Abstraction

Mason Vail
Boise State University Computer Science

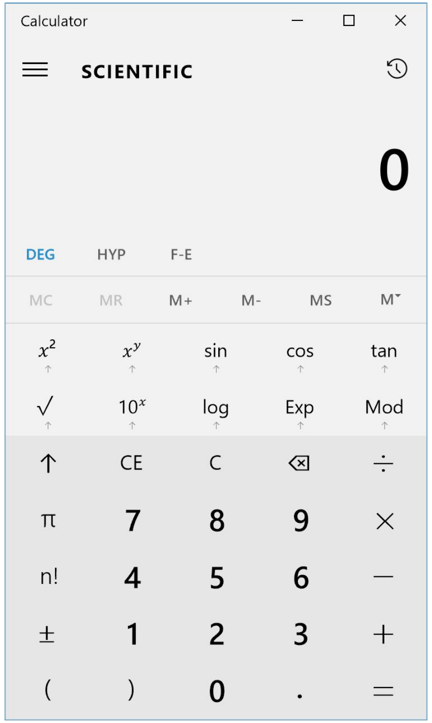
Pillars of Object-Oriented Programming

- Encapsulation
- Inheritance
- Polymorphism
- Abstraction (sometimes)

Real World Abstraction

Calculators have a well known, well defined interface with a consistent core set of arithmetic functions.

It makes no difference to you how the calculator carries out its operations, as long as you get the expected result when you input *1 + 1 =*.



Familiar Abstractions in Java

You’ve likely used many Java classes based only on descriptions of what to expect from their methods. Even if you thought to ask how some methods do their jobs, you didn’t need to know anything about their implementation to use them.

- Math
- String
- Scanner
- System.out
- Random

Object References are Abstractions

An object reference is an abstraction. It isn’t actually the object, but it knows where to find an object and it knows what you’re allowed to do with that object.

Recall that all object references are polymorphic. *Any* compatible object could be at the other end of the object reference.

Only compatible objects can be assigned to a reference, so you have use of all of the reference type’s methods. The actual object at the other end and how it carries out the functionality of the method isn’t particularly important as long as it does the job. You could swap out a different compatible object and you wouldn’t have to change how you use the reference.

Method Arguments Are Abstractions

Formal parameters of methods are object references and, hence, are still abstractions.

Consider the System.out.println(Object o) method.

The Object o argument is as abstract and polymorphic as it can be. Any kind of object will be compatible with this reference.

From the perspective of the method, it makes no difference what kind of object is passed in. All that matters is it needs to fit the Object abstraction, which has toString() as part of its functionality.

Interfaces Define Abstractions Well

Because they specify a set of methods, but have no ability to dictate implementation, Interfaces are ideal for defining an abstraction.

Documentation is critical to explaining the overall mental model and intended functionality of each method, so that expected behavior is understood by both users and those implementing the interface in a class.

Classes implementing an Interface pledge to provide a working embodiment of the abstraction, but the code behind the scenes may work differently in every class.

Abstraction for Good Design and Development

Abstraction boosts productivity. You don’t need to know the inner workings of every class you use. Programmer focus can remain on solving their own problem.

Abstraction allows us to write flexible, modular programs where compatible replacement components can be swapped out without modifying a lot of hardcoded references.

Storage Example

```
/** A Storage contains any number of Objects in no particular
 *  order and returns them, one at a time, as requested.
 */
public interface Storage {

    /** Add one Object to Storage
     *  @param o An Object to store
     */
    public void add( Object o );

    /** Remove and return one Object from Storage
     *  @return an Object from Storage
     */
    public Object removeNext( );
}
```

Storage Example

In a program requiring a `Storage` container, a `Storage` reference can be declared and assigned any object that implements `Storage`. From then on, the `Storage` reference can be used and the actual object type is irrelevant.

Later, if an alternative implementation is desired (more efficient, less memory, better name), the assignment statement can be updated and the rest of the program will continue to function without any changes necessary.

```
Storage storage = new BigIronBucket();
```

becomes

```
Storage storage = new KevlarSack();
```

and all 10,000 references to `storage` throughout the code continue to work.

Abstraction

Mason Vail
Boise State University Computer Science