Objects &
Object Oriented Programming

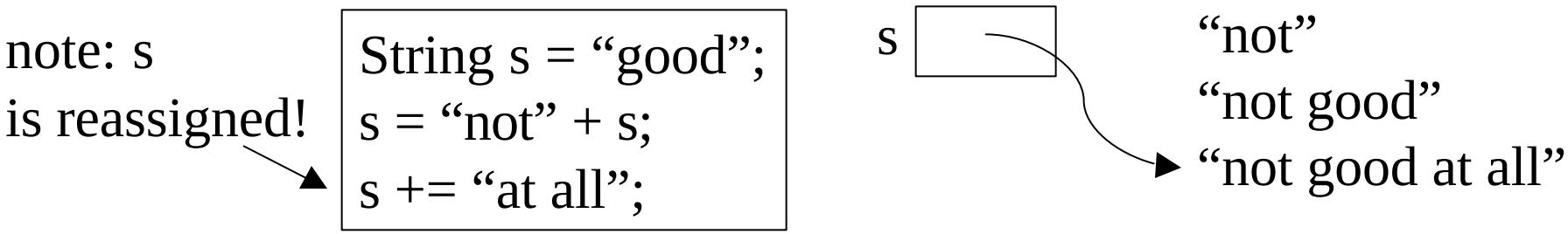
Weiss ch. 3

String

- Like any object, but:
  - has a special way of creating new ones: "…"
    - "abc" creates a new String, returns reference to it
  - has a special operator for concatenation: +
- Immutable
  - All of it's *fields* are declared *private*, so are inaccessible to users
  - None of its methods ever change the object
  - No one can create additional methods for String
  - Thought exercise: why is this important?

String Concatenation

- If either lhs or rhs is a (reference to a) String, then '+' means string concatenation; also use "+="
- String concatenation creates a new string object and returns a reference to it



- If one side not a String, it is converted:
  - e.g., "abc" + 5 → "abc5"
    1 + (2 + "3") → "123"

