COMP 3522

Object Oriented Programming in C++

Week 9

Agenda

1. Casting
   - Up, down, cross-casting
   - Static, dynamic
2. Enums
   - Scoped
   - Unscoped

COMP 3522

CASTING

Casting

- Java and C++ are **strongly typed** languages
- The type of each object (variable or constant) is defined at **compile time** and cannot be changed during execution
- We can think of an object as:
  1. Bits in memory
  2. A **type that gives these bits meaning**.

Casting is ugly in C

- We can cast an arithmetic type (char, int, float, double) to other arithmetic types
- We can cast a pointer type to another pointer type

- There is lots of room for error here
- *No check at runtime is performed to see if the cast is correct*

So we try not to cast in C

- **Notation** is hard to spot (cast to type)
- We can basically convert **anything to anything**
- There is lots of room for **error** here
- But programmers do it anyway.

C++ casting can be safer

Bjarne Stroustrup views the C cast operator:

```
Animal* a = (Animal*)canine;
```

as relaxed, not strict

He introduced some new **casting operators** to C++:

1. **dynamic_cast**
2. const_cast
3. **static_cast**
4. reinterpret_cast

Vocabulary check

- **Upcasting**: cast pointer/reference from a **derived class to a base class**
  - Always allowed for public inheritance (is-a relationship)
- **Downcasting**: casting pointer/reference from a **base class to a derived class**
  - Not allowed without explicit type cast
- **Cross-casting**: casting pointer/reference from a **base class to a sibling class**

UP, DOWN, & CROSS CASTING

Casting between base and derived classes

- Let's look at **upcasting**
- Casting up from a derived to a base class is always possible when there are no ambiguities

```
Canine* caninePtr = new Canine;
Animal* animalPtr = caninePtr; //upcast to Animal type
```

- Can be **implicit**, i.e., a function that accepts a parameter of the base class accepts all sub-classes without the need for explicit conversion
  - `void speak(Animal * animal){} //pass child bird *, canine * OK`
- Upcasting is synonymous with **polymorphism**.

Upcasting example page 1 of 2

```
struct A
{
    int value;
    virtual void foo(){}
    virtual ~A(){}
};

struct B : public A
{
    float some_value;
    int foob() { return 22; }
};
```

Upcasting example page 2 of 2

```
void f(A a) { … } // NOT POLYMORPHIC (SLICING)
void g(A& a) { … }
void h(A* a) { … }

B b;
f(b); // OK, BUT BE CAREFUL! THIS IS SLICING
g(b); // OK
h(&b); // OK
```

These are all examples of **implicit upcasting** – the object b is converted to an object of base type A automatically.

upcast_and_slice.cpp

Casting between base and derived classes

- Let's move to **downcasting**

```
Animal* animalPtr = new Canine;
Canine* caninePtr = (Canine*)animalPtr; //downcast to Canine type
```

- This is the conversion of a pointer/reference to a sub-type pointer/reference
- When the actual referred-to object is not of that sub-type the behaviour is **undefined**

Casting between base and derived classes

If we need to downcast, we should ask ourselves why:

1. How do we make sure the object is really that sub-type?
2. What should we do if the object cannot be downcast?
3. Why not just overload functions for the two types?
4. Can we redesign our classes so that we can accomplish our task with late binding and virtual functions?

If we still need a downcast, there are two good choices in C++

Dynamic and static casting

- C++ offers a **dynamic cast** and a **static cast** between classes in an inheritance hierarchy

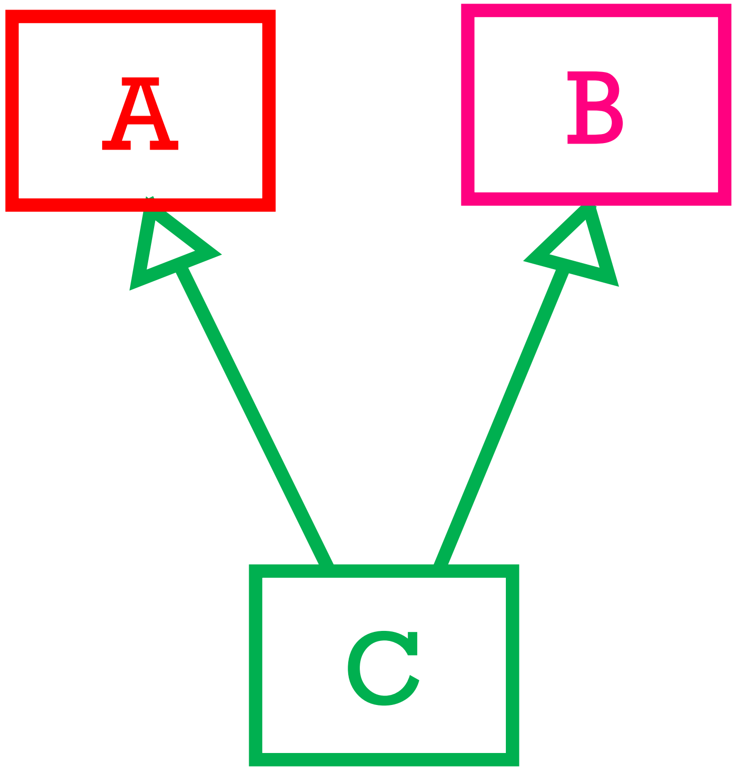
dynamic_cast<target_type pointer or reference>(variable)

static_cast<target_type pointer or reference>(variable)

Cross-casting?!

Consider this inheritance hierarchy:

```
A* an_A = new C;
B* a_B = dynamic_cast<B*>(an_A);
```



We will see this madness a few times this term.

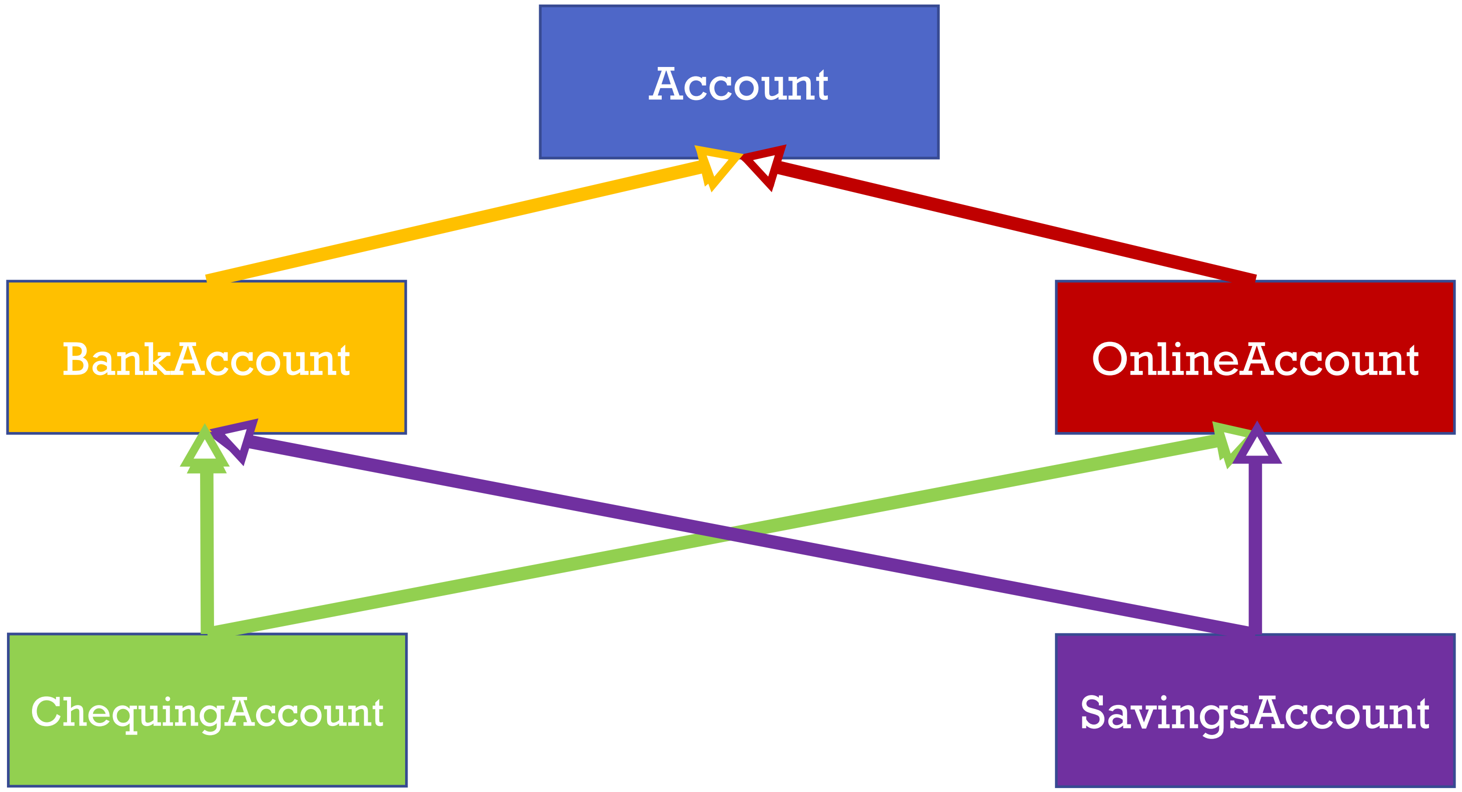
DYNAMIC CASTING

Dynamic cast

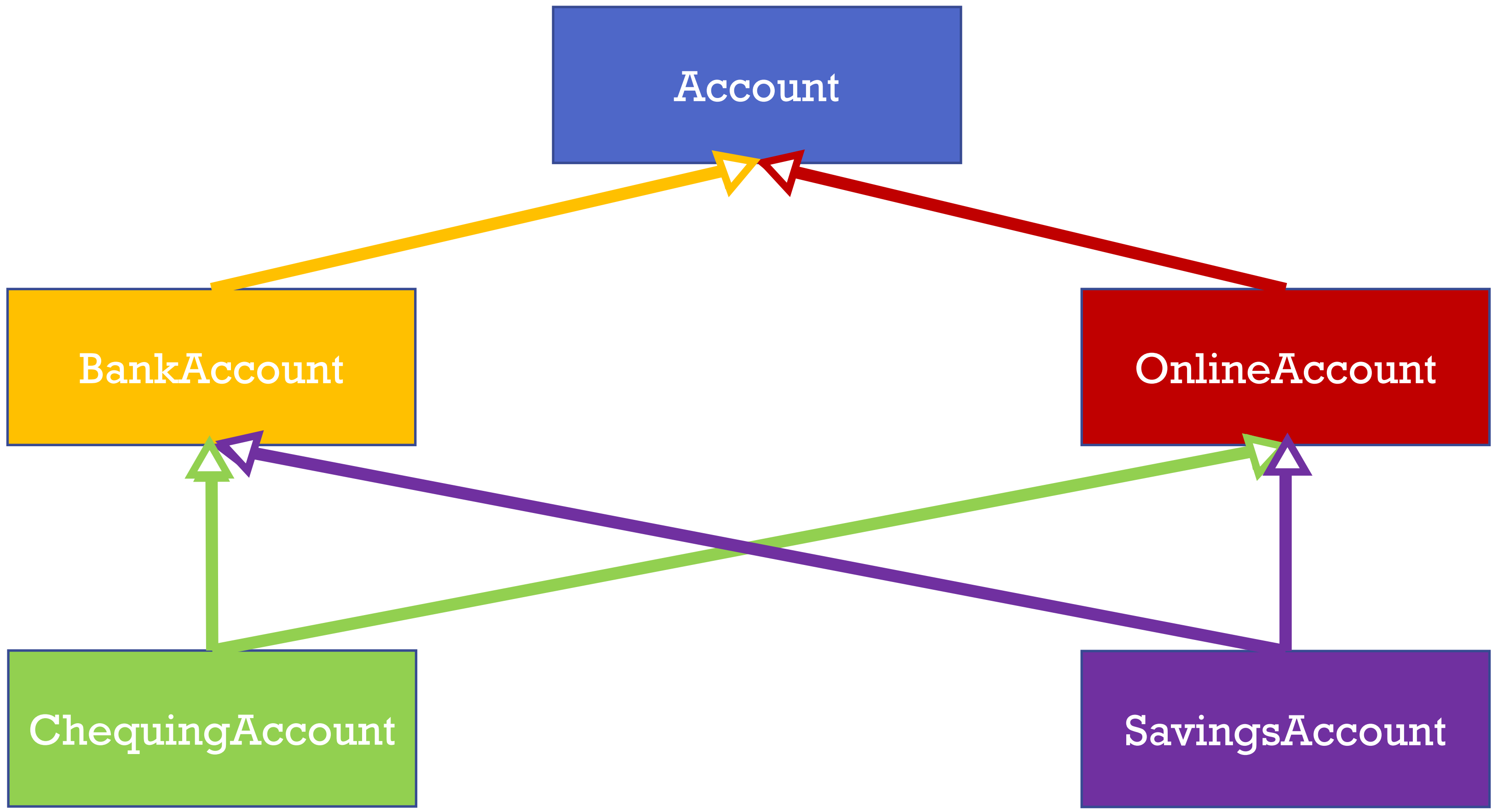
- Casts a pointer/reference of one type to a pointer/reference of another type **within an inheritance hierarchy**.
- Performs a **run-time test** to determine whether the actually casted object has the target type or a sub-type thereof
- Allowed with **pointers** **and** **references** **to polymorphic types** (recall polymorphic types are types with at least one virtual member function)
  - **Failure** to cast **pointer** target type returns **nullptr**
  - **Failure** to cast **reference** target type throw **bad_cast exception**

Dynamic cast example

- Suppose we have an **Account** class
- Account is derived by **BankAccount** and **OnlineAccount**
- **ChequingAccount** has 2 base classes, **BankAccount** and **OnlineAccount**
- **SavingsAccount** has 2 base classes, **BankAccount** and **OnlineAccount**

