POLYMORPHISM

Inheritance

A class C can derive from another class B

Class B is called the base class (also superclass)

Class C is called the derived class (also subclass)

When Inheriting

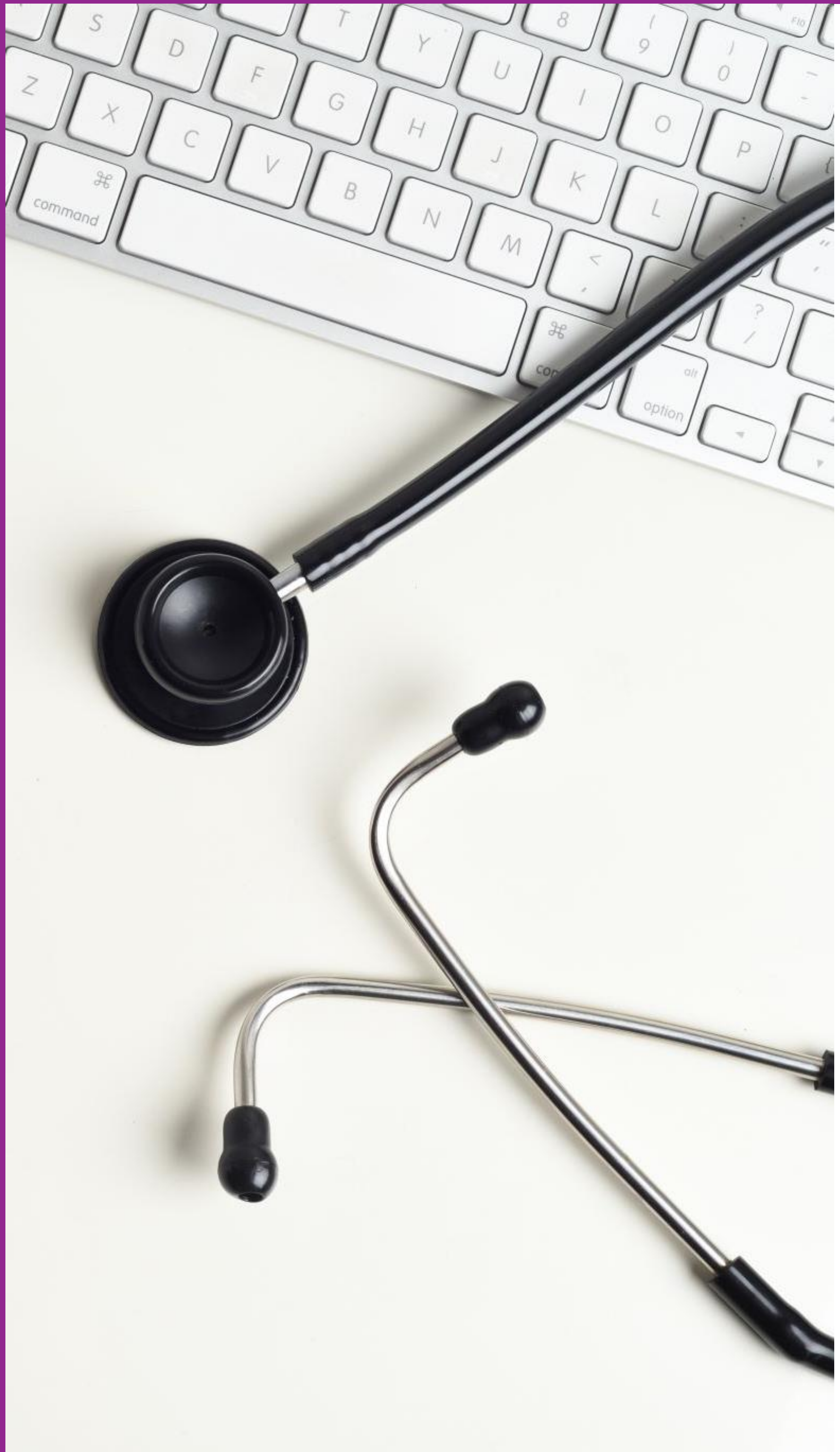
- Derived class inherits all members of base class
- Nothing can be removed
- Things can be added
- Things can be made more visible not less

Some rules of inheritance for C#

- A class can only inherit from one other class
- Classes cannot inherit from a struct
- Structs cannot inherit from anything
- If you don't specify a base class, it implies System.Object

Inheritance is used in different ways

- To reuse code
- To describe is-a relationship between entities
- To provide a supertype / subtype
- To write functions that operate on groups of types



Subtype

- An instance of Class C can be used wherever an instance of Class B is requested
- This means that C is a “subtype” of B
- We can also call C a “specialization” of B
- Conversely, B is a “supertype” of C
- We can also say B is a “generalization” of B

Multiple Kinds of Polymorphism

- Ad-hoc polymorphism – a.k.a. function overloading
- Parametric polymorphism – e.g., generic methods
- Subtyping – e.g., when implied by inheritance in C# or Java
- Discussed quite well on Wikipedia

Subtype Polymorphism

- In OOP context, the most common meaning of polymorphism
- The ability for a type identifier to represent a set of types
- Within the context of a function or interface
- In this case the base class is polymorphic
- As is any function consuming a base class
- Recommend reading about subtype polymorphism on Microsoft Learn.

