CS61B Lecture #8: Object-Oriented Mechanisms

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 or Python, one function can take an argument of any type, and then test the type (if needed).
- 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 or Python.
- But in Java, lists (such as `IntList`) and arrays have a single type of element.
- First, the short answer: any *reference* value can *cast* as (converted to) type `Object` and back, so we can use `Object` as the "generic (reference) type":

```
Object[] things = new Object[2];
things[0] = new IntList(3, null);
things[1] = "Stuff";
IntList thingsList = (IntList) things[0];  // A cast to IntList
// Both ((IntList) things[0]).head and thingsList.head == 3;
// and ((String) things[1]).startsWith("St") is true
// things[0].head                    Illegal
// things[1].startsWith("St")   Illegal
```

- Such *reference casts* don't change the value of a pointer, but rather tell the compiler how to treat it.

And Primitive Values?

- Primitive values (ints, longs, bytes, shorts, floats, doubles, chars, and booleans) are not really convertible to `Object`.
- Presents a problem for "list of anything."
- So Java introduced a set of *wrapper types*, one for each primitive type:

Prim.	Ref.	Prim.	Ref.	Prim.	Ref.
byte	Byte	short	Short	int	Integer
long	Long	char	Character	boolean	Boolean
float	Float	double	Double		

- One can create new wrapper objects for any value (*boxing*):

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

and vice-versa (*unboxing*):

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

Autoboxing

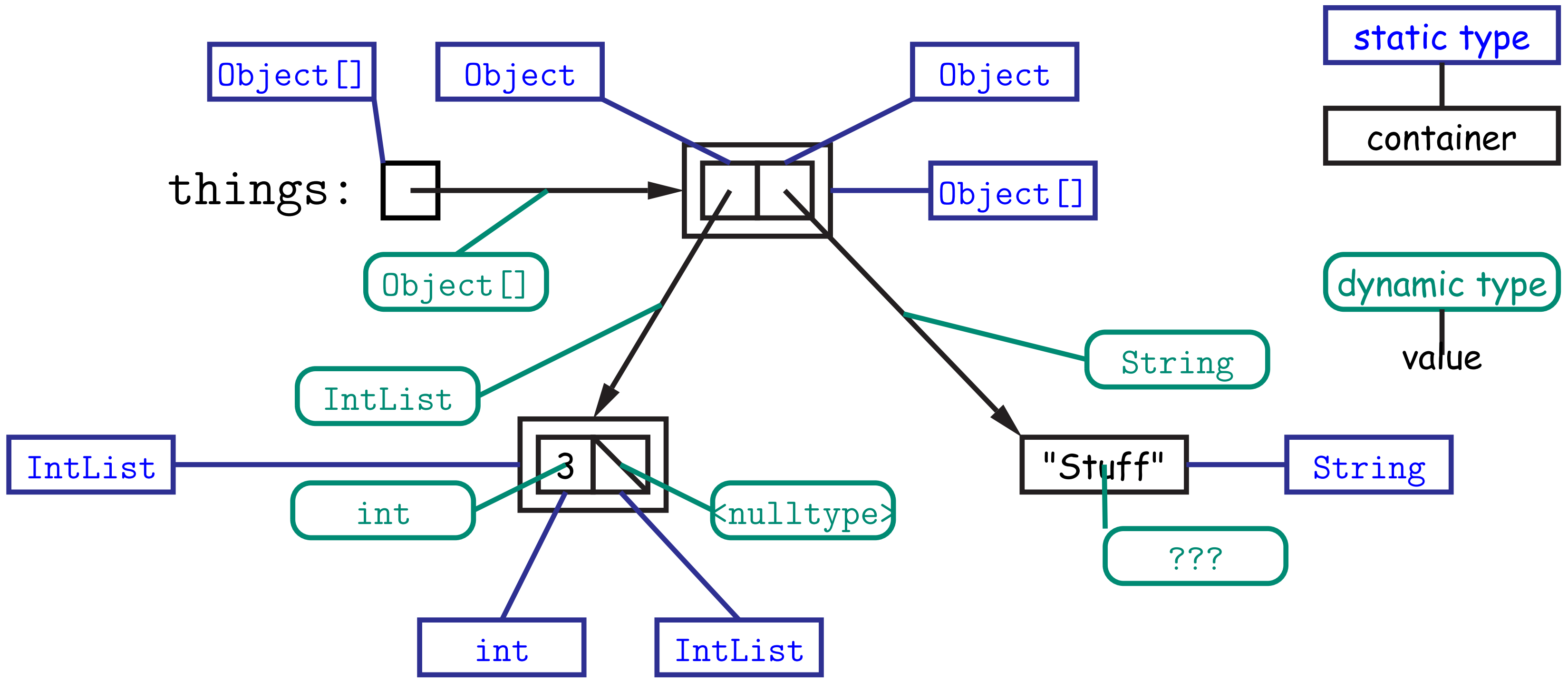
Boxing and unboxing are automatic (in many cases):

```
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]);
    // Prints Integer 1, but NOT unboxed.
```

