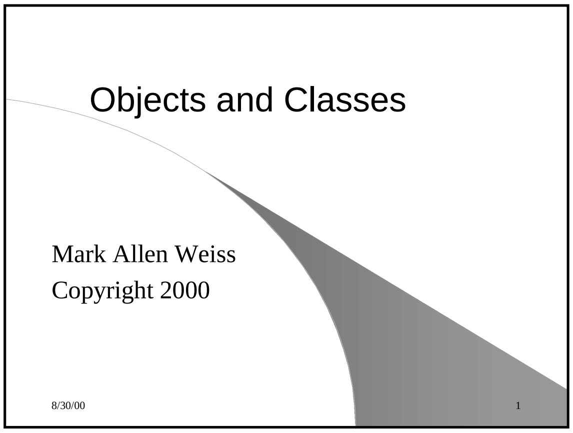
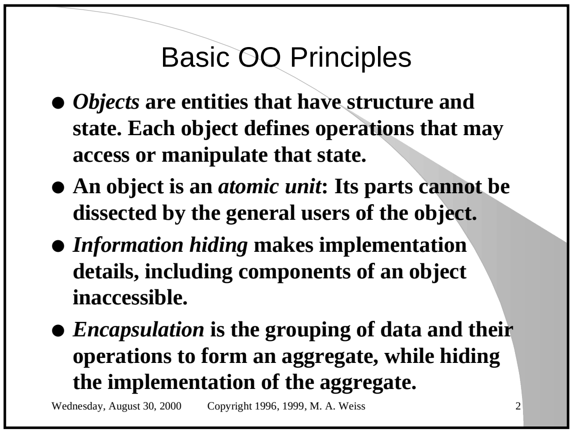


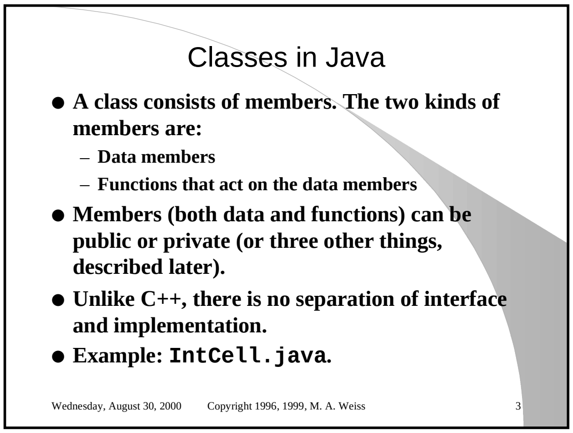
Objects and Classes

Mark Allen Weiss
Copyright 2000



Basic OO Principles

- ***Objects*** **are entities that have structure and state. Each object defines operations that may access or manipulate that state.**
- **An object is an *atomic unit*: Its parts cannot be dissected by the general users of the object.**
- ***Information hiding*** **makes implementation details, including components of an object inaccessible.**
- ***Encapsulation*** **is the grouping of data and their operations to form an aggregate, while hiding the implementation of the aggregate.**



Classes in Java

- **A class consists of members. The two kinds of members are:**
  - **Data members**
  - **Functions that act on the data members**
- **Members (both data and functions) can be public or private (or three other things, described later).**
- **Unlike C++, there is no separation of interface and implementation.**
- **Example: `IntCell.java`.**

Using an Object

- Objects are always accessed by referenced variables.
- Objects are defined by using new. This is (more or less) the only way! Example:

```
    IntCell m = new IntCell( );
```

- Note parentheses (different than old C++).
- Some objects are defined with additional parameters; this is controlled by the constructor(s) for the object (as in C++).
- = for objects is a reference assignment.

