1. a

Há diversas modalidades de polimorfismo. A questão aborda o polimorfismo de subtipo, que está presente quando há hierarquias de subtipos.

É plausível que, se uma função aceita um parâmetro de certo tipo, ela também aceite, no lugar do parâmetro, um subtipo. Se a função apresentar uma mudança de comportamento de acordo com o tipo do parâmetro, então caracteriza-se o polimorfismo. A palavra polimorfismo quer dizer “várias formas”.

No caso apresentado no enunciado, temos uma ideia única (calcular a área de uma figura), que se manifesta de diversas formas distintas (calcular a área de um quadrado é diferente de calcular a área de um círculo, etc.).

A ideia única é representada por um único nome (“calcular_area”). Esse nome precisa ser vinculado a implementações de fato. Existe uma implementação para cada subtipo.

1. b

A ideia de vínculo em linguagens de programação refere-se à ligação entre um nome e um elemento da linguagem. Em geral, nomes são resolvidos em tempo de compilação, sendo substituídos por endereços físicos de memória.

No entanto, recursos modernos de linguagens de programação impossibilitam a definição do elemento vinculado a um nome antes da execução do programa. Assim, a decisão de como resolver o vínculo (achar o elemento associado a um nome) precisa ser tomada enquanto o programa está rodando. Isso é vinculação dinâmica.

Para viabilizar a vinculação dinâmica em linguagens de programação, existe uma técnica conhecida como *dispatcher*. A ideia básica é manter uma tabela de endereços físicos que podem estar vinculados ao nome em função do tipo no qual ele for invocado. Assim, quando o nome é acionado de fato, testa-se o tipo e segue-se para o endereço correspondente na tabela.

O polimorfismo de subtipo não poderia estar presente em linguagens de programação sem a vinculação dinâmica, justamente porque ele se caracteriza pela presença de um nome que pode ser vinculado a diversas implementações diferentes.

2. Programação genérica, ou polimorfismo paramétrico. Funções, por exemplo, passam a ter um atributo a mais para indicar o tipo a que se aplicam. Nesse exemplo, uma função de ordenação passaria a ter um atributo de tipo, passando a existir as variantes “ordena<inteiro>”, “ordena<string>”, “ordena<data>”, etc. Como a ordenação se baseia no operador de comparação, e esse, por sua vez, é polimórfico por sobrecarga, a mesma implementação funciona para todos os casos acima.

3. a

Antecipa a descoberta de erros de tipo.
A anotação de tipos muitas vezes orienta o programador a entender o código.
Pensar em tipos ajuda no design.
Ferramentas de edição, compilação, etc. de programas tiram vantagem dos tipos para tornar a programação mais ágil.

3. b

Anotar tipos é uma etapa protocolar que atravanca a programação, assim eliminá-la confere agilidade ao processo.
Código sem compromissos de tipo podem se mostrar mais flexíveis e reutilizáveis.
O aspecto visual geral do programa fica mais leve.

3. c

Não há uma convergência sobre o significado da expressão “tipificação forte”. Portanto, é um conceito subjetivo.

4.

Conversão de tipos acontece quando há um problema de compatibilidade de tipos entre operandos ou parâmetros, e existe um método de conversão possível e conhecido que pode ser utilizado pelo mecanismo de run-time da linguagem de programação.

Coerção é uma conversão automática.

5. a

valor = 2, lista = {2,4,3,1,0,5}
valor = 2, lista = {2,4,3,1,0,5}
valor = 2, lista = {2,4,3,1,0,5}

Não há mudanças nos valores das variáveis, pois são passadas cópias que, depois de alteradas, são descartadas.

5. b

valor = 3, lista = {2,4,2,1,0,5}
valor = 3, lista = {2,1,2,4,0,5}
valor = 4, lista = {2,1,2,3,0,5}

Foram toleradas respostas que consideraram lista[1] como primeiro elemento do arras em vez de lista[0] como é a convenção da linguagem C. O mesmo em 5. c

5. c

valor = 3, lista = {2,4,2,1,0,5}
valor = 3, lista = {2,1,2,4,0,5}
valor = 4, lista = {2,1,2,4,3,5}

6.

Vantagem: uma computação só é realizada quando necessária.
Desvantagem: perde-se a intuição quanto ao funcionamento do programa ao se analisar o código-fonte.

7.

Transparência referencial é uma propriedade de linguagens formais, seja da lógica, da matemática, ou da programação de computadores.

A palavra “transparência" é empregada no sentido de que tudo está às claras, evidente. O oposto de transparência é “opaco”, ou seja, escondido, não visível.

Diz-se que uma linguagem de programação tem transparência referencial quando todas as condições para a compreensão do programa estão evidentes na própria linguagem. Não há mecanismos “opacos”.

Se uma linguagem tem a propriedade da transparência referencial, toda função quando é aplicada poderia ser substituída pelo código que a define. Isso não acontece se a função tiver “efeitos colaterais”, isto é, se ela realizar outra ação que não aquela que apresenta como retorno.

Um exemplo de situação de opacidade são variáveis que podem mudar de valor através de atribuições de valores. Variáveis que mudam de valor possuem estado, e o estado de uma variável nunca é evidente em si mesmo. Em uma linguagem em que a atribuição de valores está ausente, as variáveis não tem estado. Todas as menções que se faz a variável estão presentes na própria função, e, por isso, é possível saber exatamente para que serve a variável.

Como atribuição de valores está presente em linguagens de programação do paradigma imperativo, elas não têm a propriedade da transparência referencial.