String Comparison

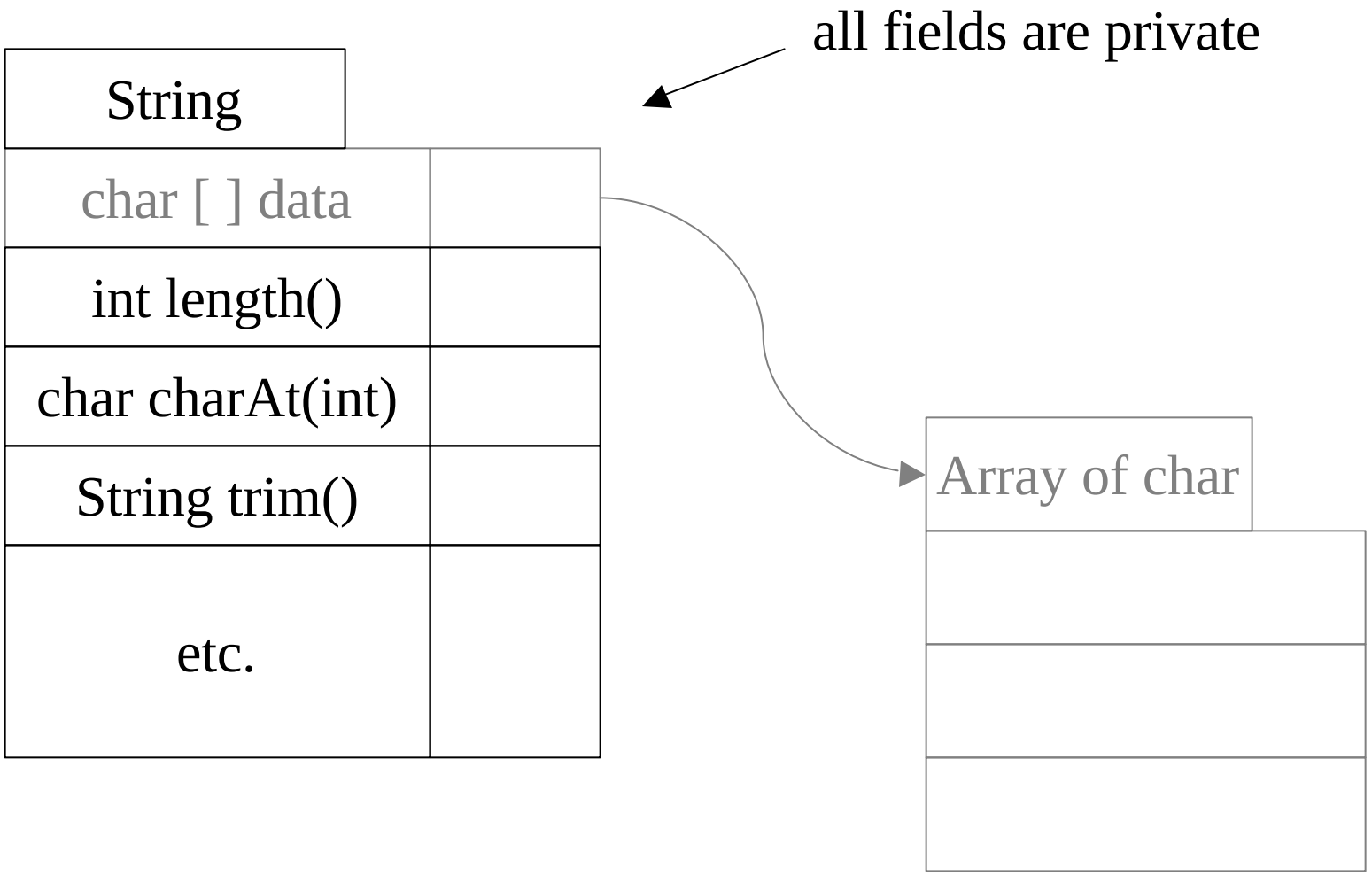
- Operators “==“ and “!=“ compare references
  - Rarely useful for strings
  - Usually want to compare contents
  - So use equals() instead

```
String s1 = “Hello”;
if (s1 == “Hello”) System.out.println(“oops”);
if (s1.equals(“Hello”) System.out.println(“better”);
if (“Hello”.equals(s1)) System.out.println(“odd, but okay”);
```

Other String Methods

- Other than equals(), there are many
  - Look in Java API (on web)
- Inspecting a string:
  - “Hello”.length() → returns int *5*
  - “Hello”.charAt(1) → returns char ‘e’
  - “Hello”.indexOf(‘o’) → returns int *4*
  - “Hello”.endsWith(“lo”) → returns boolean *true*
- Modifying a string (hey, aren’t they immutable!):
  - “cs211”.toUpperCase() → returns String “CS211”
  - “cs211”.replace(‘1’, ‘0’) → returns String “cs200”
  - “ cs211 “.trim() → returns String “cs211”

String Internals



Arrays

- Read Weiss sec. 2.4
- Arrays are objects (sort of):

```
int [ ] a = new int[5]; // initially, all zero
a[0] = 123;
a[4] = a.length;
```

- Arrays can hold references as well as primitives:

```
String [ ] staff = new String[5]; // initially, all null
staff[0] = “Kevin”; // new object, save reference in array
String [ ] people = staff; // alias to same array!
people[1] = “Eli”; // new object, save reference in array
```

Object Oriented Programming (OOP)

- Multiple Goals:
  - Modularity
  - Encapsulation
  - Information Hiding
  - Protection
  - Abstraction
  - Code Reuse
  - Genericity

First example: Internet IP Addresses

- IP address are typically 32-bits long, and usually represented as four 8-bit numbers:
  - 192.168.0.1 = = 0x…. =
- A program using IP addresses often has to:
  - look up the IP address(es) for a given host name
    e.g. www.google.com → 64.233.161.99, 64.233.161.104
  - look up IP address for the local machine
  - discover name for a given IP address
    e.g 128.84.154.170 → webclu-a.cs.cornell.edu
  - Check properties of an address (e.g., local, remote, "multicast", etc.)
- As system designers, what shall we do about this?

