CS61B Lecture #13

Administrative:

- 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: Scope rules (final java lecture, for now)

Readings for Today: *Programming Into Java*, §5.1

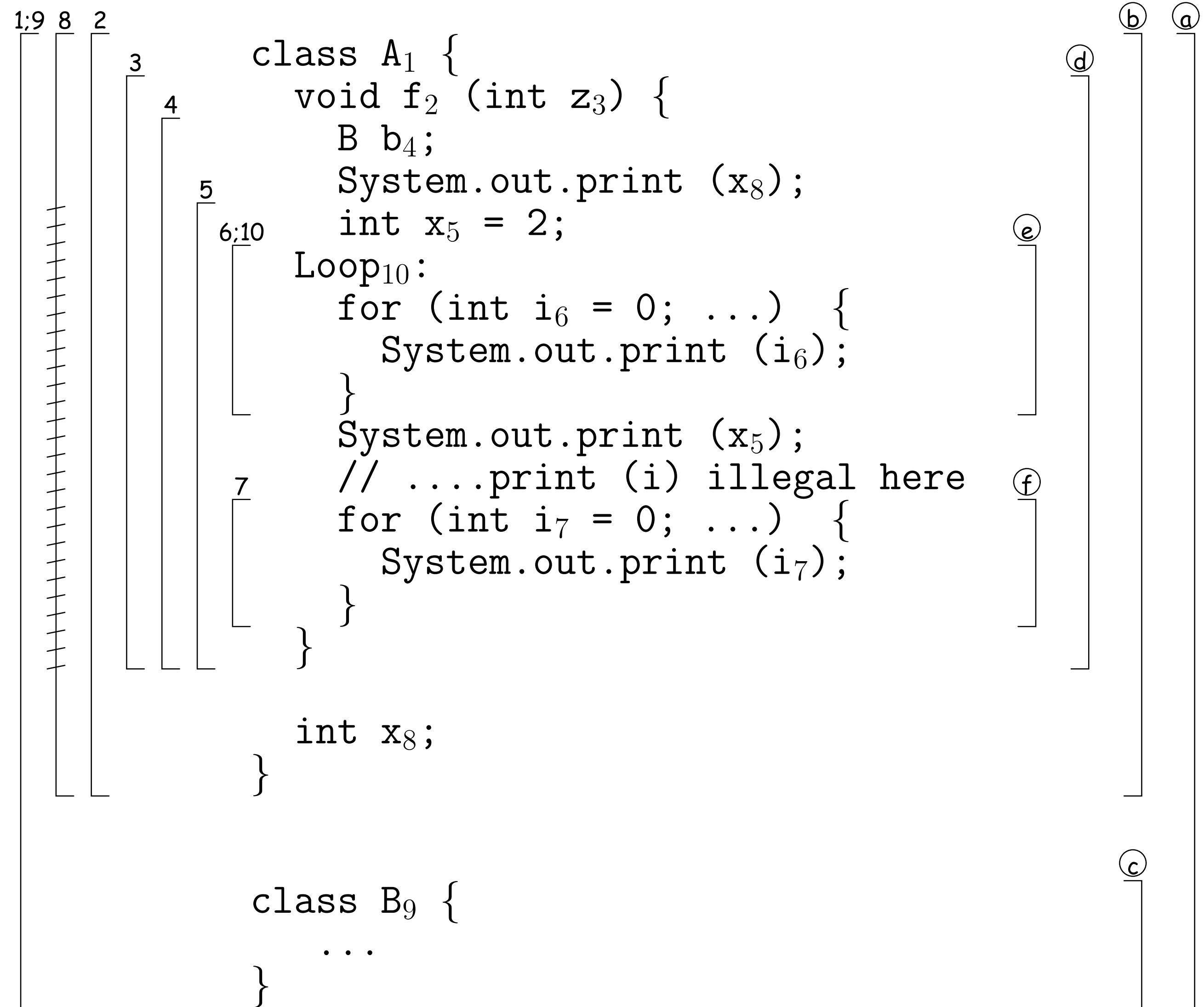
Readings for next Topic: *Data Structures (Into Java)*, Chapter 1; Goodrich and Tamassia, Chapter 3.

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

- **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.

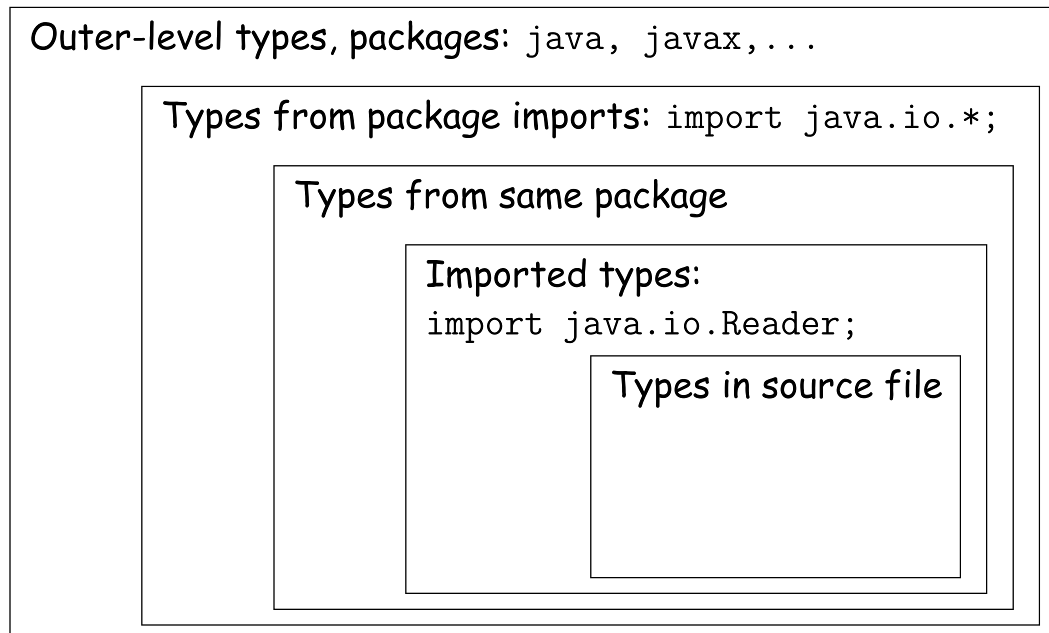


Scope of Type Names

- 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.Stack
import java.io.*;       // Can leave off java.io from all its types
import java.lang.*;     // Automatically added to all programs
```

Has no effect except to define shorthand.



Extent

- Scope and extent orthogonal concepts. Containers can exist even when out of scope: e.g., local variable 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: 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.
- Extent of local variable $\neq$ 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:

```
Collector appender (final StringBuffer buffer) {
  return new Collector () { // See Lecture #10
           public void collect (String s) { buffer.append (s); }
           public Object value () { return buffer; } }; }
```

- In

```
StringBuffer myBuffer = new StringBuffer ();
Collector myCollector = appender (myBuffer);
myCollector.collect ("Hello");
```

  The parameter **buffer** would normally go away when `appender` returns, but it is still used in later call to `myCollector.collect` (Did this sort of thing all the time in CS61A).

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

For Fun (Mostly): Reflection

- 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 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);
  /* ... and create a new instance of it (if class has default constructor) */
  Object anObj = userClass.newInstance ();
  ```