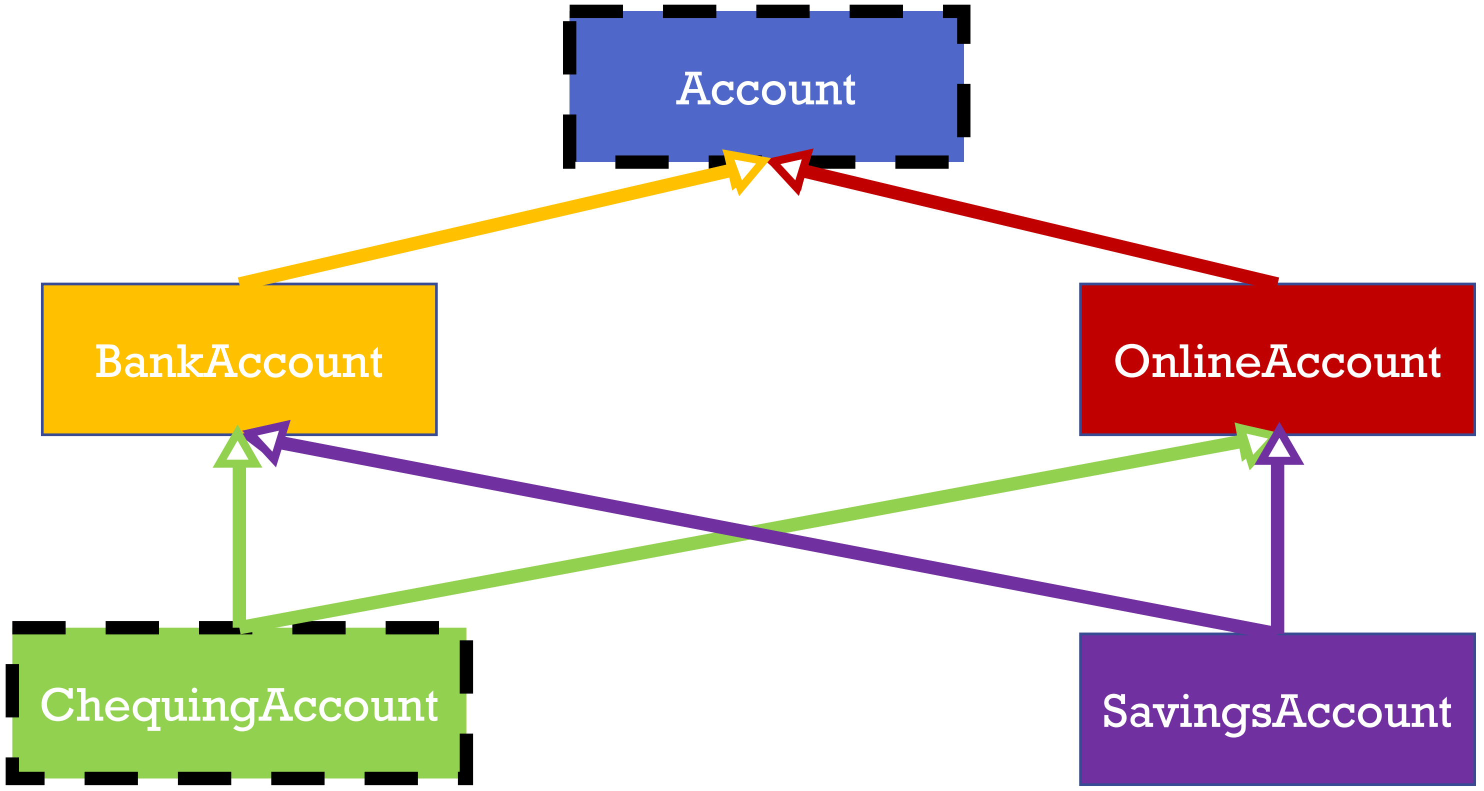


Dynamic cast example



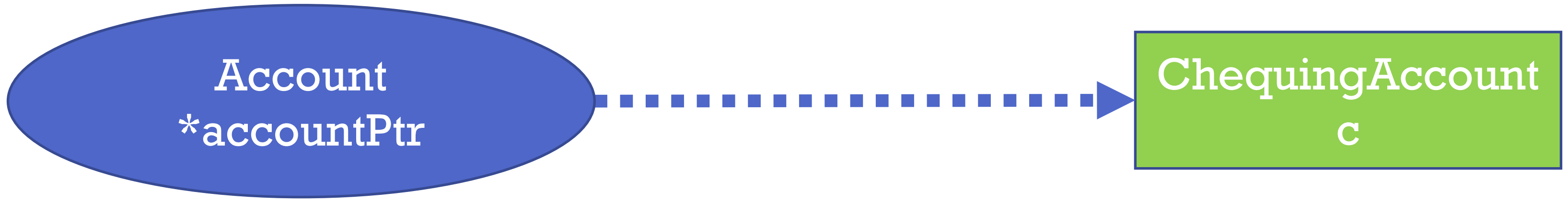
1. What kinds of pointers are allowed to point at which objects?
2. What kinds of casts are allowed?

Upcast

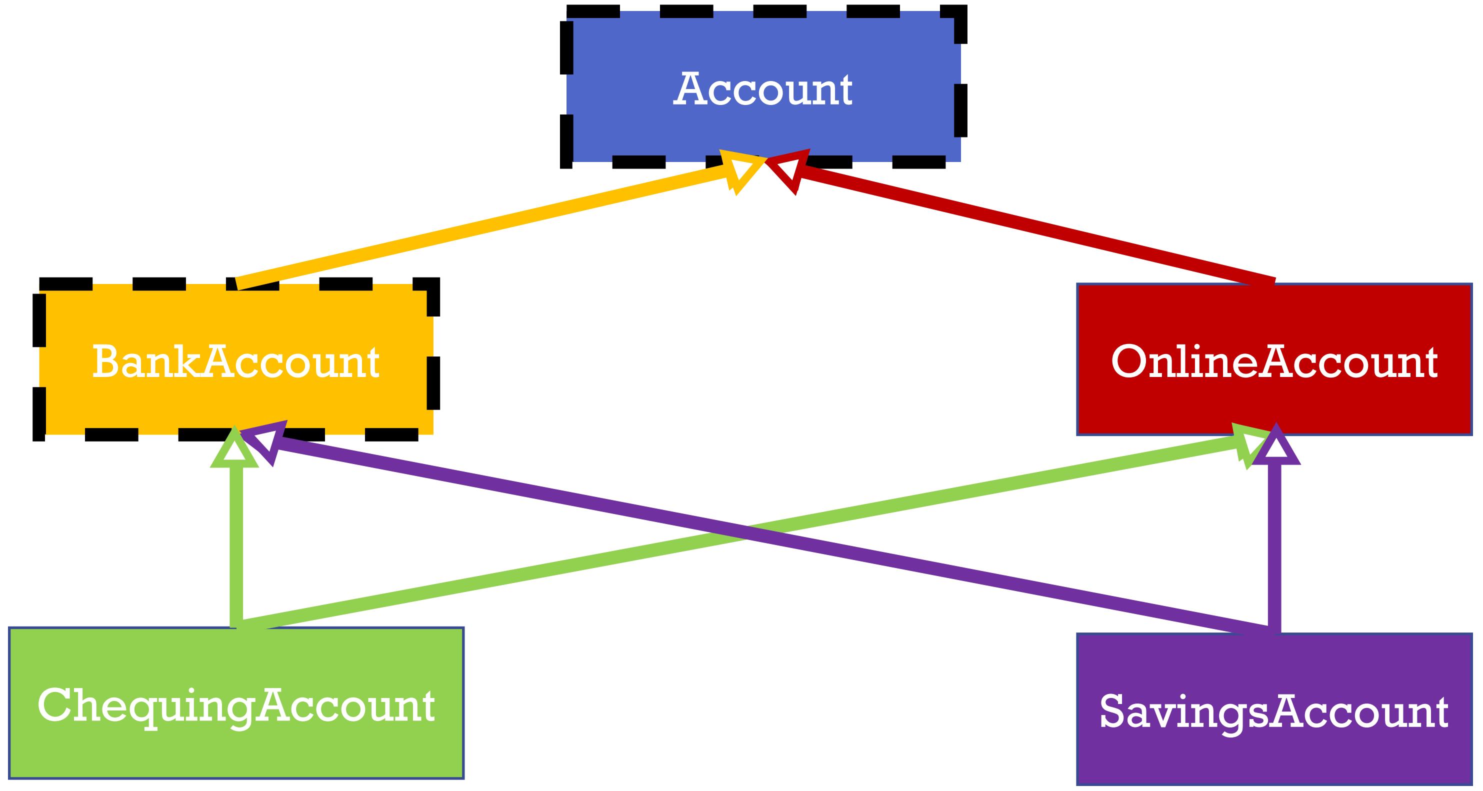


- ChequingAccount c;
- Account * accountPtr = dynamic_cast<Account *>(&c);

// Pointer upcast

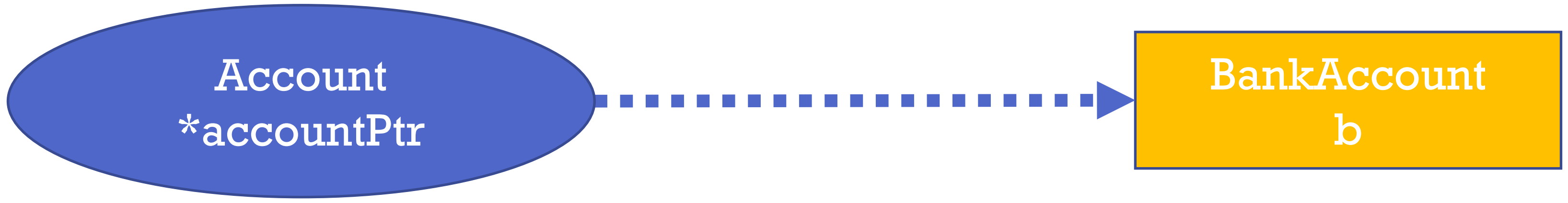


Upcast

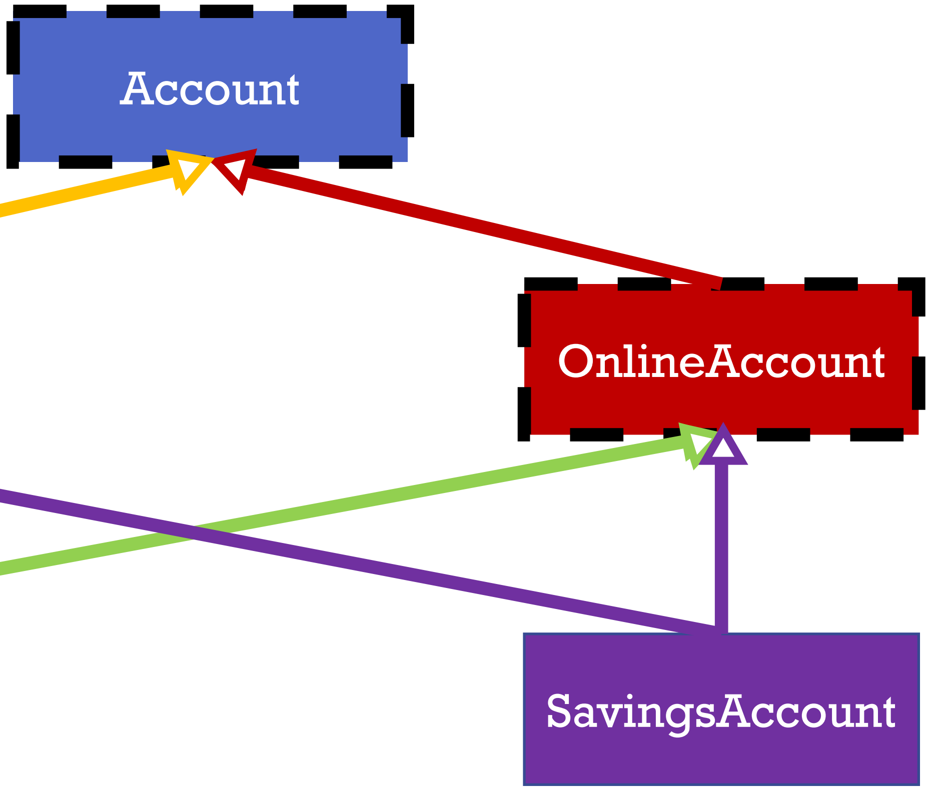


- **BankAccount b**;
- **accountPtr** = dynamic_cast<**Account** *>(&**b**);

// Pointer upcast

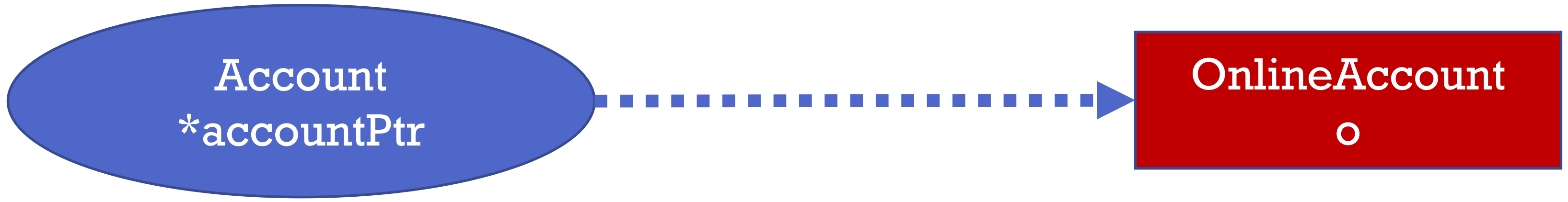


Upcast

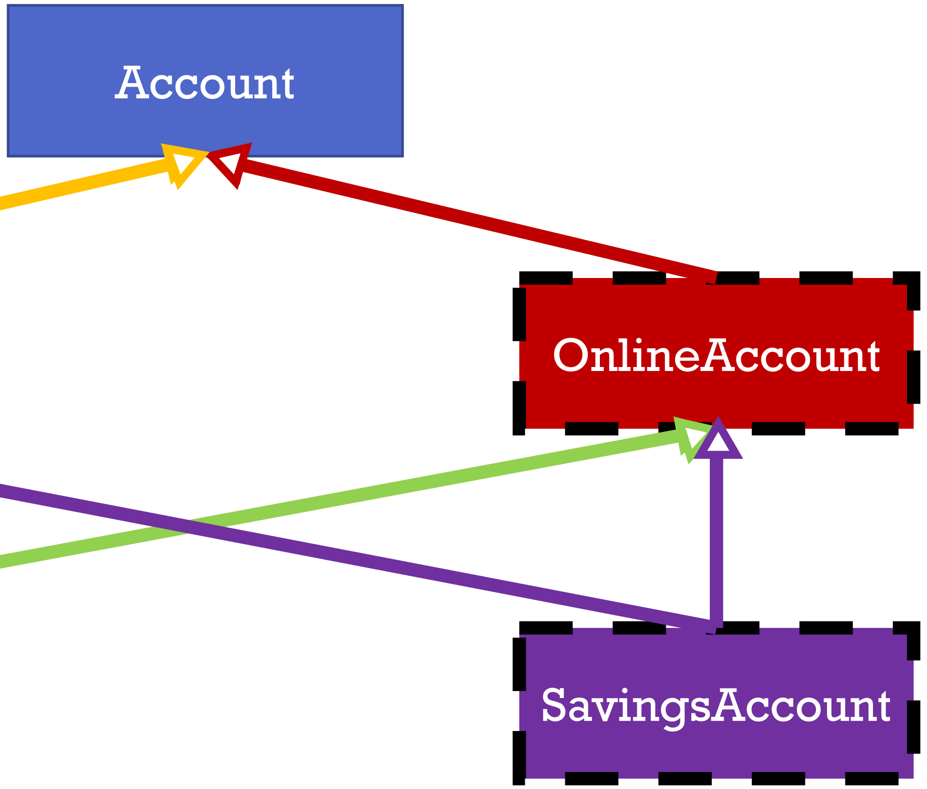


- **OnlineAccount o**;
- **accountPtr** = dynamic_cast<**Account** *>(&**o**);

