Bridging the gap

- We've finished bridging the gap from C to Java.
- Time to start thinking in object-oriented terms.
- Don't want to wind up writing bad C with Java syntax.

Objects have state

- We've said (and seen) that objects have both *state* and *behavior.*

- State refers to the values of the object's internal variables.
  - These variables are called *fields* or *instance variables.*
  - An object is an *instance* of a class, but different objects of the same class can have different values for their fields.

- For example, consider objects of the `Friends` class. They might have instance variables like:
  - `String firstName;`
  - `String lastName;`
  - … and so on

Objects have behavior

- The *behavior* of an object is represented by its internal functions, which we call *methods*.

- For example, objects of the `Friends` class might have methods like:
  - `void bestFood();`

- We've already seen many examples, like the `nextInt()` method of `Scanner`, or the `length()` method of `String`, and so on.

Method examples

- To use a method, you use the dot operator with the `object` you're calling the method on.

- `int len = str.length();`
- `float randFloat = randGen.nextFloat();`
- `int cmp = someString.compareTo(otherString);`

- Methods have zero or more parameters.

Constructors

- A constructor is a special method that is used to create an object.
- Remember how a class is a blueprint for constructing an object?
  - That blueprint is what the constructor uses.
- Java handles low-level matters like memory allocation.
- So the constructor's job is really to initialize the instance variables (fields).
  - In other words, to set the initial state of the object.

Examples

- We've seen some examples of *calling* constructors already:
  - `Scanner sc = new Scanner(System.in);`
  - `Random randGen = new Random();`

- Constructors (unlike normal methods) are only invoked using the `new` operator.

- Like normal methods, they can take zero or more parameters.
  - Typically those parameters are used to initialize fields.
  - E.g., the `Scanner` constructor tells it where to scan from, a fact that is part of the state of that `Scanner.`

State and behavior

- Behavior is usually somewhat dependent on state.
- For example, we might have two friends objects, `friend1` and `friend2`.
- This fact is expressed in their *state*, so some variable(s) are going to reflect this fact.

State and behavior

- Basically, objects of the same class have similar behavior (all the same methods).
- But the *effect* of those methods depends on the state of the particular object they're invoked on.

Example

- Let's code up the Friends class. Pay attention to:
  - Fields
  - Methods
  - Constructors

- Notice that the fields of the class are a little like global variables – but only for that class.

- Methods and constructors can access those variables freely – they don't have to be passed as function parameters.

The Inside and Outside View

- When you code up a class, think as if you're *inside* an object of it.
  - After all, you're setting up its internal structure.
  - Notice how you can call its methods and use fields *without* the dot operator.

- When using an object of the class, you think from the *outside*.
  - You basically have this variable whose methods you can call.
  - Always need to use the dot operator.

- From the outside, the internal details are not available. They are hidden away.
  - Basically, "none of your business".

Information Hiding

- A core principle of OOP design is *information hiding*.

- As an example, how does a String store the text it contains?
  - It could have an internal field that is a `char[].`
  - Maybe a more complicated object like an `ArrayList<String>`?

- The internal implementation is not relevant to a client of the class.
  - Client = someone using an object of the class.
  - So basically you, when you use a `String` object.
  - But not you when you're writing the `Friends` class! There you're on the inside.