Applying Methods

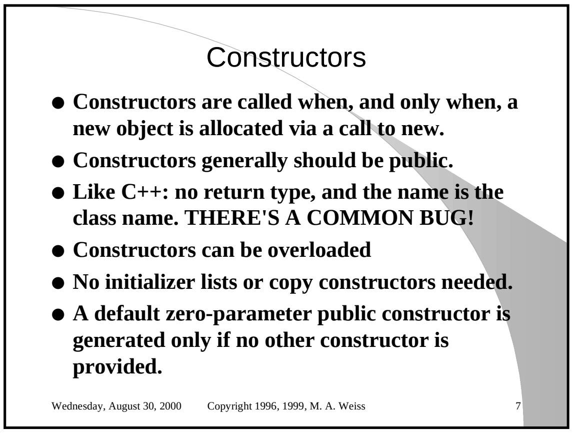
- A *method* is a class function that is applied to an object. (The C++ term is member function).
- Use the . operator to select a member:

```
    m.write( 5 );
    int n = m.read( );
```

- Private methods may not be selected by a method from another class. Public methods may be selected from anywhere. (The default, if you don’t specify public or private, is somewhere between public and private).

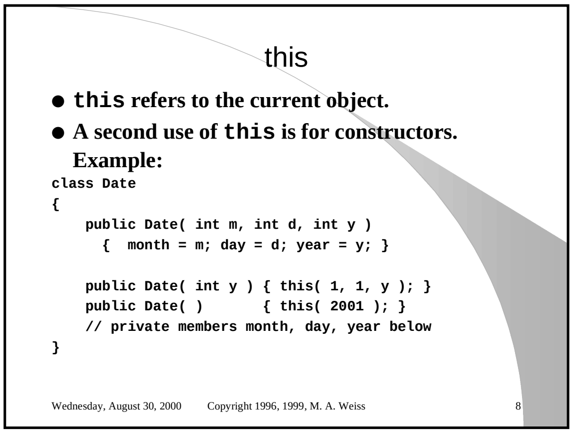
Initialization of Fields

- Fields can be initialized inline
- Can use obscure initializer block
- Can use constructors
- If none of these are done, fields will get defaults:
  - 0 for primitives
  - false for boolean
  - ‘\0’ for char
  - null for references



Constructors

- Constructors are called when, and only when, a new object is allocated via a call to new.
- Constructors generally should be public.
- Like C++: no return type, and the name is the class name. THERE'S A COMMON BUG!
- Constructors can be overloaded
- No initializer lists or copy constructors needed.
- A default zero-parameter public constructor is generated only if no other constructor is provided.

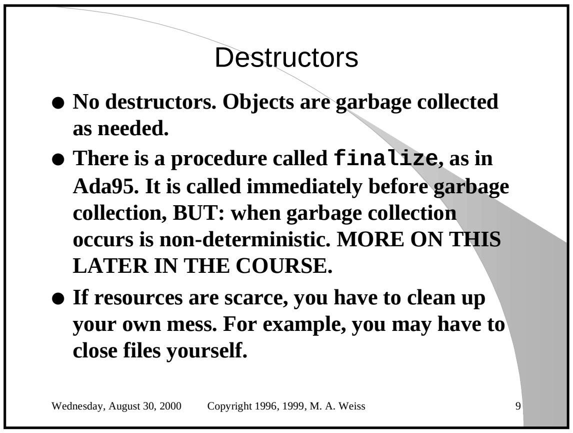


this

- `this` refers to the current object.
- A second use of `this` is for constructors. Example:

```
class Date
{
    public Date( int m, int d, int y )
      {  month = m; day = d; year = y; }

    public Date( int y ) { this( 1, 1, y ); }
    public Date( )       { this( 2001 ); }
    // private members month, day, year below
}
```



Destructors

- No destructors. Objects are garbage collected as needed.
- There is a procedure called `finalize`, as in Ada95. It is called immediately before garbage collection, BUT: when garbage collection occurs is non-deterministic. MORE ON THIS LATER IN THE COURSE.
- If resources are scarce, you have to clean up your own mess. For example, you may have to close files yourself.

