COMP 3522

Object Oriented Programming in C++

Week 9

COMP 3522

使用 C++ 进行面向对象编程

第 9 周

Agenda

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

COMP 3522

议程

1. 类型转换
   - 向上、向下、交叉转换
   - 静态、动态
2. 枚举
   - 作用域限定
   - 无作用域

COMP 3522

CASTING 投射

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**.

类型转换

- Java 和 C++ 是**强类型**语言
- 每个对象（变量或常量）的类型在**编译时**定义，并且在执行期间无法更改

- 我们可以将一个对象看作：
  1. 内存中的比特位
  2. 一种**赋予这些比特位意义的类型**.

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*

C语言中的强制转换很丑陋

- 我们可以将算术类型（char、int、float、double）强制转换为其他算术类型
- 我们可以将指针类型强制转换为另一种指针类型

- 这里非常容易出错
- 运行时不会进行任何检查以确认该强制转换是否正确

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.

所以我们尽量避免在 C 语言中进行强制类型转换

- **符号** 很难发现 (强制转换为类型)
- 我们基本上可以将 **任何东西转换为任何东西**
- 出错的可能性很大 **er ror** here
- 但程序员还是会这样做。

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

C++ 强制类型转换可能更安全

Bjarne Stroustrup 认为 C 风格的强制类型转换操作符:

```
动物* a = (Animal*)canine;
```

是宽松的，而非严格的

他向 C++ 引入了一些新的 **类型转换操作符** :

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**

词汇检查

- **向上转型**：将指针/引用从 派生类 **转换为** 基类
  - 对于公有继承（是一种关系）始终允许
- **向下转型**：将指针/引用从 基类 **转换为** 派生类
  - 没有显式类型转换则不允许
- **横向转型**：将指针/引用从 基类 **转换为** 兄弟类

UP, DOWN, & CROSS CASTING

UP, DOWN, & 跨职业施法

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**.

在基类和派生类之间进行类型转换

- 让我们来看一下 **向上转型**
- 从 派生类向 基类进行类型转换，只要不存在歧义，总是可行的

```
Canine* caninePtr = new Canine;
Animal* animalPtr = caninePtr;// 向上转型为 Animal类型
```

- 可以是 **隐式的**，即接受 基类作为参数的函数，可以接受所有 子类，而无需显式转换
  - `void speak(Animal * animal){} //传递子类 bird *, canine * OK`
- 向上转型与**多态性**同义。

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; }
};
```

向上转型示例，第 1 页（共 2 页）

```
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

向上转型示例，第 2 页（共 2 页）

```
void f(A a){…}// 非多态（切片）
void g(A& a){…}void
h(A* a){…}

    B b;f(b);// 可以，但需小心！这是切片操作
g(b);// 可以 h(&b);// 可以
```

这些全都是 **隐式向上转型** 的示例——对象 b 被自动转换为基类类型 A 的对象。

上转型_和_切片.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**

基类与派生类之间的类型转换

- 让我们转向 **向下转型**

```
Animal* animalPtr = new Canine;
Canine* caninePtr = (Canine*)animalPtr;//向下转型为 Canine 类型
```

- 这是将指向 指针/引用 转换为 子类型指针/引用
- 当实际所指对象并非该子类型时，其行为是 **未定义的**

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++

在基类和派生类之间进行类型转换

如果我们需要向下转型，就应该问自己为什么：

1. 我们如何确保该对象确实是那个子类型?
2. 如果对象无法向下转型，我们应该怎么办?
3. 为什么不直接为这两种类型重载函数呢?
4. 我们能否重新设计我们的类，以便通过晚绑定和虚函数来完成任务?

如果我们仍需要进行向下转型，在C++中有两种好的选择

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)

动态和静态类型转换

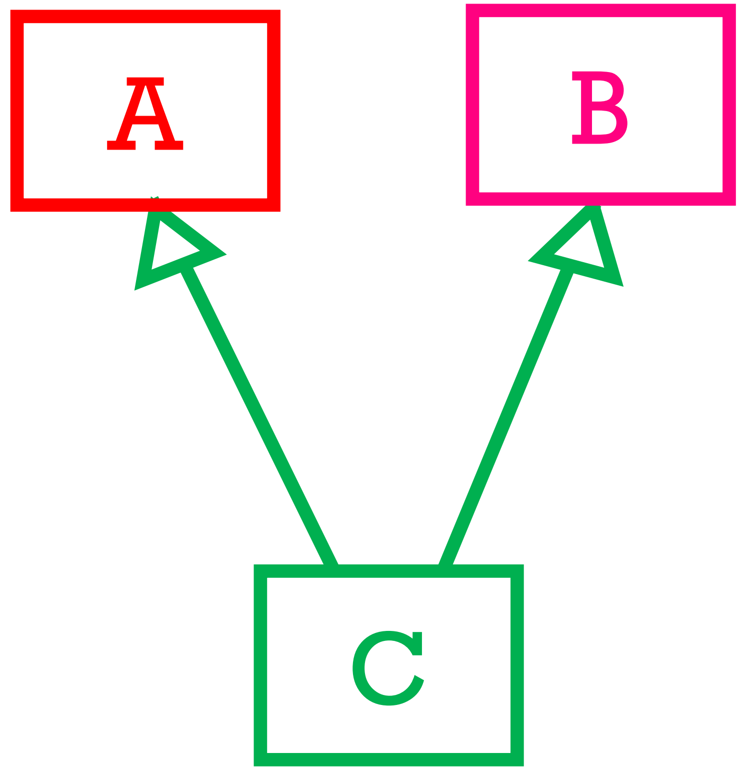
- C++ 在继承层次结构中的类之间提供**动态类型转换**和**静态类型转换**

dynamic_cast<目标_类型指针或引用>(变量)**static_cast**<目标_类型指针或引用>(变量)

Cross-casting?!

Consider this inheritance hierarchy:

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

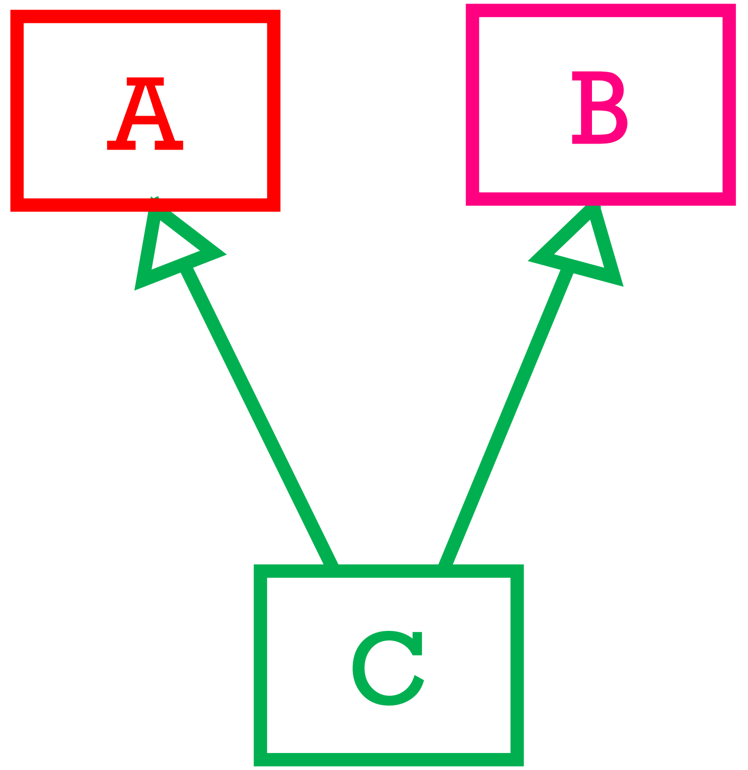


We will see this madness a few times this term.

交叉转换？！

考虑这个继承层次结构：

```
A* 一个_A = 新的 C;
B* a_B = 动态_转换<B*>(一个_A);
```



我们将在本学期多次见到这种疯狂的做法。

DYNAMIC CASTING

动态转换

Dynamic cast

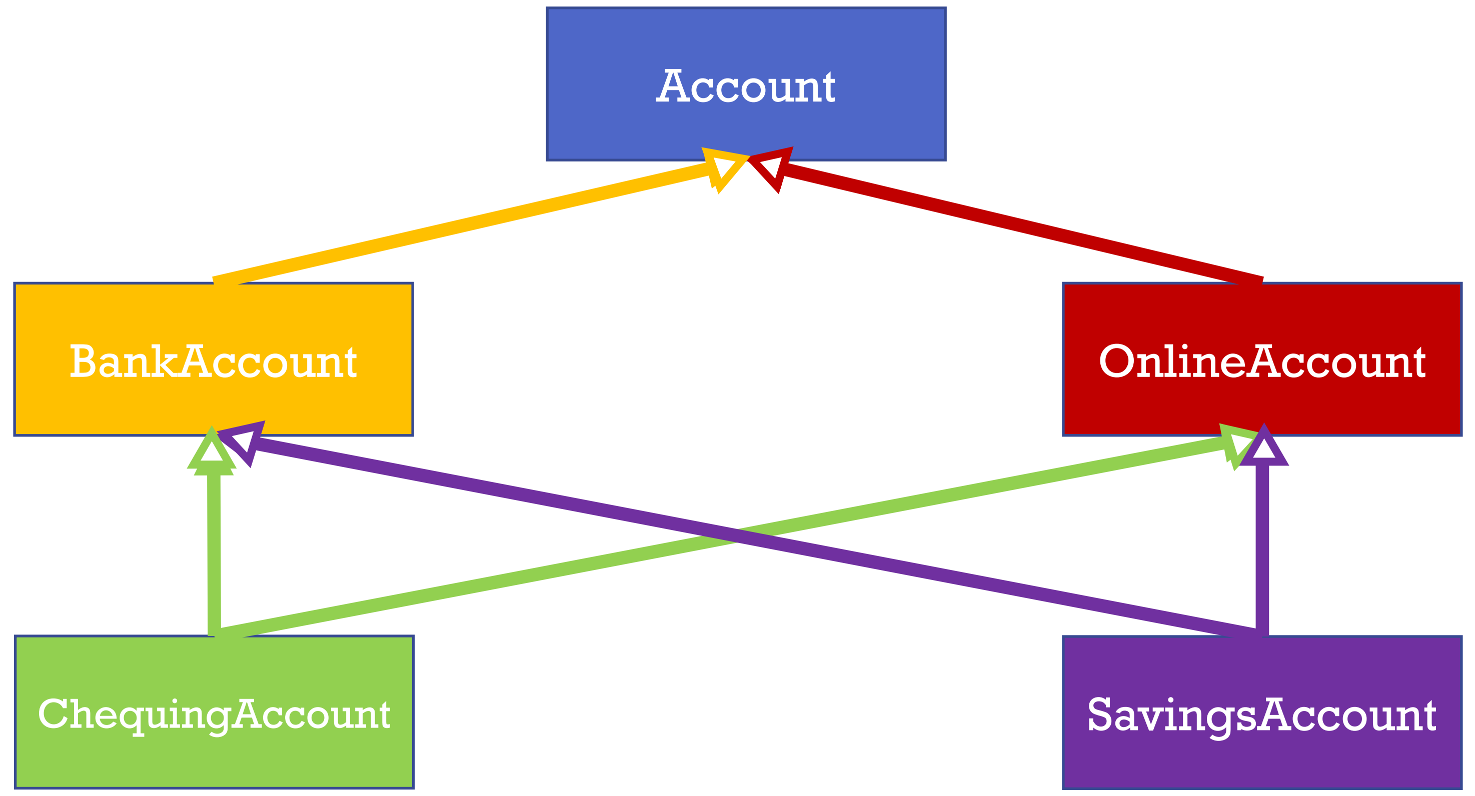
- Casts a pointer/reference of one type to a pointer/reference of another type **within an inheritance hierarchy**.
- Performs a **run-time test** to determine whether the actually casted object has the target type or a sub-type thereof
- Allowed with **pointers and references to polymorphic types** (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**

动态类型转换

- 将一种类型的指针/引用转换为继承层次结构中另一种类型的指针/引用**within an inheritance hierarchy**。
- 执行一个**运行时测试**，以确定被转换的对象是否具有目标类型或其子类型
- 允许对 **指针 和 引用 进行多态类型转换** （回顾：多态类型是指至少有一个虚成员函数的类型）
  - **转换失败** 时，指针 **目标类型返回 nullptr**
  - **转换失败** 时，引用 **目标类型将抛出** bad_cast **异常**