The static context

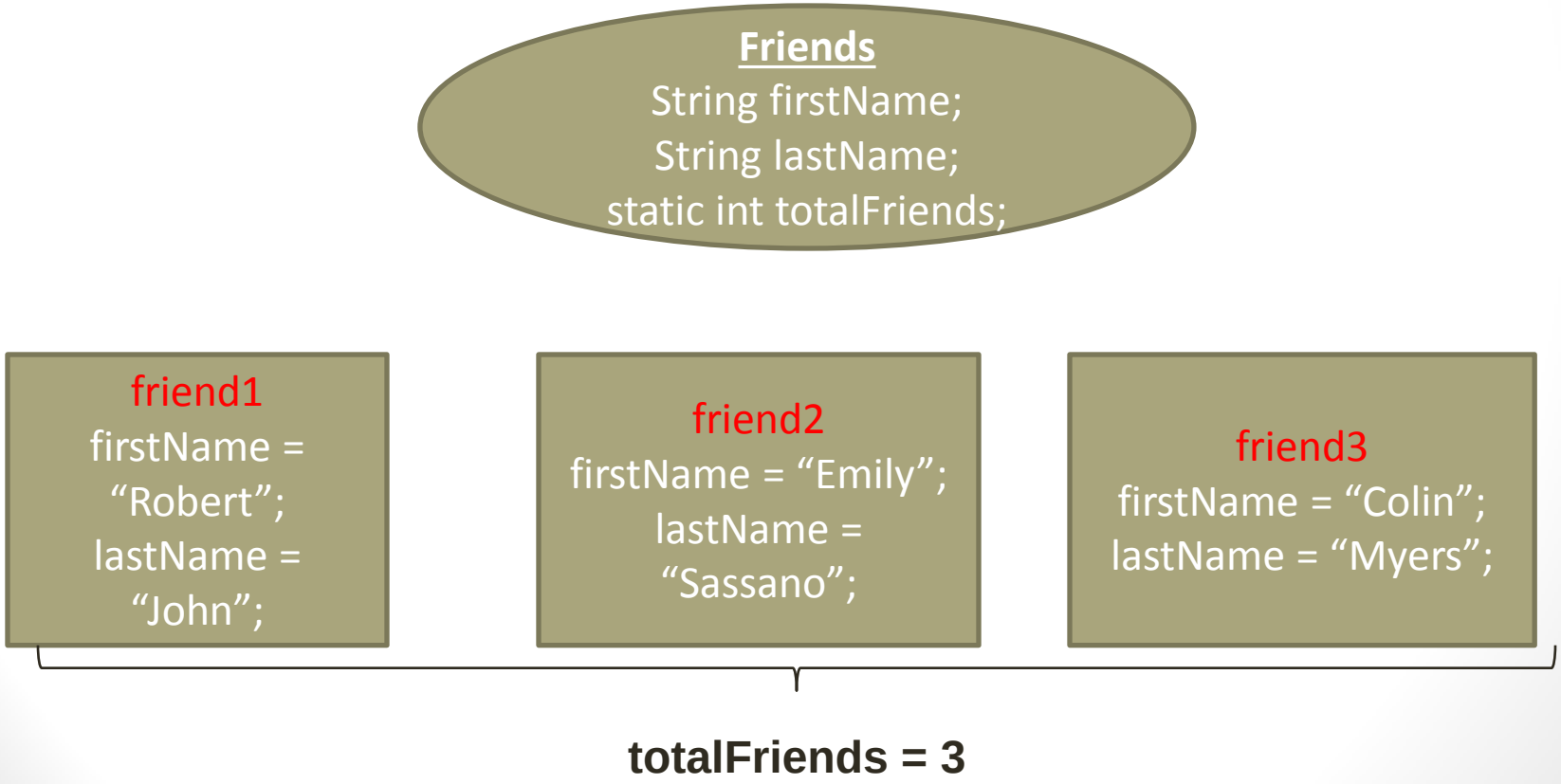
- While main() is technically part of the object it's in, in a weird way it isn't.
- Class members (fields and methods) are allowed to be `static`.
- Their behavior is somewhat different from normal fields.

Static fields

- Say Friends has a static field totalFriends (an int).
  - It counts all the Friends objects we have created.
- A count like this doesn't quite fit into the role of an instance variable.
  - It's not really part of the state of a Friends object.
  - It expresses something about *all* Friends.
- So it should really belong to the *class*, not the object!
- The entire class Friends has one single totalFriends field, even though there might be dozens of Friends objects.

Shared

- Equivalently, the totalFriends field is *shared* by all Friends objects.



Static methods

- Methods can be marked static too.
- Static methods are special, because they can *only* manipulate static fields of the class.
  - Instance variables CANNOT be touched.
  - They can create and use local variables, of course.
- Use static methods for functionality that doesn't affect or use the state of the object.
- Equivalently, functionality owned by the class, rather than an object.

Instance Methods

- This is just what we call all non-static methods.
- Instance methods can access ***all*** methods and fields of a class, **including the static ones**.
- It's only static methods that are limited.
  - We say that they're a 'static context'. In a static context, only static members can be used.

The Math class

- The Math class is a good example. All its methods are static.

- Its functions operate on their arguments, and nothing else.
  - Well, sometimes they use constants like *e* and π, but those are static fields of the Math class.

Using static members

- Instead of using the object name, we use the *class name* with a dot.

- `double rounded = Math.round(3.47);`
- `double area = Math.PI*r*r;`
- `int peopleCount = Friends.totalFriends + Family.totalFamily;`

- You *can* actually use an object name, but it makes no difference. Using the class name makes it clearer.