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.

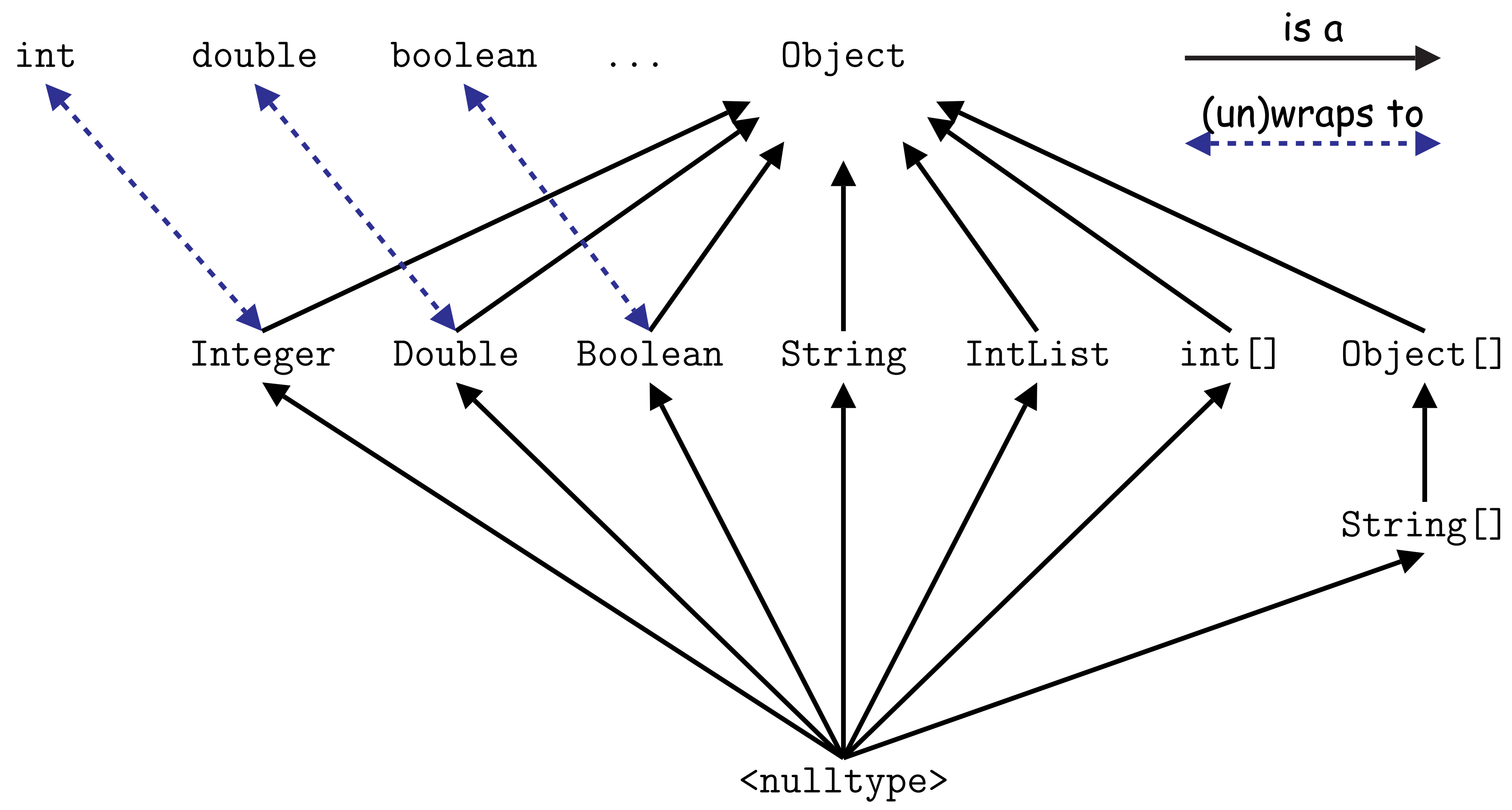
```
Object[] things = new Object[2];
things[0] = new IntList(3, null);
things[1] = "Stuff";
```



