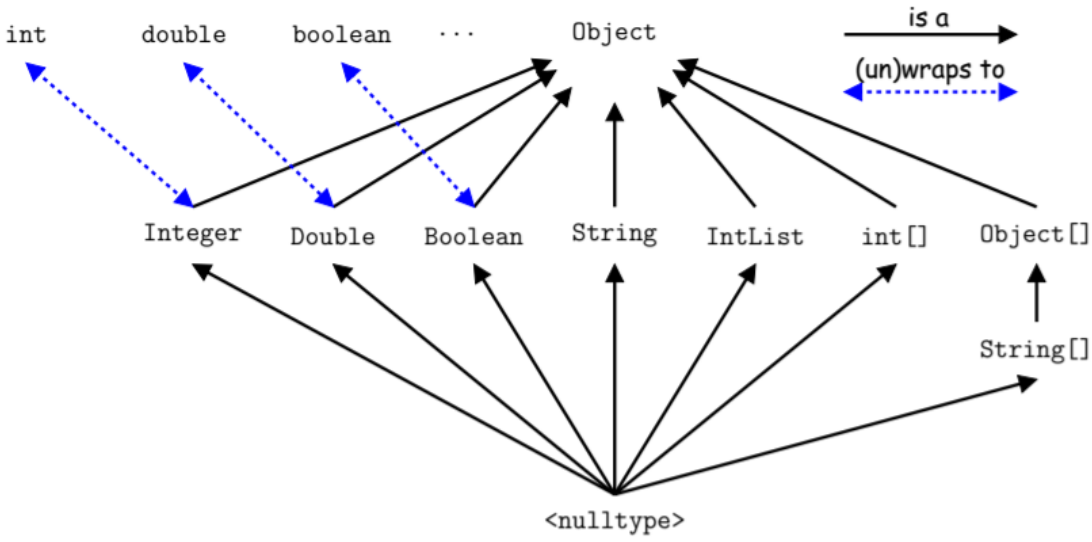
Type Hierarchies

A container with a static type `T` may only 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.

For reference types, reference types form a **type hierarchy**. Some are subtypes of others (parent classes and child classes, as we called them in 61A). `null` is a subtype of all reference types, and all reference types are subtypes of `Object`.

Here is the partial type hierarchy:



(Notice that `String[]` is a subtype of `Object[]`, which is a subtype of `Object`.)

Java is defined in such a way so that any expression of static type `T` always yields a value that is a `T`. Static types are known to the compiler, for example:

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

The compiler insists that in an assignment `L = E`, or function call `X = f(E)`, where:

```
void f(someType L) { ... }
```

the static type of `E` must be a subtype of `L`'s and `X`'s static type.

Coercions

The values of type `short`, for example, are a subset of those of `int` (`short`s are representable as 16-bit integers, `int`s as 32-bit integers). However, `short` is not a subtype of `int`, because they don't quite behave the same.

Instead, we can say that the values of type `short` can be **coerced** into a value of type `int`. The compiler will silently coerce "smaller" integer types to larger ones, e.g. `float` to `double`, and between primitive types and their wrapper types as we saw earlier.

```
short x = 3002;
int y = x;
```

The above code works because of coercion, which was added as a convenience for programmers.

Consequences of Compiler "Sanity Checks"

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

```
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
```

The compiler has figured that not every `Object` is an array, and thus will not let you do this. You must declare that `x` contains an array value: `((int[]) x)[i+1]) = 1`.

The static type of **cast** `(T) E` is `T`. If `x` isn't an array value, a runtime error will be returned.

Overriding and Extending

The notation so far is useful, but it's also clumsy. If I know for sure `Object` variable `x` contains a `String`, why can't I just write `x.startsWith("this")`? The reason is `startsWith` is only defined on strings and not all `Object`s, and the compiler isn't sure it makes sense unless you cast and tell it it's okay.

But if an operation were defined on all `Object`s, then you wouldn't need this clumsy casting. For example, `.toString()` is defined an all `Object`s, which means you can always say `x.toString()` as long as `x` is of a reference type.

The default `.toString()` function is not very useful; on an `IntList`, it would produce a string like `IntList@2f6684`. But for any subtype of `Object`, you may override the default definition.

Overriding a method

For example, if `s` is a `String`, `s.toString()` is the identity function. For any type you define, you may supply your own definition, such as the following in `IntList`:

```
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();
}
```

We can now use the function as follows:

```
x = new IntList(3, new IntList (4, null));
System.out.println(x.toString());
```

which would print out `[3,4]`. Conveniently, the `+` operator on `String`s 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 that `B` inherits from `A` in 61A terminology, or that `A` is a direct superclass of B, we write:

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

By default, `class...` extends `java.lang.Object`. The subtype will inherit all the fields and methods of its direct superclass, and passes them along to any of its subtypes.

In the subclass, you may override an instance method (but not a static method!) by providing a new definition with the exact same signature (name, return type and argument types). A method and all its overridings form a **dynamic method set**.

Conclusion: 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`.

Example

Here's an example that will illustrate all of these concepts:

```
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();
TA daniel = new TA();
Worker wPaul = paul, wDaniel = daniel;

// paul.work() is functionally equivalent to wPaul.work(), and daniel.work() is
functionally equivalent to wDaniel.work()
```

The lesson here is that instance methods select the appropriate method based entirely on the dynamic type of the object, regardless of its static type. `wDaniel`'s static type is a `Worker`, but the method called is for its dynamic type `TA`.

Fields and Static Methods

Fields hide inhierted fields of the same name, while static methods hide methods of the same signature.

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

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

Child tom = new Child();
Parent pTom = tom;
```

```
/** Here are what the calls would do. They are in a fictional, live interpreter
mode of
 *  Java for your convenience. */
>>> tom.x
no
>>> tom.y
way
>>> tom.f()
I wanna!
>>> tom.f(1)
2
>>> pTom.x
0
>>> pTom.y
1
>>> pTom.f()
Ahem!
>>> pTom.f(1)
2
```

Hiding causes confusion, however, so you should understand it, but never do it!

Conclusion

The mechanisms described here allow us to define a kind of generic method. A superclass can define a set of operations (methods) that are common 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 superclass. So when we write methods that operate on the superclass, they will automatically work for all subclasses with no extra work.