Java naming conventions

- There are some basic naming conventions that are pretty much universal across all Java programs.

- Class names always start with an uppercase letter: `String`, `Scanner, Friends, PrintStream`, etc.
  - Note that CamelCasing gets used for names that are multiple words.

- Variable and method names start with a lowercase letter, and are camel-cased too.
  - `blah, hello, thisIsACamelCasedVariableName`, etc.

Underscores

- Usually we prefer not to use underscores anywhere.

- The one exception is for fields that are constants.
  - All uppercase, words separated by underscores.

- `final int DAYS_IN_YEAR = 365;`
- `final String GREETING = “Hello World”;`

The final keyword

- Use it when the variable is declared.
  - `final int NUM_DAYS_WEEK = 7;`
  - `final double NO_VALUE_YET;`

- Such a variable can only have a value assigned to it ONCE.
  - Notice how our second example was declared, but not yet assigned.
  - You can assign a value to it once, but never after that point.

- All subsequent modifications are compiler errors.

Final with Objects

- Suppose we mark an object final.
  - `final Friends constantFriend = new Friends("Robert", "John");`
- The *state* (values of instance variables) can still be modified!
  - `constantFriend.setLastName("Jose"); // Changes the name.`
- What has been fixed is the *object* that the variable `constantFriend` refers to. (Jargon: It's an *immutable reference*)
- So we can never do:
  - `constantFriend = friend1;`
  - `constantFriend = new Friends();`

Final with Arrays

- Just like with objects.
  - Can modify *elements* of the array.
  - But the variable is bound to that array permanently.

- `final String[] words = {“nitwit”, “oddment”, “blubber”, “tweak”};`

- `words[2] = “dandelion”;  // Perfectly fine.`

- `words = new String[] {“A”, “new”, “array”}; // Illegal.`

Static final constants

- If you have a field that's going to be constant, chances are it's probably static too.
  - E.g., suppose you write a `Calendar` class.
  - `DAYS_IN_WEEK` is not going to vary from one `Calendar` object to the next.

- So mark it as both static and final.
- `static final int DAYS_IN_WEEK = 7;`

- You HAVE to initialize these when you declare them.

Final method parameters

- You can make method parameters final as well.

```
public void readInput(final Scanner sc) {
  …
}
```

- Makes `sc` an immutable reference.
  - Really makes no difference to any client of the class.
  - But someone modifying your code later will get a compile error if they try to reassign `sc` .

- Of course, they can just remove the `final` keyword then…

final on other stuff

- We can also apply the `final` modifier to a class or method declaration.
- The meaning of this is something quite different.
- We'll deal with that when we cover inheritance.

Methods

- Instance methods can access and modify:
  - Fields (including static ones!)
  - Parameters passed to them
  - Local variables they create
- Static methods can only touch static fields, but the other two cases are the same.
- The usual rules with final variables apply, of course.

The this keyword

- The keyword `this` refers to the current object.
  - From the 'inside view' I mentioned last time, `this` is the object we're inside of.
- It is accessible in instance methods and constructors.
- It is NOT accessible in static methods.
  - Makes no real sense in a static context.
- We can use the dot operator with it, regardless of public/private modifiers on class members.
  - Remember – those are only important for the outside view.

Method Overloading

- We can have multiple methods with the same name.
- These methods are said to be *overloaded*, because one name does multiple things.
- Constructors, being methods, can also be overloaded.
- Example: The substring method of the String class.
  - `String substring(int beginIndex);`
  - `String substring(int beginIndex, int endIndex);`
- Similarly, String has 7 static methods named `valueOf()`.

Method Signatures

- How does the compiler know which method you mean?
- Overloaded methods must have different *signatures*.
- Signature = name + types of parameters.
- `String substring(int beginIndex);`
  - Signature: `substring(int);`
- `String substring(int beginIndex, int endIndex);`
  - Signature: `substring(int, int)`

Method Signatures

- Parameter names are irrelevant to the signature.
- More crucially, the return type of the function is not part of the signature!
- So a class CANNOT have methods like this:
  - `String foo(int a, Scanner b);`
  - `char foo(int a, Scanner b);`
- The return types are different, but these two methods 'look the same' to the compiler.

Ordering methods

- In C and C++, we need to place the function definition somewhere before it is called.
  - Alternatively, use function prototypes.

- Java doesn't care about the relative ordering, and has no function prototypes.

The toString() method

- There is an invisible method called toString inside every object.

- It returns a String representing the object.