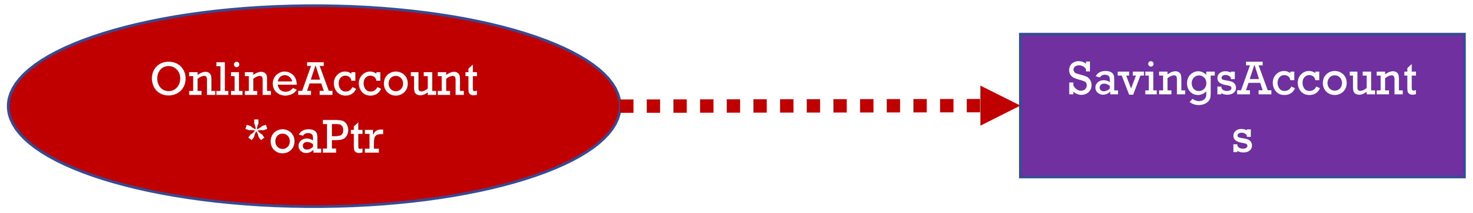
// Pointer upcast



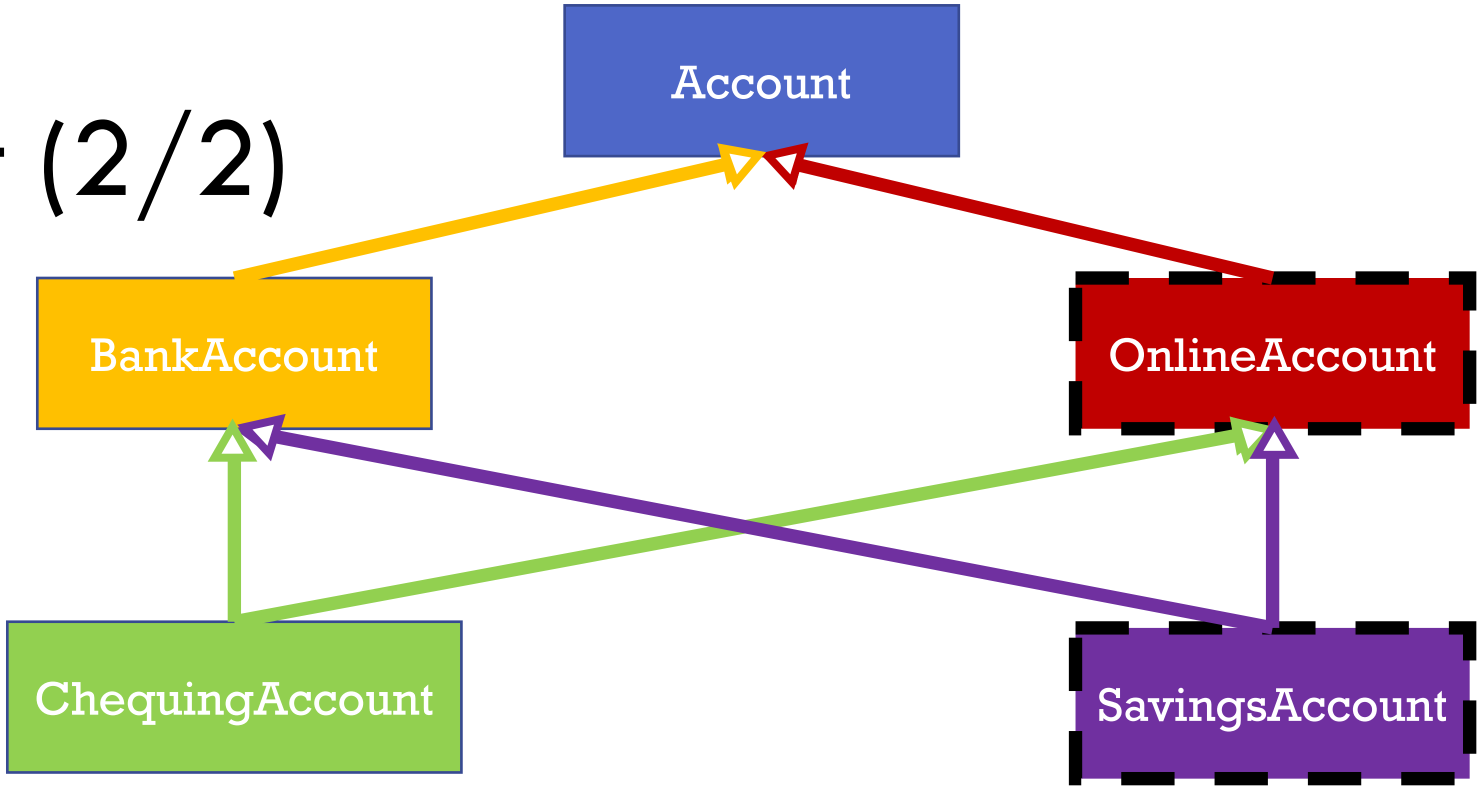
Downcast (1/2)



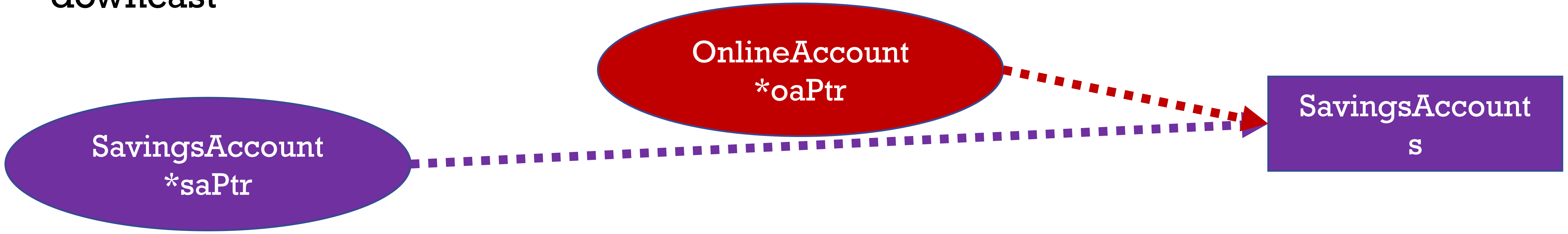
- **SavingsAccount s**;
- **OnlineAccount** * **oaPtr** = dynamic_cast<**OnlineAccount** *>(&**s**); // Pointer upcast



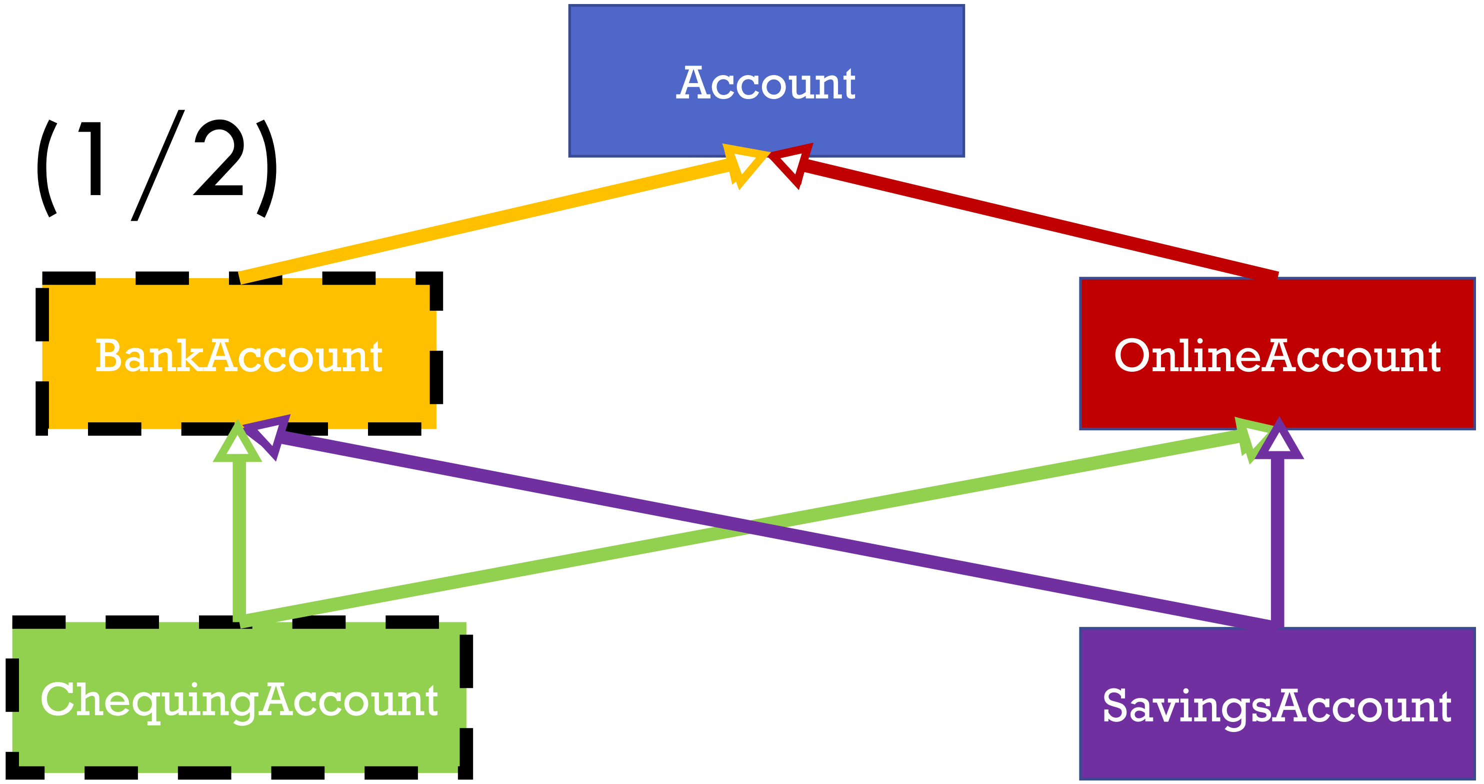
Downcast (2/2)



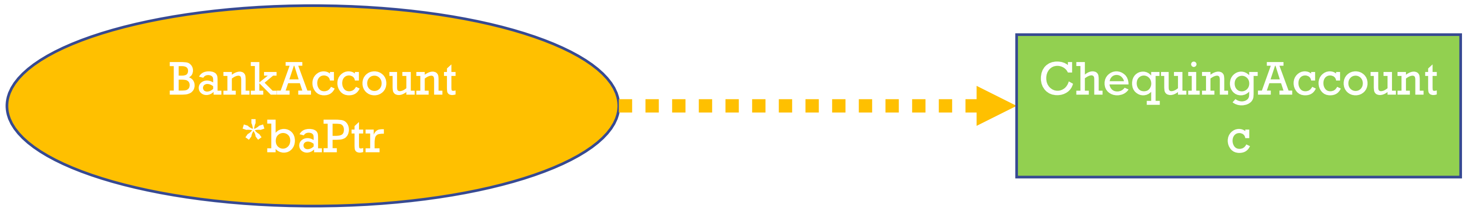
- **SavingsAccount s**;
- **OnlineAccount** * **oaPtr** = dynamic_cast<**OnlineAccount** *>(&**s**); // Pointer upcast
- **SavingsAccount** * **saPtr** = dynamic_cast<**SavingsAccount** *>(**oaPtr**); // Pointer downcast



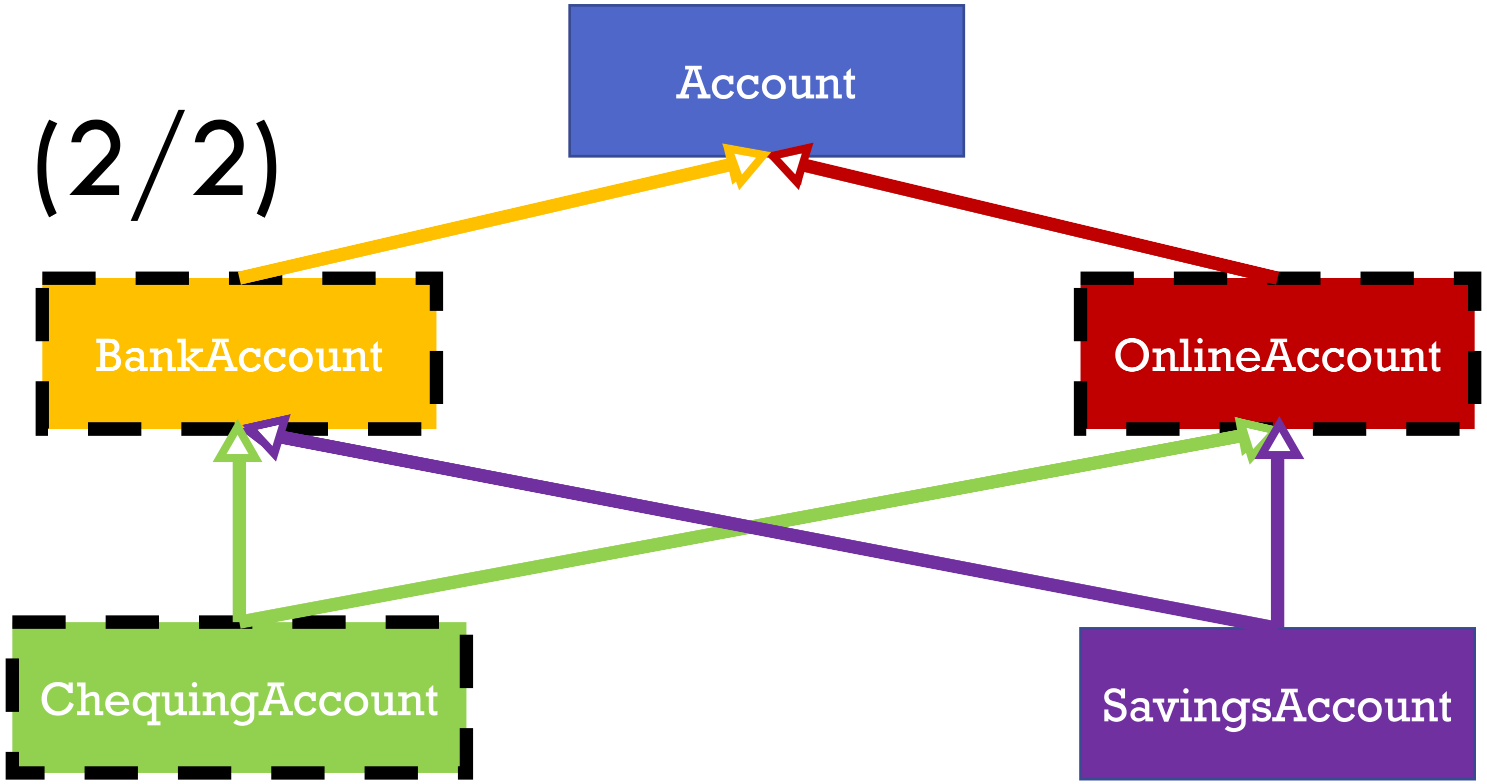
Crosscast (1/2)



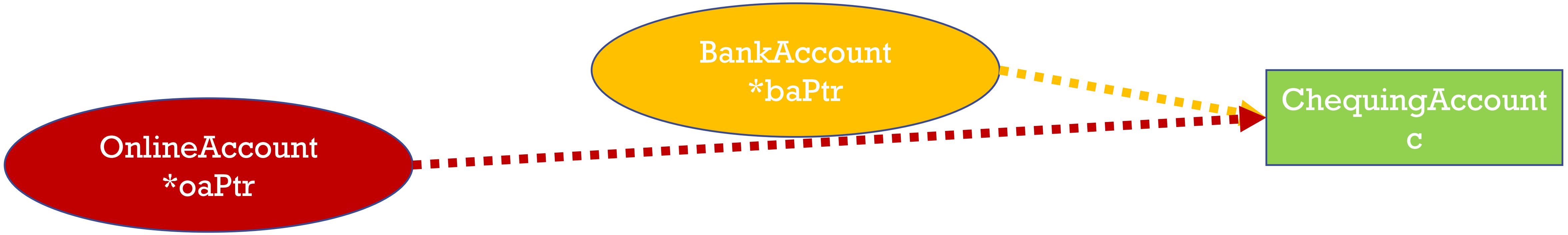
- **ChequingAccount c**;
- **BankAccount** * **baPtr** = dynamic_cast<**BankAccount** *>(&**c**); // Pointer upcast



Crosscast (2/2)