Requirements

- Code must be reusable & flexible:
  - Many programs will need to do the same operations
  - Each program is unique
- Non-solution #1: write all programs ourselves, from scratch each time
  - duplicates work; need to learn too many details each time
- Non-solution #2: provide a “specification”, and have each programmer follow the specification
  - same problems
- Non-solution #3: provide example code, and have each programmer copy-and-paste, the modify to fit
  - think: cs211 examples on course web site
  - same problems; bugs get copied too

A Traditional Approach

- Provide a mix of methods and documentation:
- Example, many unix systems:

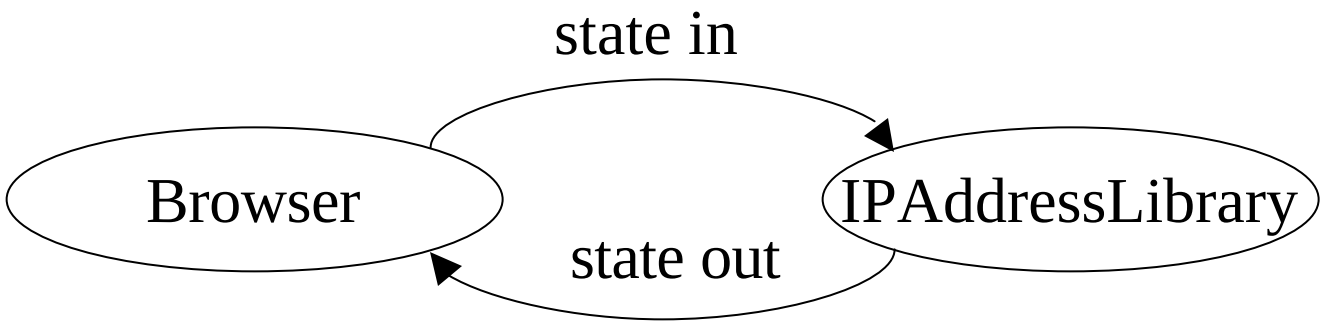
```
int inet_addr(String dots); // convert from “dots”
int inet_network(int); // extract network part
String inet_ntoa(int); // convert to “dots”
int inet_lnaof(int); // extract local part
int inet_netof(int); // extract network part
AddressInfo gethostbyname(String name);
AddressInfo gethostbyname2(String name, int type);
AddressInfo gethostbyname_r(String name);
AddressInfo gethostbyname2_r(String name, int type);
```

- Problems:
  - No organization: live in many files, no consistency, duplication
  - Much of this is historical baggage
- Java: Put everything related into a class!

- Choice
- We will provide a class “library” of code to deal with ip addresses
- But where will *state* be stored?
  - *state* in this case is the data in the ip address
  - stack
  - static area
  - heap

