# Interfaces

## C# Programming

# In this chapter we will learn ...

- Why interfaces are useful
- How to define an interface
- How to use interfaces to effect decoupling

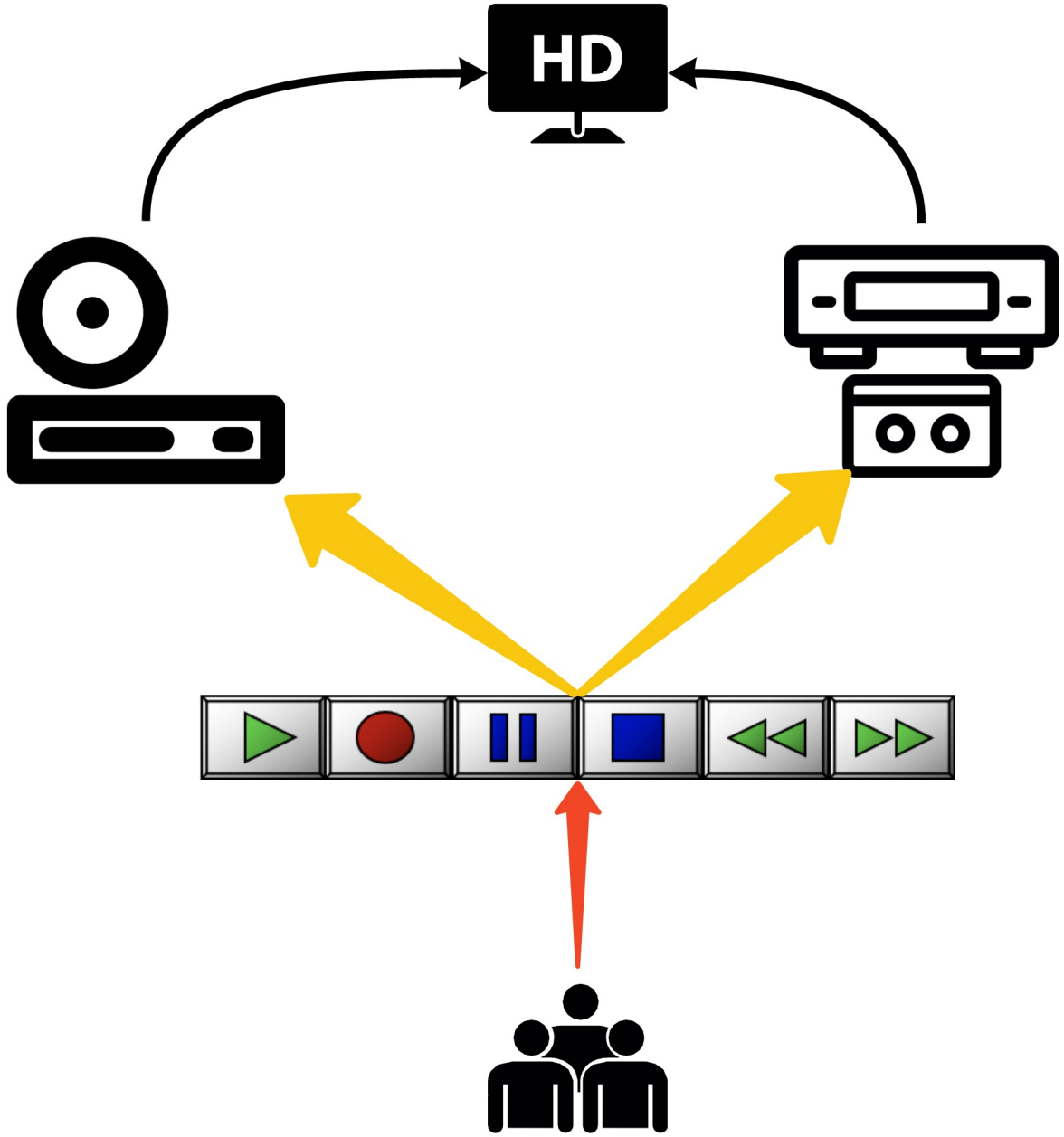
# Interfaces

# What is an Interface

- A contract?
- A way of interacting with something?
- None of the above?