Constant Things

- No constant member functions. Everything may alter the object.
- Java conventions:
  - `getMember`: an accessor
  - `setMember`: a mutator
- Instance fields can also be marked as final.
  - Must set value by end of all paths through all constructors
  - Cannot change value after constructor call

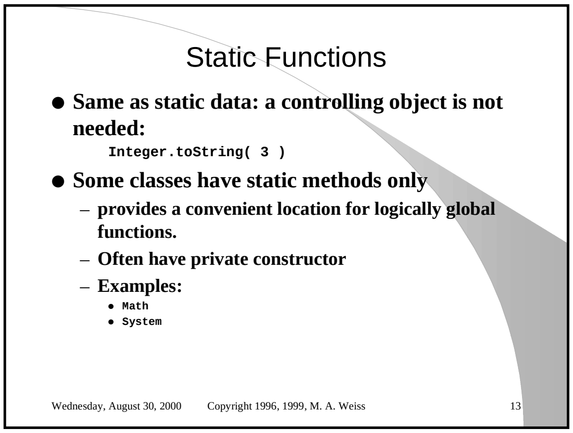
Class-Wide Things: static Members

- Like C++
- A static member (either data or function) applies to the class, rather than a particular instance of the class.
- In the example below, each Junk object has its own `x`. But there is only one shared `y`.

```
class Junk
{
    private int x;
    static private int y;
}
```

Initialization of Static Data

- Static data is initialized once, when class is first loaded (prior to creation of any objects of the class type).
- Cannot try to initialize static data in constructors -- too late; may not even be allowed to call constructors.
- Initialize fields either
  - inline when declared (if simple)
  - in static initializer (if complex)

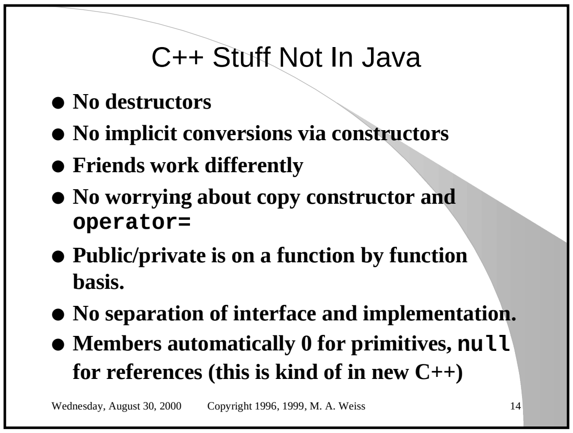


Static Functions

- Same as static data: a controlling object is not needed:

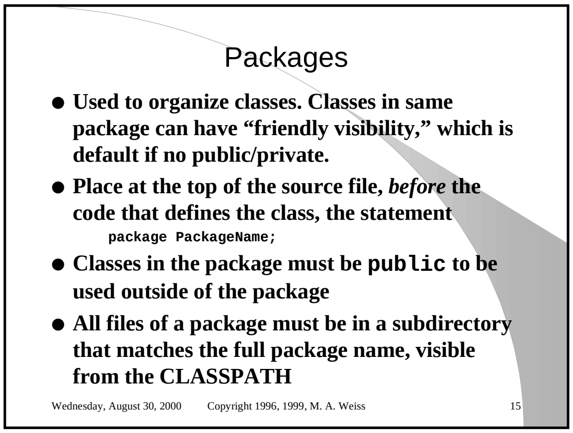
```
Integer.toString( 3 )
```

- Some classes have static methods only
  - provides a convenient location for logically global functions.
  - Often have private constructor
  - Examples:
    - `Math`
    - `System`



C++ Stuff Not In Java

- No destructors
- No implicit conversions via constructors
- Friends work differently
- No worrying about copy constructor and `operator=`
- Public/private is on a function by function basis.
- No separation of interface and implementation.
- Members automatically 0 for primitives, `null` for references (this is kind of in new C++)



Packages

- Used to organize classes. Classes in same package can have “friendly visibility,” which is default if no public/private.
- Place at the top of the source file, *before* the code that defines the class, the statement

```
package PackageName;
```

- Classes in the package must be `public` to be used outside of the package
- All files of a package must be in a subdirectory that matches the full package name, visible from the CLASSPATH