Solution #1: State on the Stack

- IP Address is a method parameter



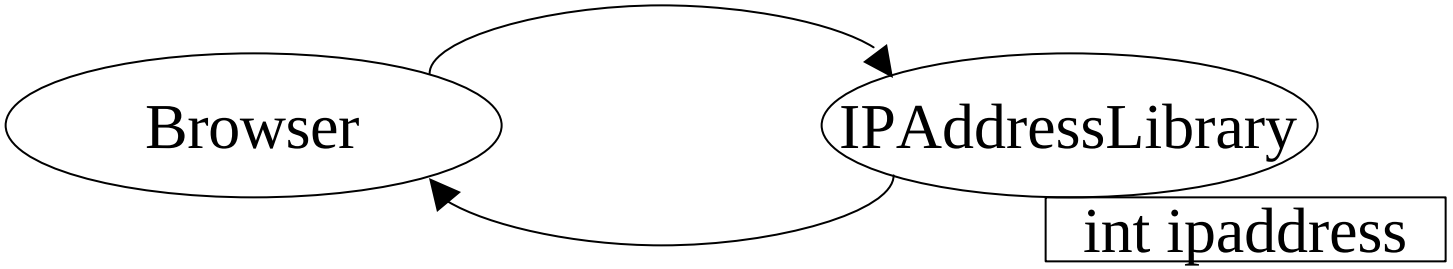

```
public class IPAddressLibrary {
  public static int convertFromDots(String s) { … }
  public static String convertToDots(int ipaddress) { … }
  public static int[ ] lookupHost(String name) { … }
  public static int networkPart(int ipaddress) { … }
  public static int localPart(int ipaddress) { … }
}
```

Solution #1: Critique

- Provides some *modularity*
  - Program split into client + library (no “inet_” prefix either)
  - All (most) code for ip addresses in a single place
- State representation is fixed
  - We used an *int* to represent state, and can never change it without rewriting all other applications
    - e.g., Maybe a 64-bit *long* would be helpful (and IPv6 needs 128 bits!)
- Clumsy and inefficient
  - Passing an *int* is fine, but what if state is larger, more complex?
- “Client” holds the state
  - They can (and will) mess it up: e.g., what does ipaddress++ do?
    - Accidental, or malicious
  - They can create their own addresses
    - “forbidden addresses” are hard to implement

Solution #2: State in Static Area

- IP Address is a *private* static variable



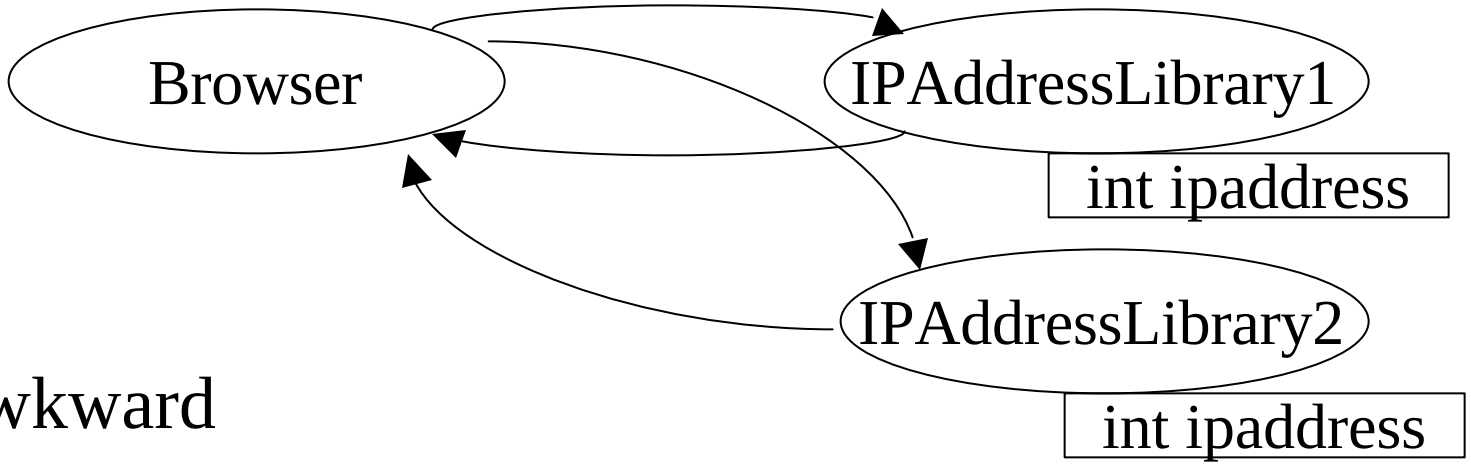

```
public class IPAddressLibrary {
  private static int ipaddress;

  public static void convertFromDots(String s);
  public static String convertToDots();
  public static void lookupHost(String name);
  public static int networkPart();
  public static int localPart();
}
```