# Question

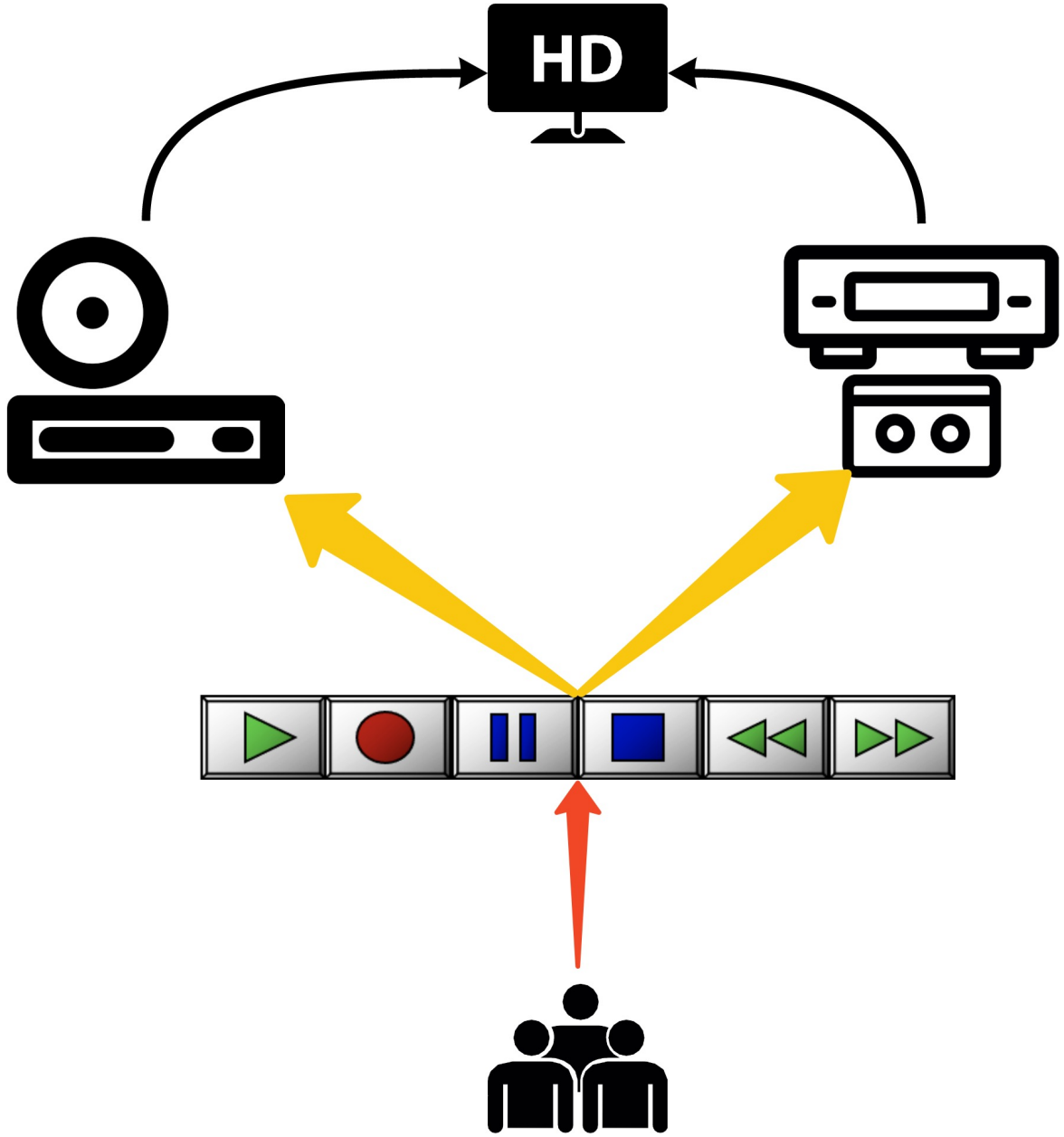
**What does the Control Panel on the right allow us to do?**