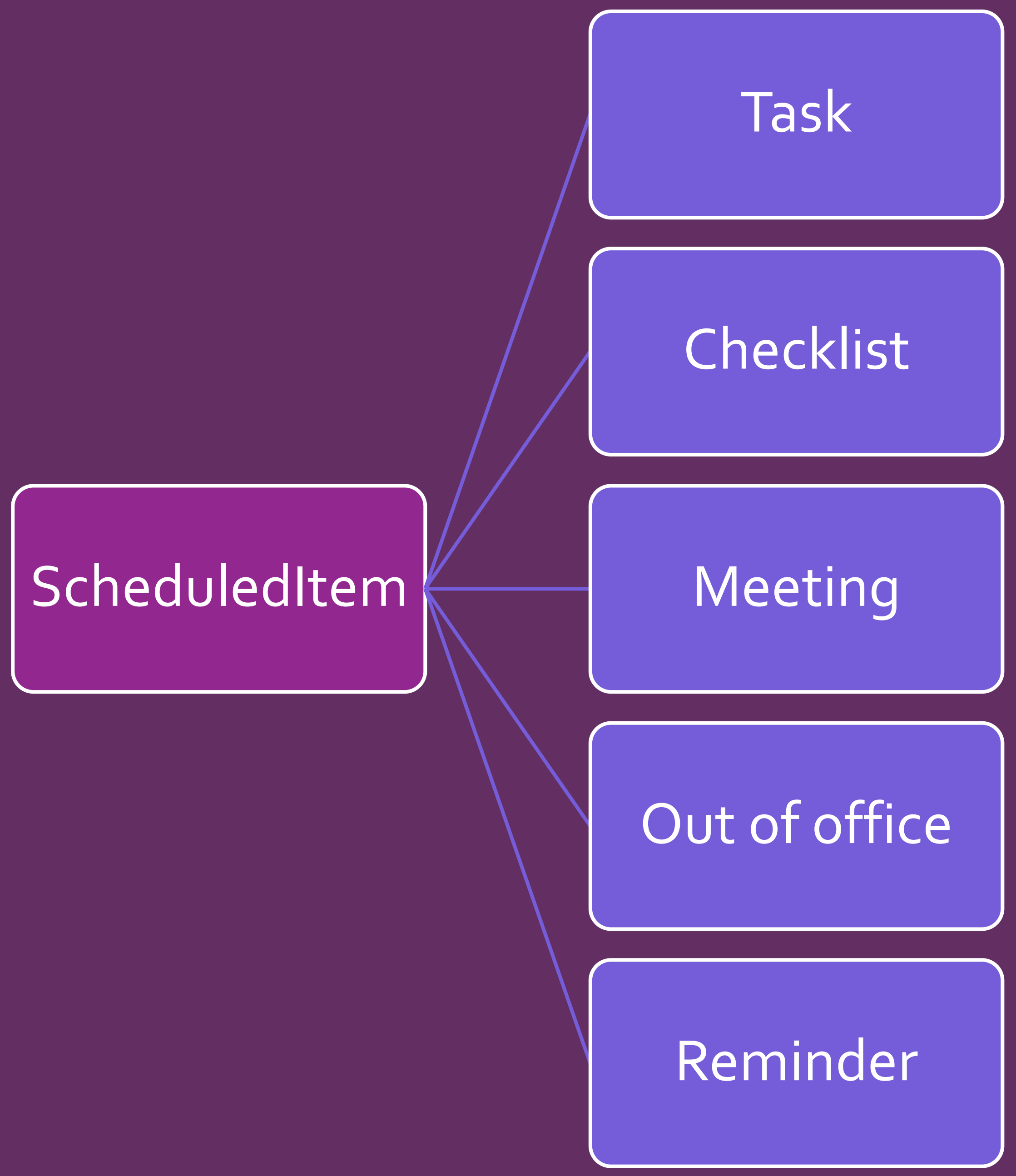
Is Inheritance Equal to Subtyping?