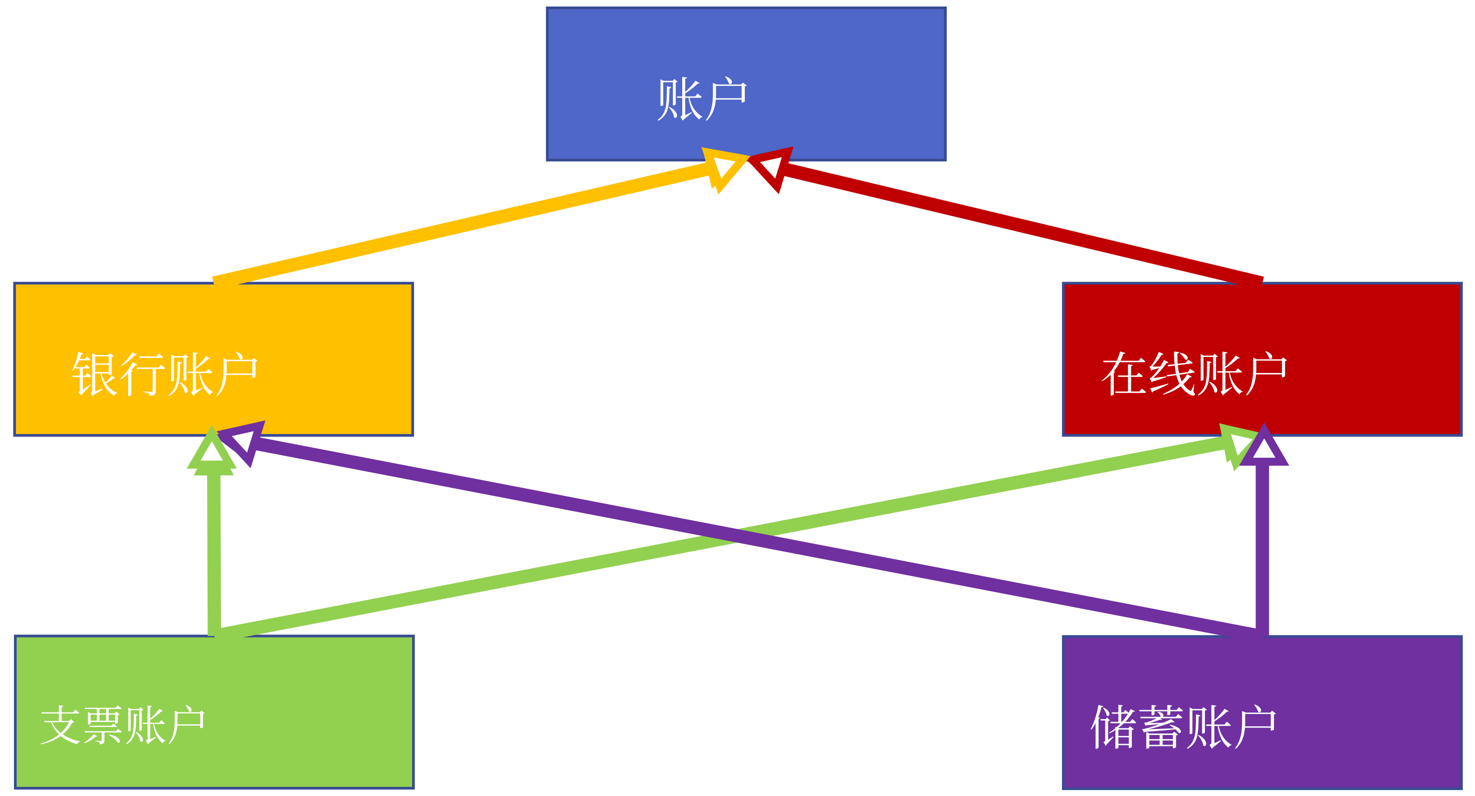
Dynamic cast example

- 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**

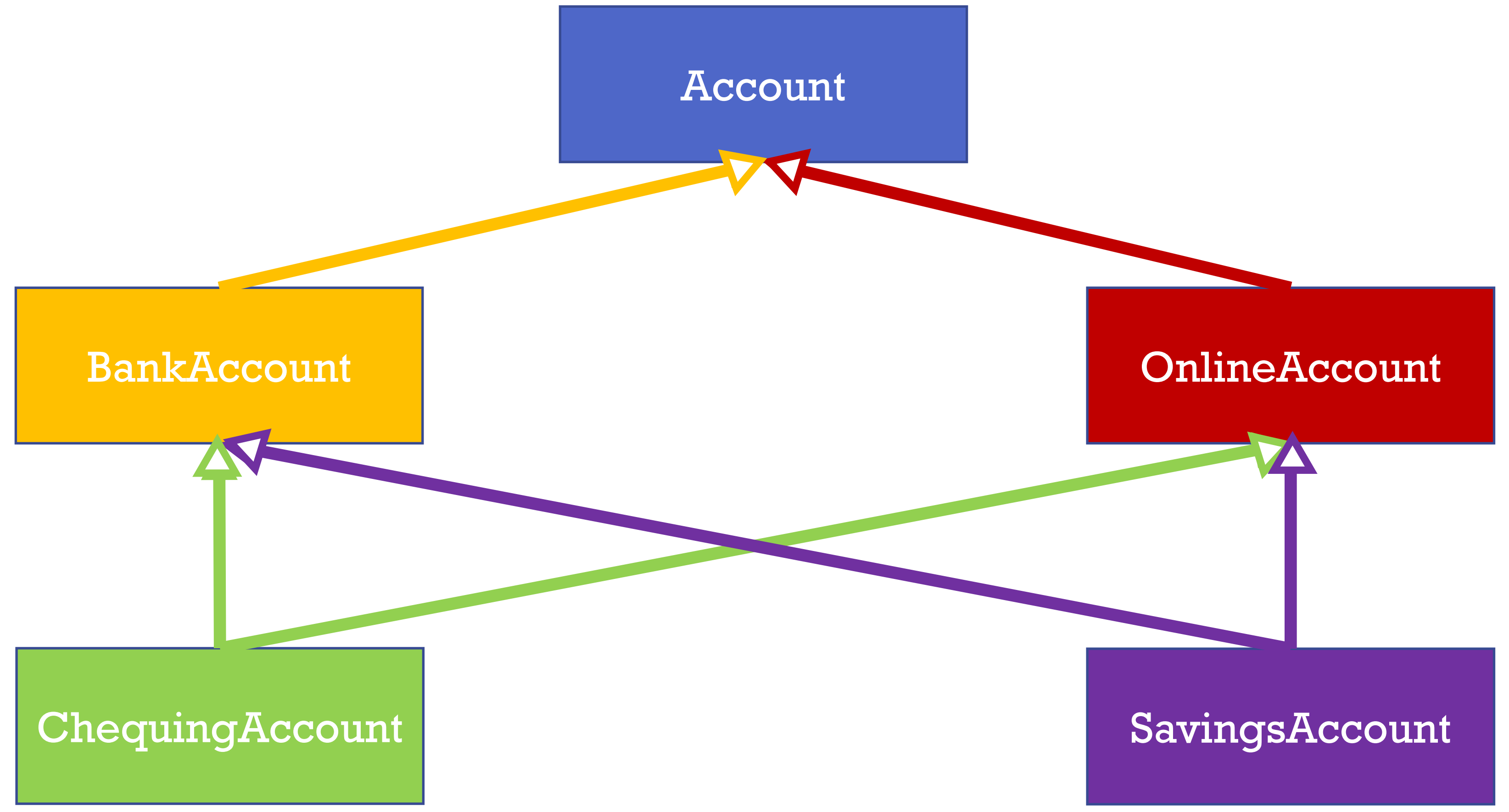


动态类型转换示例

- 假设我们有一个 **Account**类
- Account 由 **BankAccount** 和 **OnlineAccount**派生
- **ChequingAccount** 有两个基类, **BankAccount** 和**OnlineAccount**
- **SavingsAccount** 有两个基类, **BankAccount** 和 OnlineAccount

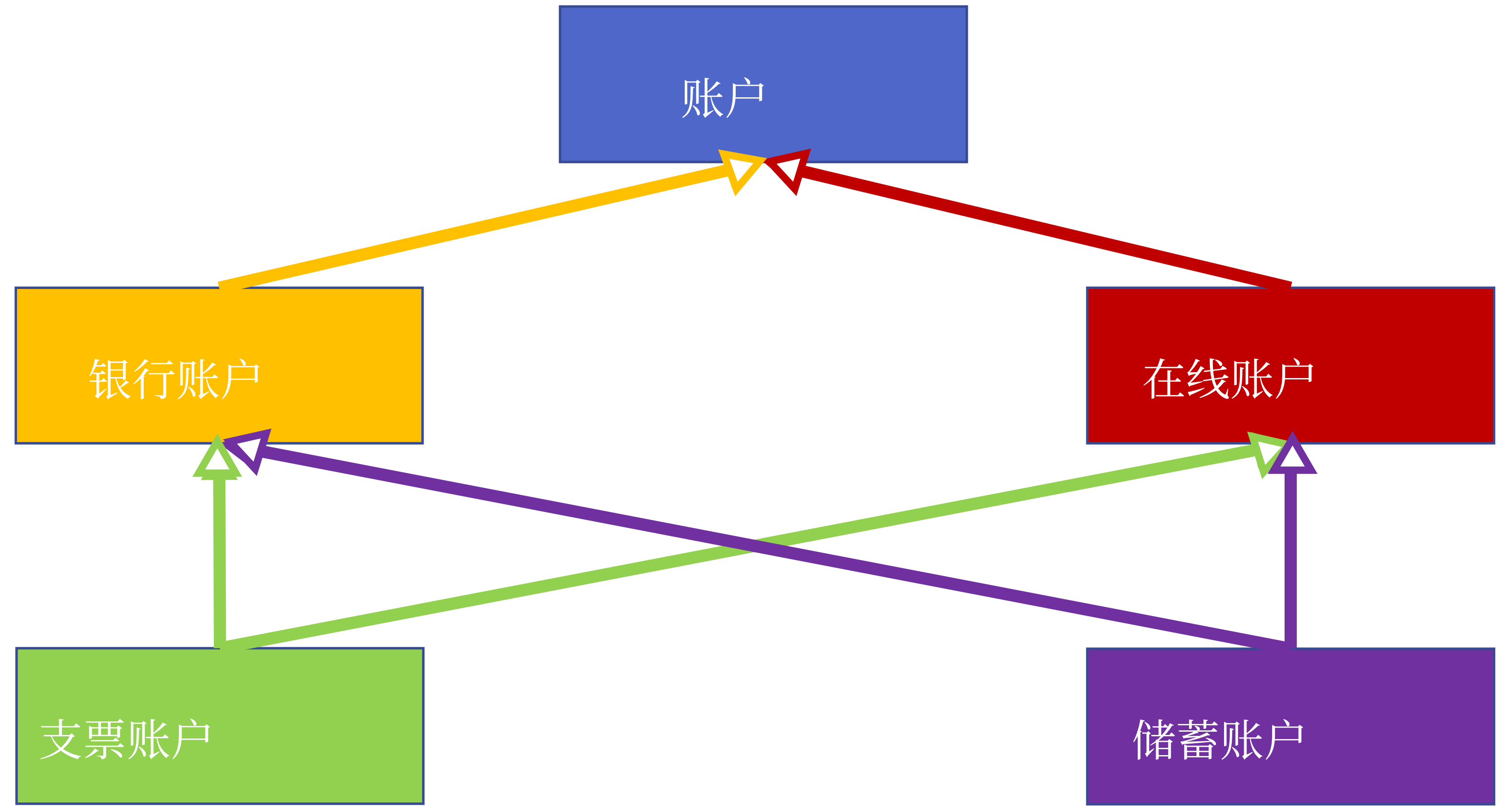


Dynamic cast example



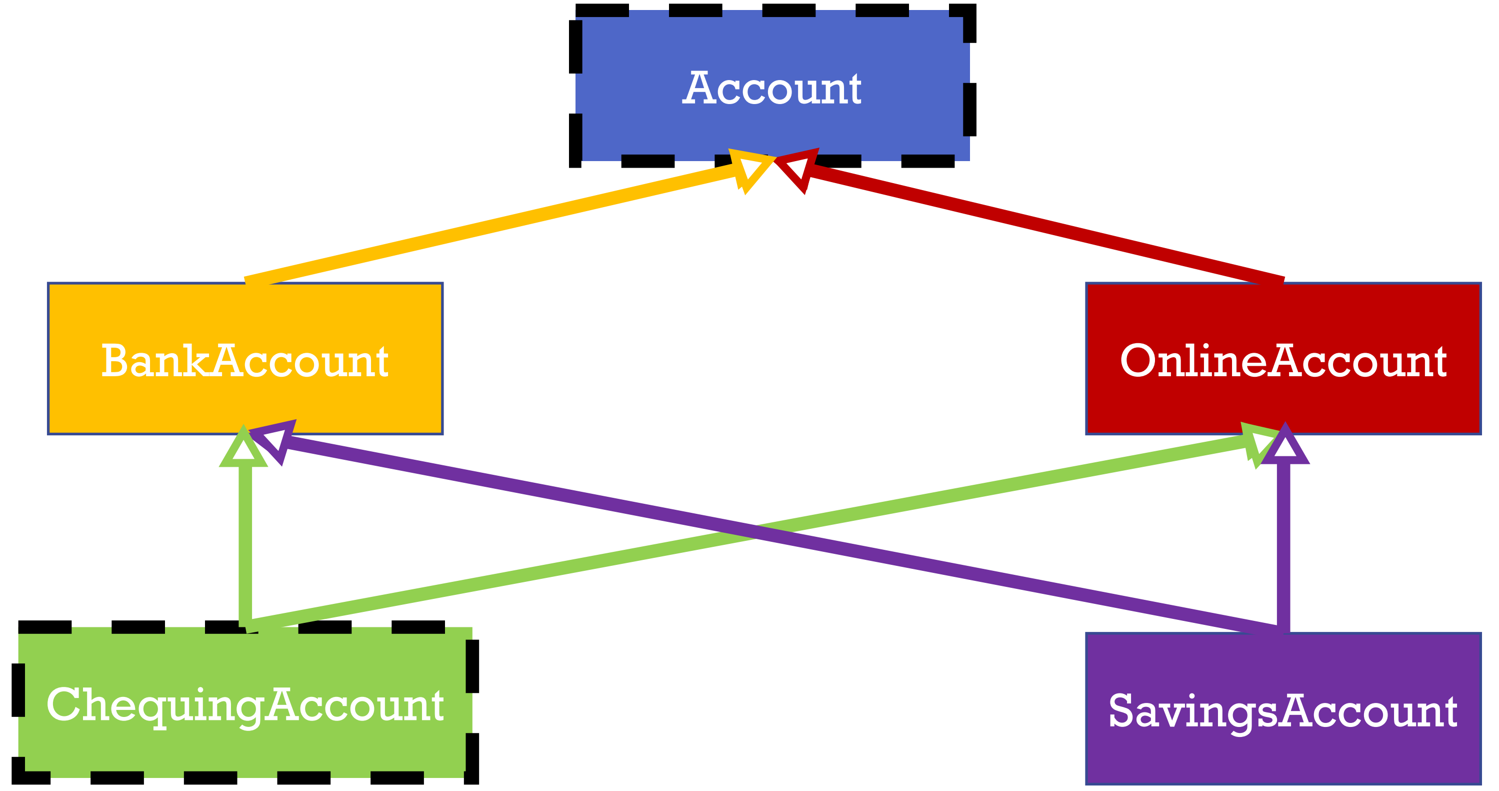
1. What kinds of pointers are allowed to point at which objects?
2. What kinds of casts are allowed?

动态类型转换示例



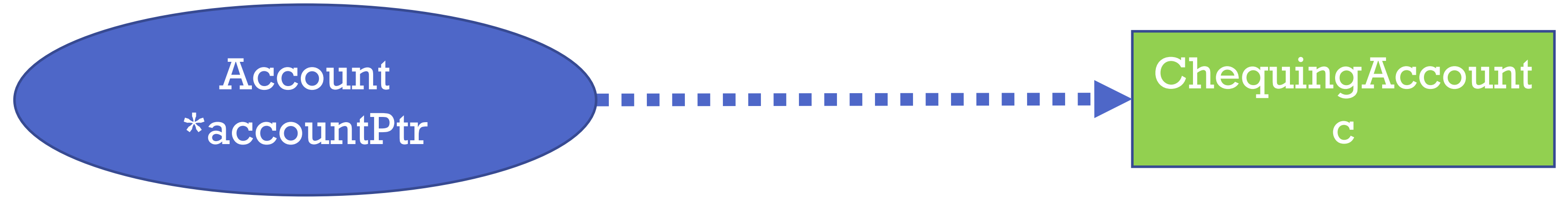
1. 哪些类型的　　p指针可以指向哪些对象?
2. 允许哪些类型的强制类型转换?

Upcast

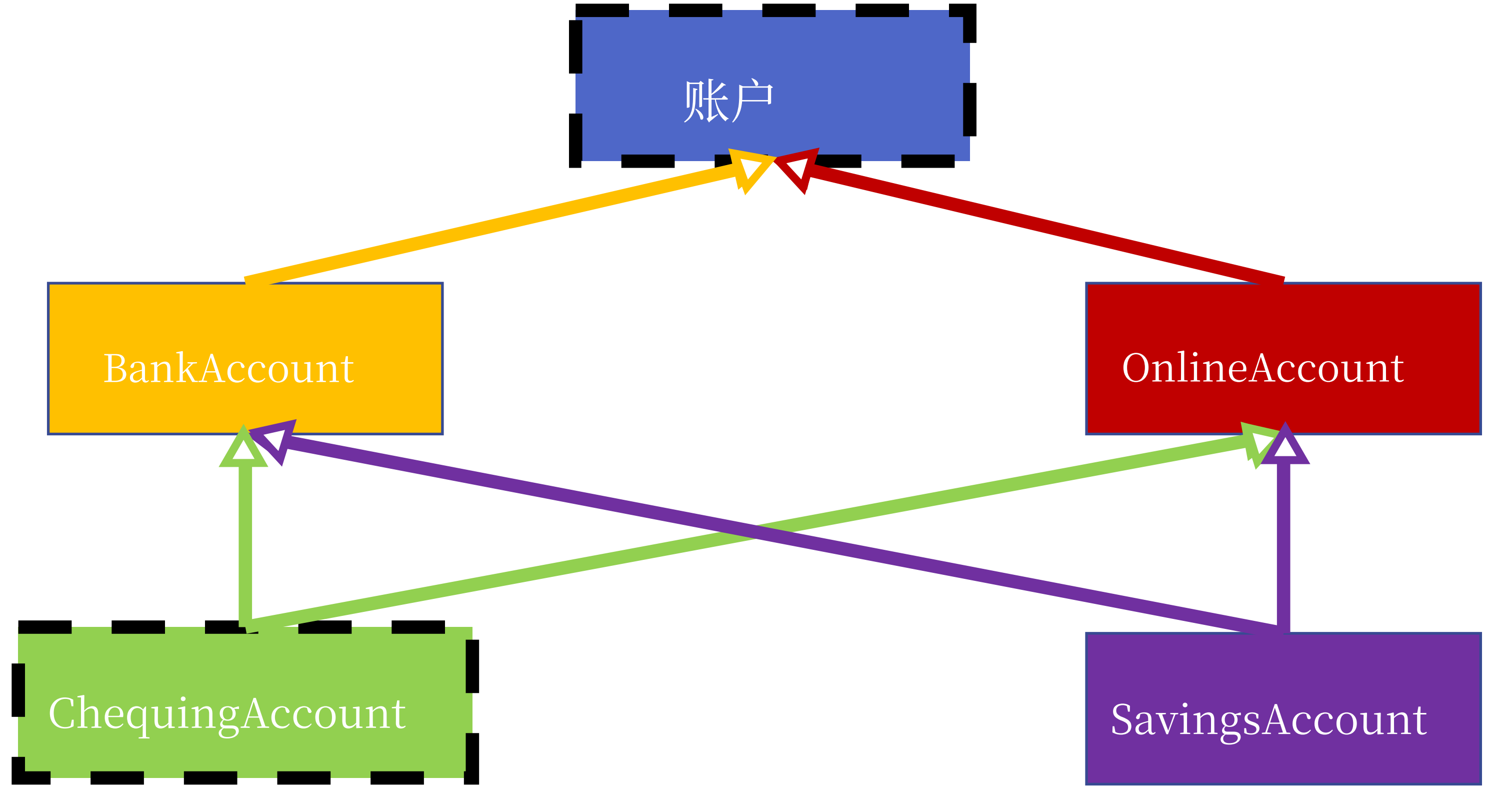


- **ChequingAccount c**;
- **Account** * **accountPtr** = dynamic_cast<**Account** *>(&**c**);