# Defining Capability

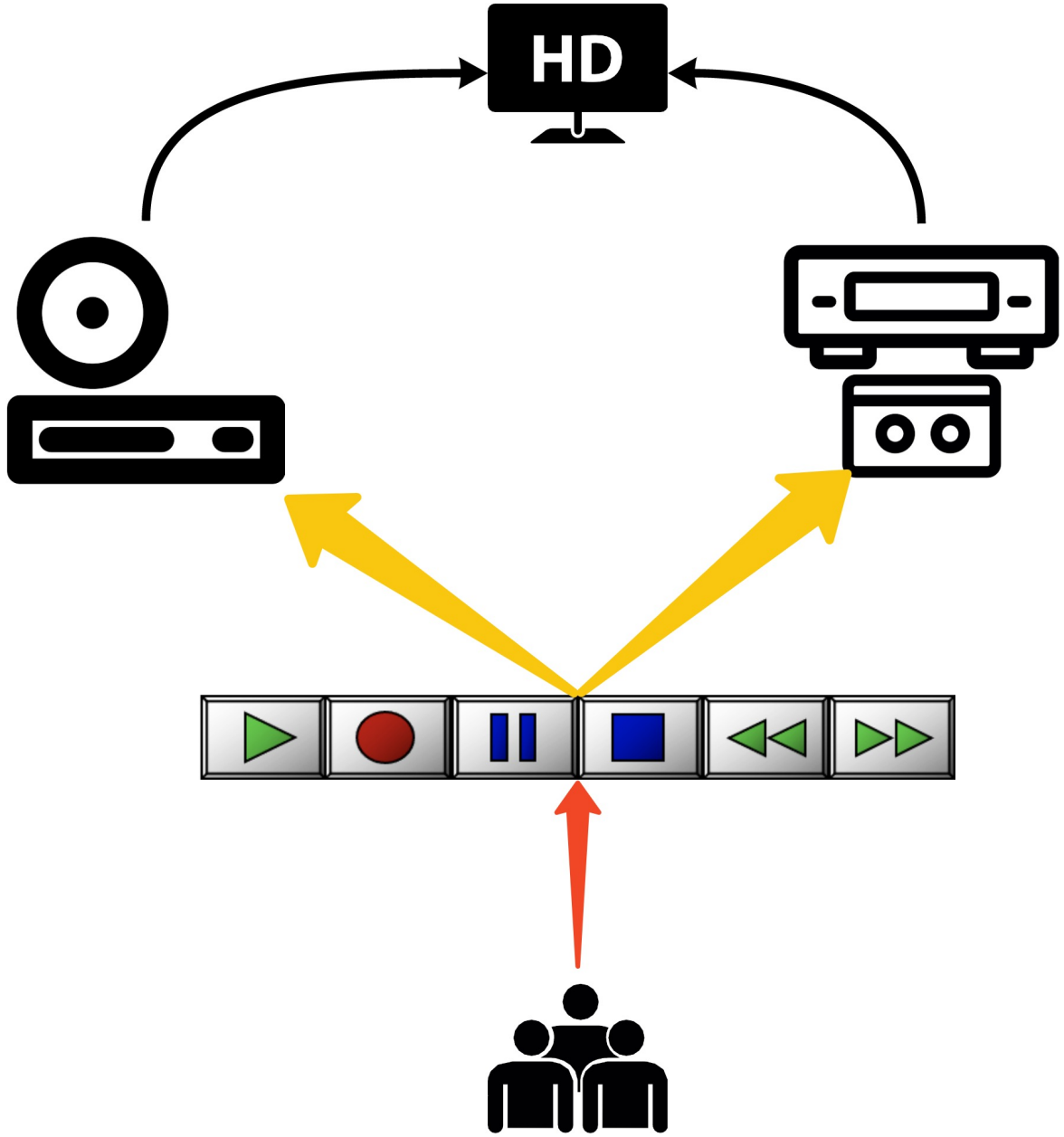
**Probably the most widely understood interface of all describes the following capabilities**

- Play
- Record
- Pause
- Stop
- Fast Forward, Rewind



# Exposing an Interface

- When an object implements an interface
  - VCR Implements Playable interface
  - DVD implements Playable interface
- Anyone who understands that interface can use it
  - Most people know how to use the ubiquitous Playable interface



# Why is that good?

**We re-use system capabilities**

- Play Media
- Spell Check Documents
- Accelerate Object
- Calculate Tax Due of all kinds of things
    - People, Cars, Houses, iPhones

**Let's write code in terms of capability not type!**

# Interfaces Benefits

## Decoupled Code

- Interfaces help break tightly bound links between classes
- Fixed dependencies can be difficult to test and change

## Flexible Code

- Interfaces help us build flexibility
- Flexibility allows us to change implementation when needed

# What is an Interface - Definition Part 1

## An interface is a code contract

```
<accessibility> interface <name> {
    <methods> ...
    <properties> ...
    <events> ...
}
```

- Defines methods properties and events an object must implement
- Usually Contains no concrete code
- Methods properties and events have no accessibility declarations

# Interface - Example

## Interface acts as a code contract

- Interface has no code associated with declarations
- Specifies what not how

```
public interface IAccelerable {
    void Accelerate(int amount);
    int Speed {get;}
    int MaxSpeed {get;}
}
```