- Formally, in C# the answer is yes
- Only from the point of view of the type system
- Remember type-system validates expressions (not values)
- Run-time is a different story

Liskov Substitution Principle

- The “L” in the SOLID principles
- A principle of substitutability
- Any instance of a type should be able to be replaced by instances of subtype
- Should not “break” the program (alter desirable characteristics)
- Applies both to type system and to run-time behavior

Example: Calendar Application



Using a Base Class in a Collection

01

A big advantage of subtyping is that we can have similar items in a collection

02

You define the collection to contain the base class

03

For example: List<ScheduledItem>

04

Operations that are common to the base class can be applied to all elements

Don’t use Inheritance to Reuse Code

- Prefer composition instead
- Inheritance implies a “is-a” relationship between entities
- Composition implies a “has-a” relationship between entities
- Small amount of extra coding (have to forward functions)
- Still better and safer

Composition

- The idea that one object may contain another
- Any class with any fields is an example of composition
- Trivial concept, but used to describe alternative to inheritance

Class Relationships

Is-a relationship – inheritance

Has-a relationship – composition

Upcast and Downcasts

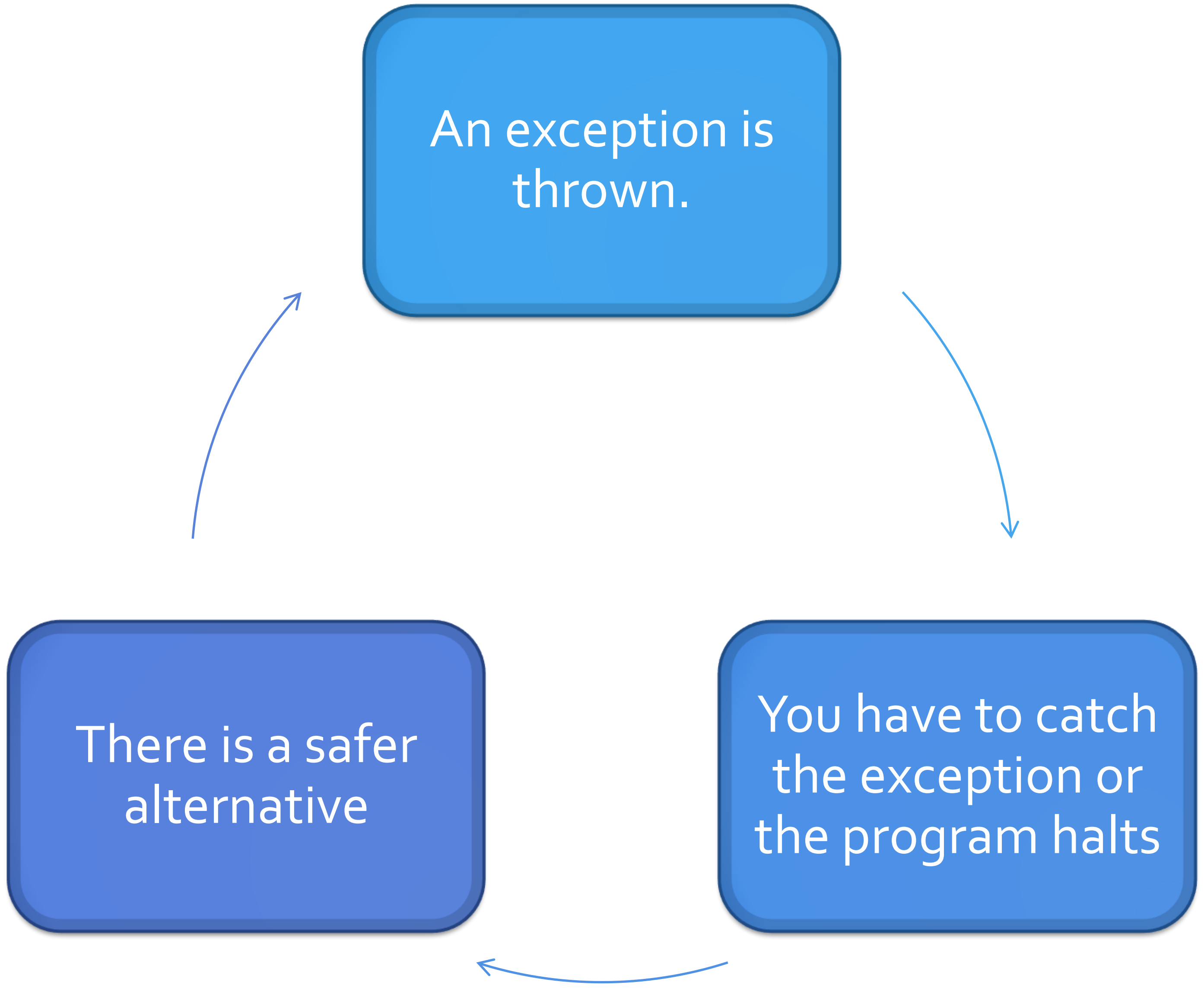
Upcasting from a class to a super type (base class) always works

Conversion happens implicitly

Downcasting from a base class to a derived class may or may not work

Require an explicit conversion

What if Explicit Conversion Fails?