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`.

Java Library Type Hierarchy (Partial)



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

  ```
  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

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

  E's static type must be a subtype of L's static type for reference types.
- Similarly, there static-type requirements for other operations: E must have an array type in `E[i]`; actual parameters must have subtypes of their formal parameters,

Primitive Types and Coercions

- Primitive types live outside the hierarchy of reference types.
- Although the values of type `short`, for example, are a subset of those of `int`, we *don't* say that `short` is a subtype of `int`, because they don't quite behave the same.
- However, values of type `short` can be *coerced* (converted) to a value of type `int`, using the same cast syntax as for reference types:

```
short x = (short) 3002;
long y = 10000L;
int z = (int) y;
long q = 10000000000000L;
int r = (int) q;
System.out.println(r);  // Prints -727379968 (?????)
```

- As the values of `r` shows, coercions of primitive types, unlike those of reference types, are computations that can change values.

Automatic Coercions, Promotions

- Certain coercions, such converting from `short` to `int`, are considered obvious and therefore intrusive.
- So the language silently coerces "smaller" integer types to larger ones, `float` to `double`, and integer types to `float` or `double`.
- These are called *promotions*.
- Finally, since the compiler can obviously tell what the value of an **int** literal is, it will convert integer literals to shorter integer types if the values fit:

```
byte x = 127;
short y = -1024;
char z = 0x0398;     // Θ
```

Consequences of Compiler's "Sanity Checks"

- These are a *conservative* rules. The last line of the following, which you might think is perfectly 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
  ```

  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:

  ```
      ((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. For example, in `IntList`, could add

```
@Override   // Compiler checks that Object really has a toString.
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 = new IntList(3, new IntList(4, null))`, then `x.toString()` is `"[3 4]"`.
- Conveniently, various operations requirihg Strings call `.toString()` for you, so for an `IntList x`, you can write:

```
"Values: " + x    System.out.println(x)    System.out.printf("%s", x);
```

Extending a Class

- To say that class `B` is a direct subtype of class `A` (or `A` is a *direct superclass* of `B`), write

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

- By default, `class ... extends java.lang.Object`.
- The subtype *inherits* all fields and methods of its direct 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).

Rule of Instance Method Calls:

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 daniel = new TA();        | daniel.work()  ==> doLab(); discuss(); ...
Worker wPaul = paul,         | wPaul.work() ==> collectPay();
       wDaniel = daniel;     | wDaniel.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?

```
class Parent {                          class Child extends Parent {
  int x = 0;                              String x = "no";
  static int y = 1;                       static String y = "way";
  static void f() {                       static void f() {
     System.out.printf("Ahem!%n");           System.out.printf("I wanna!%n");
  }                                       }
  static int f(int x) {                 }
    return x+1;
  }
}
```

```
  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 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.