# Implementing an interface

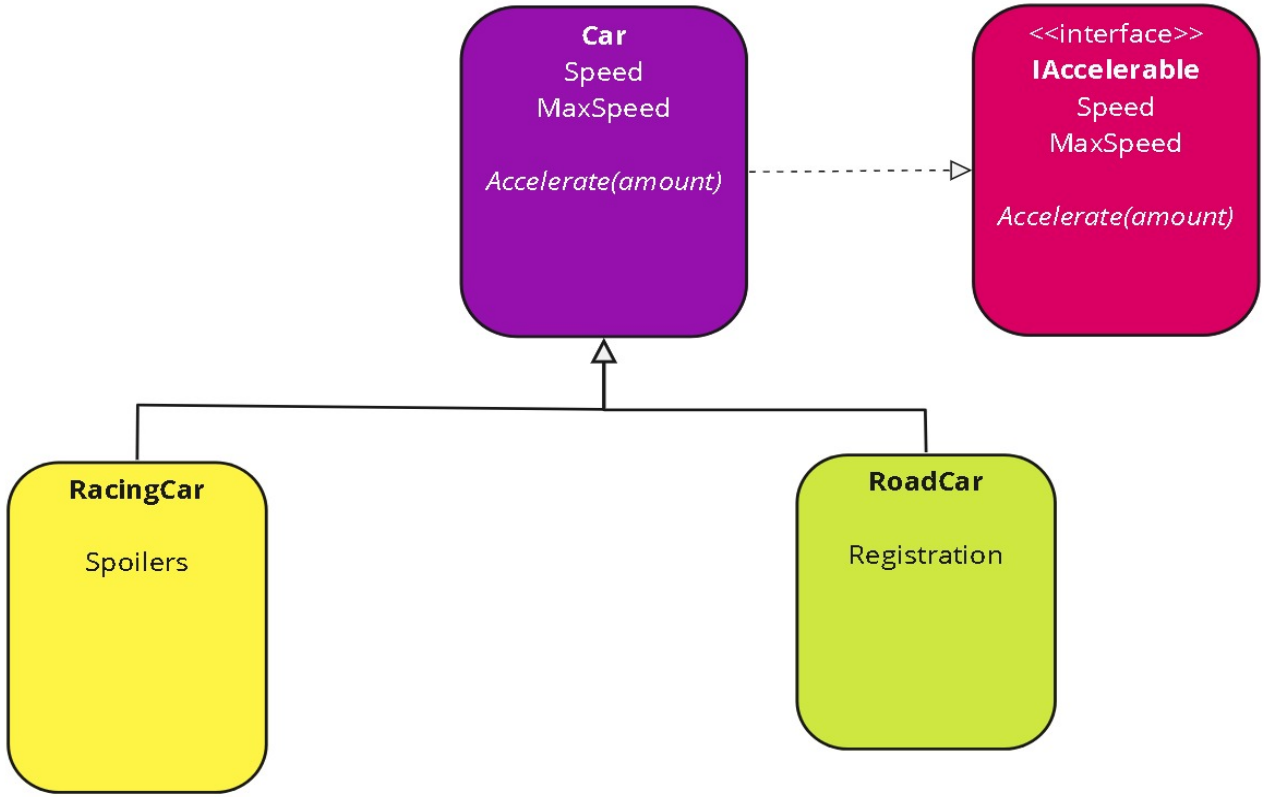
## Classes or Structs can implement one or more interfaces

- Must implement the methods, properties and events contracted
- A Compile Error will occur if Car doesn't implement the interface exactly

```
class Car: IAccelerable {
    public int Speed {get;}
    public virtual int MaxSpeed {get => 120}
    public virtual void Accelerate(int amount) {
        Thread.Sleep(80 * amount);
        Speed += amount;
    }
}
```