Solution #2: Critique

- Provides *data abstraction*
  - Still used an *int* to represent state, but client doesn't know or care
  - We can change to *long* later, or 128 bits, and client doesn't change
- Provides *modularity*
  - Program is split in two pieces: client + library
- Provides some *protection*
  - e.g., could check for "forbidden address" in a single place
- But, only a single ip address can be used at once
  - One static variable, so only one address can be stored

Solution #2+: Multiple Addresses

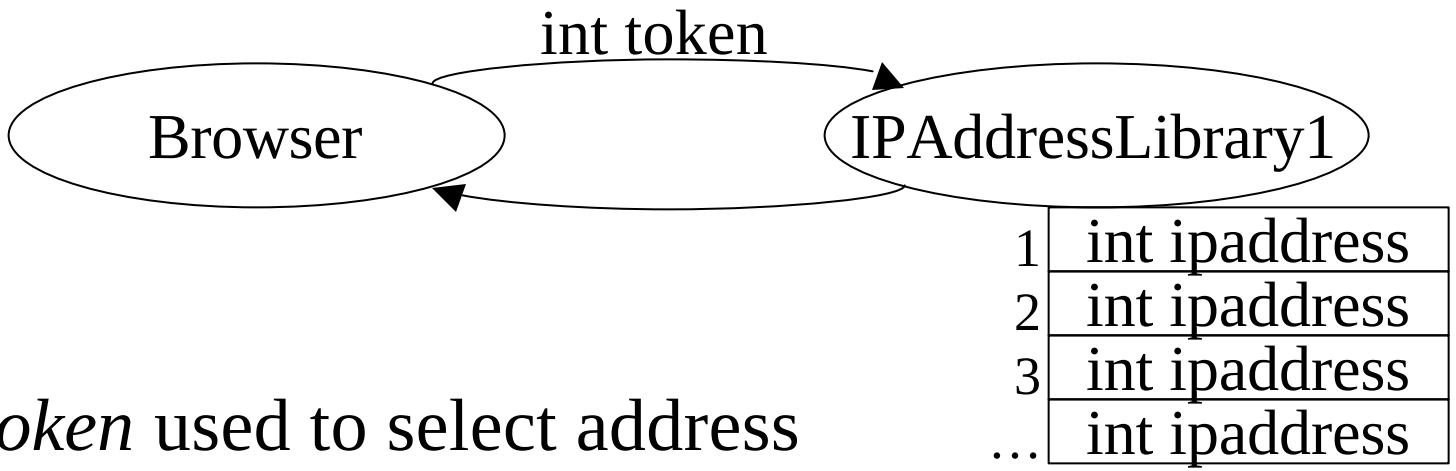
- Trick: Copy library under different names



- Awkward
- Size of program increases
- Hard to maintain as changes are introduced
- Difficult for client to use (the *code* changes, depending on which copy is being used)
- Must decide beforehand how many addresses