// Pointer upcast

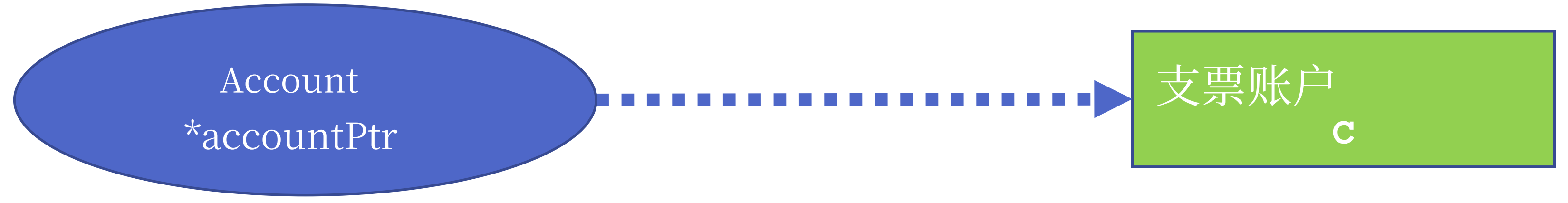


向上转型

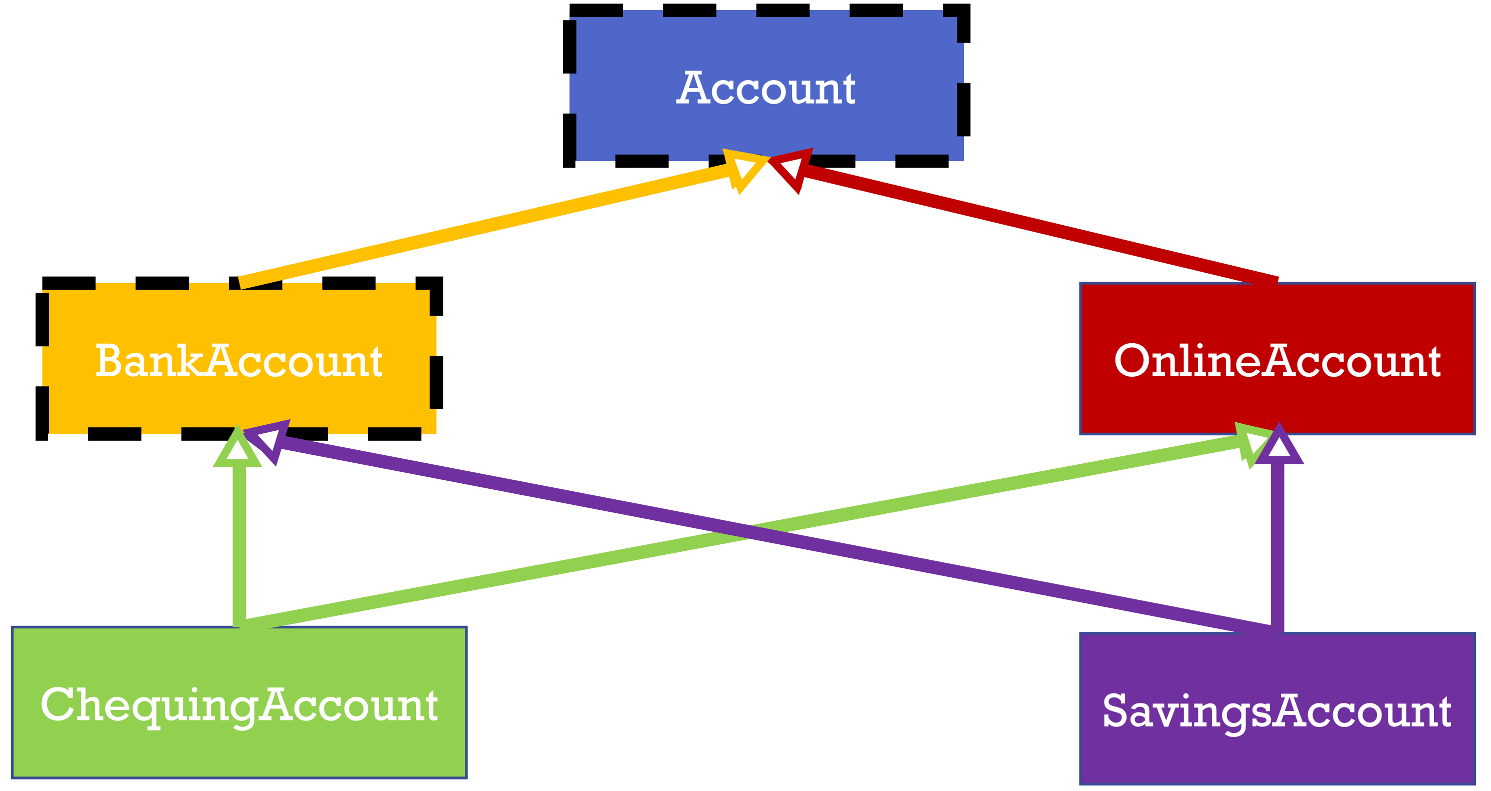


- **支票账户 c**;
- **账户** * **accountPtr** = dynamic_cast<**账户** *>(&**c**);

// 指针向上转型

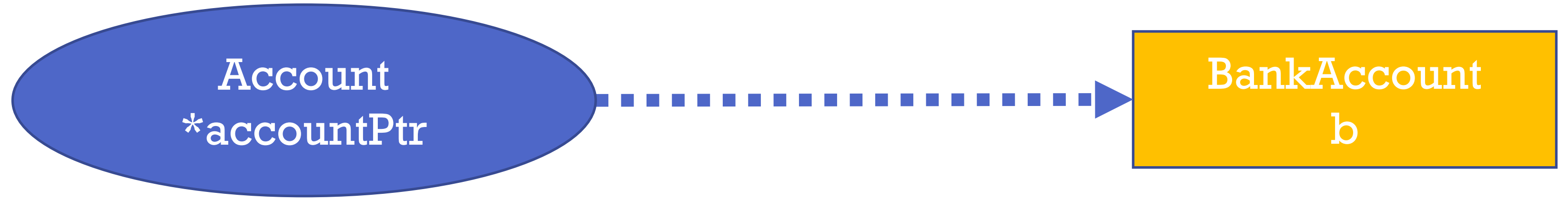


Upcast

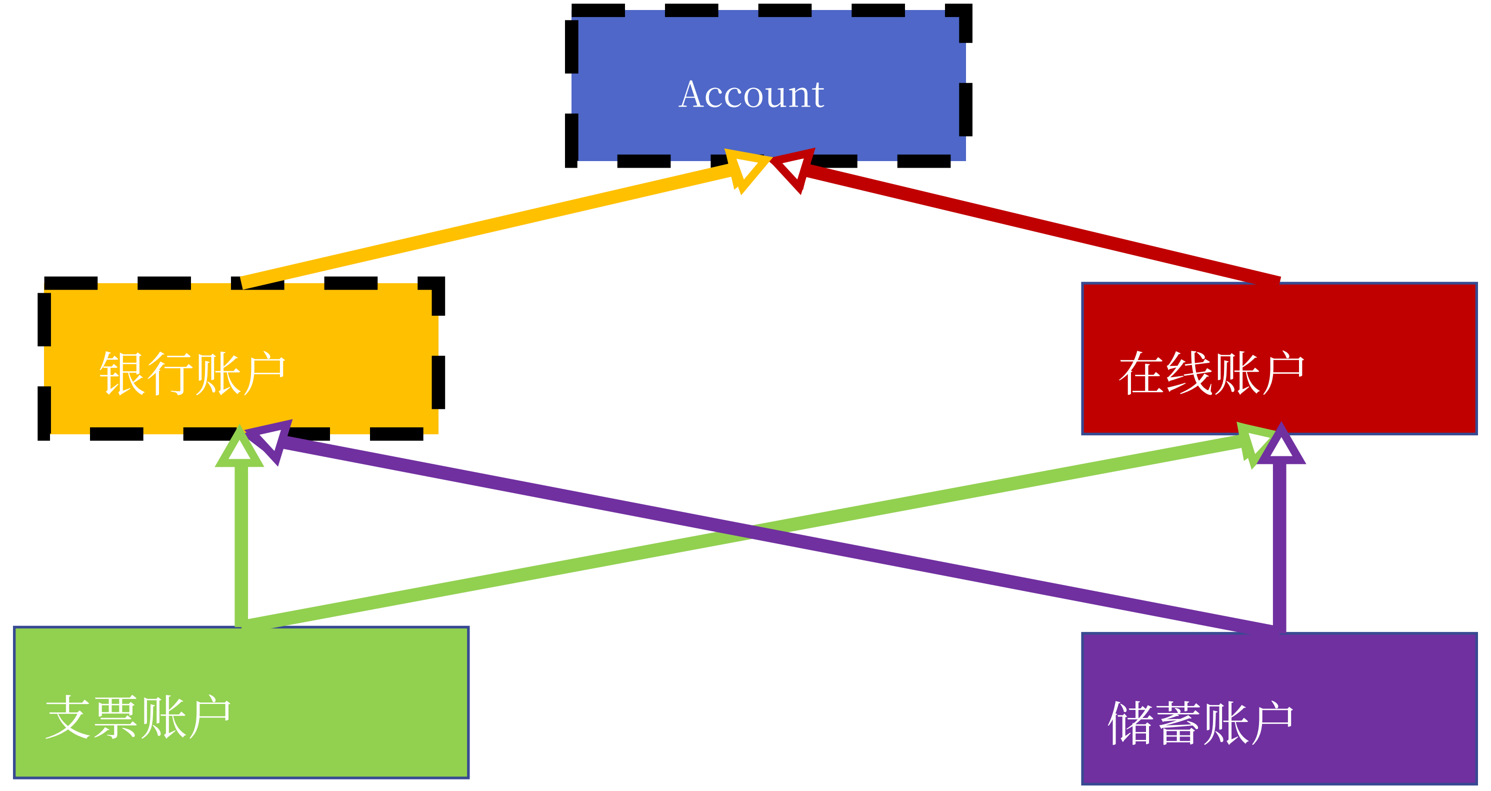


- **BankAccount b**;
- **accountPtr** = dynamic_cast<**Account** *>(&**b**);

// Pointer upcast

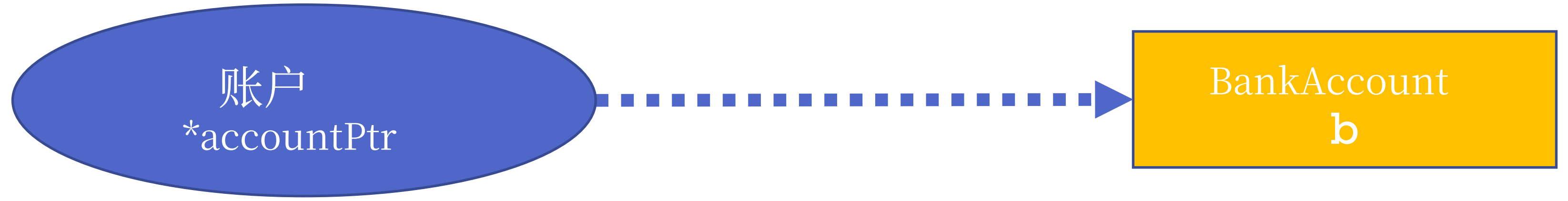


上转型

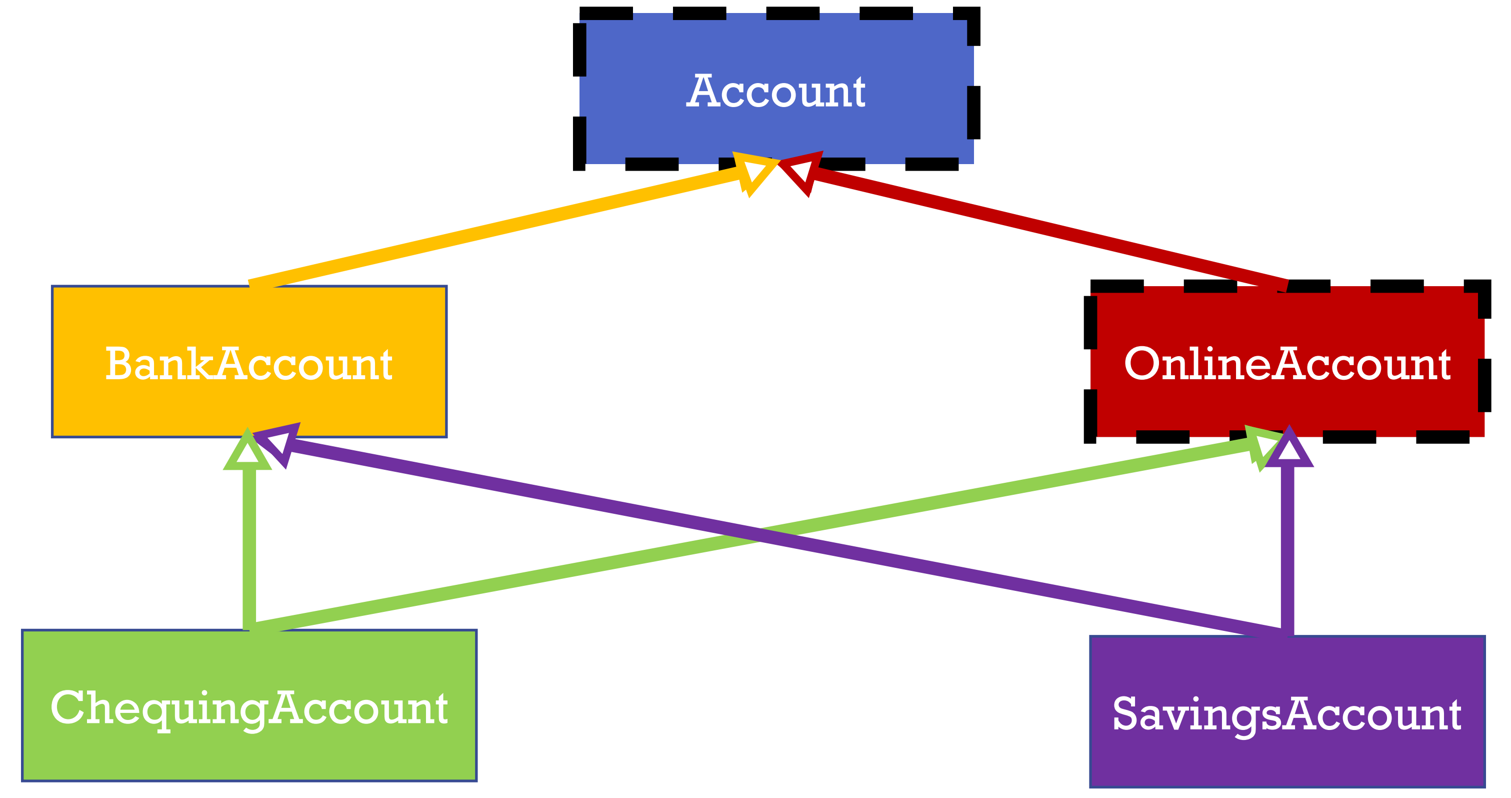


- **银行账户 b**;
- **accountPtr** = dynamic_转换<**账户** *>(&**b**);