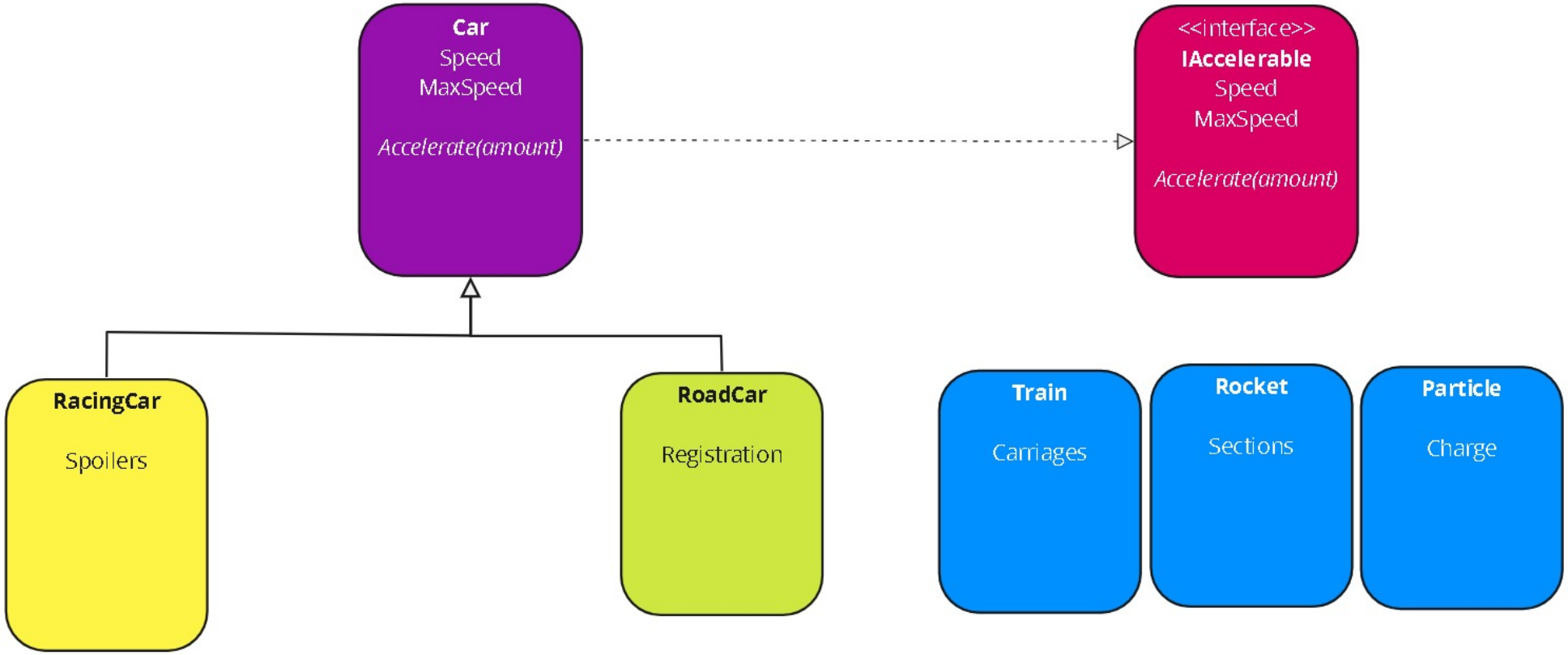
# Consider our Current Model

- Car implements IAccelerable
- RacingCar and RoadCar inherit from Car



# What if we want to Accelerate other objects?

- AccelerateToMaxSpeed is expecting to be passed Cars not Rockets



# Principle of Substitution Allows us to write this

```
class Accelerator {
    public static void AccelerateToMaxSpeed(Car item) {
        while (item.Speed < item.MaxSpeed>) {
            item.Accelerate(5);
        }
    }
}
```

## We can pass to AccelerateToMaxSpeed anything that is or derives from Car

```
Accelerator.AccelerateToMaxSpeed(new RoadCar());
Accelerator.AccelerateToMaxSpeed(new RacingCar());
```

# Problem

**We have written a useful sub system - Accelerator that can**

- Accelerator Car objects to their maximum speed

**We want to also use this sub system to Accelerate other things but**

- Substitution only allows derived types to be passed to base parameters

**We need to decouple AccelerateToMaxSpeed from its dependence on type**

- Instead of using a Car parameter we can use an interface parameter

# Something in Common

**At the moment RacingCar and RoadCar have something in common**

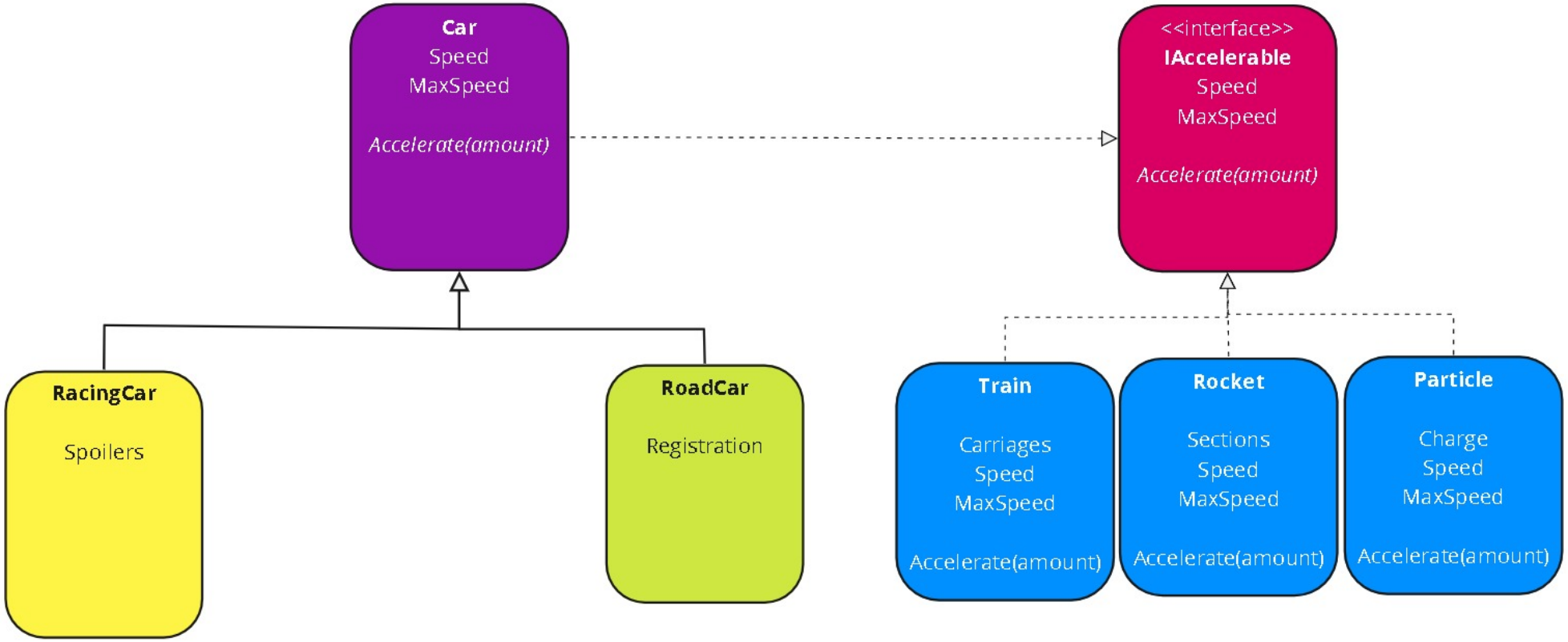
- Both inherit features from Car and can be substituted for Car

