1-

Elegiría la opción dos, ya que con un punto, el alto y el ancho se pueden sacar todos los puntos de un rectángulo, evitando información redundante y facilitando el cambio de posición de todos los demás puntos. También ayuda a que otros desarrolladores no cometan errores al querer cambiar todos los puntos en simultáneo.

2-

Por ley de Demeter, y para evitar el acoplamiento de clases, lo mejor es la segunda opción ya que no acopla a los demás objetos con el “fichero”. Lo mejor siempre es exponer un método para que dicho objeto realice la acción, en vez de exponer la estructura de datos.

3- La primera opción usa el nombre de la clase mediante reflexión, lo cual es una mala práctica ya que no se aprovecha el polimorfismo para resolver el problema y acopla una solución específica a una general, que debería de resolverse por herencia. La segunda opción utiliza strings, que si bien es más eficiente que la reflexión, sigue teniendo los mismos problemas. La tercera hace uso de la herencia pero repite código en ambas subclases. La cuarta sería la opción correcta y no le encuentro ningún error, más allá de que se realiza la extracción mediante un condicional, lo cual podría causar un efecto no deseado al no efectuarse la extracción por falta de dinero.

4-

1. Significa acceder a la variable mediante su referencia, sin intermediarios. Por ejemplo, en una clase   
   ***class MyClass {***

***var x = 10***

***method doSomething() = x \* 2***

***}***

, se podría acceder mediante “x” al valor, directamente.

1. Significa acceder al valor mediante otra referencia, como por ejemplo, un método:  
   ***class MyClass {***

***var x = 10  
 method x() = x***

***method doSomething() = self.x() \* 2***

***}***

1. En el caso de acceso directo a las variables, no permite la implementación de getters y setters ya que si lo hicieras, no sería directo. Sin embargo, esto da la ventaja de no tener tanto código para el acceso a las variables. La desventaja que posee, es que no permite que las subclases modifiquen el valor de la variable, cuando a veces es necesario.  
   En el caso del acceso indirecto a las variables, lo malo es que se genera un protocolo en el cual siempre se tiene que utilizar los getters y setters o eso podría ocasionar bugs. Muchas veces se escriben y no se utilizan, también. Lo positivo es que deja que las subclases modifiquen mediante alguna lógica el valor esperado.

5-

Es conveniente usarlo cuando se precisa inicializar un objeto con valores parametrizados. No es muy útil de usar cuando el setter tiene alguna lógica para seteado, debido a que se precisa saber si se están inicializando los valores para dejarlos sin ningún cambio. Para que esto no ocurra, se puede exponer otro método que setee todos los valores de golpe, pero al ser un método público, quita la encapsulación del objeto.

6-

Fuera del objeto, no se da acceso directo sino a las operaciones que se esperan poder hacer. Se pueden crear objetos con una interfaz de iteración y devolverla, para no tener que procesar todas las acciones de forma encapsulada y dar un poco más de libertad (como con un WriteStream). Dentro del objeto, se pueden utilizar los mismos métodos para centralizar el acceso, modificación y demás acciones con la colección.

7-

Porque con un solo método, se debería de parametrizar para que pueda setear un valor binario, como false o true, lo cual puede revelar la implementación de la representación del estado, violando la encapsulación. Al agregar dos métodos para ambos valores, se evita que el usuario le tenga que pasar el valor a setear, y clarifica la acción mediante el nombre (como prender o apagar), lo cual ayuda a la abstracción.