Using Packages

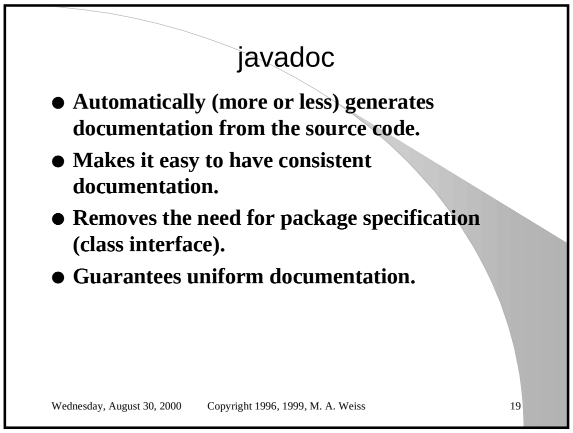
- **Use the import statement to use a package.**
- **Packages are searched for in directories that are branched off any directory named in the `CLASSPATH` variable.**
- **`CLASSPATH` almost always includes `.`, so:**
  - **In the main directory that you will work in,**
    - **Create a subdirectory that will store the various classes in the package**
    - **Place test programs in the main directory**
    - **Have the test programs import the package if you want the shorthand**
    - **If you change main directories, modify `CLASSPATH`.**

Package Visibility

- **Default visibility is package visible**
- **Packages are open-ended; anyone can join**
- **Package visibility is insecure and should be avoided if possible**

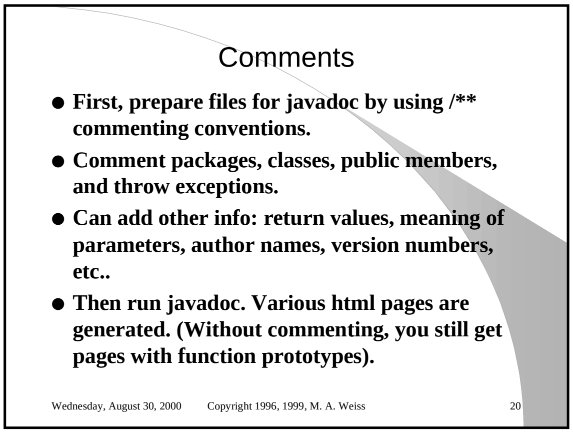
Import Directives

- **Allows class name to be used as a shorthand for the complete class name (that includes the package)**
- **Two forms:**
  - **import java.io.FileInputStream; // One shorthand**
  - **import java.io.*; // Lots of shorthands**
- **Packages do not include each other; neither do wildcard imports**



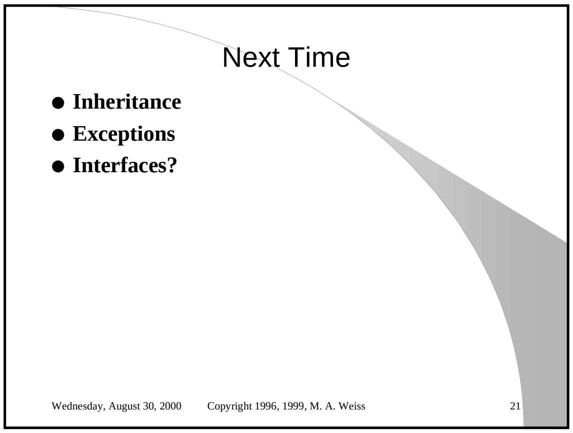
javadoc

- Automatically (more or less) generates documentation from the source code.
- Makes it easy to have consistent documentation.
- Removes the need for package specification (class interface).
- Guarantees uniform documentation.



Comments

- First, prepare files for javadoc by using /** commenting conventions.
- Comment packages, classes, public members, and throw exceptions.
- Can add other info: return values, meaning of parameters, author names, version numbers, etc..
- Then run javadoc. Various html pages are generated. (Without commenting, you still get pages with function prototypes).



Next Time

- Inheritance
- Exceptions
- Interfaces?