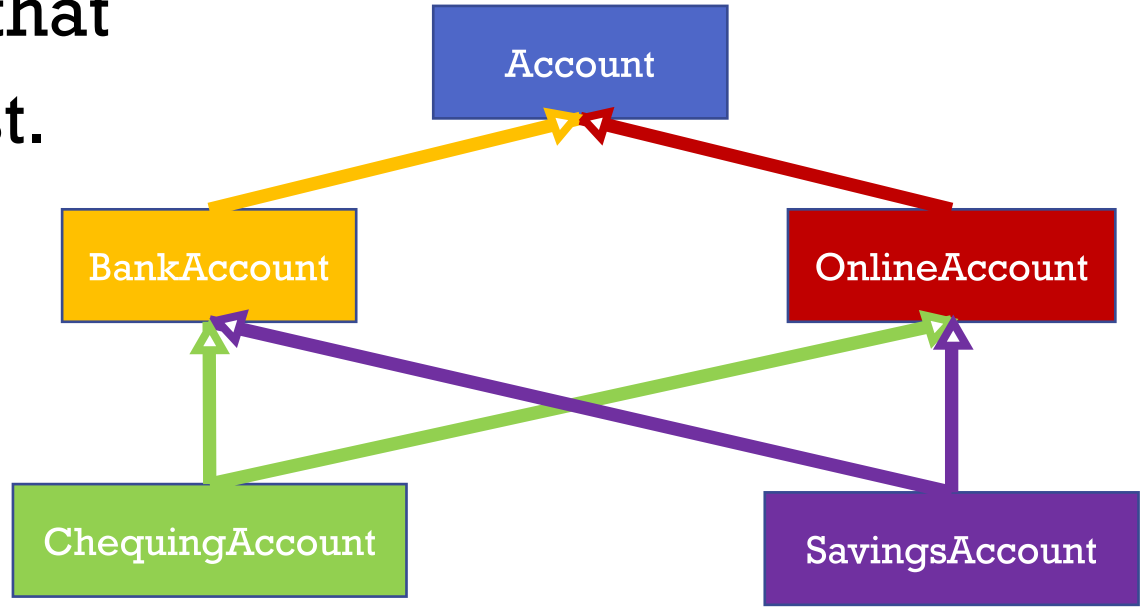
- ChequingAccount c;
- BankAccount * baPtr = dynamic_cast<BankAccount *>(&c); // Pointer upcast
- OnlineAccount *oaPtr = dynamic_cast<OnlineAccount *>(baPtr); // Pointer cross cast!



Dynamic cast example (upcast)

account.cpp

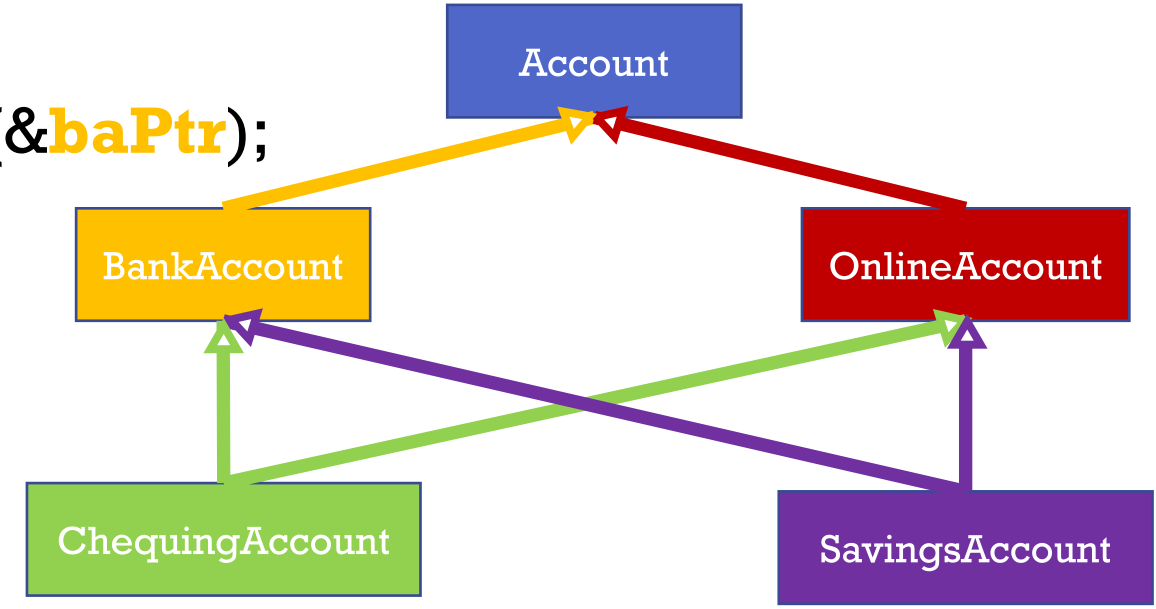
```
// Perform an upcast
accountPtr = dynamic_cast<Account *>(&c);

// Do the same thing again to show that
// no cast is required to do an upcast.
accountPtr = &c;
```



Dynamic cast example (downcast/cross cast)

account.cpp

```
// Perform a downcast
saPtr = dynamic_cast<SavingsAccount *>(oaPtr);
// Perform an upcast
BankAccount * baPtr = dynamic_cast<BankAccount *>(&c);
// Perform a cross cast
oaPtr = dynamic_cast<OnlineAccount *>(&baPtr);
```



We can say that the real purpose of dynamic-cast is to allow upcasts within an inheritance hierarchy

Note about dynamic cast

- **Only available when a class is polymorphic**
- Remember that in order to be polymorphic, a class must have at least one virtual function
- Compiler error if trying to dynamic cast classes that are not polymorphic
- **Pro tip: implement a virtual destructor with an empty implementation**

(Tangent: check out **typechecking.cpp** for a neat hack.)

STATIC CASTING

Static cast

- The static_cast operator is a compile time cast
- **Avoids the runtime checks done with dynamic_cast**
- As a result, you can use the static_cast operator with pointers and **nonpolymorphic** types
- **Only valid if the pre-cast type and the post-cast types can be implicitly converted to one another in 1 or both directions**
- You can also use it to carry out some of the conversions performed using C-style casts, generally conversions between related types
- The static_cast operator has the same syntax as dynamic_cast

ENUMS

I ♡ enumerations

```
const int RED = 0;
const int GREEN = 1;
const int BLUE = 2;
```

Is there an easier way to create a series of named constants???

I ♡ enumerations

- Enumerations restrict values to a specific named range of constants
- Underlying type is integral (like Java!)
- C++ enumerations can be **unscoped** or **scoped**
  - **Unscoped** are defined with **enum**
  - **Scoped** are defined with **enum class** or **enum struct**

I ♡ enumerations

```
const int RED = 0;
const int GREEN = 1;
const int BLUE = 2;
```

vs

```
enum color { RED, GREEN, BLUE }; //same result, less code
```

The C++ enumeration

- A user-defined type that contains a set of **named integral constants** that are known as **enumerators**.

- **Unscoped**: **enum** [**identifier**] [: **type**] { enum-list };
  - `enum color : int {RED, GREEN, BLUE}`
- **Scoped**: enum **[class | struct]** [identifier] [:**type**] { enum-list };
  - `enum class color : int {RED, GREEN, BLUE}`

- Visible in the scope in which they are declared
- Every name in the enum-list is assigned an integral value that corresponds to its place in the order of the enumeration.
- Pro-tip: enums are int by default, so it can be omitted
  - `enum color` ~~`: int`~~ `{RED, GREEN, BLUE}`

Enumerations

We can use **named constants** like this:

```
enum color { RED, GREEN, BLUE };
color r = RED;
switch(r)
{
case RED  : std::cout << “RED\n";   break;
case GREEN: std::cout << “GREEN\n"; break;
case BLUE : std::cout << “BLUE\n";  break;
}
```

Enumerations

- If not provided, the values of the enumerations begin at 0 (just like Java)
- But in C++ **we can provide the underlying values**:

```
// a = 0, b = 1, c = 10, d = 11, e = 1
// f = 2, g = 12
enum foo
{ a, b, c = 10, d, e = 1, f, g = f + c };
```

Enumerations

- Values can be **converted to integral types**:

```
enum color { RED, YELLOW, GREEN = 20, BLUE };
color my_color = RED;
int n = BLUE; // n == 21
```

Enumerations

- Values of integer, floating-point, and other enumeration types can be **converted by static_cast** or explicit cast, to any enumeration type

```
enum status { OPEN = 1, CLOSED = 2, ERROR = 3 };
status file_status = static_cast<status>(3);
cout << file_status << endl;
```

Enumerations

- We can **omit the name** of the enumeration
- We can ONLY use the enumerators in the enclosing scope

```
// defines a = 0, b = 1, c = 0, d = 2
enum { a, b, c = 0, d = a + 2};

// prints 0
cout << a << endl;
```

Enumerations

```
struct X
{
  enum direction { LEFT = 'l’, RIGHT = 'r' };
};

X x;
X* p = &x;
int a = X::direction::LEFT;
int b = X::LEFT;
int c = x.LEFT;
int d = p->LEFT;
```

SCOPED ENUMS

What about scoped enumerations?

- So far we've looked at unscoped enumerations
- There are **no implicit conversions** from the values of a **scoped enumerator** to integral types
- **static_cast** may be used to obtain the numeric value of the enumerator
- Advantage: **no namespace conflicts**

Scope Example

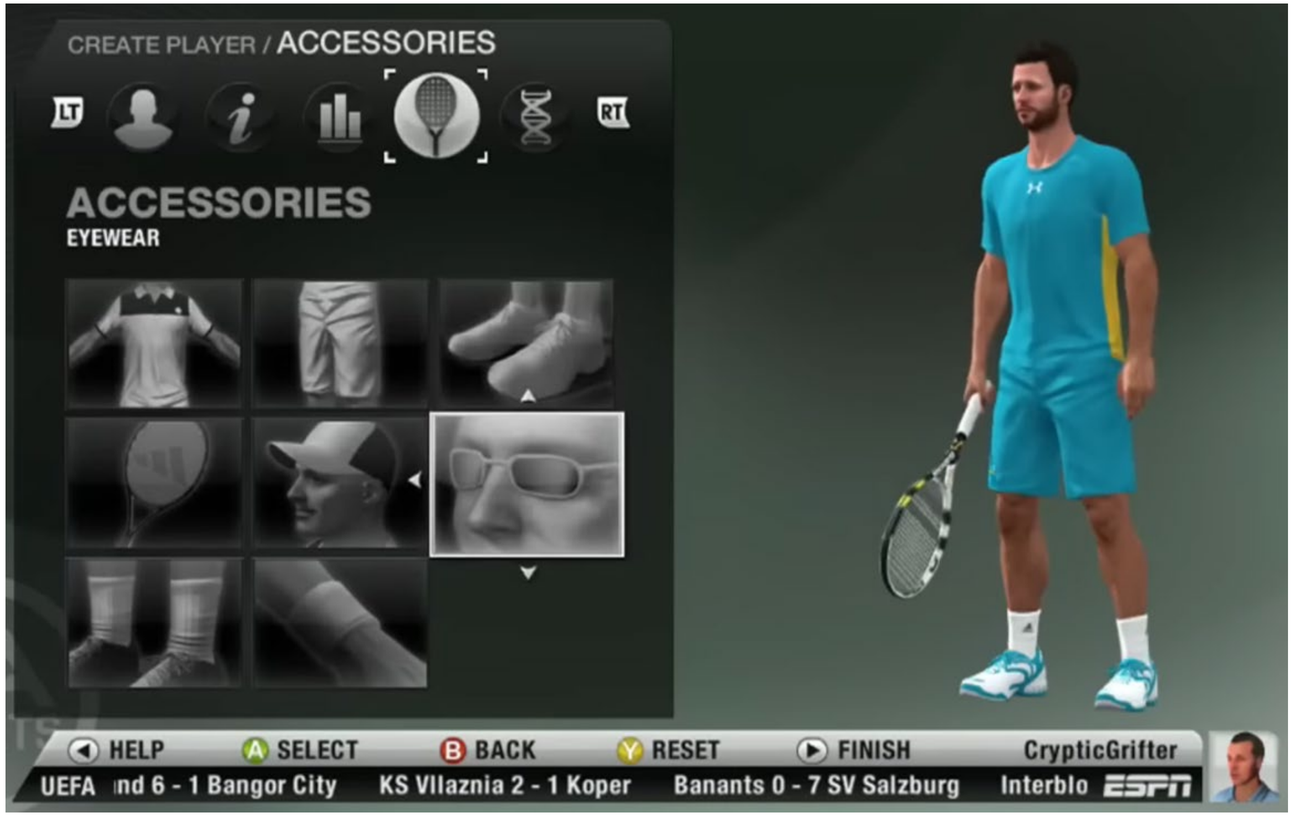
```
enum Animals {DOG, CAT, CROW};
enum FlyingAnimals {CROW, SPARROW}; //error CROW
already exists

//Compared to scoped enums
enum class Fruit {APPLE, ORANGE, LEMON}; //LEMON in
Fruit scope
enum class YellowFruit {BANANA, LEMON}; //LEMON in
YellowFruit scope OK!
```

Example

```
enum class color { RED, GREEN = 20, BLUE };
color r = color::BLUE;
switch(r){
    case color::RED  : cout << “RED\n";   break;
    case color::GREEN: cout << “GREEN\n"; break;
    case color::BLUE : cout << "BLUE\n";  break;
}
int n = r; // error: no scoped enum to int conversion
int n = static_cast<int>(r); // OK, n = 21
```

Real world example: Representing clothing in a videogame

- Need to have a finite number of clothing options in your game
- Each piece of clothing needs to have a unique id
- There must be a way to find the number of:
    - Only shoes
    - Only pants
    - All items
- How can we achieve this with only enums?