The “as” operator

Casting to a derived type can be done safely using the “as” operator.

If the run-time type of the object does not match returns null

If successful returns an expression that has the cast type

The “is” operator

Checks if a value (result of expression) is of a given type

Returns true if the run-time type matches the requested type

Note: in recent C# versions also allows pattern matching

Overview: checking the run-time type

- Use the “is” operator: expr is Type
- Use the “as” operator and check for null: (expr as Type) == null
- Call “GetType()” on the object (tricky)
- Use an explicit cast conversion: (Type)expr and catch InvalidCastException
- Try catch is a bad idea here
- See Microsoft learn documentation

Consider Functions on Every Derived Class

```
public class ScheduledItem
{
    public DateTime DateTime;
    public bool IsPast() => DateTime.Now > DateTime;
}

public class Appointment : ScheduledItem
{
    public string Location;
    public string ItemKind() => "Appointment";
}

public class Task : ScheduledItem
{
    public bool Completed;
    public string ItemKind() => "Task";
}
```

```
public static void OutputItems(IEnumerable<ScheduledItem> items)
{
    foreach (var item in items)
    {
        var kind = "";
        if (item is Appointment)
            kind = (item as Appointment).ItemKind();
        if (item is Meeting)
            kind = (item as Meeting).ItemKind();
        if (item is Task)
            kind = (item as Task).ItemKind();
        if (item is Reminder)
            kind = (item as Reminder).ItemKind();
        Console.WriteLine($"Item {kind} is scheduled for {item.DateTime}");
    }
}
```

That function can be improved

The “is” operator can also declare a variable name.

It makes an improvement in the code readability

```csharp
public static void OutputItems(IEnumerable<ScheduledItem> items)
{
    foreach (var item in items)
    {
        var kind = "";
        if (item is Appointment appointment)
            kind = appointment.ItemKind();
        if (item is Meeting meeting)
            kind = meeting.ItemKind();
        if (item is Task task)
            kind = task.ItemKind();
        if (item is Reminder reminder)
            kind = reminder.ItemKind();
        Console.WriteLine($"Item {kind} is scheduled for {item.DateTime}");
    }
}
```

Still not ideal:

- Using the function is complex
- What if there are a lot of base classes?
- What if we want many functions for ScheduledItem?
- Ideally we want complexity hidden in the class
- This is where “virtual” functions come in useful

USING A VIRTUAL FUNCTION

```
public class ScheduledItem
{
    public DateTime DateTime;
    public bool IsPast() => DateTime.Now > DateT
    public virtual string ItemKind() => "";
}

public class Appointment : ScheduledItem
{
    public string Location;
    public override string ItemKind() => "Appoin
}

public class Task : ScheduledItem
{
    public bool Completed;
    public override string ItemKind() => "Task";
}
```

```
public static void OutputItems(IEnumerable<ScheduledItem> items)
{
    foreach (var item in items)
    {
        var kind = item.ItemKind();
        Console.WriteLine($"Item {kind} is scheduled for {item.DateTime}");
    }
}
```

NOW THE FUNCTION IS SIMPLER

Understanding Virtual Functions

- When a virtual function is called on a base class
- If the run-time type of the value is different (a derived class)
- If an override of the virtual function exists (on the derived class)
- Then the override is called
- This is called “dynamic dispatch”
- It happens thanks to a “virtual method table”
- Please read the Microsoft Learn documentation for the virtual keyword for C#

Examples of Virtual Functions

- It is good practice to override virtual functions from System.Object.
- virtual string Object.ToString();
- virtual bool Object.Equals(object? other);
- virtual int Object.GetHashCode();

Should we use ScheduledItem directly?

There are many things that are a ScheduledItem but what use it the class itself?

Its main role is to describe a family of types (it's subtypes)

So perhaps we should prevent it from being used directly

Abstract Class

The abstract keyword (on a class) prevents it from being instantiated

In other words, you can’t call new.

Allows adding abstract methods

Abstract Class with Abstract Method

```
public abstract class ScheduledItem
{
    public DateTime DateTime;
    public bool IsPast() => DateTime.Now > DateTime;
    public abstract string ItemKind();
}
```

Abstract Method

An abstract method is a virtual function with no body

It is only allowed on abstract classes

All classes that derive from the class, ***must*** override the abstract method