

Overloading

to get `System.out.print(x)` to print `x`, regardless of

Python, one function can take an argument of any type, but the type (if needed).

Methods specify a single type of argument.

Example: *overloading*—multiple method definitions with the same name and different numbers or types of arguments.

Example: `out` has type `java.io.PrintStream`, which defines

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

Each is a different function. Compiler decides which to call based on arguments' types.

13-06 2016

CS61B: Lecture #8 2

And Primitive Values?

Primitive values (ints, longs, bytes, shorts, floats, doubles, chars, etc.) are not really convertible to `Object`.

Problem for "list of anything."

Introduced a set of *wrapper types*, one for each primitive

Ref.	Prim.	Ref.	Prim.	Ref.	
Byte	byte	Short	short	Integer	int
Character	char	Character	char	Boolean	boolean
Double	double	Double	double		

Created new wrapper objects for any value (*boxing*):

```
Integer three = new Integer(3);
Integer threeObj = Three;
```

Unboxing (*unboxing*):

```
Integer threeObj = Three;
int i = threeObj.intValue();
```

13-06 2016

CS61B: Lecture #8 4

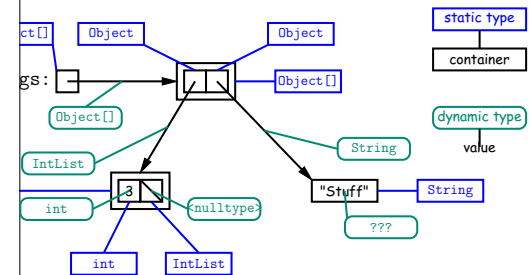
Dynamic vs. Static Types

Java has a type—its *dynamic type*.

Example: `number` (variable, component, parameter), literal, function argument, or operator expression (e.g. `x+y`) has a type—its *static type*.

Every expression has a static type.

```
Object[] gs = new Object[2];
IntList intList = new IntList(3, null);
String stuff = "Stuff";
```



13-06 2016

CS61B: Lecture #8 6

Lecture #8: Object-Oriented Mechanisms

Lecture: the bare mechanics of "object-oriented programming"

Topic is: Writing software that operates on many kinds of objects

13-06 2016

CS61B: Lecture #8 1

Generic Data Structures

How to get a "list of anything" or "array of anything"?

Problem in Scheme or Python.

Lists (such as `IntList`) and arrays have a single type of element

Short answer: any *reference* value can be converted to `Object` and back, so can use `Object` as the "generic type":

```
Object[] things = new Object[2];
IntList intList = new IntList(3, null);
String stuff = "Stuff";
IntList things[0].head == 3;
String things[1].startsWith("St") is true
intList.head Illegal
String things[1].startsWith("St") Illegal
```

13-06 2016

CS61B: Lecture #8 3

Autoboxing

Autoboxing, boxing and unboxing is automatic (in many cases):

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

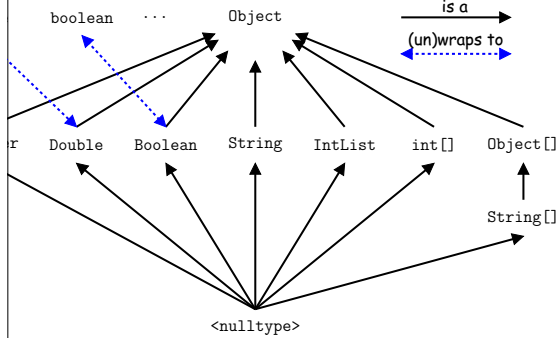
```
Integer someInts = { 1, 2, 3 };
Integer[] someIntsArray = someInts.toArray(new Integer[0]);
System.out.println(x);
```

```
System.out.println(someIntsArray[0]); // Prints 3, but NOT unboxed.
```

13-06 2016

CS61B: Lecture #8 5

A Library Type Hierarchy (Partial)



13-06 2016

CS61B: Lecture #8 8

Coercions

Short types, for example, are a subset of those that are representable as 16-bit integers, ints as 32-bit

say that short is a subtype of int, because they don't have the same.

say that values of type short can be *coerced* (converted) to a value of type int.

Right fudge: compiler will silently coerce "smaller" into larger ones, float to double, and (as just seen) between primitive types and their wrapper types.

002;

it complain.

13-06 2016

CS61B: Lecture #8 10

Overriding and Extension

that is clumsy.