CREATE PLAYER / ACCESSORIES
LT
RT
ACCESSORIES
EYEWEAR
HELP
SELECT
BACK
RESET
FINISH
CrypticGrifter
UEFA ind 6 - 1 Bangor City KS Vllaznia 2 - 1 Koper Banants 0 - 7 SV Salzburg Interblo
ESPN

https://www.youtube.com/watch?v=01Gf3Oqfb3Q&t=420s

Real world example: Representing clothing in a videogame

```
enum clothingId {
   SHOE_1, //0
   SHOE_2, //1
   SHOE_3, //2
   PANT_1, //3
   PANT_2, //4
   PANT_3, //5
   PANT_4, //6
   PANT_5  //7
};
```

- All clothing items have a unique id
- But there's no way to:
  - Get the total number of unique clothes
  - Get only the shoes
  - Get only the pants

Real world example: Representing clothing in a videogame

```
enum clothingId {
    SHOE_BEGIN,                 //0
    SHOE_1 = SHOE_BEGIN,        //0
    SHOE_2,                     //1
    SHOE_3,                     //2
    SHOE_END = SHOE_3,          //2
    PANT_BEGIN,                 //3
    PANT_1 = PANT_BEGIN,        //3
    PANT_2,                     //4
    PANT_3,                     //5
    PANT_4,                     //6
    PANT_5,                     //7
    PANT_END = PANT_5,          //7
    NUM_CLOTHES                 //8
};
```

- Create "**Bookmarks**" in the enum list
    - **SHOE_BEGIN**, **SHOE_END**, **PANT_BEGIN**, **PANT_END**, **NUM_CLOTHES** indicate special points in the enum list
    - Most have the same value as an existing enum
- Use these "**Bookmarks**" as points to separate shoes and pants
- We can now:
    - Get the total number of unique clothes
    - Get only the shoes
    - Get only the pants

clothingEnum.cpp

Agenda

Design patterns intro

1. Singleton
2. Observer

COMP 3522

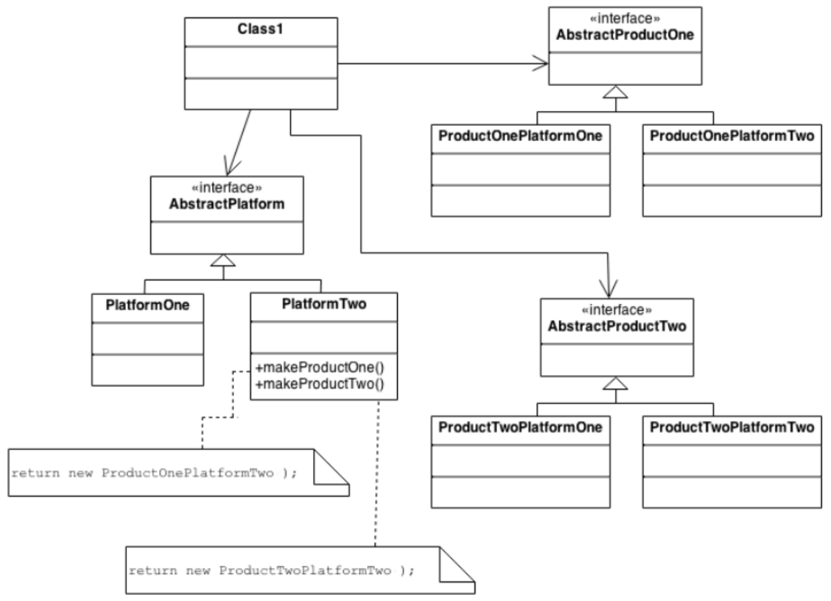
DESIGN PATTERNS

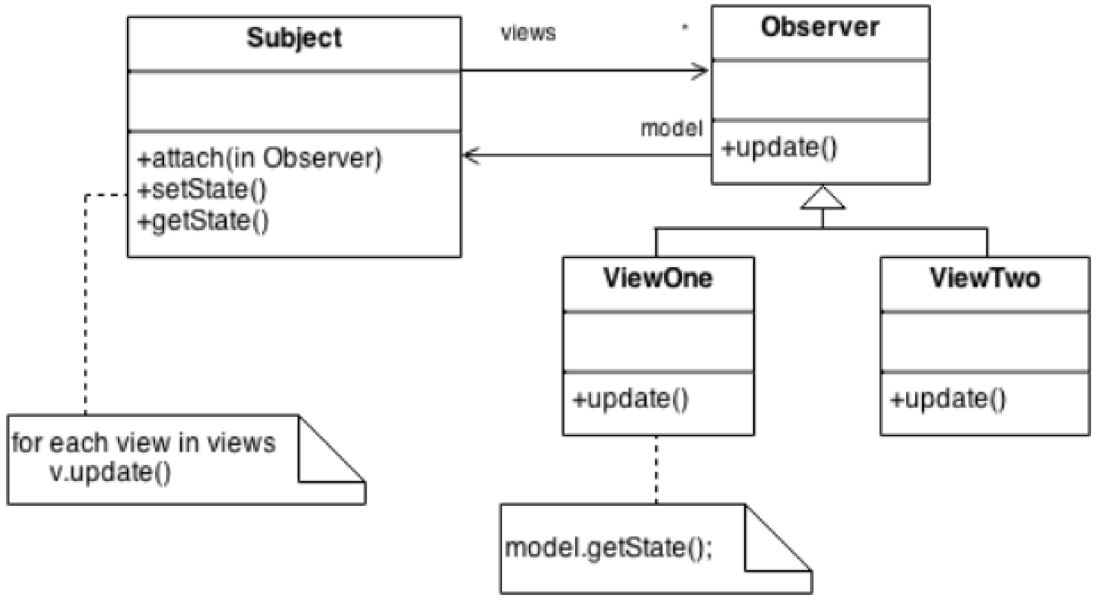
What are Design Patterns

- Common design solutions to common architectural problems
- How can I write systems so Classes can:
  - **communicate** with each other with low coupling?
  - be combined to form new **structures?**
  - be **created** with different strategies and techniques?

What are Design Patterns

- Think of these as recipes or templates to solve common design problems
- "Structuring code/classes in this way will solve specific design issues"





Design Patterns - Advantages

- Don’t re-invent the wheel, use a proven solution instead
- Are abstract, and can be applied to different problems
- Communicate ideas and concepts between developers
- Language agnostic. Can be applied to most (if not all) OOP programs.

Design Patterns - Disadvantages

- Can make the system more complex and harder to maintain. Patterns are deceptively 'simple'.
- The system may suffer from pattern overload.
- All patterns have some disadvantages and add constraints to a system. As a result, a developer may need to add a constraint they did not plan for.
- Do not lead to direct code re-use.

Categorizing Design Patterns

- **Behavioural**

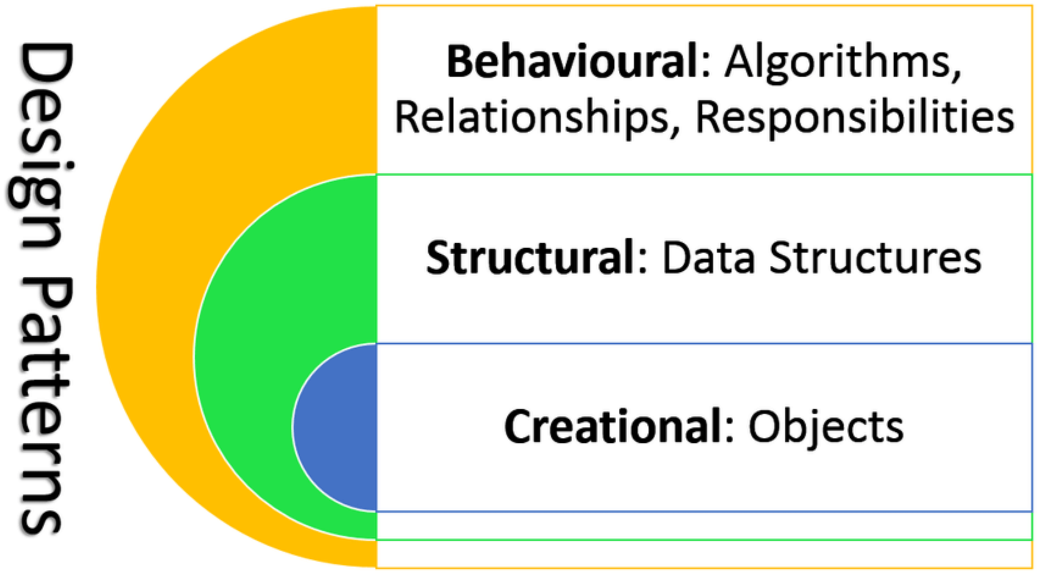
Focused on **communication and interaction between objects.** How do we get objects talking to each other while minimizing coupling?

- **Structural**

How do classes and objects **combine to form structures** in our programs? Focus on architecting to allow for maximum flexibility and maintainability.

- **Creational**

All about class instantiation. Different **strategies and techniques to instantiate an object, or group of objects**



Design Patterns
Behavioural: Algorithms, Relationships, Responsibilities
Structural: Data Structures
Creational: Objects

Picking a Pattern

Step 1	• Understand the problem you are facing in terms of dependencies, modularity and abstract concepts.
Step 2	• Identify if this is a behavioural, structural or creational issue?
Step 3	• Are there any constraints that I need to follow?
Step 4	• Is there a simpler solution that works? If not, pick a pattern.

SINGLETON

Introduction

- I need a system where there is a single object that is accessible from anywhere in the code

- **How do I do this without global variables?**

Singleton design pattern: a really easy one!

- **Design pattern category: Creational**
- Sometimes we want to guarantee that only a **single instance** of a class will ever exist
- We want to **prevent more than one copy from being constructed**
- We must write code that enforces this rule
- We want to employ the **Singleton Design Pattern**

Singleton pattern

1. **Instantiates** the object on its first use
2. **Ideally hides** a private constructor
3. **Reveals** a public `get_instance` function that returns a reference to a static instance of the class
4. **Provides** "global" access to a single object

Why/how do we use it?

Use the singleton pattern **when you need to have one and <u>only one</u> object of a type** in a system.

Singleton is a globally accessible class where we guarantee only a single instance is created

<u>That's it.</u>

<u>Really, that's all there is to it.</u>

Code sample (so easy!)

```
class singleton
{
    public:
        static singleton& get_instance() //allows public access to singleton
        {
            static singleton instance; //static enforces only one singleton instance exists
            return instance; // Instantiated on first use.
        }
    private:
        int test_value;
        singleton() {} //hides constructor in private visibility

    public:
        singleton(singleton const&)       = delete; //prevents copying singleton
        void operator=(singleton const&)  = delete; //prevents copy assigning of singleton
        int get_value() { return test_value++; }
};
```

singleton.cpp

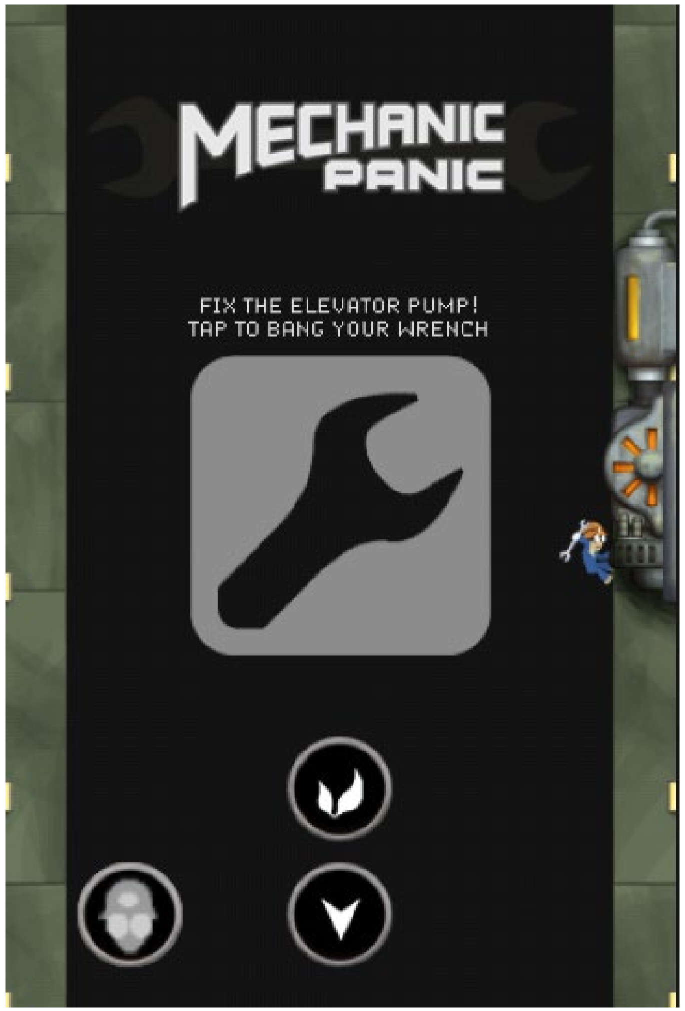
Application – Game screen management

- Game has multiple screens
  - Start, gameplay UI, game over, store, etc
- Different screens must be able to be displayed at various places in the code
  - Store logic wants to show store screens
  - Gameplay logic wants to show start/gameplay/game over
  - Settings logic wants to show settings screen
- Need a **central place** to call and load specific screens on demand

Application – Game screen management

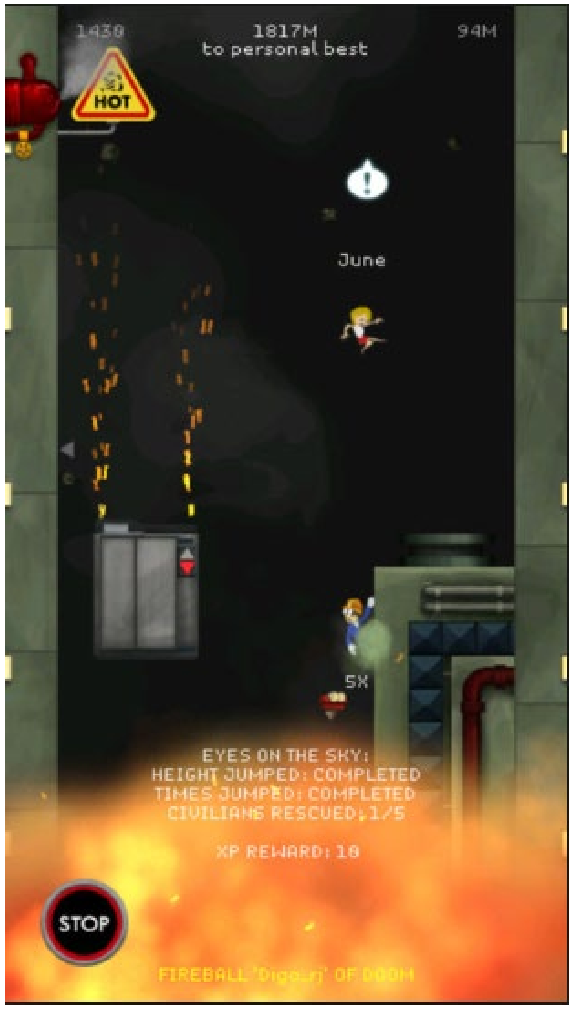
- Create enums for every game state: `MAIN_MENU`, `GAMEPLAY`, `GAME_OVER`
- Create Screen classes for different screens: MainMenu, Gameplay, GameOver
- Create a **ScreenManager singleton**
  - Has:
    - map of GameState enums to Screens
    - Screen stack
  - Responsible for pushing/popping screens off a Screen stack
- Have a function `show(GameState gs)` that accepts a GameState enum and pushes a screen from a map to the screen stack

Mechanic Panic – Singleton screens example



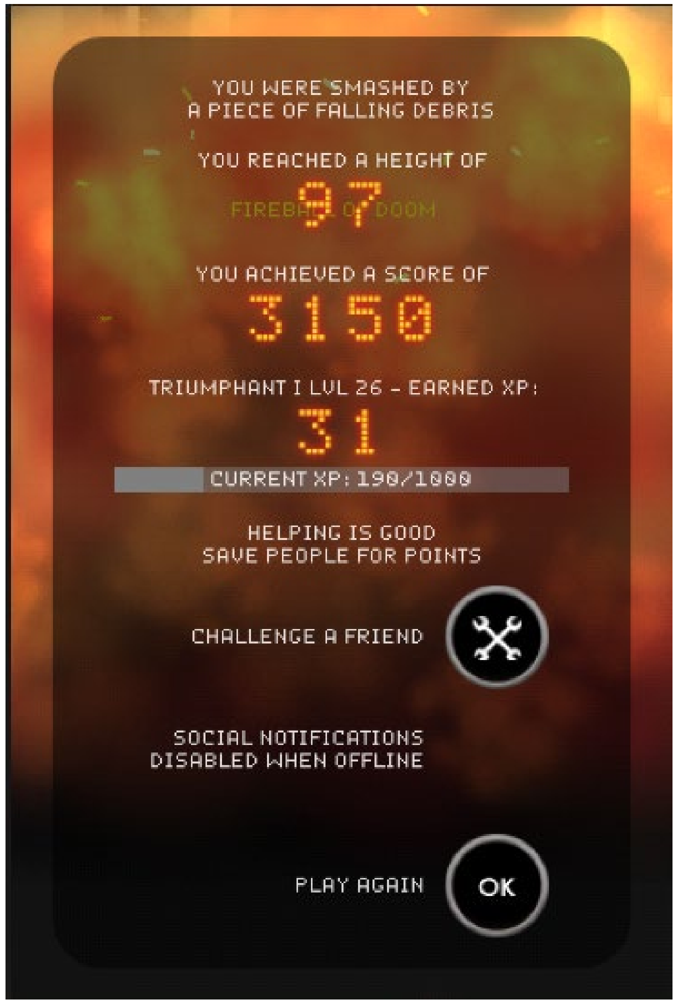
FIX THE ELEVATOR PUMP!
TAP TO BANG YOUR WRENCH