// 指针上转型

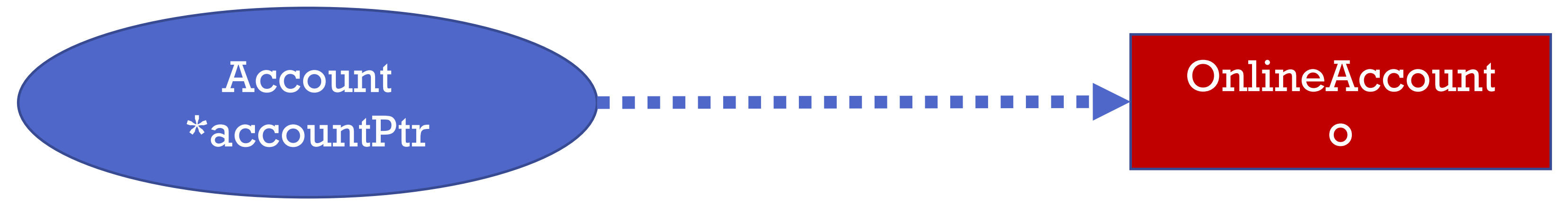


Upcast

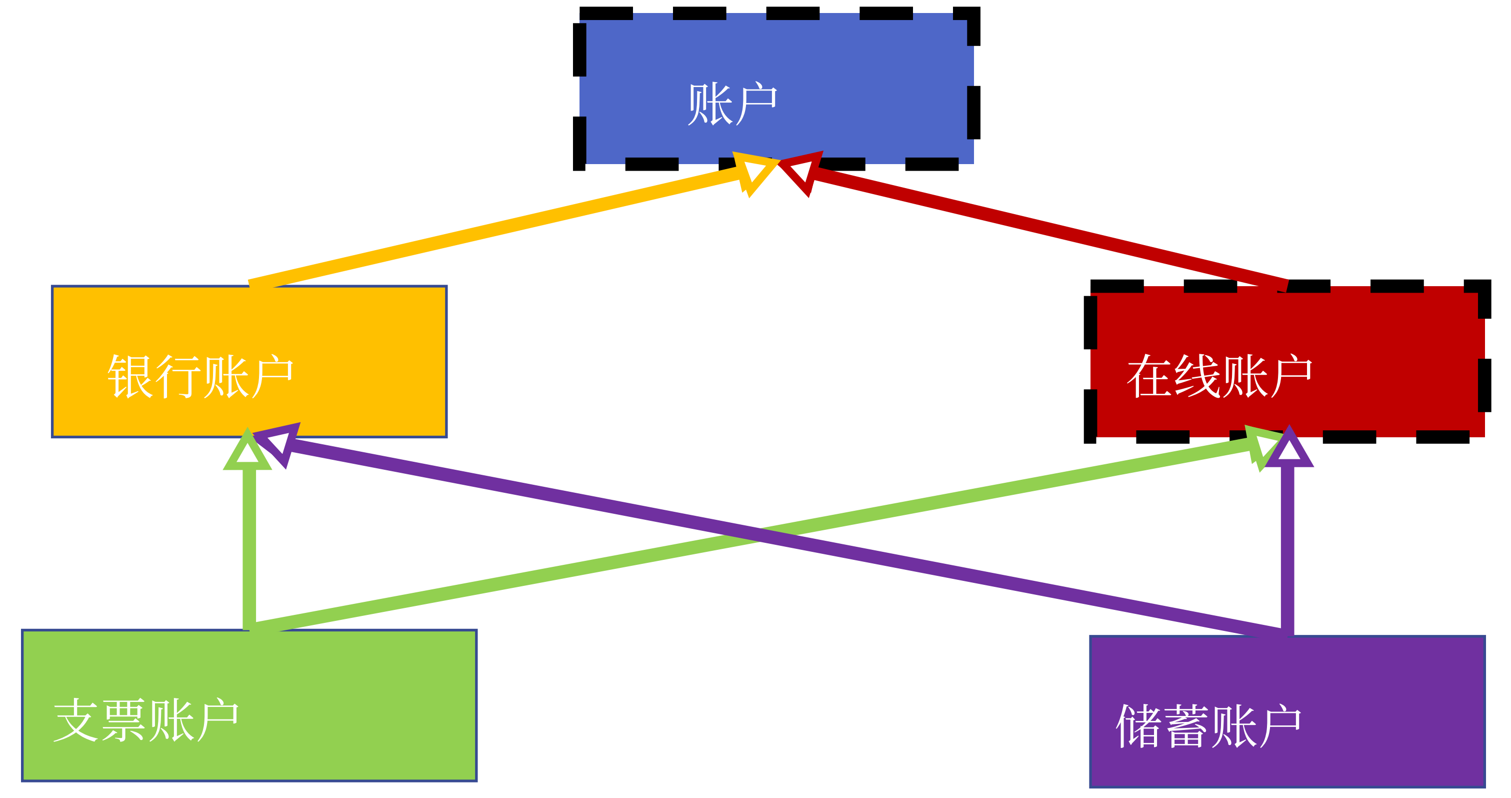


- **OnlineAccount o**;
- **accountPtr** = dynamic_cast<**Account** *>(&**o**);

// Pointer upcast

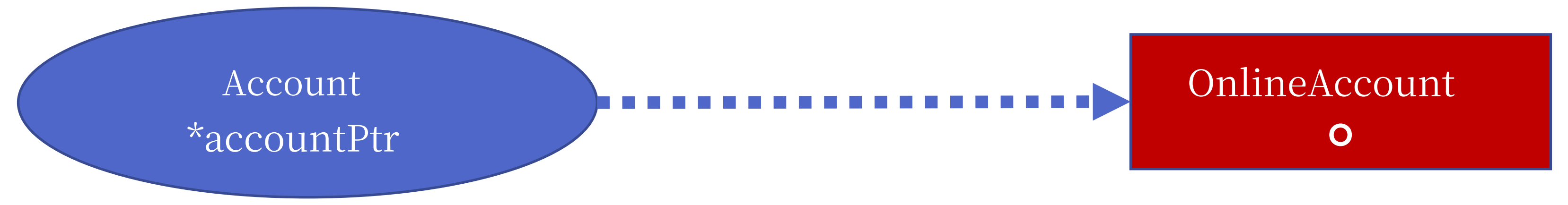


向上转型

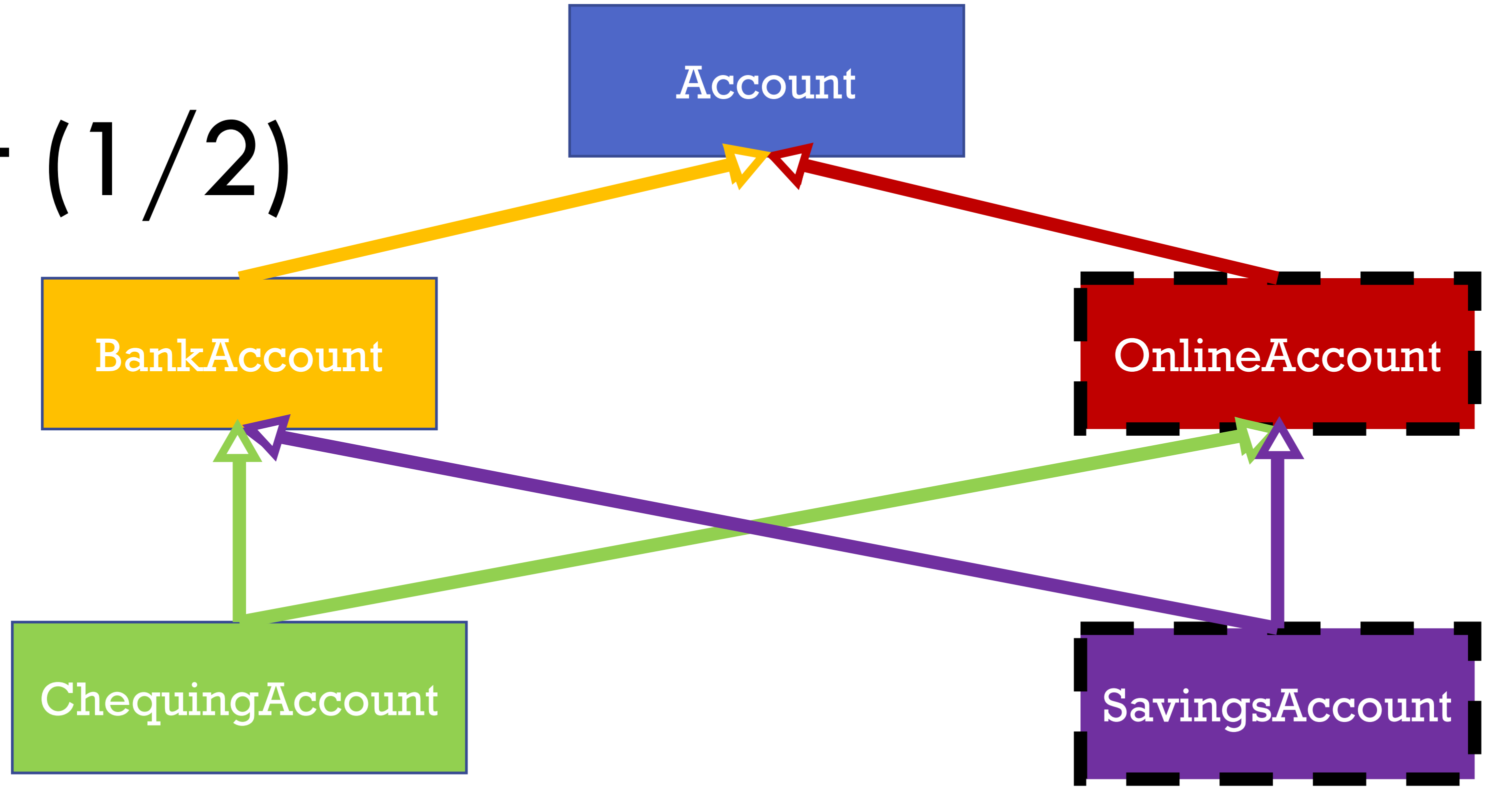


- **在线账户 o**;
- **accountPtr** = dynamic_cast<**Account** *>(&**o**);

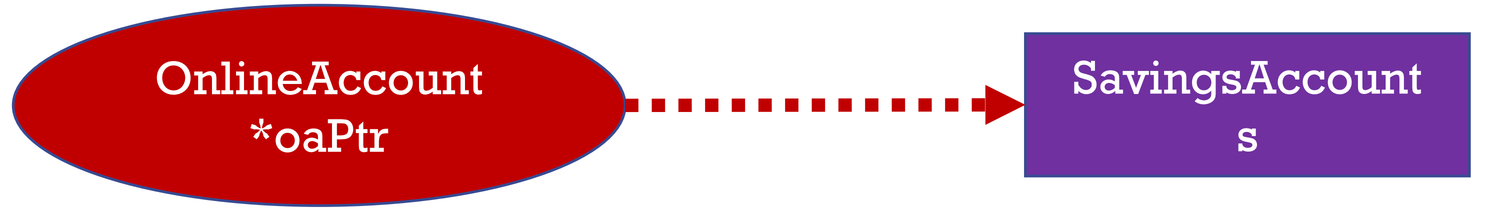
// 指针上转型



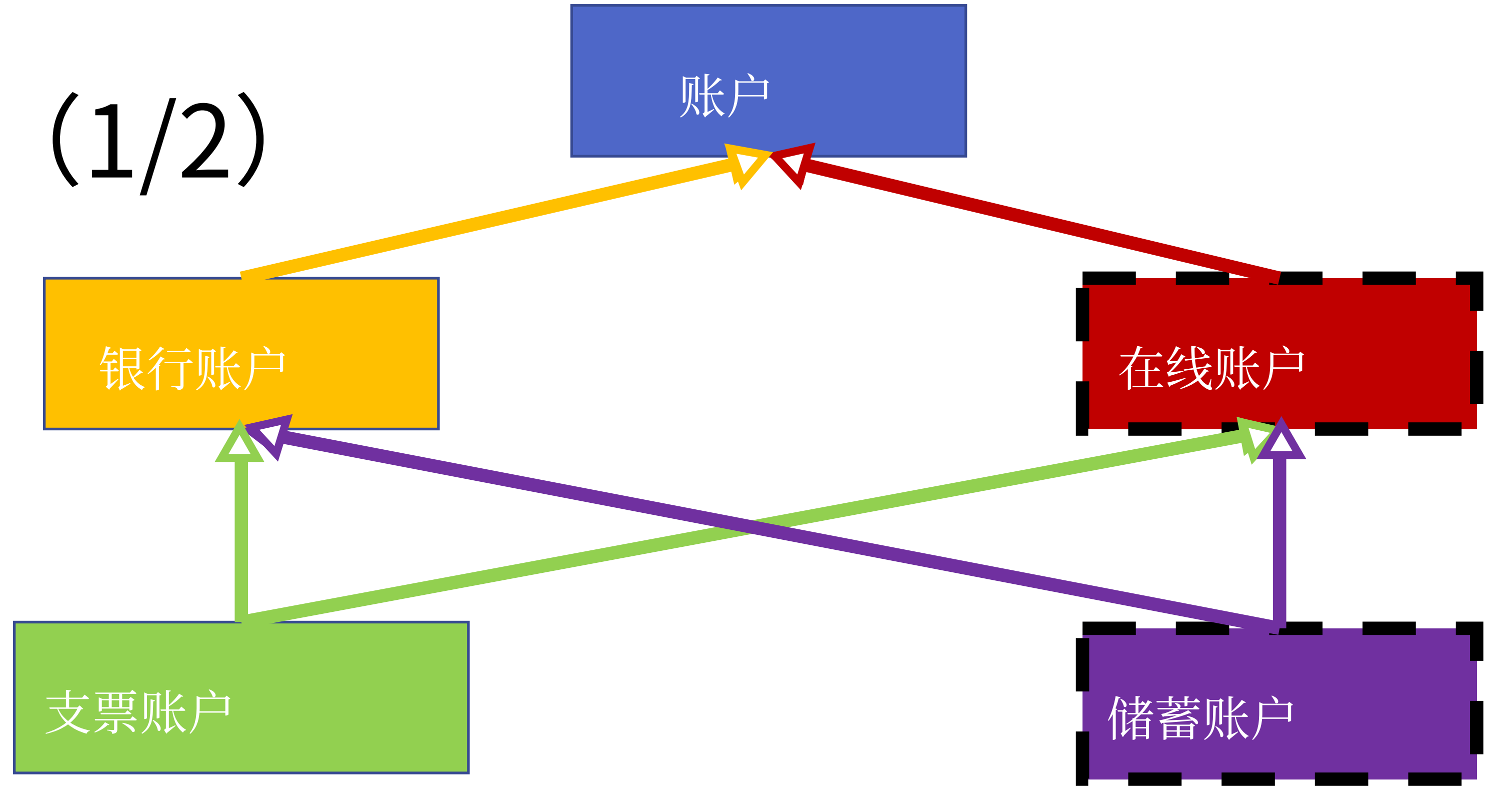
Downcast (1/2)



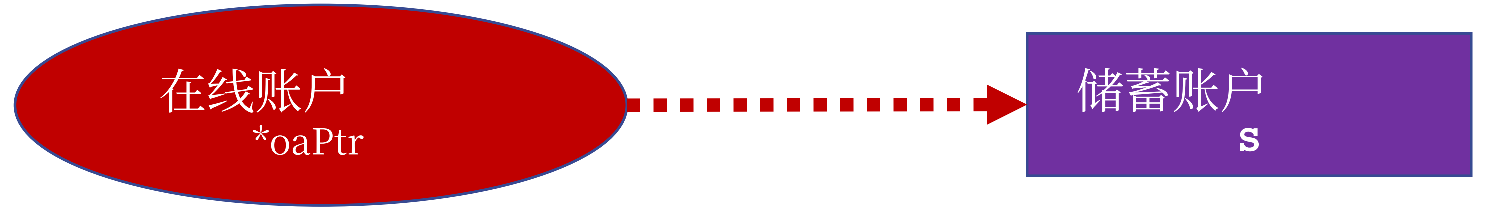
- **SavingsAccount s**;
- **OnlineAccount** * **oaPtr** = dynamic_cast<**OnlineAccount** *>(&**s**); // Pointer upcast