**Let's give all of the types something in common**

- Make them all implement IAccelerable

# Defining Common Capability across unrelated types

- All types below implement *IAccelerable*



# Rocket is IAccelerable

## Let's make Rocket IAccelerable

```
class Rocket: IAccelerable {
    public int Speed {get;}
    public virtual int MaxSpeed {get => 1000}
    public virtual void Accelerate(int amount) {
        Thread.Sleep(5 * amount);
        Speed += amount;
    }
}
```

# Interface references

## An Interface reference can reference any object that implements the interface

```
IAccelerable thing1 = new Car();
IAccelerable thing2 = new Rocket();
...
thing1.Accelerate(5); // Car object would accelerate by 5
```

## We could also do this

```
List<IAccelerable> things = new List<IAccelerable> () {
    new RoadCar(), new Rocket(), new RacingCar(), new Train()
}
foreach(IAccelerable thing in things) {
    thing.Accelerate(5);
}
```

# Decoupling our Code with Interface References

```
class Accelerator {
    public static void AccelerateToMaxSpeed(Car item) {
        while (item.Speed < item.MaxSpeed>) {
            item.Accelerate(5);
        }
    }
}
```

## Above code can be decoupled by changing Car parameter to IAccelerable

```
class Accelerator {
    public static void AccelerateToMaxSpeed(IAccelerable item) {
        ... // AccelerateToMaXSpeed can accelerate any object that implements IAccelerable
    }
}
```

# Working with Decoupled Code

## AccelerateToMaxSpeed

- Can work with any object that implements IAccelerable
- Is decoupled from type dependence
- Cares only what an object "Can Do" == Its capability

## So we can do this

```
Accelerator.AccelerateToMaxSpeed(new RacingCar());
Accelerator.AccelerateToMaxSpeed(new Rocket());
Accelerator.AccelerateToMaxSpeed(new Train());
```

# What is an Interface - Definition Part 2

An interface is

1. A code contract
2. A decoupling mechanism allowing us to write algorithms that focus not on the type of an object but its capability

# Breaking Interfaces

**Ideally once an interface is published we don't change it**

- Changing an interface will cause implementing code to break

**If an interface absolutely must change we can**

- Provide a default implementation for the breaking change
- Allow implementing types to continue to work
- Implementing types don't have direct access to the default implementation
    - Only way to access it is via an interface reference

# Interface - Default Implementations - C# 8.0

**Assume this is the published interface is changed to code below**

```
public interface IAccelerable {
    void Accelerate(int amount);
    int Speed {get;}
    int MaxSpeed {get;}
}
```

```
public interface IAccelerable {
    void Accelerate(int amount);
    int Speed {get;}
    int MaxSpeed {get;}
    void Boost() => Accelerate(20);
}
```

# Interface - Default Implementations Continued

## Default Implementation prevents implementing types breaking

```
Car myCar = new Car();
myCar.Accelerate(5); // OK
myCar.Boost(); // Compilation Error! Boost() isn't available from Car reference
```

- Car needs to implement Boost explicitly

## We can access Boost() from the interface

```
IAccelerable myCar = new Car();
myCar.Boost(); // OK
```

# SOLID

# SOLID OOD Principles

**Help us create well designed loose coupled, easier to test types**

- S - Single Responsibility
- O - Open Closed Principle
- L - Liskov Substitution Principle
- I - Interface Segregation
- D - Dependency Inversion

**By Robert C Martin (Uncle Bob)**

# Single Responsibility

**A class should have only one reason to change**

- A class should have just one job
- If it needs to change its because its job has changed

**Imagine an Bank Account class**

- We might change how the bank account rules work
- We shouldn't need to change the bank account class to alter how it talks to a database

**Know a class's responsibility and stick to it!**

# Single Responsibility

## Consider the following Code

- Does it break Single Responsibility?

```
class Car {
    public void Accelerate(int amount) {
        Speed += amount;
        LogMetric("speed", Speed);
    }
    protected void LogMetric(string metric, object value) {
        Console.WriteLine($"{metric}:{value}");
    }
}
```

# ANSWER

## YES

## REASON

- Car should be responsible for managing the Car simulation
- It shouldn't have the responsibility for Logging
  - If we wanted to change how we did logging is that a valid reason to change Car?
  - What happens when we want to Log to a Database or to HTTP?