GameState enum: MAIN_MENU
ScreenManager::**getInstance()**.show(MAIN_MENU);



EYES ON THE SKY:
HEIGHT JUMPED: COMPLETED
TIMES JUMPED: COMPLETED
CIVILIANS RESCUED: 1/5
XP REWARD: 10

FIREBALL 'Digger' OF DOOM



YOU WERE SMASHED BY
A PIECE OF FALLING DEBRIS

YOU REACHED A HEIGHT OF
97
FIREBALL OF DOOM

YOU ACHIEVED A SCORE OF
3150

TRIUMPHANT I LVL 26 - EARNED XP:
31
CURRENT XP: 190/1000

HELPING IS GOOD
SAVE PEOPLE FOR POINTS

CHALLENGE A FRIEND

SOCIAL NOTIFICATIONS
DISABLED WHEN OFFLINE

PLAY AGAIN
OK

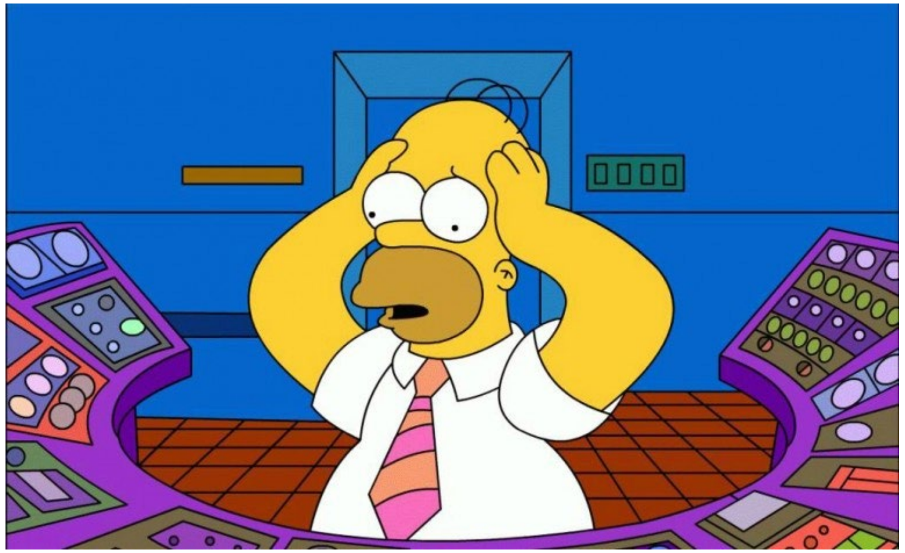
GameState enum: GAME_OVER
ScreenManager::**getInstance()**.show(GAME_OVER);

GameState enum: GAMEPLAY
ScreenManager::**getInstance()**.show(GAMEPLAY);

OBSERVER

Introduction

- I want to create a system where one object can **broadcast information** to **multiple objects**
- How do I notify a bunch of different kinds of objects if the state of one part of the system changes without **tightly coupling** that part of the system with the rest?



Introduction

- We like to partition our systems into cooperating classes
- Those classes share information
- We need to maintain consistency
- But we can't couple them tightly because that reduces their flexibility
- We use an idiom you will see often in programming called Publish-Subscribe:
    - The **Subject** publishes notifications without knowing who **observes**
    - Any number of **Observers** can subscribe to receive notifications*

* Sounds a little like Java GUI listeners to me!

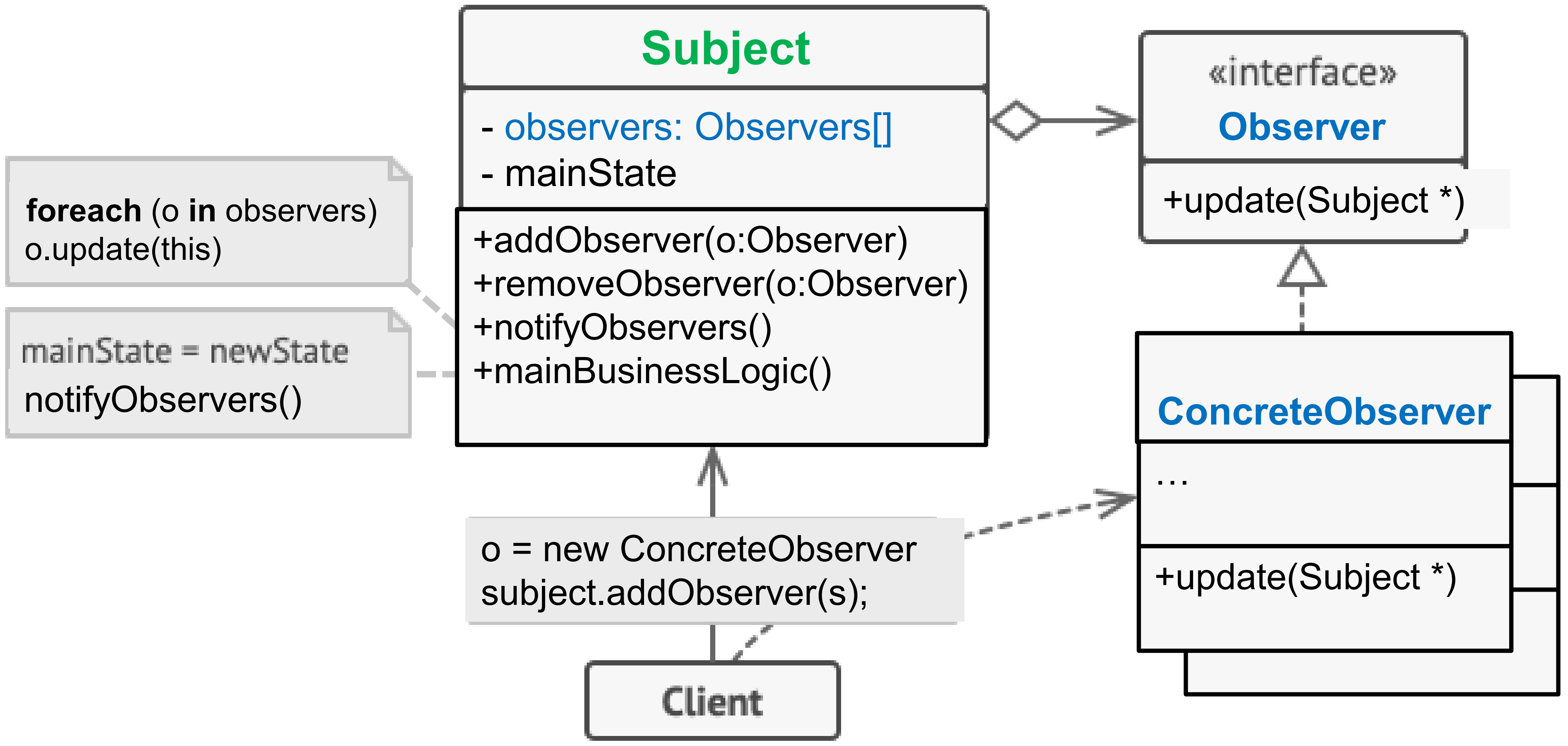
Observer design pattern

- **Design pattern category: Behavioral**
- The **Observer pattern** describes how to establish these relationships
- There are two key objects:
  1. **Subject** may have any number of dependent observers
  2. **Observers** are all notified whenever the subject undergoes a change of state
- Each observer queries the subject to synchronize their states
- This pattern ensures that when a subject changes state all its observers are automatically notified

Observer design pattern

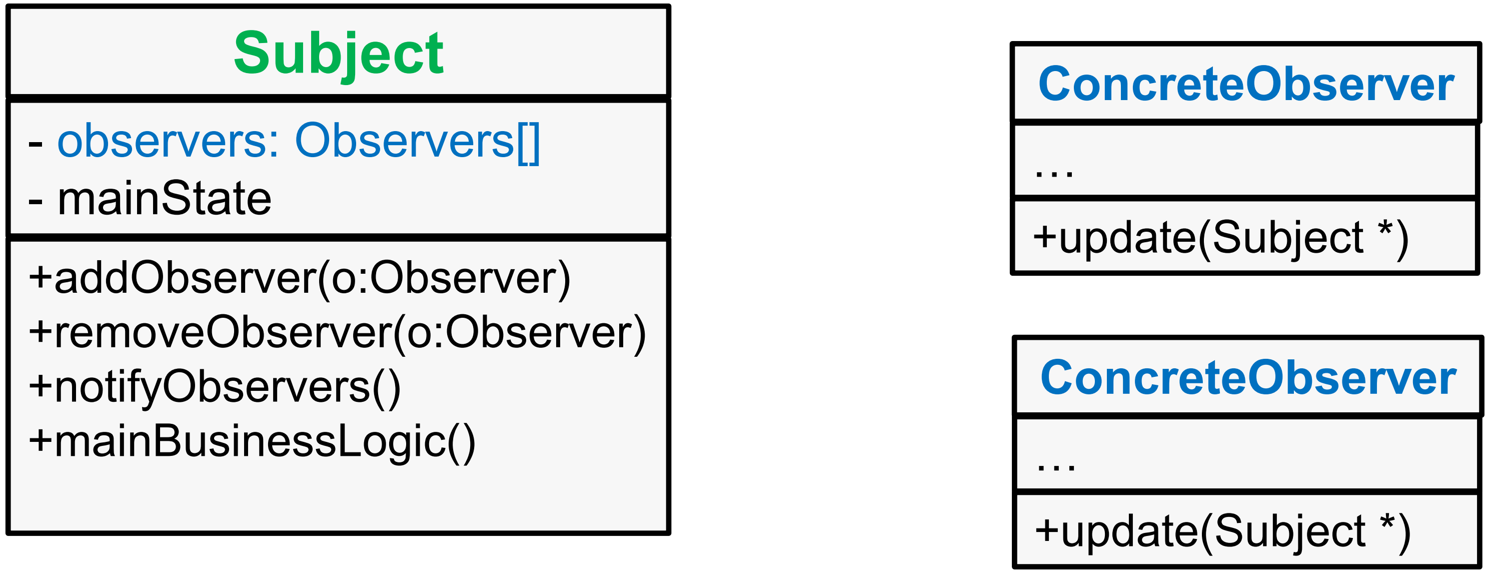
1. (Abstract) **Subject**
   - Knows its Observers
   - Any number of Observers may observe a subject
   - Provides an **interface for attaching and detaching** Observers
2. (Abstract) **Observer**
   - Defines an **updating interface** for objects that should be notified of changes in a subject
3. **ConcreteSubject**
   - Sends **notification** of its changed state to its observers
4. **ConcreteObserver**
   - Maintains a reference to a ConcreteSubject object
   - Stores state that needs to be consistent with the subject's
   - Implements the Observer updating interface