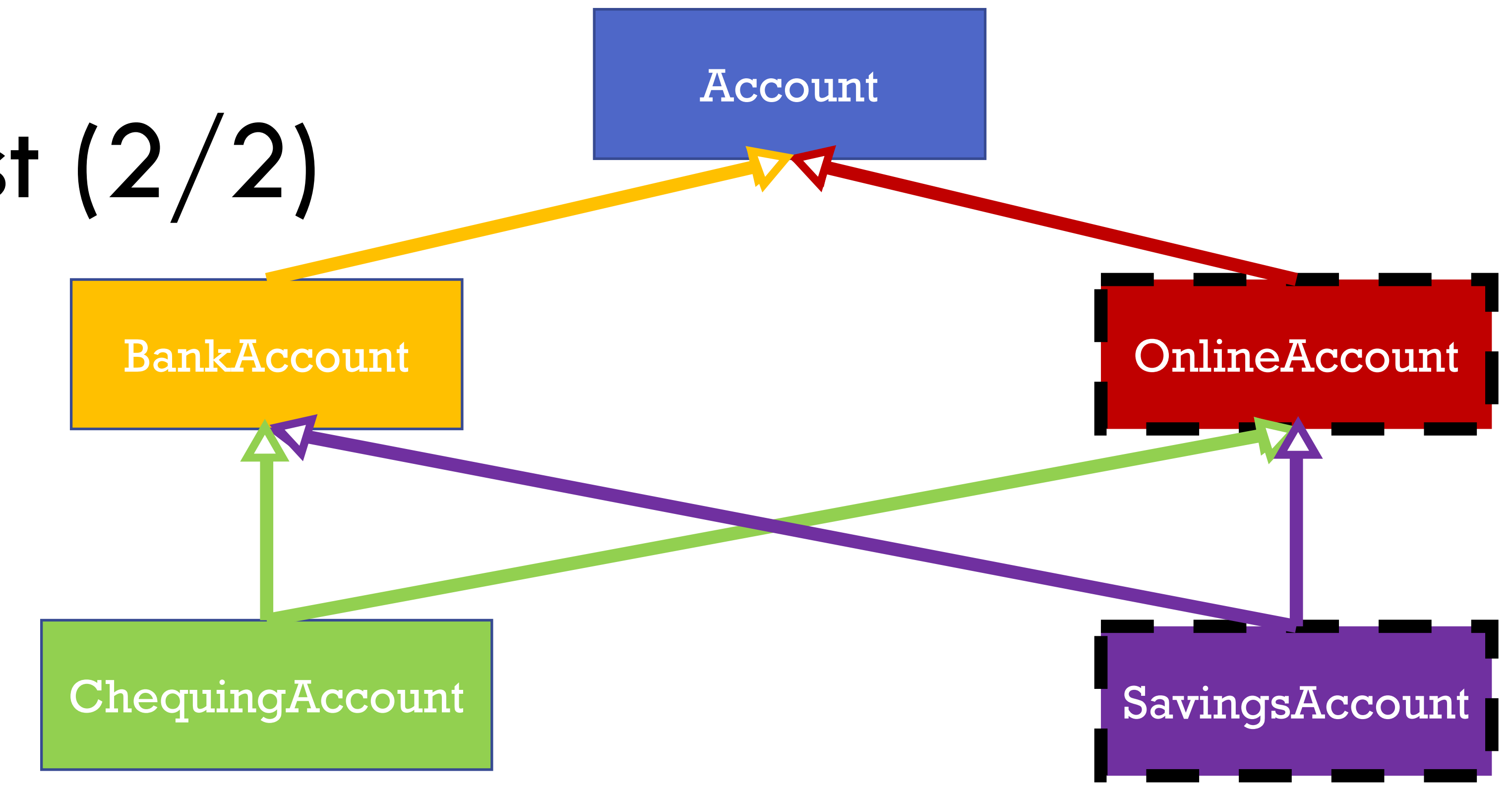
向下转型（1/2）



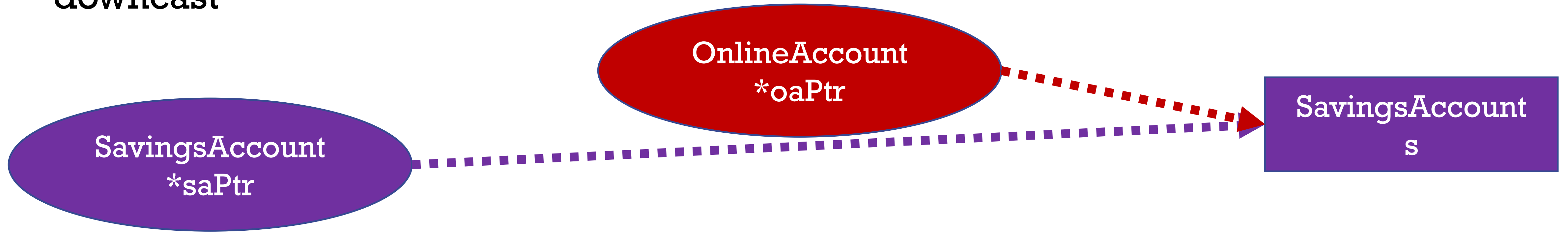
- **储蓄账户 s**;
- **OnlineAccount** * **oaPtr** = dynamic_cast<**OnlineAccount** *>(&**s**); // 指针向上转型



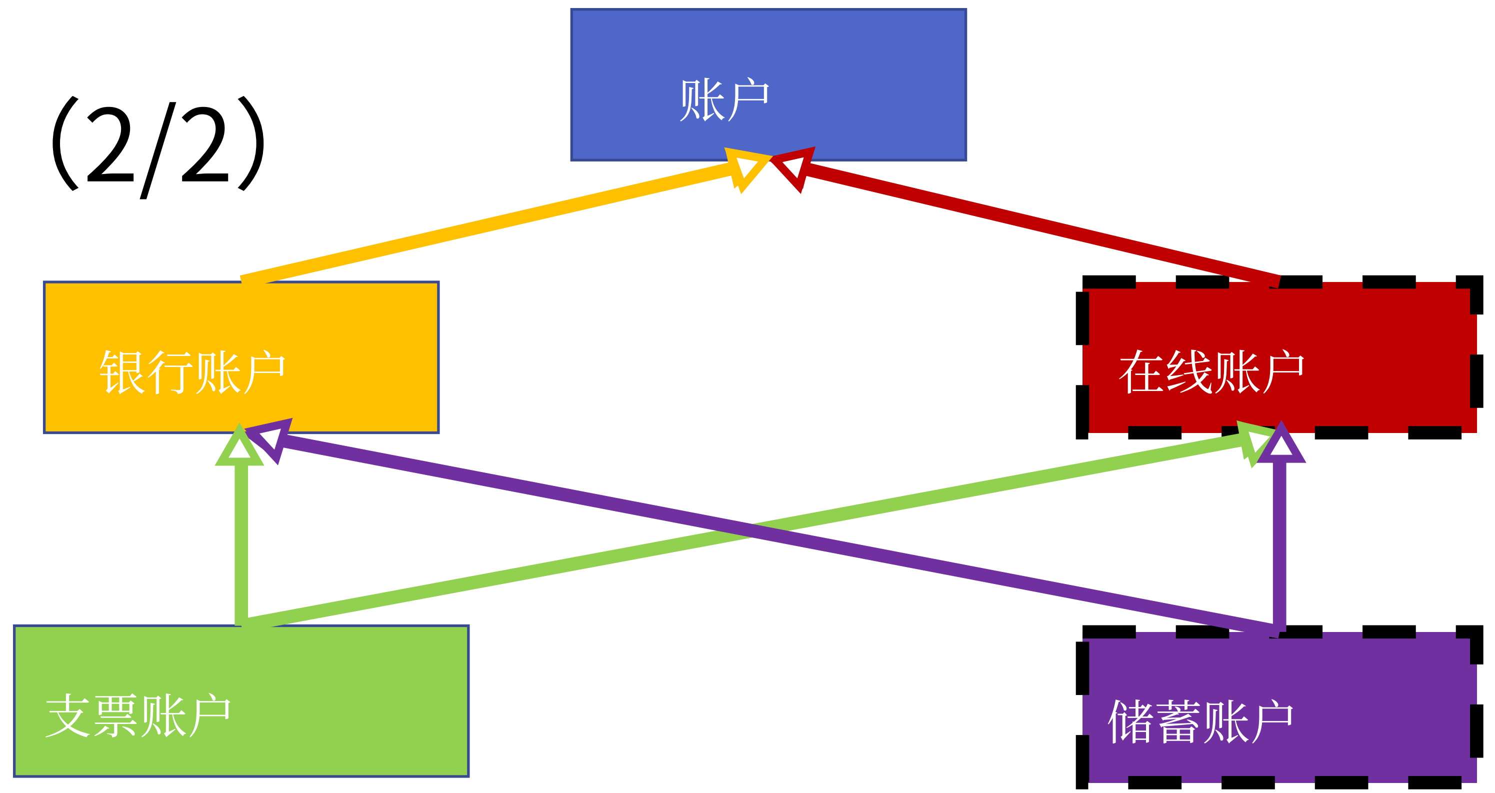
Downcast (2/2)



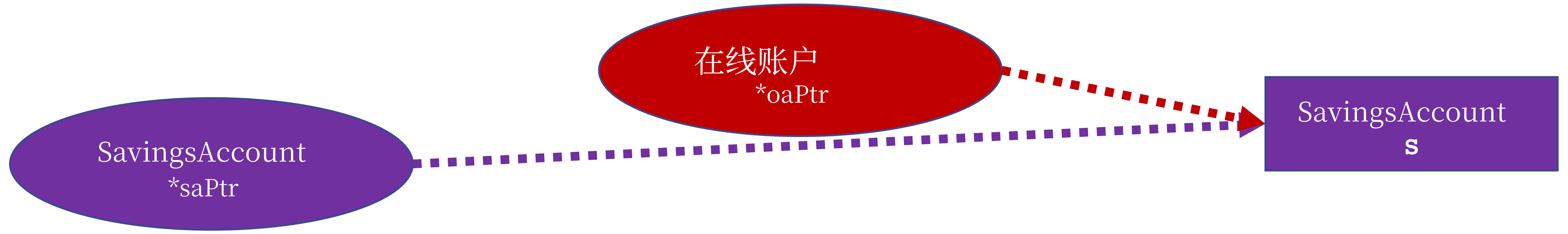
- **SavingsAccount s**;
- **OnlineAccount** * **oaPtr** = dynamic_cast<**OnlineAccount** *>(&**s**); // Pointer upcast
- **SavingsAccount** * **saPtr** = dynamic_cast<**SavingsAccount** *>(**oaPtr**); // Pointer downcast



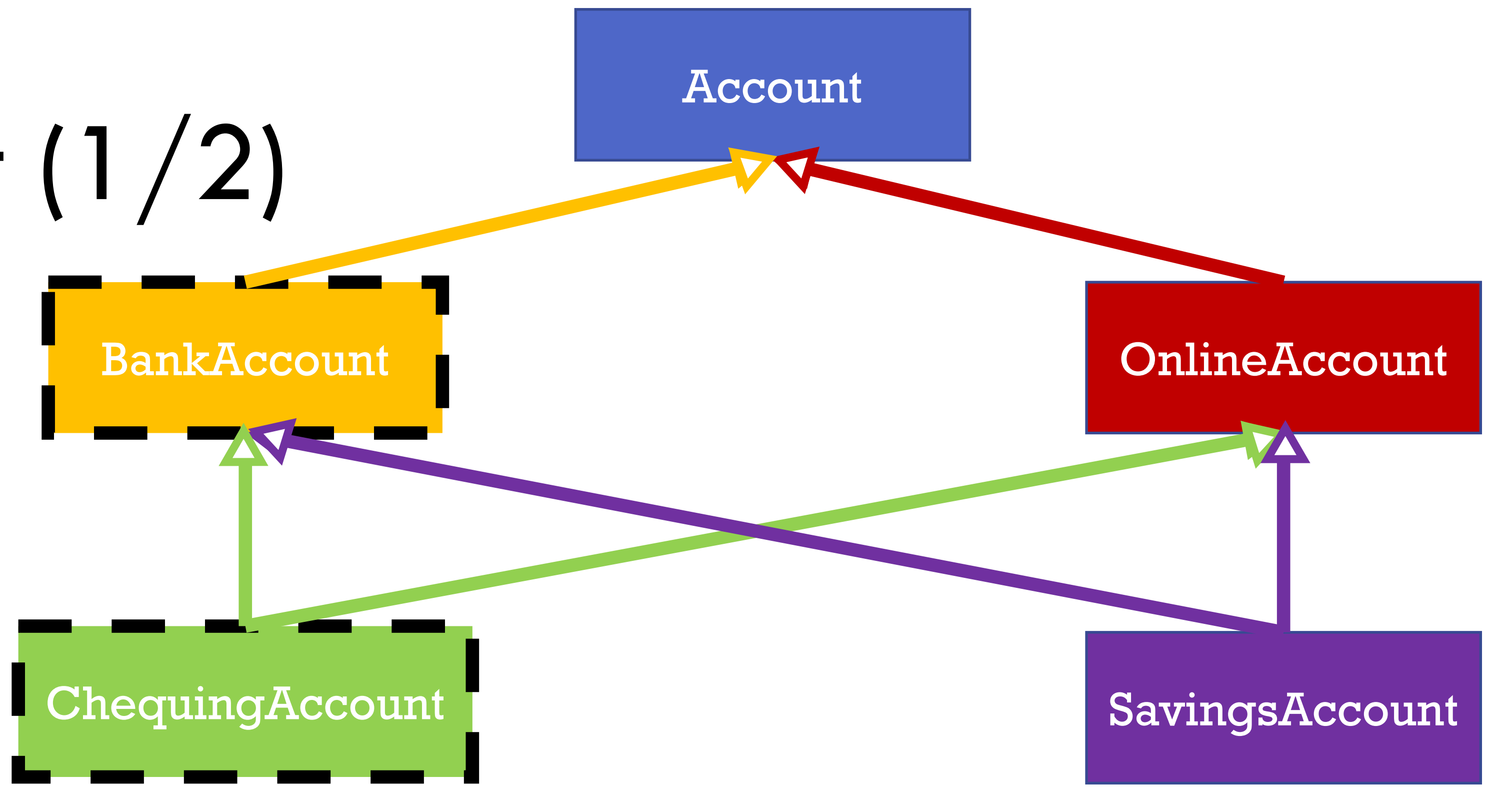
向下转型（2/2）



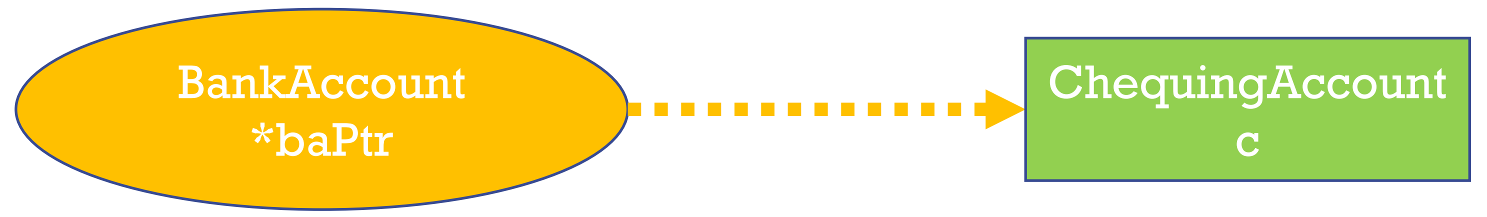
- **储蓄账户 s**;
- **OnlineAccount** * **oaPtr** = dynamic_cast<**OnlineAccount** *>(&**s**); // 指针上转型
- **SavingsAccount** * **saPtr** = dynamic_cast<**SavingsAccount** *>(**oaPtr**); // 指针下转型