## SOLUTION

- Give the job of logging to another type and allow Car to reuse the logging services

# Towards Single Responsibility

- Give *Logger* the responsibility for logging
- Car can re-use *Logger* to access logging services

```
class Logger {
    public void LogMetric(string metric, object value) {
        Console.WriteLine($"{metric}:{value}");
    }
}
class Car {
    Logger logger = new Logger();
    public void Accelerate(int amount) {
        Speed += amount;
        logger.LogMetric("speed", Speed);
    }
}
```

# Open/Closed Principle

## Code should be

- Open for Extension yet Closed for modification

## We should be able to

- Add new functionality to the application without having to change existing code

## Usually achieved using

- Inheritance
- Design patterns (such as Strategy pattern)
- Writing flexible/adaptable methods

# Open Closed Principle - Inheritance

## Here the MaxSpeed calculation is fixed to 120

```
class Car {
    ...
    public int MaxSpeed => 120;
}
```

## But if we allowed this we could extend Car by inheritance

```
class Car {
    ...
    public virtual int MaxSpeed => 120;
}
```

# Extension by Inheritance

## Below Car is extended into RacingCar

- Car does not need to be changed
- RacingCar is used to extend the application as a new type
- Because MaxSpeed is virtual it can be overridden in RacingCar

```
class Car {
    ...
    public virtual int MaxSpeed => 120;
}
class RacingCar: Car {
    public override int MaxSpeed => base.MaxSpeed + 80;
}
```

# Benefits of Open Closed Principle

## Existing code doesn't need to change

- Existing tests are still valid
- Code Coverage of existing tests is valid

## Focus can be placed on proving correctness of extension

- New test assets can be developed
- Acceptable test coverage can be achieved for extension

# Liskov Substitution

**Objects of a base class should be replaceable by objects of a derived class**

**Requirements**

- Derived Objects should not break the application
- Overridden methods should keep the same parameters as the base
- Overridden methods should not be more restrictive than the base
- Overridden methods can be less restrictive than the base if desired

# Liskov - Reference Substitution

## A base class reference variable can point to

- Any derived object in inheritance hierarchy

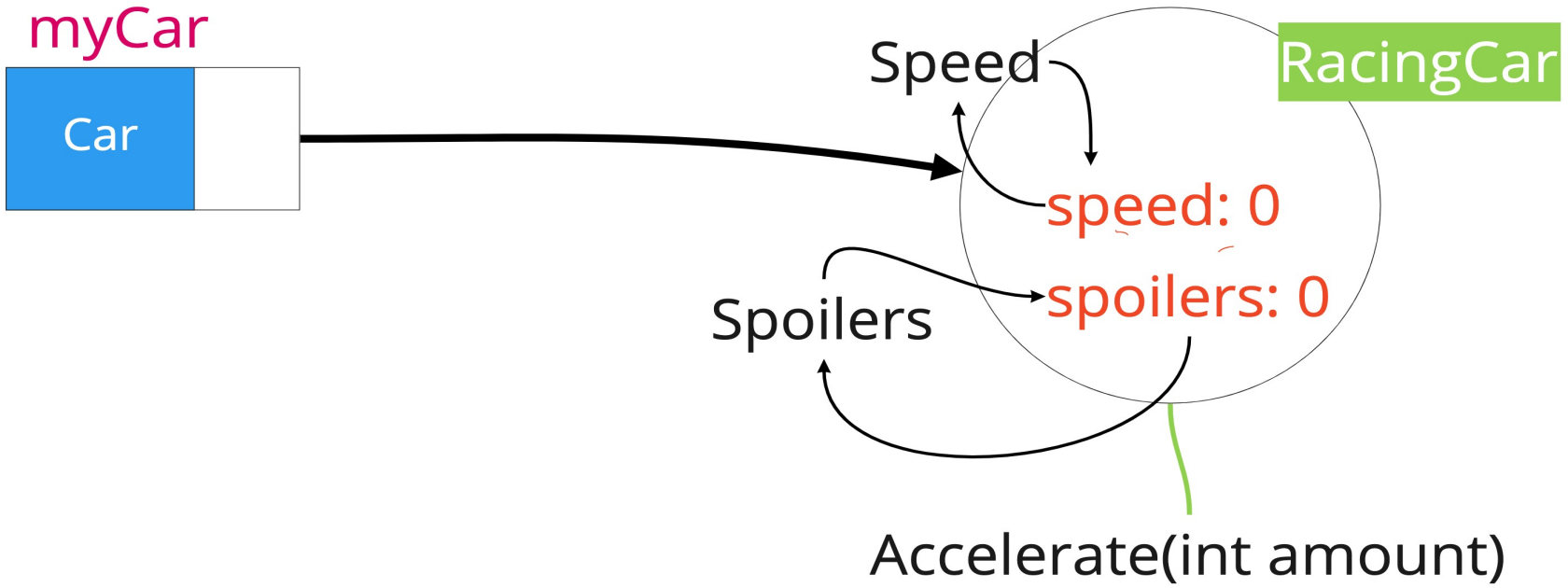
## That means we can do this

```
Car myCar = new RacingCar();
```

# Reference Substitution - Under the hood

```
Car myCar = new RacingCar();
```

- Car reference points to RacingCar object



# Interface Segregation

**"Types should not have to depend on interfaces they cannot use"**

## Broadly defined interfaces are difficult to implement

- Many badly related methods often make no sense to implementing types
- Better to have multiple compact and more tightly defined interfaces