Observer design pattern: Class diagram



https://refactoring.guru/design-patterns/observer

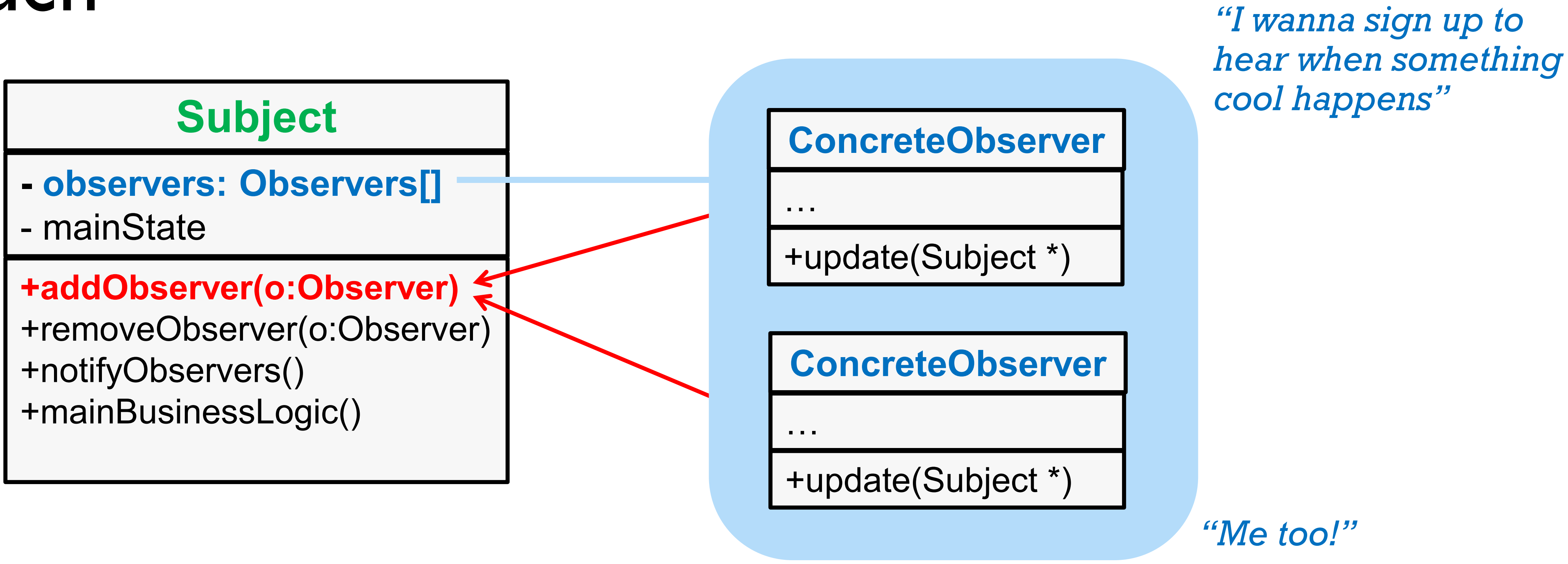
1. Create subject and observers



Instantiate subject, and 2 observer objects

https://refactoring.guru/design-patterns/observer

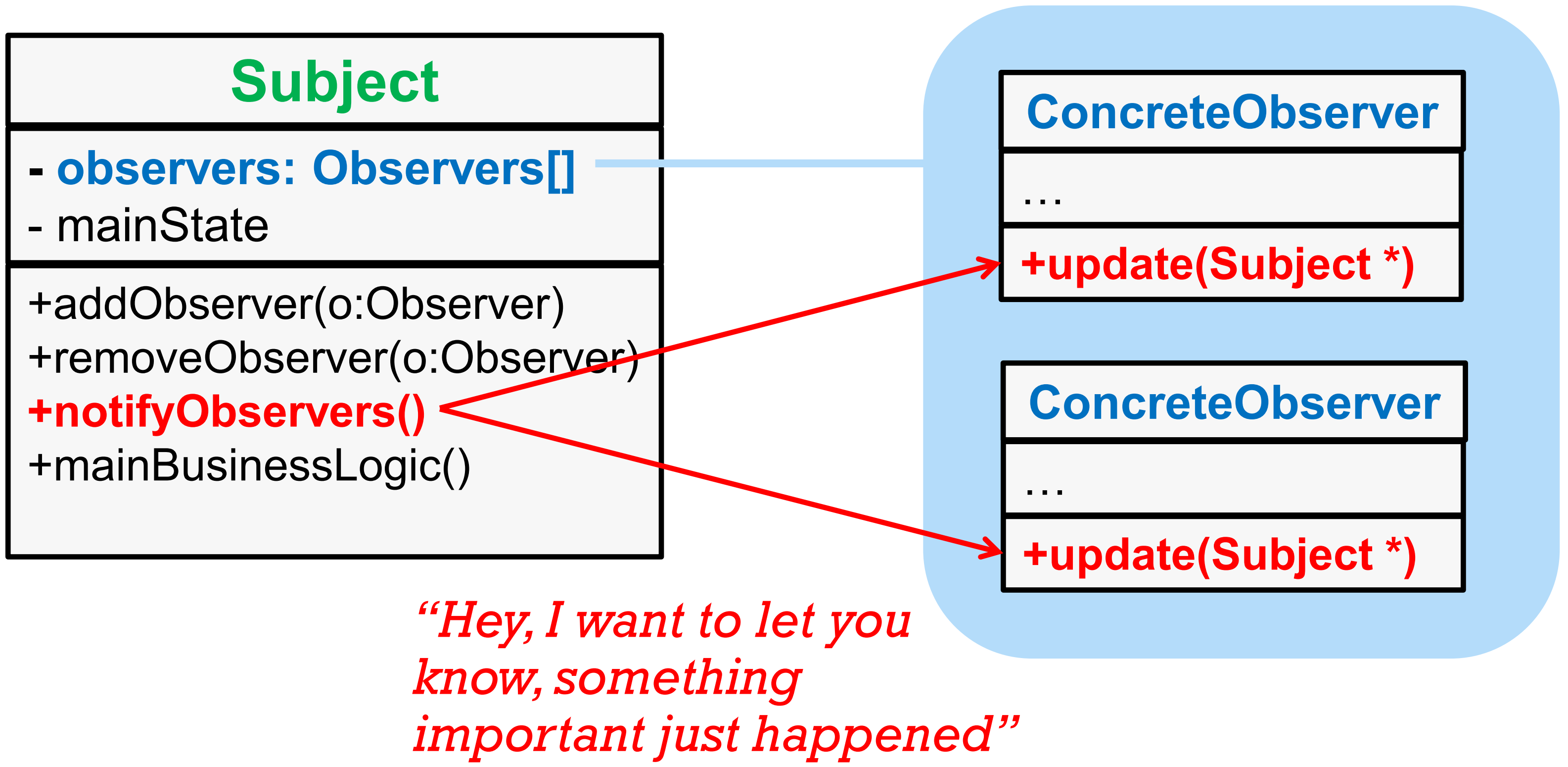
2. Attach



Observer objects can be added to a Subject to "listen in" to important events

https://refactoring.guru/design-patterns/observer

3. Update



Subject notifies observer objects when important event happens

https://refactoring.guru/design-patterns/observer

Use cases

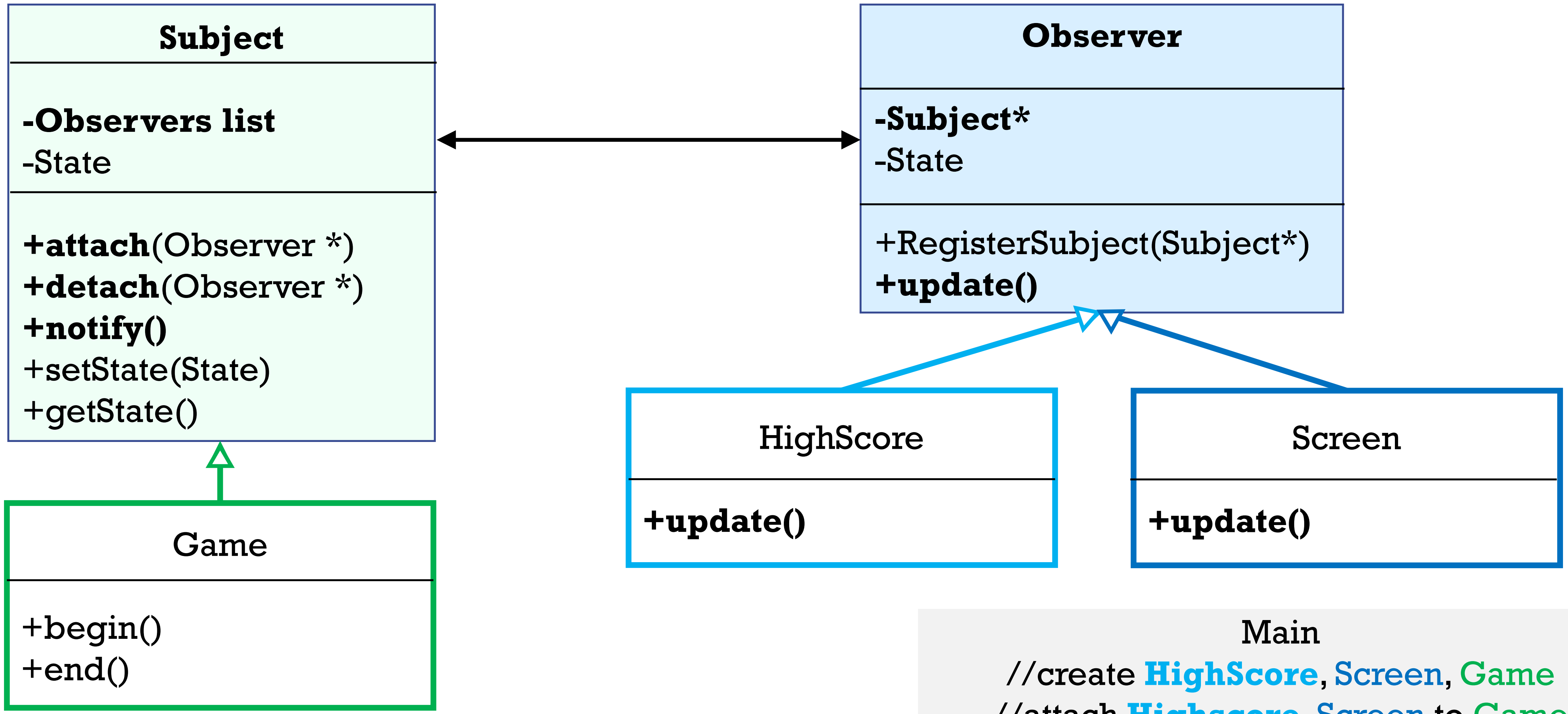
- Use the Observer pattern when:
  - A **change to one subject** requires **changing others**, and you don’t know how many objects needs to be changed
  - An **object** needs to **notify other objects** without making any assumptions about their concrete type (loose coupling!)
- I need the professor to be **notified** when a student joins his/her class
- I want the display to **update** when the size of a window is changed
- I need the schedule view to **update** when the database is changed

Game example

- Game class will notify observers when game begins, and game ends
  - Game begin – HighScore closed, game start screen shown
  - Game end – HighScore displayed, game end screen shown

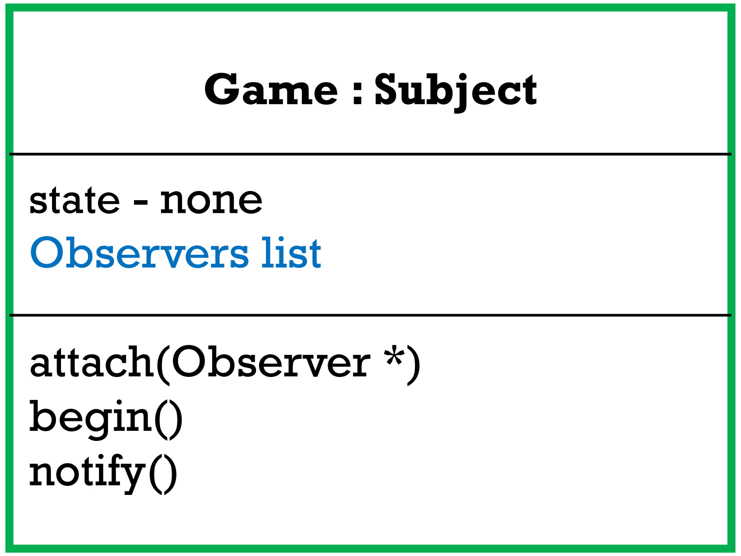
- **Game** class is a **Subject**
  - Subject contains a vector of observer pointers
  - Subject's **notify()** will call all observers' **update()**
- HighScore and Screen class are **Observers**
  - All observers have an **update()**
  - Perform own logic when this **update()** called by **Subject**

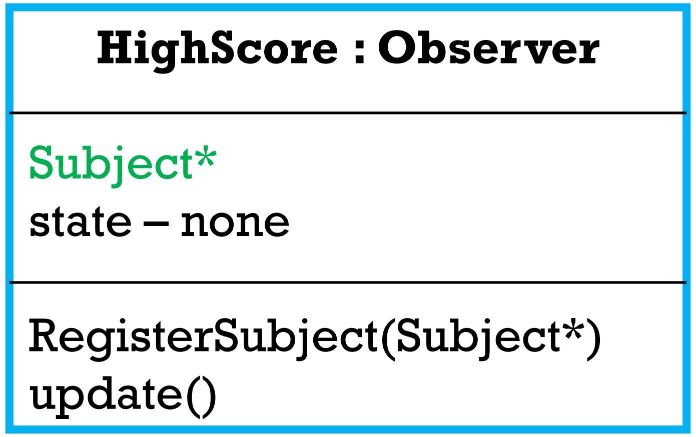
Game example

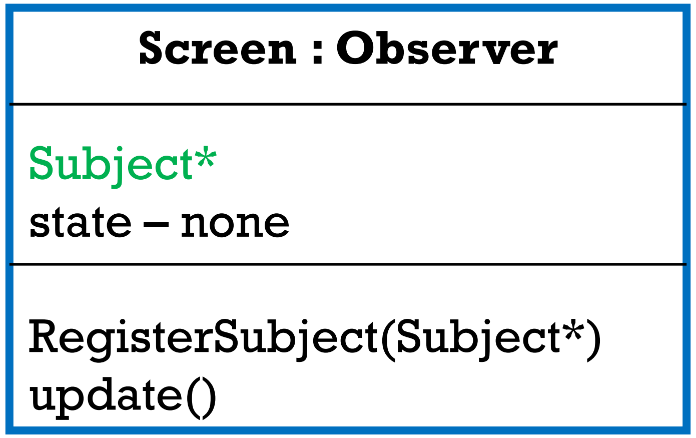


Main
//create **HighScore**, Screen, Game
//attach **Highscore**, Screen to Game
//call Game to begin and end

