In an naive \textit{interpreted} implementation, each object is represented by a heap-allocated record, containing
\begin{itemize}
  \item Name and values of each instance variable.
  \item Pointer to class description record.
\end{itemize}
Each class is represented by a (essentially static) record with:
\begin{itemize}
  \item Name and code pointer for each class method.
  \item Pointer to super-class’s record.
\end{itemize}

\section*{Example}
(based on the code from slides 12,13,17 of previous lecture)

\begin{itemize}
  \item \textbf{class Object}
  \item \textbf{class DisplayObject}
  \item \textbf{class Line}
  \item \textbf{class Text}
  \item \textbf{class Bitmap}
\end{itemize}

\section*{Interpreted Implementation of Operations}
To perform a message \texttt{send} (function call) at runtime, the interpreter does a method lookup, starting from the receiver object, as follows:
\begin{itemize}
  \item Use class pointer to find class description record.
  \item Search for method in class record. If found, invoke it; otherwise, continue search in superclass record.
  \item If no method found, issue “Message Not Understood” error. (Can’t happen in strongly-typed languages; more later.)
\end{itemize}

Instance variables are accessed in the object record. Pseudo-variable \texttt{this} always points to the receiver object record; \texttt{super} always points to the superclass.
How about “compiling” OO languages? Dynamic binding makes compilation difficult:

- method code doesn’t know the precise class to which the object it is manipulating belongs,
- nor the precise method that will execute when it sends a message.

Instance variables are not so hard.

- Code that refers to instance variables of a given class will actually operate on objects of that class or of a subclass.
- Since a subclass always extends the set of instance variables defined in its superclass, compiler can consistently assign each instance variable a fixed (static) offset in the object record; this offset will be the same in every object record for that class and any of its subclasses.
- Compiled methods can then reference variables by offset rather than by name, just like ordinary record field offsets.

(Multiple inheritance schemes cause problems.)

Handling message sends is harder, because methods can be overridden by subclasses.

Simple approach: keep a per-class static method table (or vtable) and “compile” message sends into indirect jumps through fixed offsets in this table.

Example: Classes in slides 12,13,17 of previous lecture all have this vtable structure:

```
<table>
<thead>
<tr>
<th>(offset 0)</th>
<th>draw code ptr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>(offset 1)</td>
<td>translate code ptr.</td>
</tr>
</tbody>
</table>
```

These tables can get large, and much of their contents will be duplicated between a class and its superclasses. Still, this approach is used by C++, Java. (Again, multiple inheritance – and Java interfaces – cause complications.)

Unlike Java, C++ uses the “direct model” for representing object values.

So in declarations like this

```
DisplayObject x(10,20);
Line y(30,40,2,4);
```

variables x and y represent actual instance records, not pointers to records. In order to allocate storage space for such instances, the compiler uses the declared type of the variable, so x has space for only x0 and y0, not delx and dely.

Now, an assignment like

```
... x = y; ...
```

works by truncating the Line to fit in the space of a DisplayObject; the delx and dely fields are lost.

Not likely to be what you wanted!
TYPES VS. CLASSES

As noted, many languages use a single class hierarchy to describe both subtyping and inheritance.

- Historically, it took a long time for language designers to figure out that subtyping and inheritance are fundamentally distinct ideas.
- From a software engineering point of view, inheritance is very mixed blessing, because it tends to break encapsulation.
- Availability of interfaces in Java and C# is a major improvement.

We can define sensible subtyping rules without reference to classes at all, just based on object structure.

In fact, subtyping makes sense even without static types, if we think of it as a discipline that will prevent our getting runtime errors like “method not understood” or “field not defined.”

PRINCIPLES OF SUBTYPING

Let’s model object interfaces by record types with named fields and methods (using ML-like notation).

Example (corresponding to slides 9 of previous lecture):

```plaintext
type DisplayObject =
  { x0: int ref, y0: int ref,
    draw : unit -> void,
    translate : int * int -> void }

type Line =
  { x0: int ref, y0: int ref,
    del_x: int ref, del_y: int ref,
    draw : unit -> void,
    translate : int * int -> void }
```

We write $t' <: t$ to mean $t'$ is a subtype of $t$. For example, we would expect that Line $<$: DisplayObject.