# Interface Segregation - Example

```
interface IAccelerable {
    int Speed {get;set;}
    void Accelerate(int amount);
    void Brake();
}
```

## If IAccelerable is be implemented by

- Car - it all makes sense
- Rocket - Brake() makes no sense on a Rocket

# Solution

## Segregate the Interfaces into separate definitions

- Selectively implement the relevant interface(s)

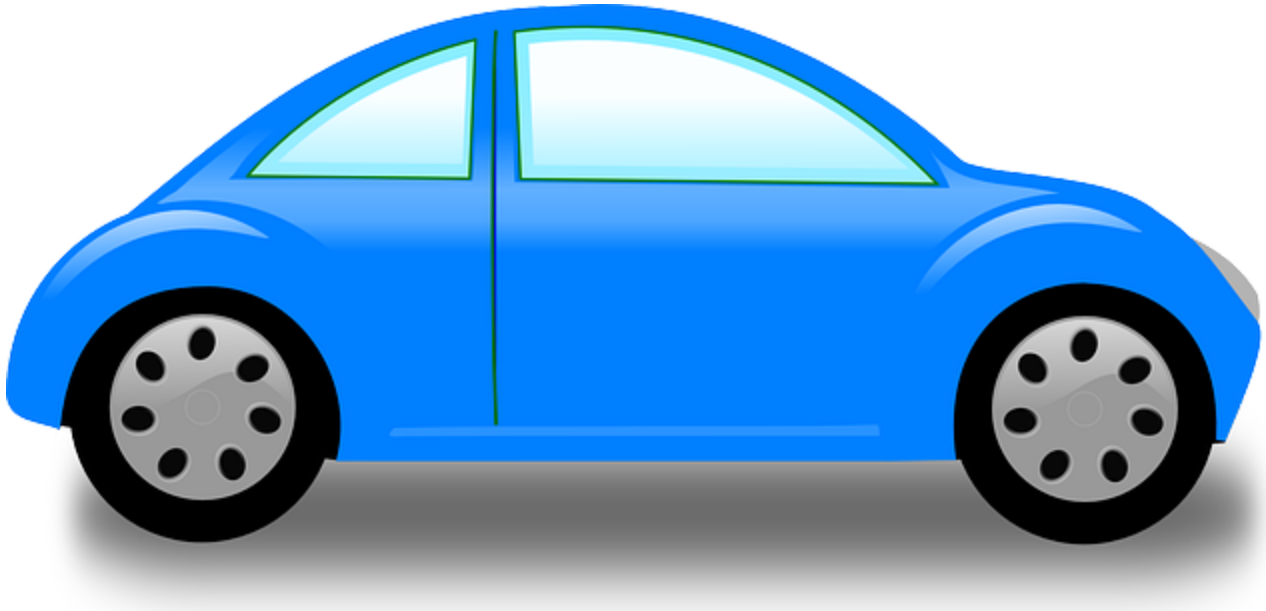
```
interface IAccelerable {
    int Speed {get;set;}
    void Accelerate(int amount);
}
interface IDecelerable {
        void Brake();
}
class Car: IAccelerable, IDecelerable {
    // Implement both IAccelerable and IDecelerable
}
class Rocket: IAccelerable { // Implement just IAccelerable
}
```

# Dependency Inversion

**Imagine a Car with a combustion engine built directly into the engine bay**

## What problems would that cause?

- Mmmmm.....



# Fixed Dependencies

## Here Car has a fixed dependance on Logger

- It can only ever use Logger to do logging
- That means it can only ever Log to the console

```
class Car {
    Logger logger = new Logger();
    public void Accelerate(int amount = 5) {
        Speed += amount;
        logger.Log("speed", Speed);
    }
}
```

# Let's Invert the Dependency

- A Logger Object is now passed in on creation of a Car
  - We could also pass any object that derives from Car

```
class Car {
    Logger logger;
    public Car(Logger logger) {
        this.logger = logger;
    }
    public void Accelerate(int amount = 5) {
        Speed += amount;
        logger?.Log("speed", Speed);
    }
}
Car myCar = new Car(new Logger());
```

# Decoupling Dependency Inversion

## To Avoid using Fixed Dependencies we typically use Interfaces

```
interface ILogger {
    void Log(string metric, object value);
}
```

## Car is declared using ILogger interface == NO Fixed Dependency

```
class Car {
    ILogger logger;
    public Car(ILogger logger) {
        this.logger = logger;
    }
}
```

# Injecting Dependencies

## Types implementing the dependent interface can be injected

```
class ConsoleLogger: ILogger {
    public void Log(string metric, object value) {
        Console.WriteLine($"{metric}:{value}");
    }
}
class FileLogger: ILogger { ... }
```

Dependency is now injected from the outside

```
Car myCar = new Car(new ConsoleLogger);
Car yourCar = new Car(new FileLogger);
```

# In this chapter we learned ...

- Why interfaces are useful
- How to define an interface
- How to use interfaces to effect decoupling