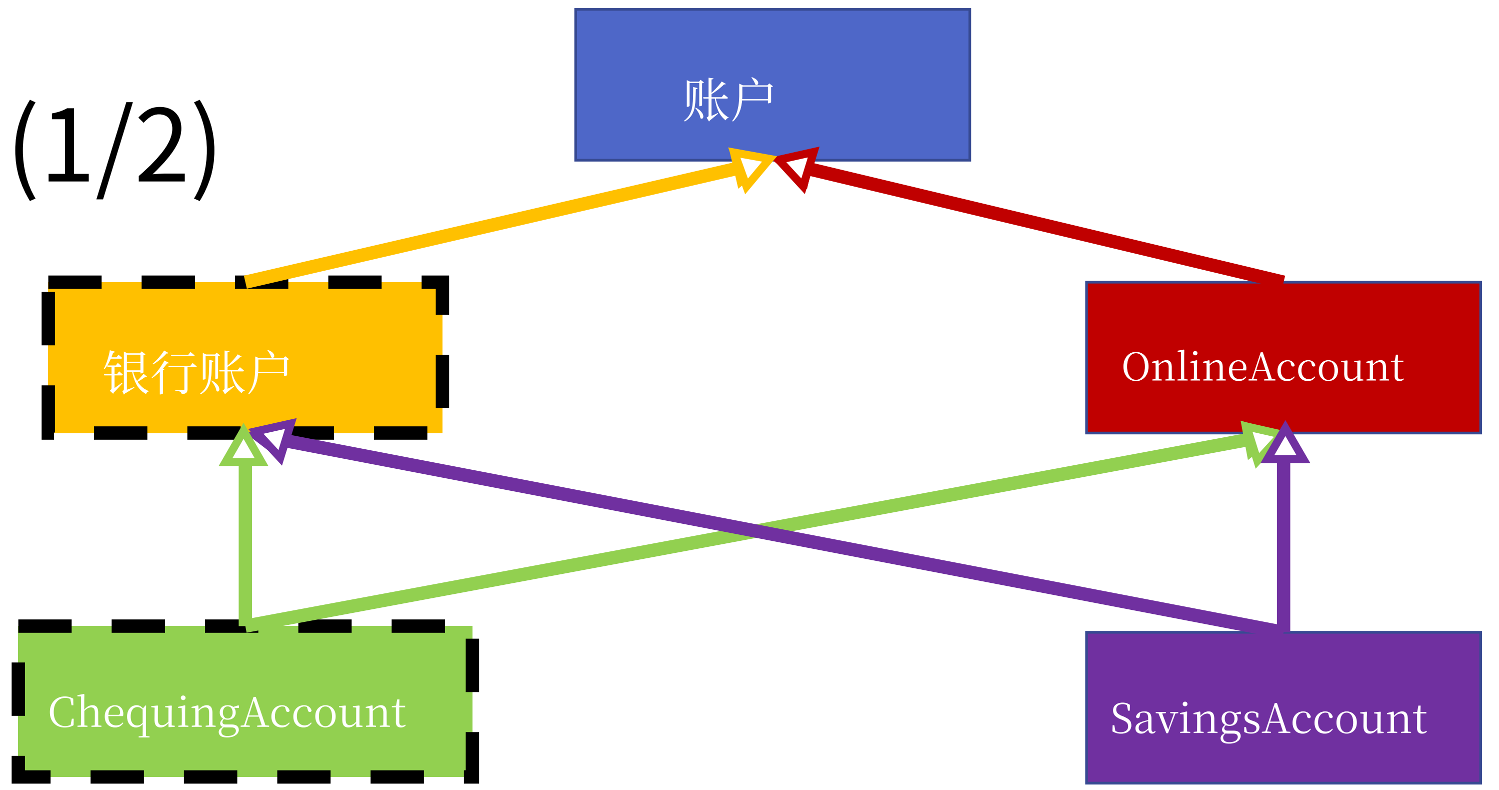
Crosscast (1/2)



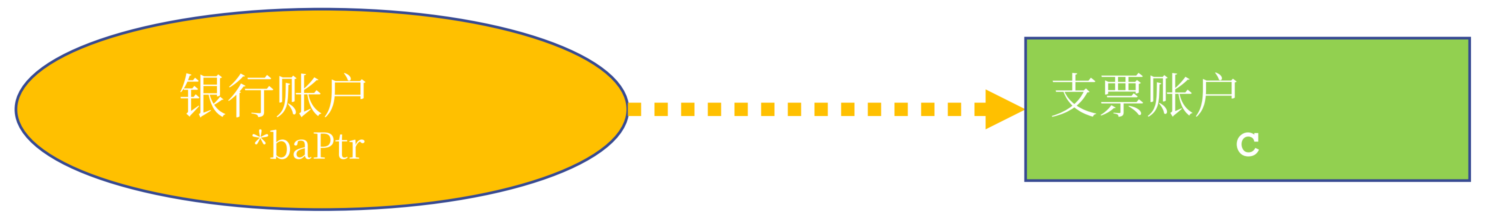
- ChequingAccount c;
- BankAccount * baPtr = dynamic_cast<BankAccount *>(&c); // Pointer upcast



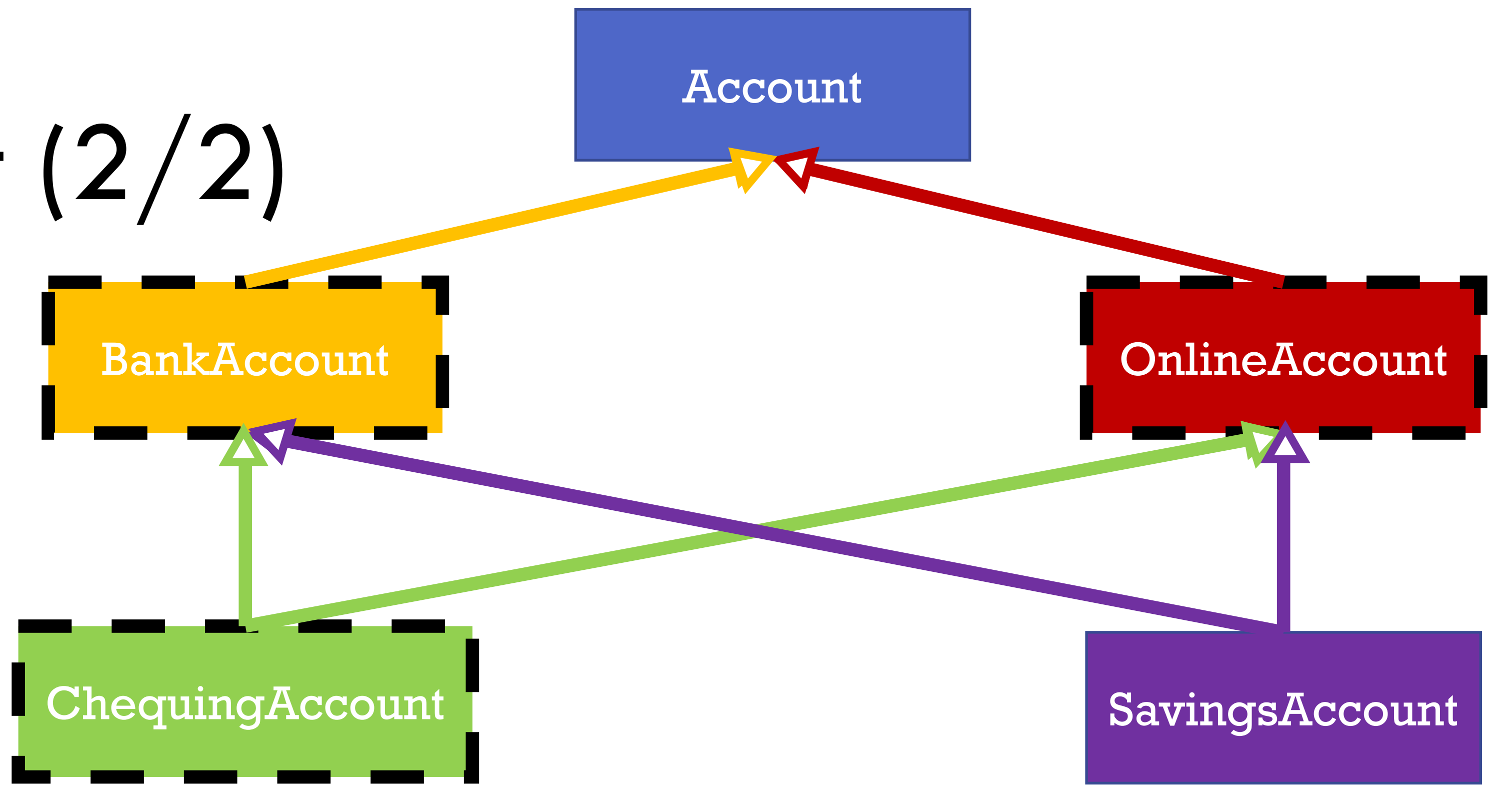
交叉广播 (1/2)



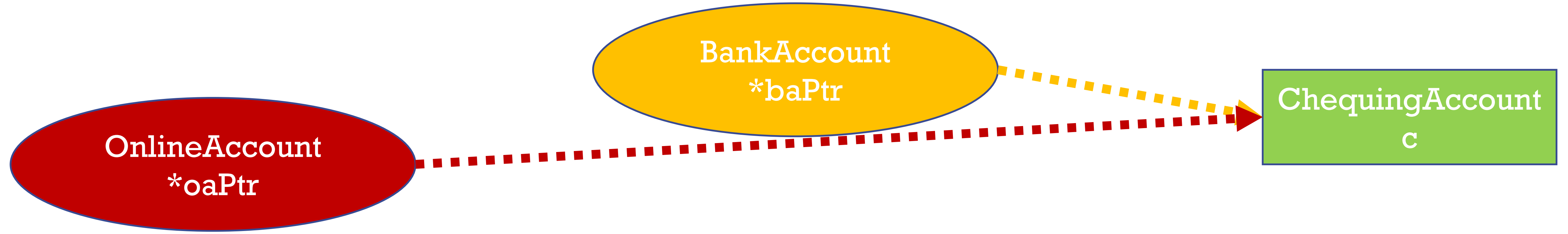
- 支票账户 c;
- BankAccount * baPtr = dynamic_cast<BankAccount *>(&c); // 指针上转型



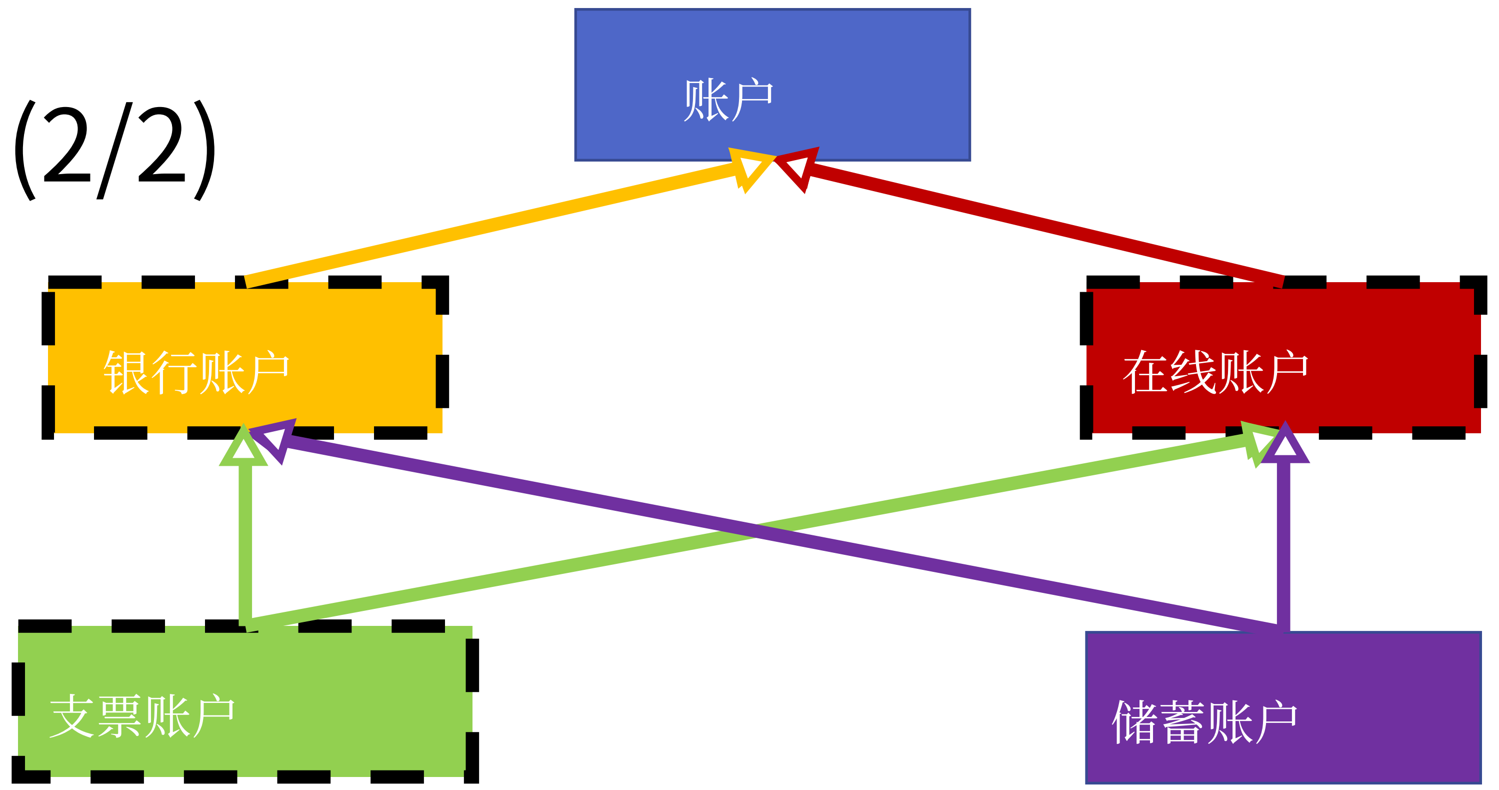
Crosscast (2/2)



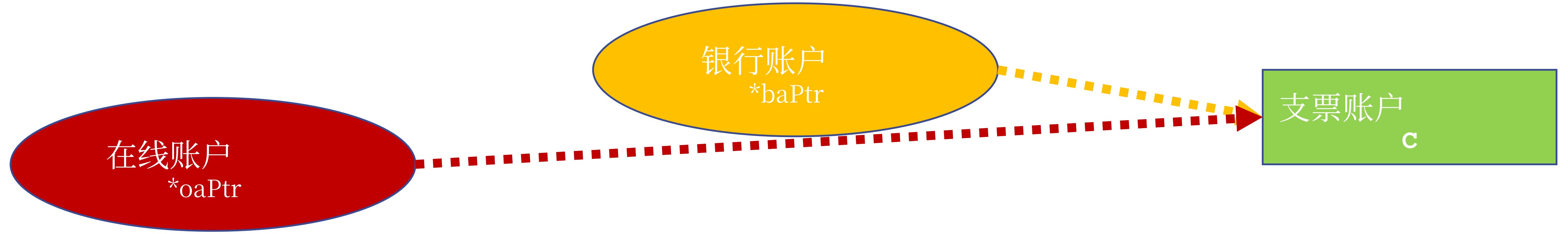
- **ChequingAccount c**;
- **BankAccount** * **baPtr** = dynamic_cast<**BankAccount** *>(&**c**); // Pointer upcast
- **OnlineAccount** ***oaPtr** = dynamic_cast<**OnlineAccount** *>(**baPtr**); // Pointer cross cast!



交叉广播 (2/2)



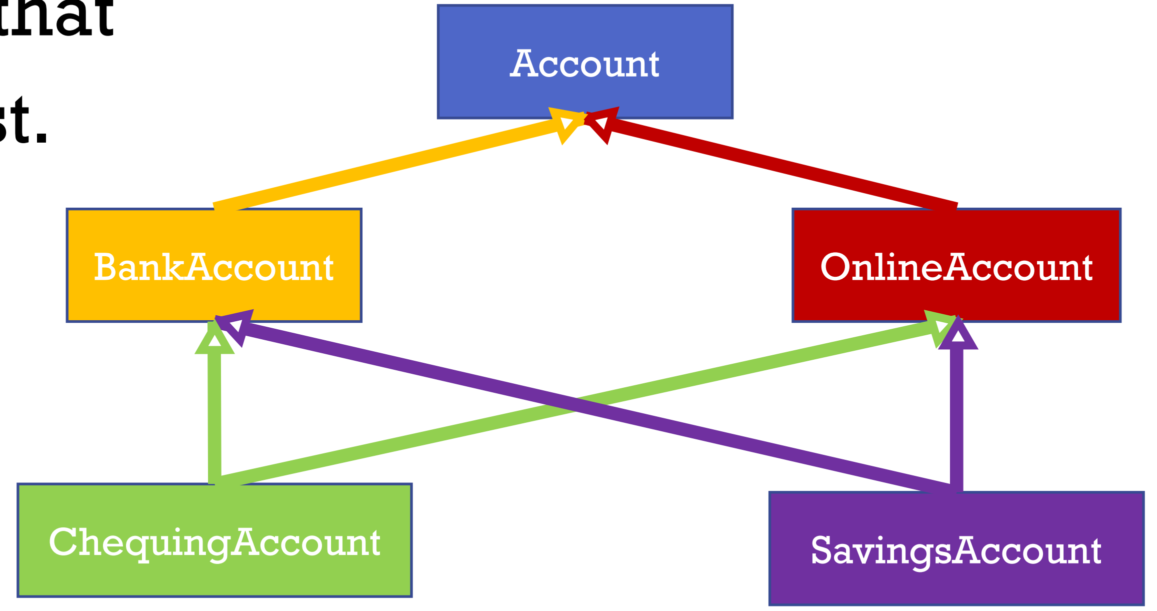
- **支票账户 c**;
- **银行账户** * **baPtr** = dynamic_cast<**银行账户** *>(&**c**); // 指针向上转换
- **在线账户** ***oaPtr** = dynamic_cast<**在线账户** *>(**baPtr**); // 指针交叉转换!