SUBTYPING RULES

We extend our language’s type system with a subsumption rule:

$$
egin{align*}
T E , e & : t' \\
& t' <: t
\end{align*}
$$

(Sub)

As usual, we want this rule to be sound, i.e., using it to prove a program type-correct won’t allow well-typed programs to go wrong at runtime. In effect, this codifies the principle of safe substitution.

Now we must define the $<$: relation so that subsumption remains sound. We assume

$$
\begin{align*}
t & <: t \\
\end{align*}
$$

(Reflexive)

$$
\begin{align*}
t'' & <: t' \\
t' & <: t \\
\Rightarrow \\
\frac{t'' \ll : t'}{t'' <: t} \\
\end{align*}
$$

(Transitive)

MORE SUBTYPING RULES

Two easy rules for records:

$$
\begin{align*}
\{ l_1 : t_1, \ldots, l_n : t_n : t \} & <: \{ l_1 : t_1, \ldots, l_n : t_n \} \\
& \text{(W)}
\end{align*}
$$

$$
\begin{align*}
t'_k & <: t_k \\
\{ l_1 : t_1, \ldots, l_k : t'_k, \ldots, l_n : t_n \} & <: \{ l_1 : t_1, \ldots, l_k : t_k, \ldots, l_n : t_n \} \\
& \text{(D)}
\end{align*}
$$

The (Width) rule says that we can always make a subtype by adding fields to a record. The (Depth) rule says that we can always make a subtype by replacing the type of any field with a subtype.

(Also, we assume that order in records doesn’t matter.)
Making a sound rule for function subtyping is harder. A first guess:

\[
\frac{t'_1 <: t_1 \quad t'_2 <: t_2}{t'_1 \rightarrow t'_2 <: t_1 \rightarrow t_2} \quad \text{(wrong!)}
\]

But this would imply, for example, that given

\[
f : \text{DisplayObj} \rightarrow \text{DisplayObj} \\
g : \text{Line} \rightarrow \text{Line}
\]

then typeof\(g\) <: typeof\(f\), so we should be able to replace any use of \(f\) by \(g\) without destroying type-correctness.

Suppose we have \(x = f(y)\), so \(x\) and \(y\) both have type \(\text{DisplayObj}\). If we try to replace \(f\) by \(g\):

- The result value of \(g\) is a Line, which is a subtype of \(\text{DisplayObj}\), so that's ok to bind to \(x\).
- But the argument value of \(g\) must be a Line, and \(y\) is only guaranteed to be a \(\text{DisplayObj}\)!

The rule we really want is:

\[
\frac{t_1 <: t'_1 \quad t'_2 <: t_2}{t'_1 \rightarrow t'_2 <: t_1 \rightarrow t_2} \quad \text{(Fun)}
\]

For example, given

\[
f : \text{DisplayObj} \rightarrow \text{DisplayObj} \\
h : \text{Object} \rightarrow \text{Line}
\]

(where \(\text{Object}\) is the universal supertype), we could validly replace \(f\) by \(h\).

We say that the <: relation is **covariant** on the result type of functions, and **contravariant** on the argument type.

What about subtyping for mutable variables (ref cells) and array elements? When can we safely replace a variable \(x : t_x\) by a variable \(y : t_y\)? A mutable variable can appear in two contexts:

- It can be stored into, e.g. \(x := e\). To safely replace \(x\) by \(y\), \(y\) must be able to hold any value that \(x\) can, i.e., we must have \(t_x <: t_y\).
- It can be fetched from, e.g. \(z := x\). To safely replace \(x\) by \(y\), the value in \(y\) must be containable by \(z\), which expects a value from \(x\), i.e., we must have \(t_y <: t_x\).

Combining these requirements, we see that \(t_x\) and \(t_y\) must be equal. So no non-trivial subtyping should be permitted on references and array elements.
Java appears to break this rule. If \( B \) is a subclass of \( A \), then \( B[] \) is treated as a subtype of \( A[] \). This is fine when fetching, but can be unsound when storing, because it allows an \( A \) value to be stored as an element of an \( B[] \), e.g.

```java
B[] bx = new B[100];
A[] ax = bx; // permitted because B[] <: A[]
ax[0] = new A(); // oops!
```

To guard against this, Java inserts an (expensive) \textit{runtime} check on every array store operation to make sure that the stored value actually belongs to the same class as the array.