Object variable x contains a String, why can't I write, x.hashCode("this")?

hashCode() is only defined on Strings, not on all Objects, so the compiler is sure it makes sense, unless you cast.

hashCode() were defined on all Objects, then you wouldn't need casting.

hashCode() is defined on all Objects. You can always say x.hashCode() if x has a reference type.

toString() function is not very useful; on an IntList, toString() returns a string like "IntList@2f6684"

As a subtype of Object, you may override the default definition.

13-06 2016

CS61B: Lecture #8 12

Type Hierarchies

A class with (static) type T may contain a certain value only if the value is "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.

Classes are subtypes of themselves (& that's all for primitive types)

Classes form a *type hierarchy*; some are subtypes of others, and some are subtypes of all reference types.

Object types are subtypes of Object.

13-06 2016

CS61B: Lecture #8 7

The Basic Static Type Rule

is designed so that any expression of (static) type T always evaluates to a value that "is a" T.

Variables are "known to the compiler," because you declare them,

```

class C {
    // Static type of field
    T s; { // Static type of call to f, and of parameter
        // Static type of local variable
    }
}
    
```

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

Remember that in an *assignment*, L = E, or function call, f(E),

```

SomeType L { ... },
    
```

E must be subtype of L's static type.

Rules apply to E[i] (static type of E must be an array) and array operations.

13-06 2016

CS61B: Lecture #8 9

Consequences of Compiler's "Sanity Checks"

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

```

new int[2];
A; // All references are Objects
    // Static type of A is array...
    // But not of x: ERROR
    
```

Remember that not every Object is an array.

How do you know that x contains array value?

The compiler will tell you, like this:

```

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

The static type of cast (T) E is T.

Remember that x isn't an array value, or is null?

Remember that you will have runtime errors—exceptions.

13-06 2016

CS61B: Lecture #8 11

Extending a Class

class B is a direct subtype of class A (or A is a direct supertype of B), write

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

class ... extends java.lang.Object.

class B inherits all fields and methods of its superclass (and also of any of its subtypes).

class B may override an instance method (not a static method), providing a new definition with same signature (name, return type, and parameter types).

class B and all its overrides form a *dynamic method set*.

When you call `x.f(...)` is an instance method, then the call `x.f(...)` uses the *dynamic method set* of `x`. If `x` is a reference variable, the *dynamic method set* of `x` is the *dynamic method set* of the object that `x` refers to, not the static type of `x`.

What About Fields and Static Methods?

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

class Child extends Parent {
    // inherits x and y from Parent
    // inherits f() from Parent
}

int main() {
    Parent p = new Parent();
    Child c = new Child();
    p.x; // "no"
    p.y; // "way"
    p.f(); // "I wanna!"
    c.x; // "no"
    c.y; // "way"
    c.f(); // "I wanna!"
}
```

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

Child hides inherited fields of same name; static methods of the same signature. Hiding static methods causes confusion; so understand it, but don't do it!

Overriding toString

If `s` is a `String`, `s.toString()` is the identity function.

When you define a class, you may supply your own definition. For example, `IntList`, could add

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

If you create `IntList(3, new IntList(4, null))`, then `x.toString()` returns `"[3 4]"`.

The `+` operator on `Strings` calls `.toString` when asked to concatenate two `Strings`, and so does the `%s` formatter for `printf`.

When you write `printf("%s", x)`, you can supply an output function for any type you want.

Illustration

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

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

Prof(); | paul.work() ==> collectPay();
TA(); | daniel.work() ==> doLab(); discuss(); ...
Paul; | wPaul.work() ==> collectPay();
daniel; | wdaniel.work() ==> doLab(); discuss(); ...
```

When you write `TA().work()`, you call `work()` on the object that `TA()` returns, not the static type of `TA()`. If you write `Paul.work()`, you call `work()` on the object that `Paul` refers to, not the static type of `Paul`. If you write `daniel.work()`, you call `work()` on the object that `daniel` refers to, not the static type of `daniel`. In general, when you write `x.work()`, you call `work()` on the object that `x` refers to, not the static type of `x`. This is true for all instance methods (only), select method based on dynamic state, but we'll see it has profound consequences.

What's the Point?

The `toString` method described here allows us to define a kind of *generic* method.

We can define a set of operations (methods) that are common to many different classes.

We can then provide different implementations of these methods, each specialized in some way.

Subclasses will have at least the methods listed by the superclass.

When you write methods that operate on the superclass, they will work for all subclasses with no extra work.