University of Toronto, Department of Electrical and Computer Engineering

ECE 1724-F1 — Performant Software Systems with Rust

Quiz 4 — Sample Quiz 3

Wednesday, November 11, 2026

Note: Each question has exactly one correct answer. All snippets are complete programs. Ignore compiler warnings. Examples are simplified teaching adaptations of Rustrace; no knowledge of its code base is required.

1. What happens when this program is compiled and run?

```
struct CappedWriter<W> { inner: W }

impl CappedWriter<u32> {
    fn doubled(&self) -> u32 { self.inner * 2 }
}

fn main() {
    let writer = CappedWriter { inner: 3_u16 };
    println!("{}", writer.doubled());
}
```

Ⓐ It prints `6`; the receiver's `u16` is converted to `u32`.

Ⓑ It prints `3`; a shared reference prevents multiplication.

Ⓒ It does not compile: `doubled` is defined only for `CappedWriter<u32>`.

Ⓓ It does not compile: a generic struct can only have generic implementation blocks.

2. What does this program print?

```
trait EditorEffects { fn count(&self) -> u32; }
struct Log(u32);
impl EditorEffects for Log {
    fn count(&self) -> u32 { self.0 }
}
struct SyntaxEffects<E> { inner: E }
impl<E: EditorEffects> EditorEffects for SyntaxEffects<E> {
    fn count(&self) -> u32 { self.inner.count() + 1 }
}
fn main() {
    let effects = SyntaxEffects {
        inner: SyntaxEffects { inner: Log(3) },
    };
    println!("{}", effects.count());
}
```

Ⓐ `5`

Ⓑ `3`

Ⓒ `4`

Ⓓ It does not compile: a generic wrapper cannot implement a trait.

3. Which statement about this program is correct?

```
trait EditorEffects {}
struct Noop;
struct Recording;
impl EditorEffects for Noop {}
impl EditorEffects for Recording {}

fn make_effects(record: bool) -> impl EditorEffects {
    if record { Recording } else { Noop }
}

fn main() {
    let _effects = make_effects(true);
}
```

Ⓐ It compiles because both returned structs implement `EditorEffects`.

Ⓑ It compiles because this call only selects the `true` branch.

Ⓒ It fails because the same trait cannot be implemented for two different structs.

Ⓓ It fails because the branches return different concrete types behind `impl EditorEffects`.

4. What does this program print?

```
struct Versions { current: u32 }
impl Iterator for Versions {
    type Item = u32;
    fn next(&mut self) -> Option<Self::Item> {
        self.current += 1;
        Some(self.current)
    }
}
fn main() {
    let mut versions = Versions { current: 4 };
    println!("{:?} {:?}", versions.next(), versions.next());
}
```

Ⓐ `Some(4) Some(5)`

Ⓑ `Some(5) Some(6)`

Ⓒ `5 6`

Ⓓ It does not compile: `Self::Item` cannot name the associated type.

5. What happens when this program is compiled and run?

```
trait EditorEffects {}
trait WorkspaceEffects: EditorEffects {
    fn ready(&self) -> bool { true }
}
struct Log;
impl EditorEffects for Log {}

fn open<T: WorkspaceEffects>(effects: T) -> bool {
    effects.ready()
}
fn main() {
    println!("{}", open(Log));
}
```

Ⓐ It does not compile: `Log` does not implement `WorkspaceEffects`.

Ⓑ It prints `true`: implementing the supertrait automatically implements its subtrait.

Ⓒ It does not compile: a supertrait must declare at least one method.

Ⓓ It does not compile: a default method cannot be called through a generic parameter.