Recall: object-oriented terminology

<table>
<thead>
<tr>
<th>term</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>interface</td>
<td>what an object can do</td>
</tr>
<tr>
<td>type</td>
<td>a description of an object’s interface</td>
</tr>
<tr>
<td>subtype</td>
<td>a type that extends the interface of another type (its supertype)</td>
</tr>
<tr>
<td>implementation</td>
<td>how an object does its thing</td>
</tr>
<tr>
<td>class</td>
<td>a description of an object’s implementation</td>
</tr>
<tr>
<td>subclass</td>
<td>a class that extends the implementation of another class (its superclass)</td>
</tr>
</tbody>
</table>

Inheritance

Inheritance enables code reuse in two ways:

- Reuse for the benefit of providers: The provider of a new class can define that class by explaining how it is different from an existing class. (This kind of reuse is subclassing.)
- Reuse for the benefit of clients: The client of a type can write code that can be used with multiple implementations of that type. (This kind of reuse is subtyping.)

In most OO languages, when you use inheritance, you define a subclass and a subtype.

**Good programming practice:** use inheritance only if the old class and the new class have an is-a relationship. Otherwise, it’s probably better for the new class to contain a field whose type is the old class.

Declared type vs actual type, in Java

The actual type must match or be a subtype of the declared type. In other words, the actual object must support a superset of the methods described by its type. That way, we can be sure that every method call on the object is valid.

When we compile a program, the type checker looks at the declared type (not the value) of an object to see whether the program’s method calls are legal.

When we run a program, Java uses the actual object (not the declared type of the object) to choose which method to run.

For example, consider the following code:

```java
Dog spot = new FrenchPoodle("Spot", 4, 99);
spot.sayHello();
```

(where FrenchPoodle inherits from Dog). Dog is the declared type, and FrenchPoodle is the actual type. When we compile the program, the type checker will make sure that the Dog type defines a sayHello method. When we run the program, Java will call the version of sayHello that is defined in the FrenchPoodle class.
Memoization

Memoization, like dynamic programming, is an optimization technique that avoids doing redundant work.

Here's how memoization works, assuming we have a function \( f \) that performs some possibly redundant work:

1. Create a empty “cache”, which we’ll use as a map: input \( \rightarrow f(\text{input}) \).
2. Given an input I, use the input as a key in the cache, to see if we’ve already computed the result.
   
   A. If we’ve already computed the result, just return the result from the cache.
   
   B. If we haven’t already computed the result:
   
      a. compute \( f(I) \)
      
      b. store the result in the cache
      
      c. return the result

Note that memoization assumes that \( f \) does not have side effects.

Commentary

- Memoization can save us time, when our code makes multiple calls to a function using the same input.

- Memoization and dynamic programming have benefits and drawbacks, relative to one another. As with all optimization techniques, to decide which (or whether) to use, we often need to combine an understanding of the problem, the context in which we’ll be solving the problem, a theoretical model of the cost, and empirical data on the actual performance.

- For a single invocation of an algorithm (e.g., to make change for 42¢), dynamic programming might be slightly more efficient than memoization, in part because there are fewer total function calls.

- For multiple invocations of an algorithm that provide the same input (e.g., repeatedly making change for 42¢), memoization will probably be more efficient than dynamic programming because dynamic programming recomputes the answer each time.

- If solving one instance of a problem does not require solving instances of all the smaller versions of the problem, memoization might be more efficient than dynamic programming because memoization will solve only those subproblems that are necessary.

Next time: Prolog!