CS61B Lecture #13

Administrative:

- Pick up Data Structures reader at Vick Copy.
- Before Project #1 due date, will run auto-grader tests *Wednesday night only*. If you get something submitted by then, you'll see the results of our testing on it. You can still resubmit until deadline.
- Otherwise (if you finish at the last minute), you aren't penalized, but you'll have to rely on your own testing.

Today:

- Final Java Lecture (for now)
- Scope rules

Readings for Today: *Programming Into Java*, §4.1, §5.6.3

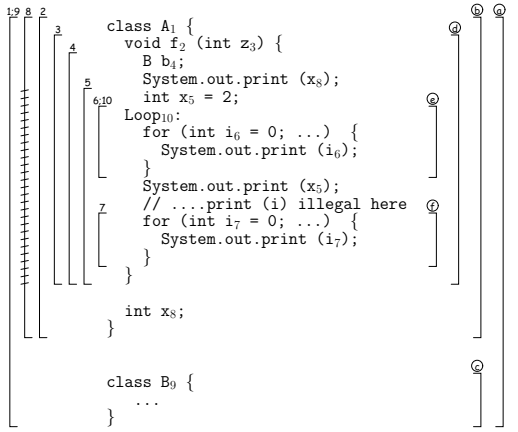
Readings for next Topic: *Data Structures*, Chapter 1.

Terminology

- *Scope rules:* what *declaration* governs each *use* of an identifier.
- *Scope of a declaration:* section of *program text* where it applies—where it defines the meaning of its identifier.
- *Declarative Region:* construct that contains declarations and defines a common scope for them.
- *Extent of a container:* the *period of time* during which a container (local variable, parameter, field, object) exists.
- Like Java, most modern programming languages use two varieties of scope rule:
  - *block-structured scope* to govern unqualified identifiers (like the `i` in `i+1` or the `A` in `A.B`);
  - *selection* (like the `B` in `A.B`). Scope of declarations in `A` (or in `A`'s type) extends to the immediate right of the dot in `A.`.

Block-Structured Scope

- The rule: declarations in a given declarative region govern identifiers in that region, and in smaller regions nested inside it.
- In case of ambiguity, the smallest (innermost, most deeply nested) declarative region's declaration wins.

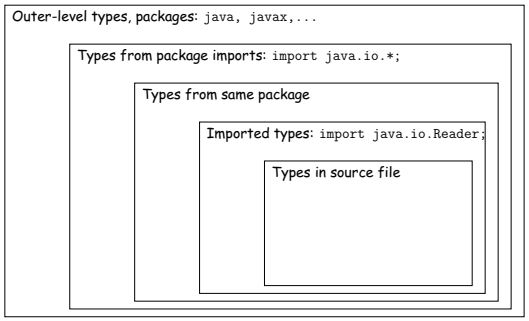


Scope of Type Names

- Type declarations (classes, interfaces) "leak out" of their file into surrounding package.
- Otherwise, available by selection.
- As a shorthand, can be imported:

```
import java.util.Stack; // 'Stack' now short for java.util.
import java.io.*;       // Can leave off java.io from all i
import java.lang.*;     // Automatically added to all progr
```

Has no effect except to define shorthand.



Extent

- Scope and extent orthogonal concepts.
- Containers can exist even when out of scope: local variable still exists during call to another function.
- Three basic kinds:
  - *Static extent:* exist during entire execution.
  - *Automatic extent:* exist from time of declaration until end of execution of declarative region that contains declaration.
  - *Dynamic extent:* exist from expression that creates them until (in C or C++) deleted or (in Java) until no longer reachable.
- In Java,
  - static variables come closest to having have static extent. Usually exist from time their containing class is first used until end of program.
  - Parameters, local variables have automatic extent.
  - Objects created by **new** have dynamic extent.
- **Common error:** don't confuse extent of local variable with extent of object it points to:

```
{ StringBuffer s = new StringBuffer(); ... return s; }
```

Now `s` is gone, but *not* the new `StringBuffer`.

"Dangling References"

- What if variable in scope, but extent terminated?
- Senseless to use the variable, but apparently possible.
- Normally, avoided by block structure.
- However, there is one case in Java:

```
Actor appender (final StringBuffer buffer) {
  return new Actor () {
           public void act (String s) {
             buffer.append (s);
           }
         };
}
```

(interface `Actor` is in Lecture #7).

- In

```
StringBuffer myBuffer = new StringBuffer ();
Actor myActor = appender (myBuffer);
myActor.act ("Hello");
```

The parameter **buffer** would normally go away when `appender` returns, but it is still used in later call to `myActor.act` (Routine in CS61A).

- Allowed in Java if `buffer` is **final** (not assignable): new object can keep copy of `buffer`'s value.

New Subject: instanceof

- The boolean test

```
if (x instanceof AType)
  S
```

  executes $\mathcal{S}$ iff the dynamic type of `x` (must be a reference value) is a subtype of `AType`.

- Java uses this internally for multiple exceptions:

```
void f () throws InterruptedException {
  try { ... }
  catch (FileNotFoundException e) { S1 }
  catch (IOException e) { S2 }
}
```

  is roughly equivalent to

```
try { ... }
catch (Exception e0) {
  if (e0 instanceof FileNotFoundException) {
    FileNotFoundException e = (FileNotFoundException) e0;
    S1
  } else if (e0 instanceof IOException)
   { IOException e = (IOException) e0; S1 }
  else if (e0 instanceof InterruptedException)
   // Just pass it on
   { throw (InterruptedException) e0; }
}
```


Instanceof: When You Should Use It

- **Ans:** almost never. Don't use without a really good reason (and that's rare).
- Instead, organize programs so that calls to overridden instance methods provide all the data dependence that's needed.
- For example, don't need to say

```
if (x instanceof House)
  System.out.print (((House) x).toString ());
else if (x instanceof Car)
  System.out.print (((Car) x).toString ());
  ...
```

  but rather just `System.out.print (x.toString ())`, because `toString` defined on all `Objects`.

- Likewise, you should arrange your types the same way, so that the type of variables used like `x` have some common interface, and the variable behaviors are all in the methods.
- In project, therefore, don't push `Doubles` and `Quantities` on the stack; use `Quantity` alone, or arrange all values on the stack implement a certain interface, which in turn defines all needed methods.

New Subject: Reflection

- [This material mostly for fun.]
- Java has *reflection:* programs can "look at themselves."
- E.g., objects of type `java.lang.Class` represent Java types.
- For any non-null reference value, `x`, `x.getClass()` returns `Class` object that stands for `x`'s dynamic type.
- `Class` objects are *not* types, they simply stand for (*reflect*) them. *Can't* write

```
x.getClass() y;   // WRONG
```

  to declare a local variable `y` whose type is the same as `x`'s dynamic type.

- Here's are things you *can* do:

```
/* Print the name of x's dynamic type. */
System.out.print (x.getClass (). getName ());
/* Get a Class for the type name typed in by the user. */
input.nextToken ();
Class userClass = Class.forName (input.sval);
/* Create a new one */
Object anObj = userClass.newInstance ();
```