Dynamic cast example (upcast)

account.cpp

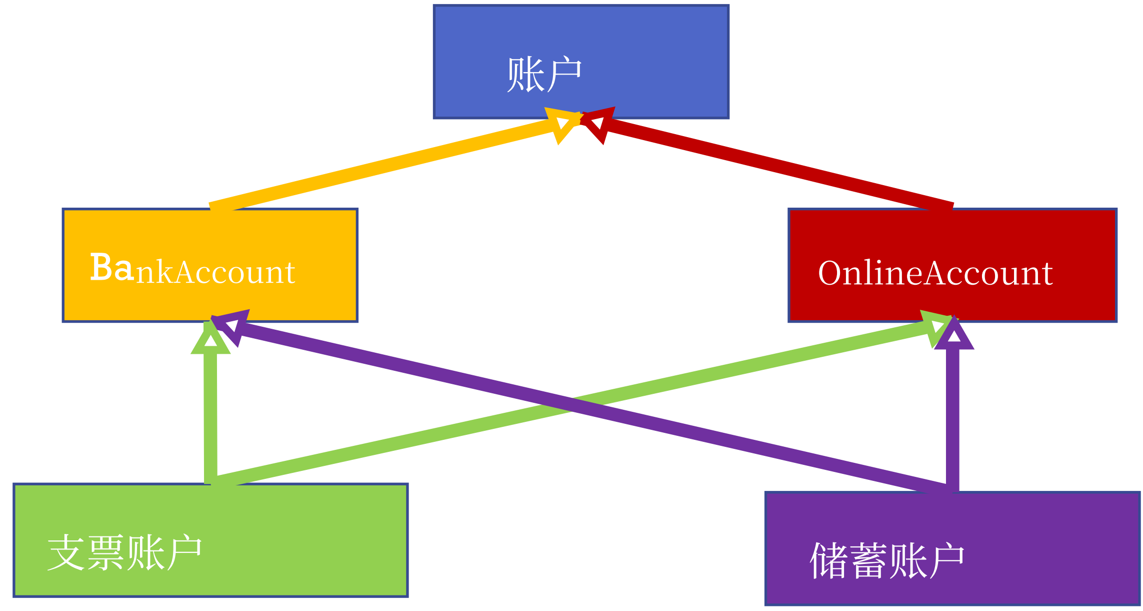
```
// Perform an upcast
accountPtr = dynamic_cast<Account *>(&c);

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



动态类型转换示例（上行转换）

account.cpp

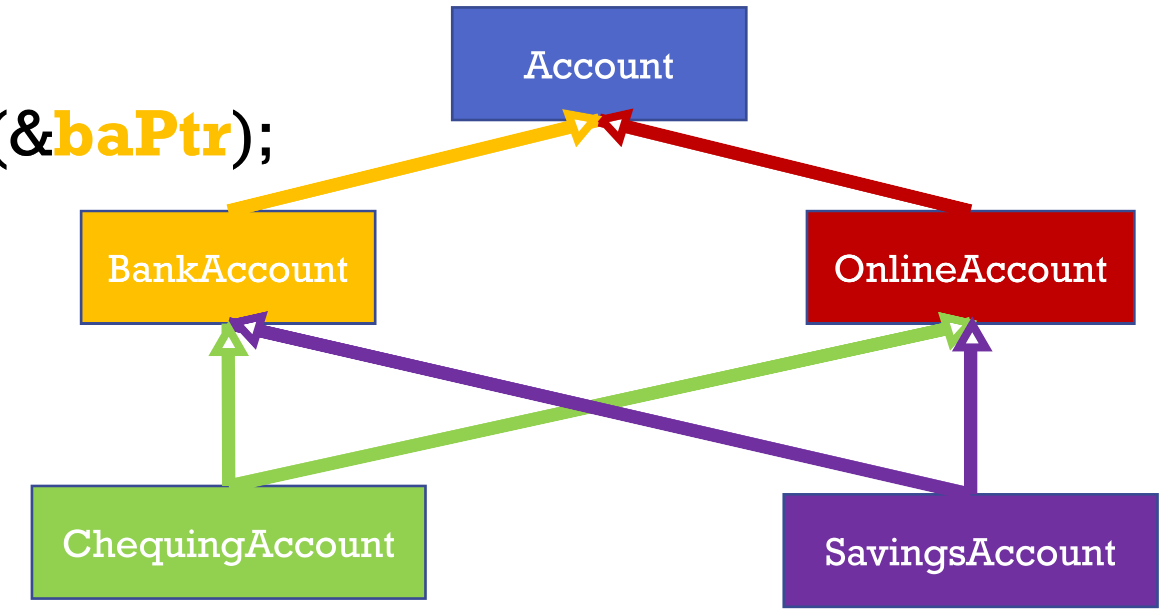
```
// 执行一次上行转换
账户指针 = 动态_转换<账户 *>(&c);

// 再次执行相同操作，以表明
// 上行转换无需进行显式类型转换。
账户指针 = &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);
```

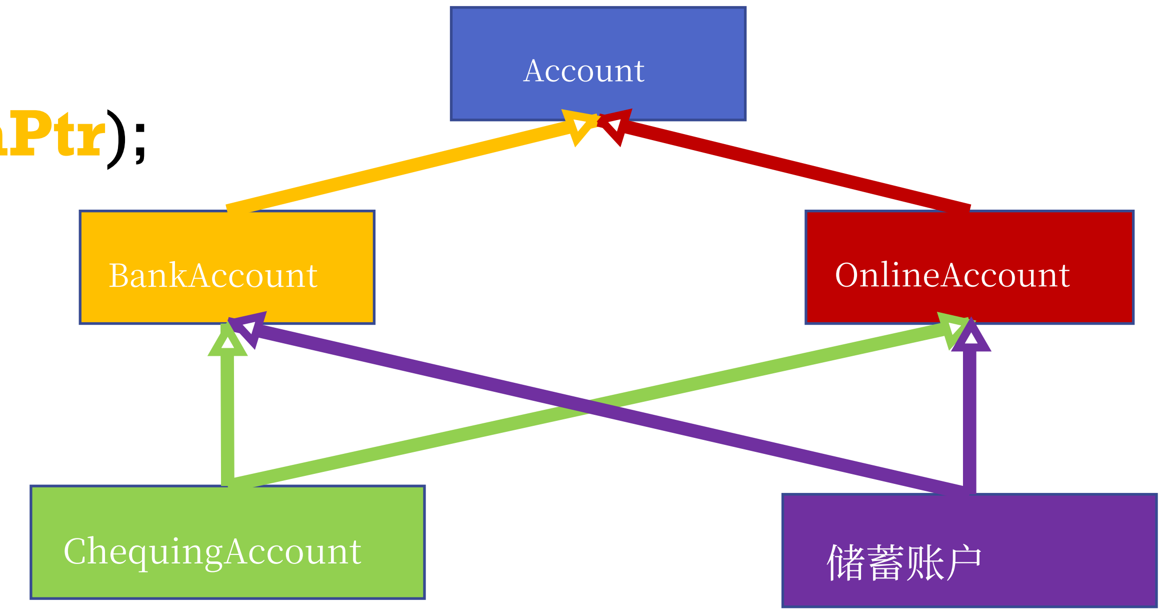


We can say that the real purpose of dynamic-cast is to allow upcasts within an inheritance hierarchy

动态类型转换示例（向下转换/横向转换）

account.cpp

```
// 执行向下转换
saPtr = 动态_转换<储蓄账户 *>（oaPtr）；
// 执行向上转换
银行账户 * baPtr = 动态_转换<银行账户 *>(&c);
// 执行横向转换
oaPtr = 动态_转换<OnlineAccount *>(&baPtr);
```



我们可以说，dynamic-cast 的真正目的是允许在继承层次结构中进行向上转型

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
- **<u>Pro tip: implement a virtual destructor with an empty implementation</u>**

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

关于动态类型转换的说明

- **仅当类为多态时才可用**
- 请记住，要成为多态类，该类必须至少有一个虚函数
- 如果尝试对非多态类进行动态类型转换，将导致编译器错误
- **专业提示：实现一个具有空实现的虚析构函数**

（题外话：查看 **typechecking.cpp** 以了解一个巧妙的技巧。）

STATIC CASTING

静态转换

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

静态转换

- static_cast 运算符是一种编译时转换
- **避免了 dynamic_cast 所做的运行时检查**
- 因此，你可以将 static_cast 运算符用于指针和 **非多态** 类型
- **只有当转换前的类型与转换后的类型可以在一个或两个方向上隐式相互转换时，该转换才有效**
- 你也可以使用它来执行一些使用 C 风格转换所进行的转换，通常是相关类型之间的转换
- static_cast 运算符的语法与 dynamic_cast 相同

ENUMS

枚举

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 ♡ 枚举

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

有没有更简单的方法来创建一系列命名常量？？？

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 ♡ 枚举

- 枚举将值限制为特定的命名常量范围
- 底层类型为整型（与 Java 相同！）
- C++ 语言的枚举可以是 **非限定作用域** 或 **限定作用域**
  - **非限定作用域** 使用 **enum** 定义
  - **限定作用域** 使用 **enum class** 或 **enum struct** 定义

I ♡ enumerations

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

vs

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

I ♡ 枚举

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

vs

```
enum color { RED, GREEN, BLUE }; //同样的结果，更少的代码
```

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}

C++ 枚举

- 一种用户定义的类型，其中包含一组 **命名的 整型 常量**，这些常量被称为 **枚举器**。

- **无作用域限定的**: **enum** [标识符] [: 类型] { 枚举列表 };
  - **枚举 颜色 : 整数** {红色, 绿色, 蓝色}
- **有作用域限定的**: enum **[class |struct]** [标识符] [:类型] { 枚举列表 };
  - **枚举 类 颜色 : 整型** {RED, GREEN, BLUE}

- 在声明的作用域内可见
- 枚举列表中的每个名称都被赋予一个整数值，该值对应其在枚举顺序中的位置。

- 提示：枚举默认为 int 类型，因此可以省略
  - 枚举 颜色 : **整型** {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;
}
```

枚举

我们可以像这样使用**命名常量**:

```
枚举 颜色 { 红色, 绿色, 蓝色 };
颜色 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 };
```

枚举

- 如果未提供，则枚举值从 0 开始（就像 Java 一样）
- 但在 C 语言中++ **我们可以指定底层值**:

```
//a = 0, b = 1, c = 10, d = 11, e = 1
//f = 2, g = 12
枚举 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
```

枚举

- 值可以**转换为整数类型**:

```
枚举 颜色 { 红色, 黄色, 绿色 = 20, 蓝色 };颜色 我的_颜
色 = 红色; int n = 蓝色; // 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;
```

枚举

- 整数、浮点数及其他枚举类型的值可以通过**静态_转换**或显式 转换，转为任意枚举类型

```
枚举 状态 { OPEN = 1, CLOSED = 2, ERROR = 3 };状态 文
件_状态 = 静态_转换<状态>(3);cout << 文件_状态 =
<< 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;
```

枚举

- 我们可以 **省略枚举的名称**
- 我们只能在封闭的作用域中使用枚举符

```
// 定义了 = 0、b = 1、c = 0、d = 2
枚举 { a, b, c = 0, d = a + 2};

// 输出 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;
```

枚举

```
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 作用域枚举

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**

关于作用域枚举呢?

- 到目前为止，我们已经了解了无作用域的枚举
- **没有隐式转换**从**作用域枚举器**的值到整型
- **static_cast** 可用于获取枚举器的数值
- 优点：**无命名冲突**

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!
```

作用域示例

```
枚举 动物 {狗, 猫, 乌鸦};
enum FlyingAnimals {CROW, SPARROW}; //错误 CROW 已存在

//与作用域枚举相比
enum class Fruit {APPLE, ORANGE, LEMON}; //LEMON 在
Fruit 作用域中
    enum class YellowFruit {BANANA, LEMON}; //
LEMON 在 YellowFruit 作用域中，没问题！
```

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
```

enums.cpp

示例

枚举 **类** 颜色 { 红色, 绿色 = 20, 蓝色 }; 颜色 r = 颜色::
蓝色; switch(r){

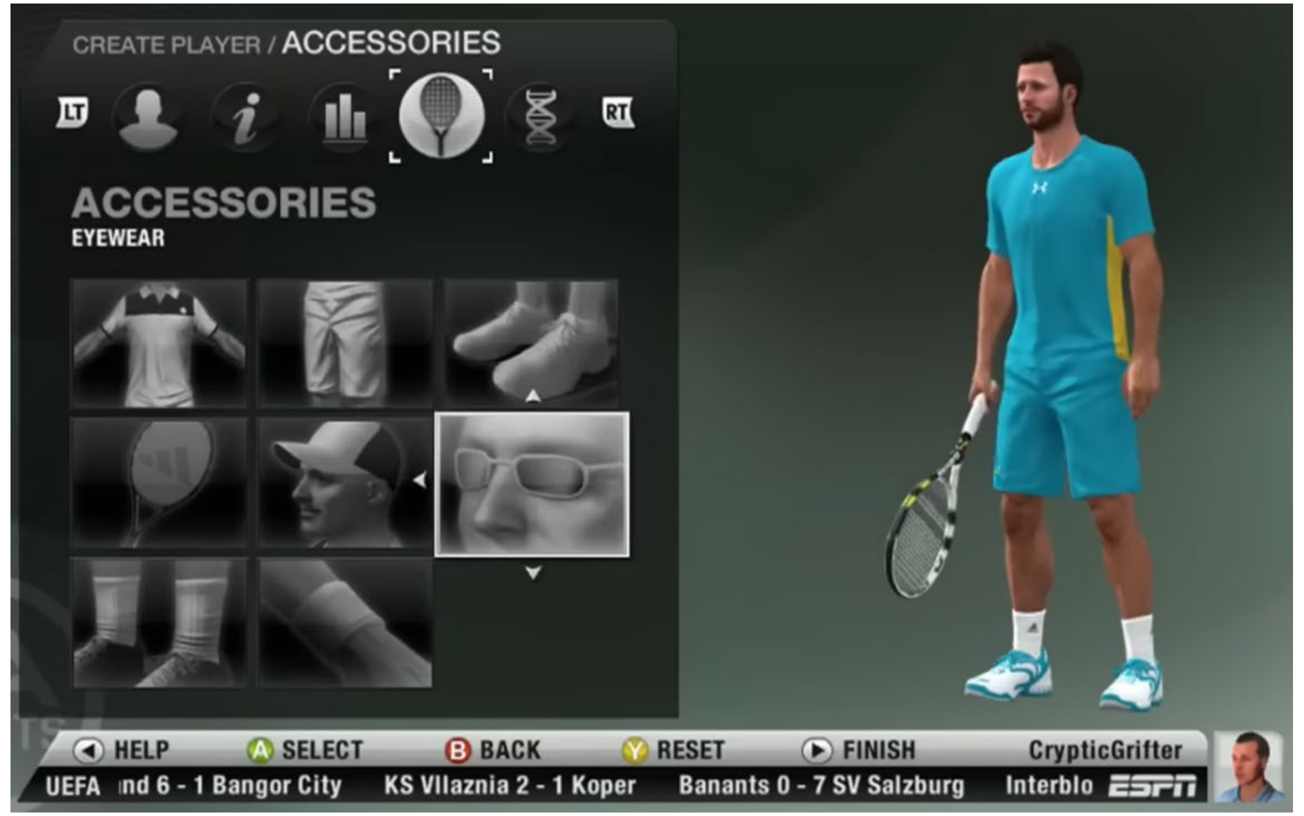
情况 颜色::红色 : cout << "红色\n"; 中断; 情况 颜色::绿色:
cout << "绿色\n"; 中断; 情况 颜色::蓝色 : cout << "蓝色\n"; 中
断;

}int n = r; // **错误：没有作用域枚举到 int 的转换** int n =
静态_转换<int>(r); // 正确， n = 21

enums.cpp

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?



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

现实世界示例：在电子游戏中表示服装

- 你的游戏中需要有有限数量的服装选项
- 每件服装都需要有一个唯一的 ID
- 必须有一种方法来找出以下类别的数量：
    - 仅鞋子
    - 仅裤子
    - 所有项目
- 我们如何仅用枚举实现这一点?

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

实际案例：在视频游戏中表示服装

