Homework 7

Problem 1

a)

Template approach

Benefits:

- instantiated at compile time so they have good performance
- type checking also occurs at compile time — good for checking errors before running
- reusable

Drawbacks:

- compile time could be long if many instances of the template
- debugging template code can be tricky
- code size can increase

b)

Generics approach

Benefits:

- resusable code
- type checking at compile time - less runtime errors
- code size doesn’t drastically increase

Drawbacks:

- error checking with types can be difficult to resolve
- performance overhead

c)

Duck Typing

Benefis:

- high flexibility — only fails at runtime if method doesn’t exist or is incompatible
- less type declaration code to be written

Drawbacks:

- type checking at runtime — leads to undefined behavior
- performance overhead with dynamic method resolution

If I was defining a new language, I would choose either of these depending on the typing that I have defined. If statically typed, then templating or generics would be preferred. If dynamic typing, then duck typing would be preferred.

Problem 2

Dynamically-typed languages perform type checking at runtime — this inherently allows functions to operate on any type of argument, as long as the operation being performed is valid for that type. Therefore, parametric polymorphism would just be performing duck typing instead of requiring a specific approach for handling it.

Problem 3

You can specify the behavior (methods or operations) that a type must have to be used with a particular template. This means you can write a function that works with any type that supports a certain set of operations, even though you don't know what the actual type will be.

This is similar but there are a few differences:

- performance is better in statically typed because types are known at compile time
- the approach is probably safer as it catches type errors during compile time, but it can be less flexible since you have to define the required behaviors in advance

Problem 5

You can create protoype objects that can be used as a blueprint for other object. This object wouldnt be a class, but rather just an Object type that you use to create new objects of the same type. This works similarly to regular subtype inheritance.

Problem 7

No it cannot be used to instantiate concrete objects — IShape is an abstract base class. it isn't a complete class, rather it is a template for other classes. Therefore you can point to an IShape, but creating one would result in a compile time error. In order to create such an object, you would need to create a new class that inherits from IShape and implements the methods get_area and get_perimeter.

Problem 8

Interfaces in statically typed languages serve as a definition for an object while specifying a set of methods that such a class must implement.

In a dynamically typed language, duck typing resolves the need for an interface. You don't need to specify an object implements a certain interface, if it has a method that's also in the interfaces method list, then it acts just like an interface without requiring to declare one as one.

Problem 10

You would prefer interface inheritance where you want to inherit the properties of a class, but not the implementation of its methods. For example, if you have a Shape interface you can inherit it with the classes Circle and Rectangle, which have very separate properties and implementations for key methods and calculations. You wouln't want to do a traditional subtype, or the circle and rectangle class may have properties that don't meet the expected behavior of the functions.