Solution #2++: Multiple Addresses

- Trick: Maintain a table of addresses



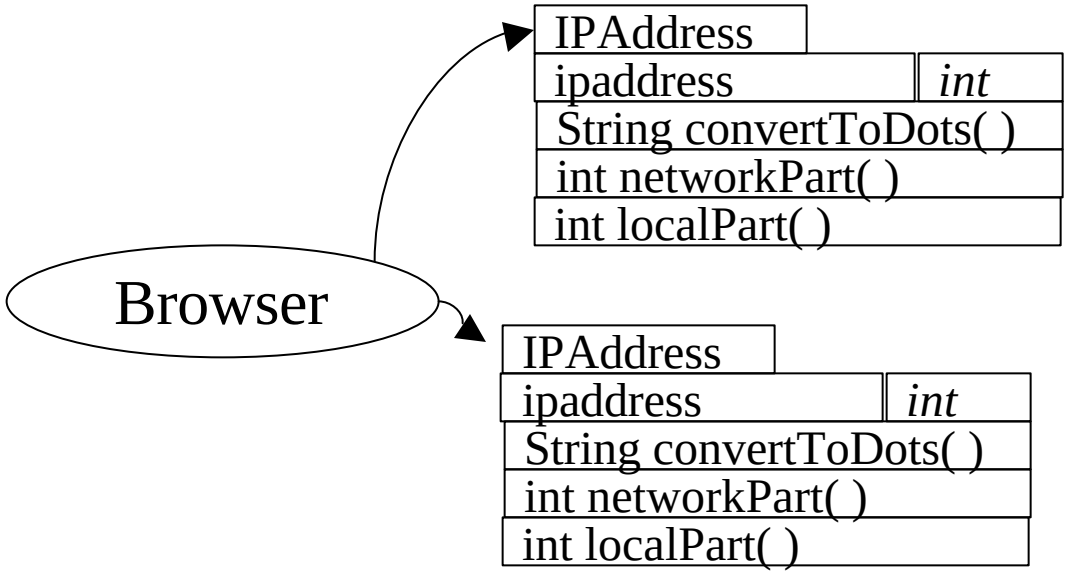
- a *token* used to select address
- a very often-used trick in traditional libraries
- client can (and will) mess things up:
  - if *int ipaddress* is a token, then *ipaddress*++ means something!
- hard to determine when to reclaim a slot in the table

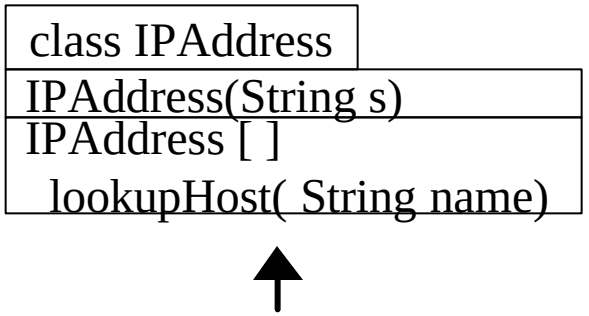
Goals

- *modularity*
  - Code for IP address separate from browser
  - Each can change independent of the other
- *data abstraction*
  - client does not need to know about state representation, just the operations available
- *information hiding*
  - client is not allowed to fiddle with internal state representation
- *protection*
  - library should be able to enforce various policies concerning its data
- multiple objects (IP addresses)
  - Want a “name” or token to refer to each IP address
  - Token itself should not contain any state
  - Client should not be allowed to manipulate tokens

One Solution: Object Oriented Programming

- A *class* is a template for making *objects*
  - each instance stores the state for a single IP Address
  - references used as the name for each object
- Class definition specifies:
  - static methods & static variables: methods & state that are not specific to a particular object (IP address)
  - constructors: methods that initialize newly-created objects
  - instance methods: methods that operate on a single object
  - instance variables: state associated with a single object





(only *ONE* copy of the static members and constructors)

```
public class IPAddress {
  private int ipaddress; // private instance variable

  public IPAddress(String s) { ... }; // construct a new address given dot notation
  public String convertToDots(); // convert instance into dot notation

  public static IPAddress [] lookupHost(String name); // get instances for a name

  public int networkPart(); // get network part of an instance
  public int localPart(); // get local part of an instance
}
```

Encapsulation

- (A fundemental concept in OOP)
- An object *encapsulates* state + code for a data type
  - state is mostly invisible to outside world
  - all operations grouped into single place, along with the data that is operated upon
  - static variables & methods used to share data and functionality between objects, if necessary