When to use public, private, and protected class variables and functions?

A good rule of thumb is: make everything as private as possible. This makes your class more encapsulated, and allows for changing the internals of the class without affecting the code using your class.

If you design your class to be inheritable, then carefully choose what may be overridden and accessible from subclasses, and make that protected. But be aware that, as soon as you accept to have subclasses of your class, and there is a protected field or method, this field or method is part of the public API of the class, and may not be changed later without breaking subclasses.

A class that is not intended to be inherited should be made final. You might relax some access rules (private to protected, final to non-final) for the sake of unit-testing, but then document it, and make it clear that although the method is protected, it's not supposed to be overridden.

If you "code to an interface rather than implementation" then it's usually pretty straightforward to make visibility decisions. In general, variables should be private or protected unless you have a good reason to expose them. Use public accessors (getters/setters) instead to limit and regulate access to a class's internals.

To use a car as an analogy, things like speed, gear, and direction would be private instance variables. You don't want the driver to directly manipulate things like air/fuel ratio. Instead, you expose a limited number of actions as public methods. The interface to a car might include methods such as `accelerate()`, `deccelerate()`/`brake()`, `setGear()`, `turnLeft()`, `turnRight()`, etc.

The driver doesn't know nor should he care how these actions are implemented by the car's internals, and exposing that functionality could be dangerous to the driver and others on the road. Hence the good practice of designing a public interface and encapsulating the data behind that interface.

This approach also allows you to alter and improve the implementation of the public methods in your class without breaking the interface's contract with client code. For example, you could improve the `accelerate()` method to be more fuel efficient, yet the usage of that method would remain the same; client code would require no changes but still reap the benefits of your efficiency improvement.