Consider the last two lines of the example in the following slide:

- In $L_1$, $S$ points to a Shape object, but it could just as well have pointed to an object of any one of Shape’s subtypes, Square and Circle.
- If, for example, $S$ had been a Circle, the assignment $C := S$ would have been perfectly OK. In $L_2$, however, $S$ is a Shape and the assignment $C := S$ is illegal (a Shape isn’t a Circle).

```plaintext
VAR S : Shape; Q : Square; C : Circle;
BEGIN
  Q := NEW (Square);
  C := NEW (Circle);
  S := Q; (* OK *)
  S := C; (* OK *)
  Q := C; (* Compile-time Error *)
L1: S := NEW (Shape);
L2: C := S; (* Run-time Error *)
END;
```
Typechecking Rules

TYPE
T = CLASS ··· END;
U = T CLASS ··· END;
S = T CLASS ··· END;

VAR t,r : T; u : U; s : S;

A variable of type T may refer to an object of T or one of T’s subtypes.

<table>
<thead>
<tr>
<th>Assignment</th>
<th>Compile-time</th>
<th>Run-Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>t := r;</td>
<td>Legal</td>
<td>Legal</td>
</tr>
<tr>
<td>t := u;</td>
<td>Legal</td>
<td>Legal</td>
</tr>
<tr>
<td>u := t;</td>
<td>Legal</td>
<td>Check</td>
</tr>
<tr>
<td>s := u;</td>
<td>Illegal</td>
<td></td>
</tr>
</tbody>
</table>

Run-time Type Checking

The Modula-3 runtime-system has three functions that are used to implement typetests, casts, and the TYPECASE statement.

- **NARROW** takes a template and an object as parameter. It checks that the type of the object is a subtype of the type of the template. If it is not, a run-time error message is generated. Otherwise, **NARROW** returns the object itself.

- **ISTYPE(S,T : Template) : BOOLEAN;**
- **NARROW(Object, Template) : Object;**
- **TYPECODE(Object) : CARDINAL;**

Run-time Checks

Casts are turned into calls to **NARROW**, when necessary:

```
VAR S : Shape; VAR C : Circle;
BEGIN
  S := NEW (Shape); C := S;
END;
⇓
VAR S : Shape; VAR C : Circle;
BEGIN
  S := malloc (SIZE(Shape));
  C := NARROW(S, Circle$Template);
END;
```
Implementing ISTYPE

We follow the object's template pointer, and immediately (through the templates' parent pointers) gain access to it's place in the inheritance hierarchy.

```pascal
PROCEDURE ISTYPE (S, T : TemplatePtr) : BOOLEAN;
BEGIN
  LOOP
    IF S = T THEN RETURN TRUE; ENDIF;
    S := S^.parent;
    IF S = ROOT THEN RETURN FALSE; ENDIF;
  ENDLOOP
END ISTYPE;
```

Implementing NARROW

NARROW uses ISTYPE to check if S is a subtype of T. Of so, S is returned. If not, an exception is thrown.

```pascal
PROCEDURE NARROW(T:TemplatePtr; S:Object):Object;
BEGIN
  IF ISTYPE(S^.template, T) THEN
    RETURN S (* OK *)
  ELSE WRITE "Type error"; HALT;
  ENDIF;
END NARROW;
```

Run-time Checks — Example

```
TYPE T = CLASS [...];
S = T CLASS [...];
U = T CLASS [...];
V = U CLASS [...];
X = S CLASS [...];
Y = U CLASS [...];
Z = U CLASS [...];
VAR x : X;
```

![Run-time Checks — Example diagram](image-url)
Run-time Checks – An $O(1)$ Algorithm

- The time for a type test is proportional to the depth of the inheritance hierarchy. Two algorithms do type tests in constant time:
  1. Norman Cohen, “Type-Extension Type Tests can be Performed in Constant Time.”
  2. Paul F. Dietz, “Maintaining Order in a Linked List.”
- The second is more efficient, but requires the entire type hierarchy to be known. This is a problem in separately compiled languages.
- SRC Modula-3 uses Dietz' method and builds type hierarchies of separately compiled modules at link-time.
- These algorithms only work for single inheritance.

