:cture #8: Object-Oriented Mechanisms

lecture: the bare mechanics of "object-oriented pro-

topic is: Writing software that operates on many kinds

Overloading

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

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

hods specify a single type of argument.

ion: *overloading*—multiple method definitions with the nd different numbers or types of arguments.

out has type java.io.PrintStream, which defines

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

e is a different function. Compiler decides which to call of arguments' types.

Generic Data Structures

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

blem in Scheme or Python.

lists (such as IntList) and arrays have a single type of

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

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

And Primitive Values?

ues (ints, longs, bytes, shorts, floats, doubles, chars,) are not really convertible to Object.

roblem for "list of anything."

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

ef.	Prim.	Ref.	Prim.	Ref.
yte	short	Short	int	Integer
ong	char	Character	boolean	Boolean
oat	double	Double		

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

```
hree = new Integer(3);
reeObj = Three;
```

sa (*unboxing*):

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

Autoboxing

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

```
e = 3;
 Three;
ree + 3;

omeInts = { 1, 2, 3 };
 someInts) {
ut.println(x);


println(someInts[0]);  // Prints Integer 1, but NOT
```

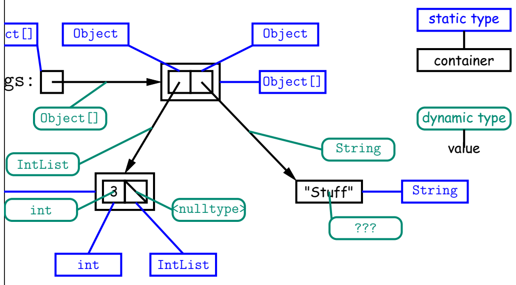
Dynamic vs. Static Types

has a type—its *dynamic type.*

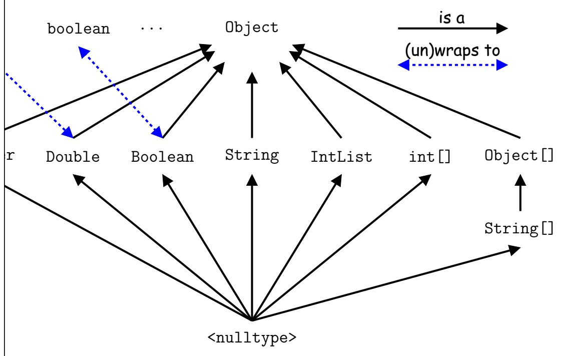
ner (variable, component, parameter), literal, function rator expression (e.g. x+y) has a type—its *static type.*

very *expression* has a static type.

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



a Library Type Hierarchy (Partial)



boolean

...

Object

is a

(un)wraps to

r

Double

Boolean

String

IntList

int[]

Object[]

String[]

<nulltype>

Type Hierarchies

with (static) type T may contain a certain value only if s a" T—that is, if the (dynamic) type of the value is a T. Likewise, a function with return type T may return hat are subtypes of T.

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

ypes form a *type hierarchy;* some are subtypes of oth- pe is a subtype of all reference types.

e types are subtypes of Object.

Coercions

f type short, for example, are a subset of those of are representable as 16-bit integers, ints as 32-bit

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

say that values of type short can be *coerced* (con- value of type int.

ight fudge: compiler will silently coerce "smaller" inte- larger ones, float to double, and (as just seen) be- ive types and their wrapper types.

```
002;
```

t complaint.

The Basic Static Type Rule

gned so that any expression of (static) type T always e that "is a" T.

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

```
        // Static type of field
t s)  { // Static type of call to f, and of parameter
        // Static type of local variable
```

re-declared by the language (like 3).

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

```
SomeType L) { ... },
```

e must be subtype of L's static type.

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

Overriding and Extension

far is clumsy.

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

th is only defined on Strings, not on all Objects, so the t sure it makes sense, unless you cast.

eration *were* defined on all Objects, then you *wouldn't* casting.

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

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

subtype of Object, you may *override* the default defi-

uences of Compiler's "Sanity Checks"

servative rule. The last line of the following, which you s perfectly sensible, is illegal:

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

ures that not every Object is an array.

know that x contains array value!?

till must tell the compiler, like this:

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

type of cast (T) E is T.

isn't an array value, or is null?

ve have runtime errors—exceptions.

Overriding toString

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

: you define, you may supply your own definition. For
ntList, could add

```
ng toString() {
uffer b = new StringBuffer();
d("[");
tList L = this; L != null; L = L.tail)
ppend(" " + L.head);
d("]");
b.toString();
```

IntList(3, new IntList(4, null)), then x.toString()

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

ick, you can supply an output function for any type you

Extending a Class

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

```
 extends A { ... }
```

lass ... extends java.lang.Object.

inherits all fields and methods of its *superclass* (and
along to any of its subtypes).

u may *override* an instance method (*not* a static method),
a new definition with same *signature* (name, return
nt types).

a method and all its overridings form a *dynamic method*

f f(...) is an instance method, then the call x.f(...)
r overriding of f applies to the *dynamic type* of x, re-
the static type of x.

Illustration

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

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

```
Prof();      | paul.work()  ==> collectPay();
TA();        | daniel.work()  ==> doLab(); discuss(); ...
aul,         | wPaul.work() ==> collectPay();
 daniel;     | wDaniel.work() ==> doLab(); discuss(); ...
```

stance methods (only), select method based on *dynamic*
state, but we'll see it has profound consequences.

t About Fields and Static Methods?

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

```
ew 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
```

hide inherited fields of same name; static methods
f the same signature.
ding causes confusion; so understand it, but don't do it!

What's the Point?

sm described here allows us to define a kind of *generic*

can define a set of operations (methods) that are com-
different classes.

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

s will have at least the methods listed by the super-

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