- Whenever you try to treat an object as a String, the object calls its `toString()` and the result gets used.
  - This happens if you try to print an object, for instance.

Overriding toString

- The default implementation of toString() is pretty useless.
  - Except for Strings, which just return themselves.

- So we *override* the default version by writing our own.

```
public String toString() {
   …
   …
}
```

- It must have *exactly* this signature, return type, and access modifier.

What should toString output?

- A String that meaningfully describes the object.

- E.g., suppose we wrote a `Point` class for a graphics or geometry library.
  - It's state will likely contain x and y coordinates of the point.
  - So maybe return those in String form, like: `"(3.5, 14.1)"`
  - This is a fairly clear text representation of a `Point` object.

- The `String.format()` method is useful here.

The equals() method

- Just like `toString()`, there is another invisible method called `equals()`.

- It is used to see if two objects are equal.

- Why not just use the == operator?
  - Works for primitives, after all.

- Unfortunately, the Java == operator compares the *references*, not the actual objects

Object equality

- If we have two String variables, a and b, what does a == b mean?

- It checks to see if a and b are the same object.
  - Not the same as them having the same contents.

- To see if they have the same contents, we need to use the test `a.equals(b)` instead.

Object equality

- For example:
  - `String a = “blah”;`
  - `String b = a;`
  - `String c = “blah”;`

- Here `a` and `b` are the same object, while `c` has the same *contents* as them (but is not the same object).

- Use `equals()` to be certain.

Object equality

- `String a = “blah”;`
- `String b = a;`

Object equality

- `String a = “blah”;`
- `String b = a;`
- `String c = “blah”;`

String annoyance

- Annoying fact: Strings (but not other objects) are *interned* to save space, which means that sometimes == will work!
  - Interning strings means internally storing only one copy.
- The JVM makes interning decisions at runtime.
- If you use == to compare Strings, it may work fine one day, and fail the next.
- The compiler doesn't point this out, so it's up to you to remember to use `equals()` for comparing objects.

Access control

- There are four modifiers that control who can access class members (i.e., fields and methods)
  - `public`
  - `private`
  - `protected`
  - default (no modifier written)

- We'll see the `protected` modifier later.

Who sees what?

- The modifiers affect which fields and methods can be accessed by *clients* of the class.
  - Members with `public` visibility can be directly accessed with the dot operator.
  - Members with `private` visibility cannot be directly accessed by clients.

- To the outside world, the private members could be anything (or nothing).
  - Rarely even described in documentation.
  - Part of the *information hiding* design principle.

Getter and setter methods

- We saw th Getter and Setter methods in previous lectures.
- They just allow manipulation (getting and setting the value) of a private variable.
- Gives us control over what the client does to potentially important pieces of the class.
- For example, we can stop them from assigning a negative age, or an age that's too high. (Say we have "int age" in Friends)
  - Just catch those cases in `setAge()` method.

Default access

- If no access modifier is specified, the member has default access (sometimes called *package* access).

- Basically, java groups related classes into packages.
  - A package is really just a glorified folder containing classes (and other packages).
  - There is a naming scheme to make it easy to import packages.

- Members with default access are public to anyone in the same package, and private to everyone else.
  - Fairly convenient (no get/set needed) because these classes often operate together quite closely.
  - Can usually relax information hiding since it's trusted code.

Constructors

- Constructors have some special behaviors of interest.
- What happens if we don't write a constructor?
- The compiler invisibly makes a default no-parameter constructor.
  - It does absolutely nothing.
  - All parameters initialized to default values (zeros and nulls).
- This default will go away if you write any constructors!
  - Have to explicitly write your own no-parameter constructor now.

Visibility

- Constructors are usually public.
  - After all, a client needs to see it to call it and make an object.
- Private constructors are usually used to *prevent* the class from being instantiated.
- For example, `Math` has a private constructor.
  - Does nothing, as far as I know.
- This eliminates the default constructor, and makes the user unable to make a `Math` object.

Visibility

- Bit more useful – constructors with default access.
- Can only be called by classes in the same package.
- Useful to provide ways of making the object that we don't want a client to use.
  - But still convenient for internal use.
- Note that different constructors can have different access modifiers.
  - Can have a special default-access constructor, as well as public ones for the client to use.

Another use for this

- Just like methods, constructors can use the `this` keyword to refer to the current object.

- However, they can also use it to call other constructors.
  - Here `this` gets treated like a function instead.

- Example
  - We'll modify the default Friends constructor to call the more complex one.

- Advantage: Can call private constructors this way.