Recall: object-oriented terminology

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>interface</td>
<td><em>what</em> 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><em>how</em> 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>

Recall: 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 existing 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 existing class.

Types in Java

The **declared type** of a variable is the type that appears in the code, to the left of the variable, when it is declared. For example:

```java
Dog buddy; // buddy's declared type is Dog
Animal buddy; // buddy’s declared type is Animal
```

The value of variable can always be used anywhere that type is expected, e.g., as an argument to a function, as a return value, etc.

Subtyping

Subtyping creates an “is-a” relationship: an instance of the subtype “is” an instance of the supertype. Here is how we can create is-a relationships (i.e., subtypes) in Java:

- Implementing an interface establishes an is-a relationship.
- Extending an interface establishes an is-a relationship.
- Extending a class establishes an is-a relationship.

Also, every variable “is an” instance of its declared type.

The “is-a” relationship is **transitive**: if A “is a” B and B “is a” C, then A “is a” C.
Actual types, subtyping, and substitutability

The actual type of a variable 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. As a result, subtyping allows us to substitute an instance of a subtype for an instance of a supertype because we know that the subtype supports all the operations in the supertype. In other words, if A is a subtype of B, then an instance of A can replace an instance of B in any situation that calls for a B. This idea is often referred to as the “Liskov substitution principle”, named after the researcher, Barbara Liskov, who initially introduced it.

Declared type vs actual type, in Java

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.