Game flow example

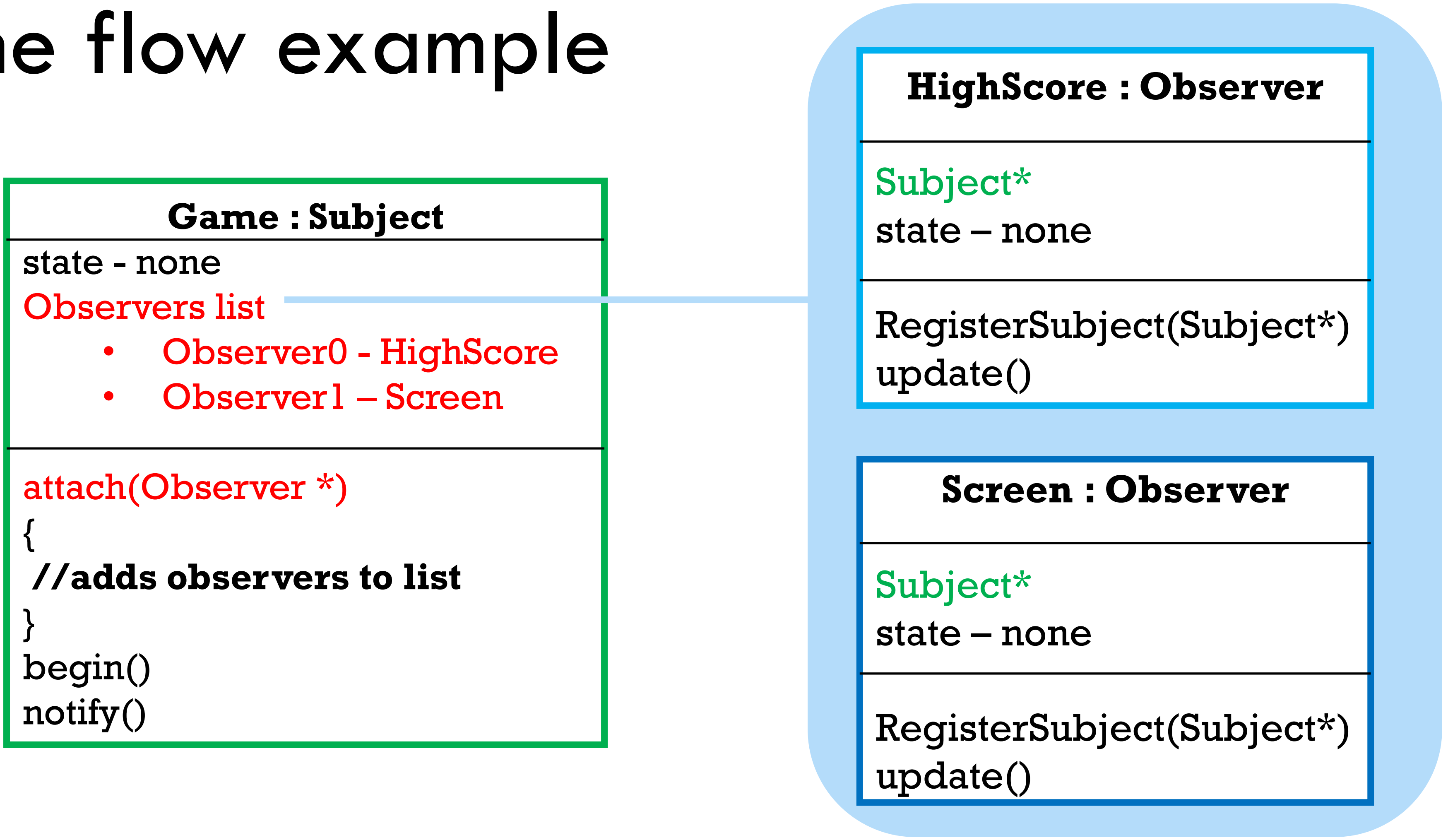






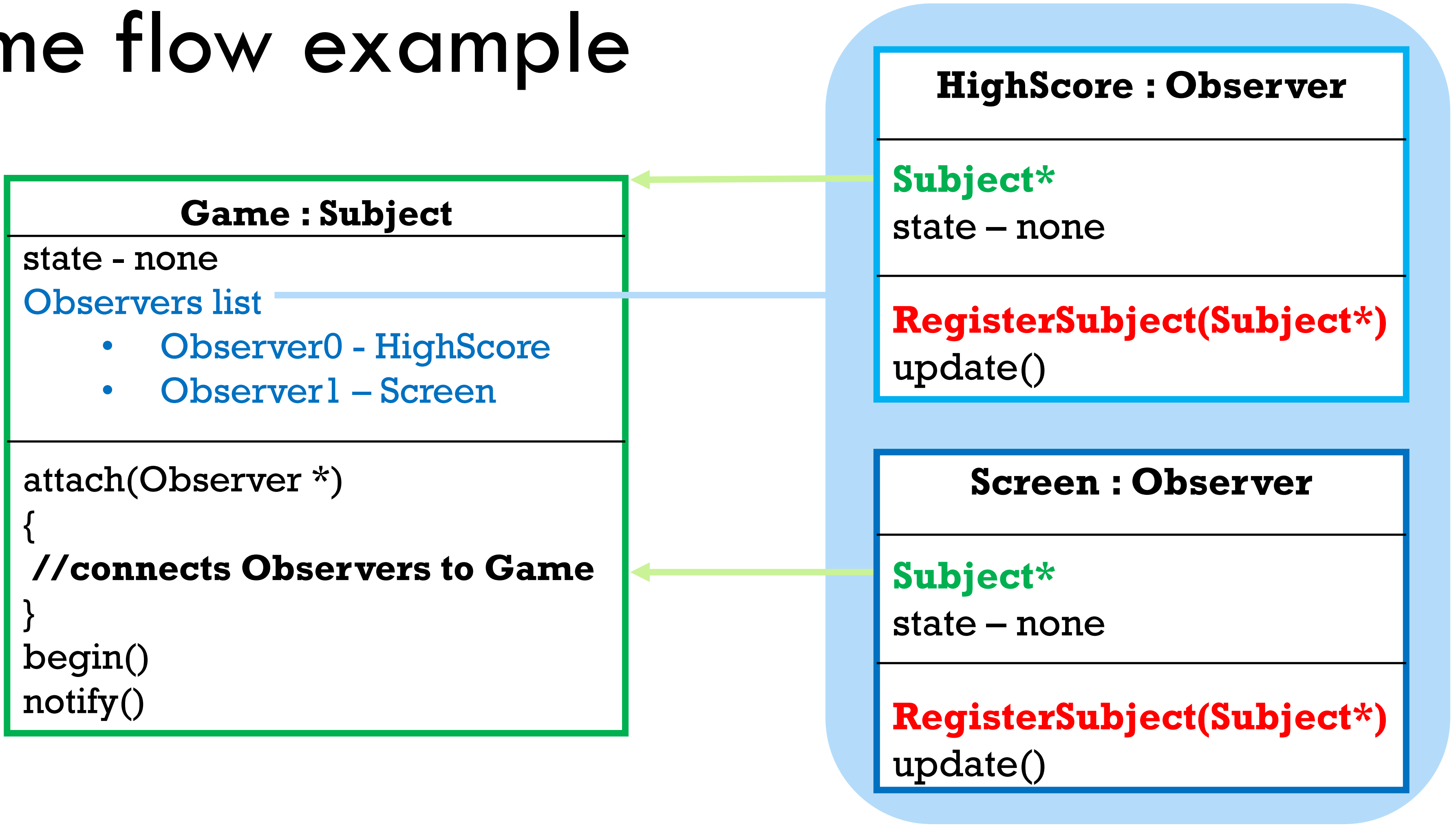
- Main
  - Create **Game**, **HighScore**, and **Screen** objects

Game flow example



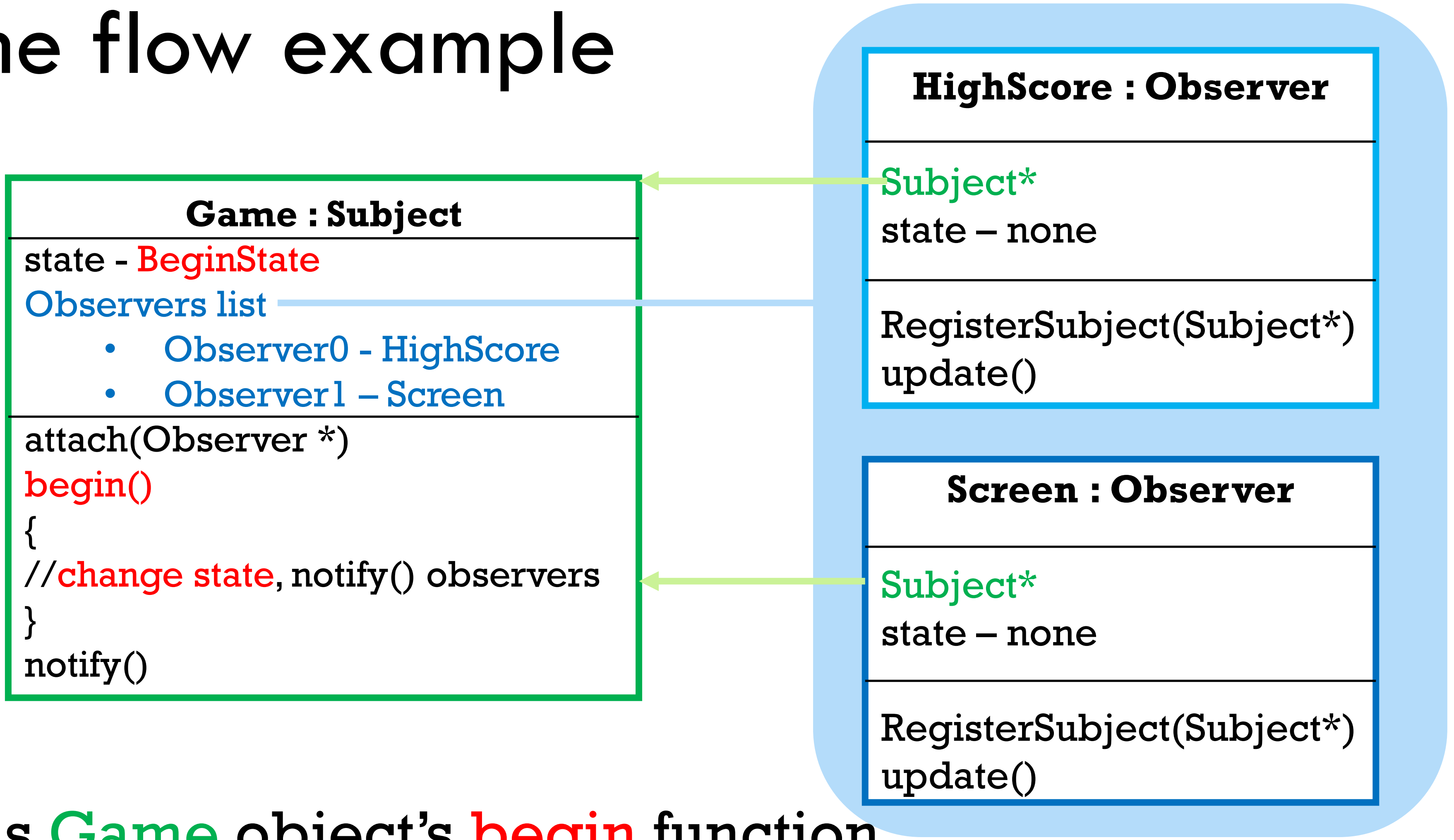
- Attach **HighScore** and **Screen** to **Game** object
  - Pass HighScore and Screen to **attach(Observer*)**
  - Game doesn't know they're HighScore and Screen objects
  - Game sees them as the **abstract Observer type**

Game flow example



- **Game** now connected to **HighScore** and **Screen**
- Need to connect **HighScore** and **Screen** to **Game**
  - Pass **Game** to **RegisterSubject(Subject*)** in each Observer

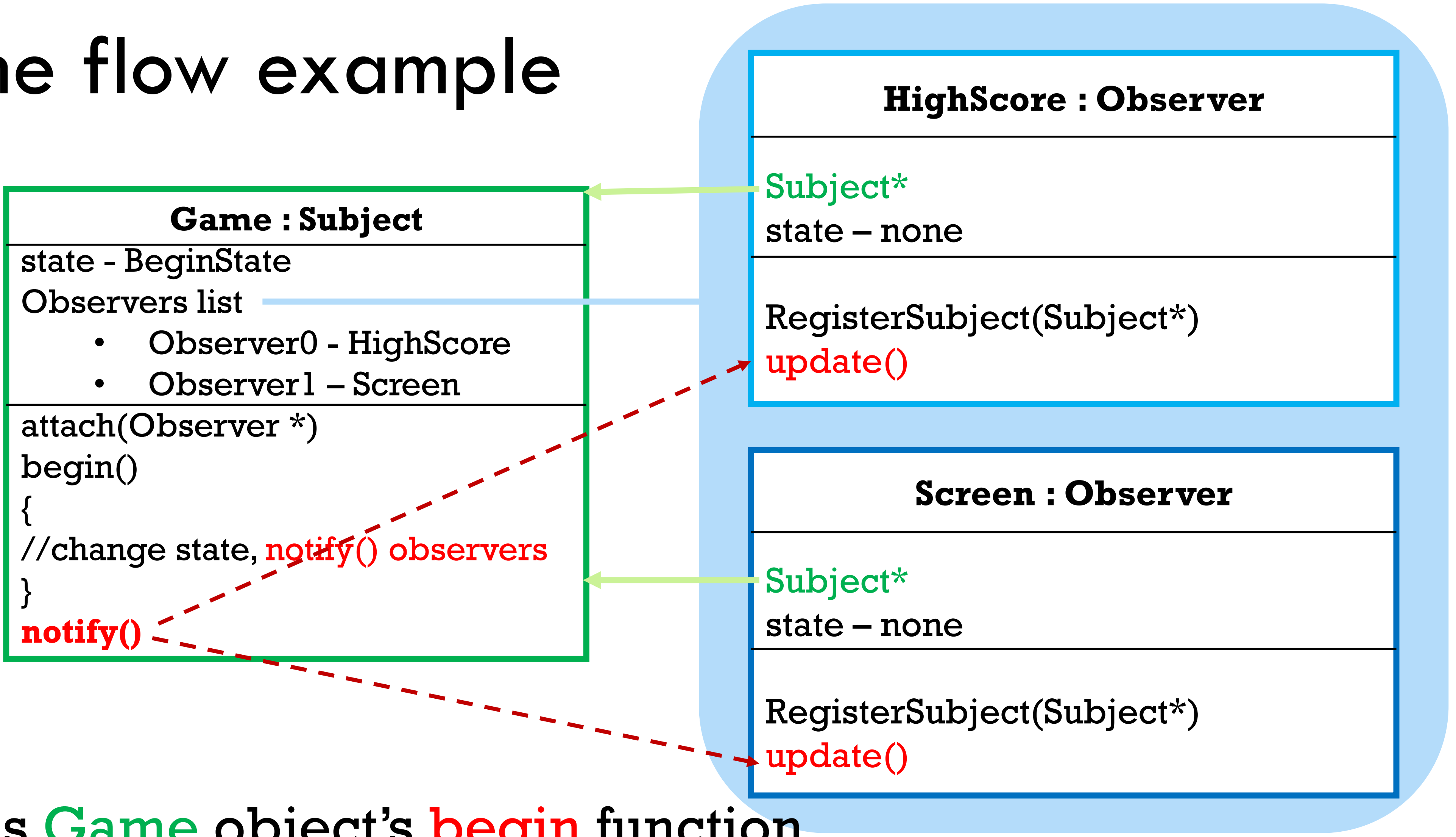
Game flow example



- Client calls Game object's begin function
    - Change the game's state to BeginState

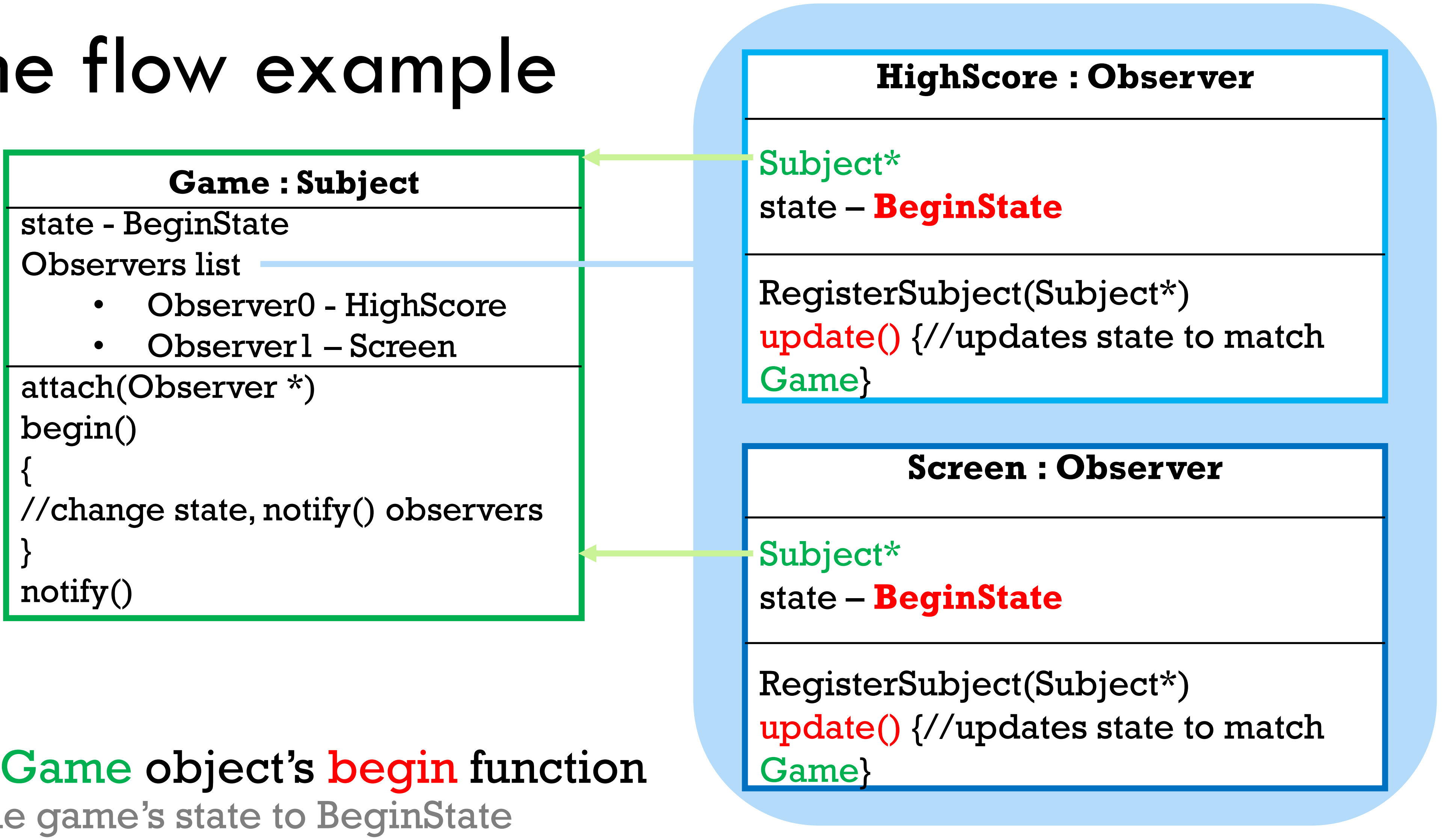
SubjectObserver.cpp

Game flow example



- Client calls Game object’s begin function
  - Change the game’s state to BeginState
  - Notify all observers in list that something has changed

Game flow example



- Client calls Game object's begin function
  - Change the game's state to BeginState
  - Notify all observers in list that something has changed
  - Observers query Game's inherited getState() to get its current state
  - Sets internal state to match Game's BeginState