Primitive Obsession

1. **Primitive Obsession** is a code smell where primitive (like string, int, Guid) are used for domain concepts, leading to ambiguity and errors.
2. Using Guid or string for an orderId, customerId, or productId makes it easy to mix these identifiers up, as they're all of the same type.

Primitive Obsession is a code smell that occurs when primitive data types (like `int`, `string`, `bool`, etc.) are overused to represent domain concepts, instead of creating meaningful domain-specific classes or structs. This can lead to less readable, less maintainable, and error-prone code, as the semantics of the values are not explicitly encoded in the type system.

Passing multiple loosely related primitive parameters

```
public void CreateOrder(string productName, string customerName, string
customerEmail)
{
    // Primitives are used instead of a `Customer` and `Product` class.
}
```

Why Is It a Problem?

1. **Lack of Type Safety**: You can easily mix up variables or pass invalid values.
2. **Duplication**: Validation logic, like checking an email format, is repeated everywhere.
3. **Reduced Maintainability**: Code becomes harder to understand and maintain.

Validation logic, like checking an email format, is repeated everywhere.

When using primitives like `string` to represent a domain concept (e.g., an email address), any associated logic for that concept, such as validating the format of the email, often gets repeated across the codebase. This creates duplication and inconsistency, as developers might implement the validation differently in various places, leading to potential bugs.

Example

```
public void RegisterUser(string email)
{
    if (!email.Contains("@")) throw new ArgumentException("Invalid email
format.");
```

```
    // Other logic...
}

public void NotifyUser(string email)
{
    if (!email.Contains("@")) throw new ArgumentException("Invalid email
format.");
    // Notification logic...
}
```

How can we Solve Primitive Obsession?

Strongly Typed IDs

How to Fix Primitive Obsession

1. **Encapsulate Primitive Values**: Create domain-specific value objects to represent concepts.

```
public class Email
{
    public string Value { get; }

    public Email(string value)
    {
        if (!IsValid(value)) throw new ArgumentException("Invalid email
format.");
        Value = value;
    }

    private bool IsValid(string email)
    {
        // Perform validation
        return email.Contains("@");
    }
}
```

Strong Typed Ids Pattern

- Creating distinct types for each kind of ID in your Domain.
- This makes your code more expressive and less error-prone.
- It clarifies which type of ID is expected and prevents accidentally using one type ID (like a productId) where another (like an orderId) is intended.

Benefits of This Approach

1. **Type Safety**: You cannot accidentally mix `CustomerId` with other GUIDs (e.g., `OrderId`).
2. **Validation Centralization**: All validation logic is encapsulated in the `CustomerId` type, reducing code duplication.
3. **Domain Clarity**: It makes the domain model more expressive and easier to understand.
4. **Immutability**: Prevents accidental changes to the `CustomerId` once created.
5. **Consistency**: Ensures that all `CustomerId` instances across the application conform to the same validation rules.

This pattern is a cornerstone of writing robust, maintainable, and domain-focused code in C#.

Terms

1. Encapsulation of Value

- **Purpose**: The `CustomerId` record encapsulates a `Guid` value that uniquely identifies a customer in your domain.
- **Why?** Instead of using a raw `Guid` throughout the code, you wrap it in the `CustomerId` type to provide clear semantic meaning. This avoids confusion and reduces the risk of passing incorrect or unrelated `Guid` values.

---

2. Validation Logic

- **Purpose**: The static `Of` method centralizes validation logic for creating a `CustomerId`.

```
if (value == Guid.Empty)
    throw new DomainException("CustomerId cannot be empty.");
```

Why? By enforcing rules for creating valid `CustomerId` values (e.g., not allowing empty GUIDs), you prevent invalid states from propagating through your application. This improves domain integrity and reduces bugs.

3. Immutability

- **Purpose**: The `record` keyword ensures that `CustomerId` is immutable, meaning its value cannot be changed after creation.

```
public record CustomerId
```

- **Why?** Immutability is a hallmark of value objects. It ensures that once created, the `CustomerId` remains consistent, which is particularly important in concurrent or distributed systems.

4. Factory Method (`Of`)

- **Purpose**: The `Of` method is a factory method for creating instances of `CustomerId`. It abstracts the creation process and ensures proper validation.

```
public static CustomerId Of(Guid value)
```

- **Why?** This prevents direct instantiation of `CustomerId` with invalid data. By using the factory method, you enforce a single point of entry for constructing valid objects.

5. Error Handling

- **Purpose**: The `Of` method uses `ArgumentNullException.ThrowIfNull` and a custom `DomainException` to provide meaningful error messages when invalid data is encountered.
- **Why?** This helps identify and handle invalid states early, improving debugging and error reporting.

---

6. Domain-Specific Type

- **Purpose**: The `CustomerId` type is meaningful within the domain. It reflects the business rule that every customer has a unique identifier.
- **Why?** Using a domain-specific type like `CustomerId` makes the code more expressive and easier to understand. It also enhances type safety by distinguishing between different GUID usages (e.g., `OrderId` vs. `CustomerId`).