Run-time Checks – Alg. II (b)

In the Compiler (or Linker):
1. Build the inheritance tree.
2. Perform a preorder traversal and assign preorder numbers to each node.
3. Similarly, assign postorder numbers to each node.
4. Store $T$'s pre- and postorder numbers in $T$'s template.

In the Runtime System:

```
PROCEDURE ISTYPE (S, T: TemplatePtr): BOOLEAN;
BEGIN
RETURN (T.pre <= S.pre) AND (T.post >= S.post);
END ISTYPE;
```

Run-time Checks – Alg. II (c)

```
<table>
<thead>
<tr>
<th>TYPE</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>T = CLASS [···];</td>
<td></td>
</tr>
<tr>
<td>S = T CLASS [···];</td>
<td></td>
</tr>
<tr>
<td>U = T CLASS [···];</td>
<td></td>
</tr>
<tr>
<td>V = U CLASS [···];</td>
<td></td>
</tr>
<tr>
<td>X = S CLASS [···];</td>
<td></td>
</tr>
<tr>
<td>Y = U CLASS [···];</td>
<td></td>
</tr>
<tr>
<td>Z = U CLASS [···];</td>
<td></td>
</tr>
<tr>
<td>√ISTYPE(Y, U)</td>
<td>U.pre ≤ Y.pre</td>
</tr>
<tr>
<td></td>
<td>U.post ≥ Y.post</td>
</tr>
<tr>
<td>ISTYPE(Z, S)</td>
<td>S.pre ≤ Z.pre</td>
</tr>
<tr>
<td></td>
<td>S.post ≠ Z.post</td>
</tr>
<tr>
<td>√ISTYPE(Z, T)</td>
<td>T.pre ≤ Z.pre</td>
</tr>
<tr>
<td></td>
<td>T.post ≥ Z.post</td>
</tr>
</tbody>
</table>
```

Run-time Checks – Alg. II (d)

- Consider U:
  1. U’s pre-number is ≤ all it’s children’s pre numbers.
  2. U’s post-number is ≥ all it’s children’s post numbers.
Consider a method invocation $m.P()$. The actual procedure called will depend on the run-time type of $m$.

If more than one method can be invoked at a particular call site, we have to inline all possible methods. The appropriate code is selected by branching on the type of $m$.

To improve on method inlining we would like to find out when a call $m.P()$ can call exactly one method.

**Inlining Methods — Example**

```plaintext
TYPE T = CLASS [f : T][
    METHOD M (); BEGIN END M;
];

TYPE S = CLASS EXTENDS T [
    METHOD N (); BEGIN END N;
    METHOD M (); BEGIN END M;
];

VAR x : T; y : S;
BEGIN
    x.M();
    y.M();
END;
```
For each type $T$ and method $M$ in $T$, find the set $S_{T,M}$ of method overrides of $M$ in the inheritance hierarchy tree rooted in $T$.

If $x$ is of type $T$, $S_{T,M}$ contains the methods that can be called by $x.M()$.

We can improve on type hierarchy analysis by using a variant of the Reaching Definitions data flow analysis.

```
TYPE T = class []
  method M (); begin end M;
];

TYPE S = class extends T []
  method N (); begin end N;
  method M (); begin end M;
];

VAR x : T; y : S;
BEGIN
  x.M(); ⇐ $S_{T,M} = \{T.M, S.M\}$
  y.M(); ⇐ $S_{S,M} = \{S.M\}$
END;
```

**Summary**

- The time for a type test is proportional to the depth of the inheritance hierarchy. Many algorithms do type tests in constant time:
  1. Norman Cohen, “Type-Extension Type Tests can be Performed in Constant Time.”
  2. Paul F. Dietz, “Maintaining Order in a Linked List”.

**Readings and References**

- Read Scott: 529–551, 554–561, 564–573
What happens when both a class and its subclass have an instance variable with the same name?

- The subclass gets both variables. You can get at both of them, directly or by casting. Here's an example in Java:

```java
class C1 { int a; }
class C2 extends C1 { double a; }
class C {
    static public void main(String[] arg) {
        C1 x = new C1(); C2 y = new C2();
        x.a = 5; y.a = 5.5;
        ((C1)y).a = 5;
    }
}
```