```
enum clothingId {
   鞋_1, //0
   鞋_2, //1
   鞋_3, //2
   PANT_1， //3
   裤子_2, //4
   PANT_3， //5
   裤子_4, //6
   裤子_5  //7
};
```

- 所有服装都有唯一的 ID
- 但无法做到：
  - 获取所有不同服装的总数
  - 仅获取鞋子
  - 仅获取裤子

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

现实世界示例：在视频游戏中表示服装

```
枚举 clothingId {
  SHOE_开始,               //0
  鞋_1 = 鞋_开始,          //0
  SHOE_2,                  //1
  鞋_3,                    //2
  鞋_结束 = 鞋_3,          //2
  裤子_开始,               //3
  裤子_1 = 裤子_开始,      //3
  裤子_2,裤子              //4
      _3,                  //5
  PANT_4,                  //6
  裤子_5,                  //7
  PANT_结束 = 裤子_5,      //7
  衣物数量_                //8
};
```

- 在枚举列表中创建"**书签**"
  - **鞋_开始**, **SHOE_END**, **PANT_开始**, **PANT_结束**, **数字_CLOTHES** 表示特殊的枚举列表中的点
  - 大多数具有与现有枚举相同的值
- 使用这些"**书签**"作为分隔鞋子和裤子的点
- 我们现在可以:
  - 获取不同服装的总数
  - 仅获取鞋子
  - 仅获取裤子

clothingEnum.cpp

Agenda

Design patterns intro

1. Singleton
2. Observer

COMP 3522

议程

设计模式简介

1. 单例模式
2. 观察者模式

COMP 3522

DESIGN PATTERNS

设计模式

What are Design Patterns

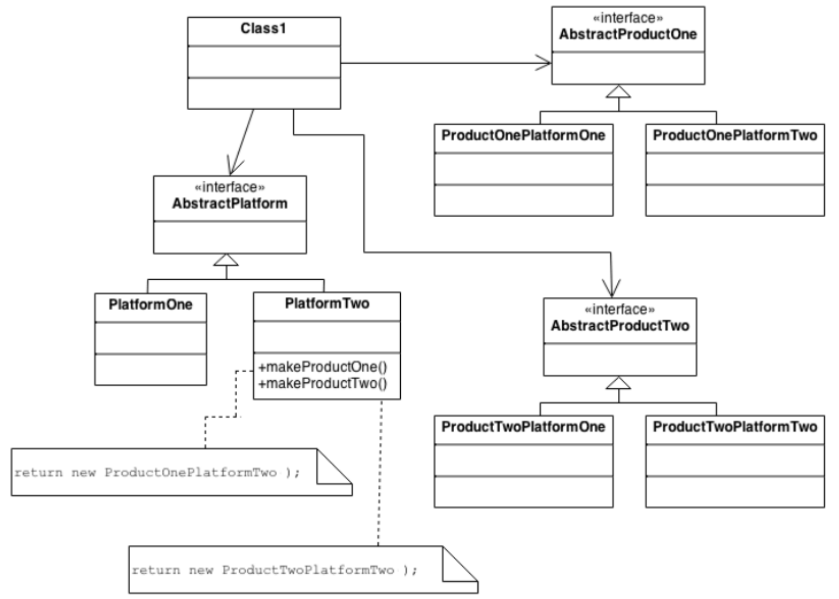
- Common design solutions to common architectural problems
- How can I write systems so Classes can:
  - **communicate** with each other with low coupling?
  - be combined to form new **structures?**
  - be **created** with different strategies and techniques?

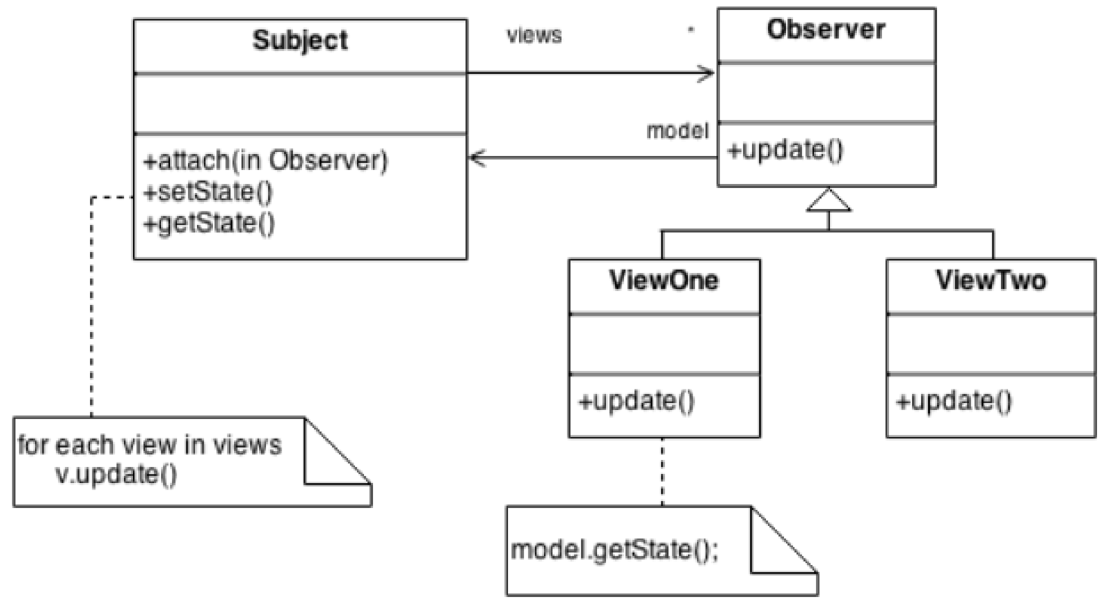
什么是设计模式

- 针对常见架构问题的通用设计方案
- 如何编写系统，使得类可以：
  - **相互通信**且保持低耦合?
  - 以不同方式组合形成新的 **结构?**
  - 使用不同的策略和技术被**创建**?

What are Design Patterns

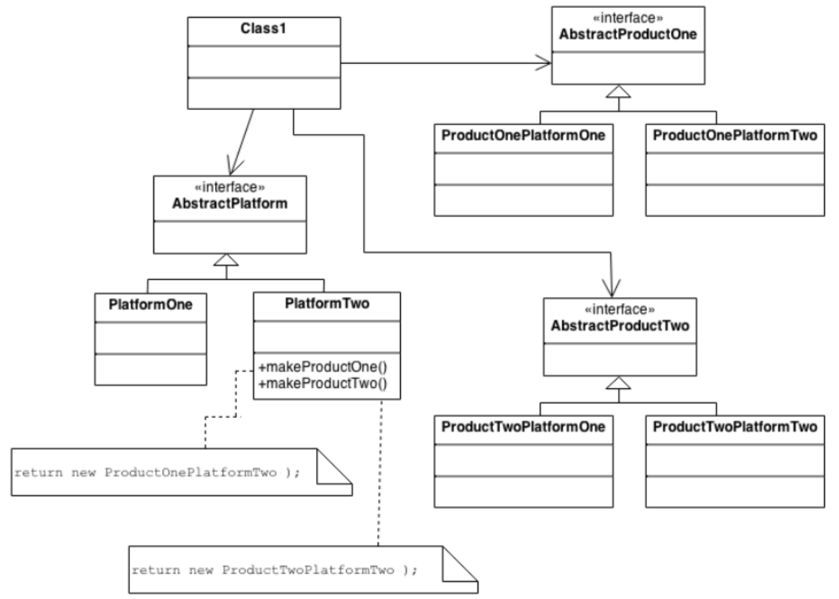
- Think of these as recipes or templates to solve common design problems
- “Structuring code/classes in this way will solve specific design issues”

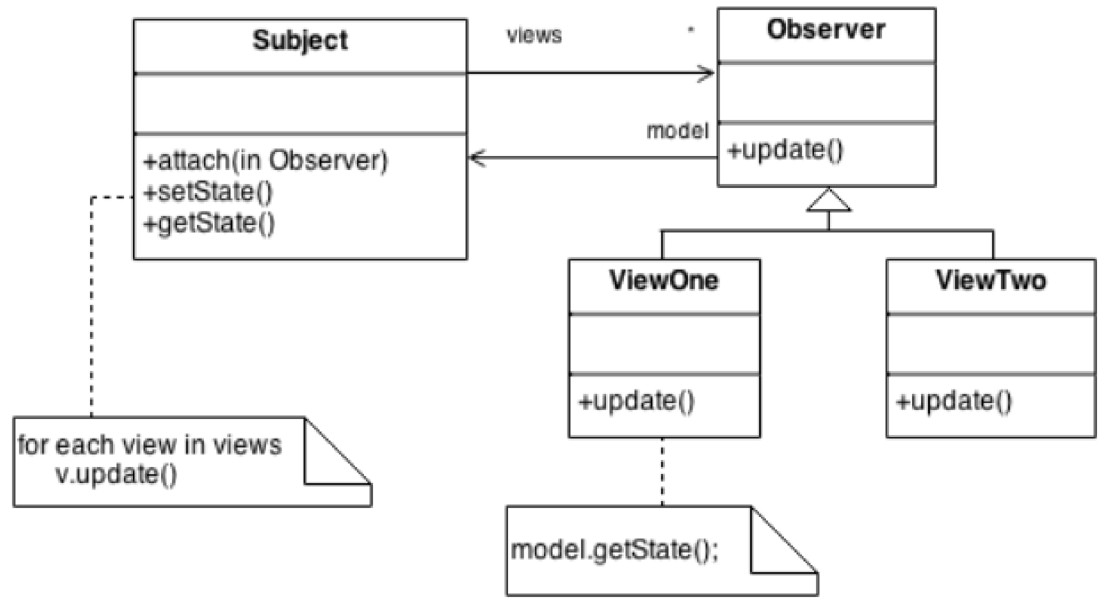




什么是设计模式

- 将这些视为解决常见设计问题的配方或模板
- “以这种方式组织代码/类将解决特定的设计问题”





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.

设计模式 - 优势

- 不要重复造轮子，使用经过验证的解决方案
- 具有抽象性，可应用于不同的问题
- 促进开发者之间的思想和概念交流
- 与语言无关，可应用于大多数（即使不是全部）面向对象程序

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.

设计模式 - 缺点

- 可能使系统更加复杂，更难维护。模式看似‘简单’，实则具有误导性。
- 系统可能会遭受模式过度使用的问题。
- 所有模式都有一些缺点，并会为系统增加约束。因此，开发人员可能需要添加原本未计划的约束。
- 无法直接促进代码复用。

Categorizing Design Patterns

- **Behavioural**

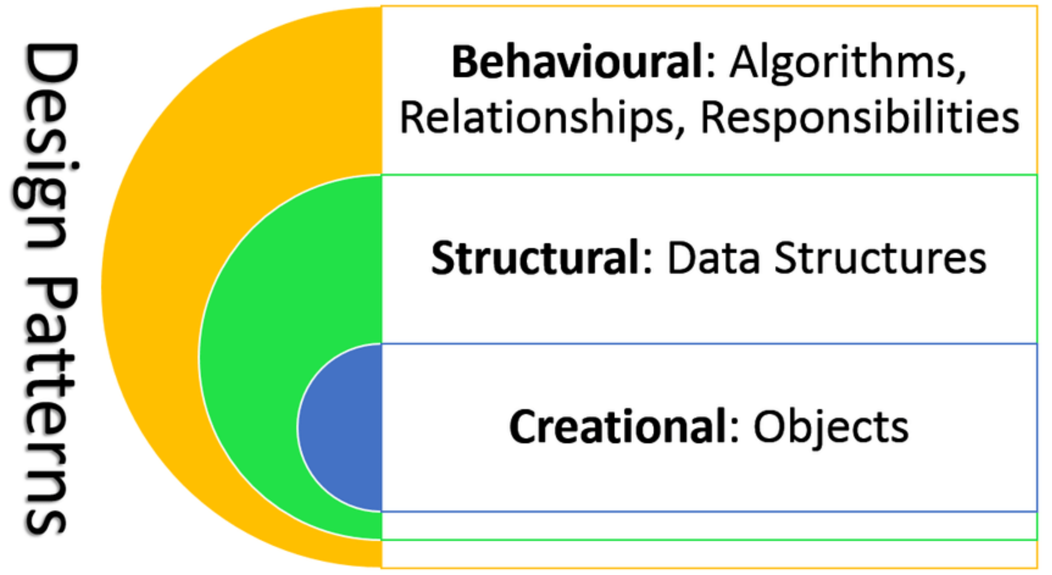
Focused on **communication and interaction between objects.** How do we get objects talking to each other while minimizing coupling?

- **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

设计模式的分类

- **行为型**

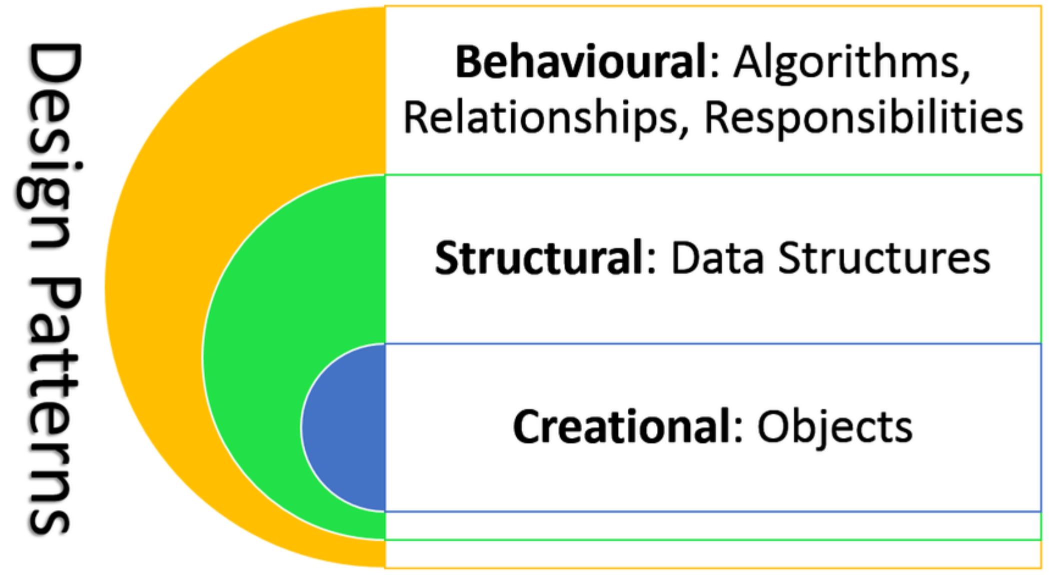
专注于 **通信和交互** **tion**
对象之间的。我们如何在最小化耦合的同时
让对象之间进行通信?

- **结构型**

类和对象如何 **组合以形成**
结构 在我们的程序中? 应专注于架构设计，以
实现最大程度的灵活性和可维护性。

- **创建型**

关于类实例化的所有内容。不同的
实例化对象或一组对象的策略和技术



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.

选择一种模式

Step 1	• 从业务依赖、模块化和抽象概念的角度来理解你所面临的问题。
Step 2	• 判断这是一个行为型、结构型还是创建型问题?
Step 3	• 是否有必须遵守的约束条件?
Step 4	• 是否存在可行的更简单解决方案?如果不存在,则选择一种模式。

SINGLETON

单例

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?**

简介

- 我需要一个系统，其中存在一个可从代码任何位置访问的单一对象

- **如何在不使用全局变量的情况下实现这一点?**

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**

单例设计模式：非常简单的一种！

- **设计模式分类：创建型**
- 有时我们希望确保某个类 **只有一个实例** 被创建
- 我们希望 **防止构造出多个副本**
- 我们必须编写代码来强制执行这一规则
- 我们希望采用 **单例设计模式**

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

单例模式

1. **在首次使用时实例化**该对象
2. **理想情况下隐藏**私有构造函数
3. **公开**一个公共的 `get¯instance` 函数，该函数返回类的静态实例引用
4. **提供**对单一对象的“全局”访问

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

That's it.
Really, that's all there is to it.

我们为什么/如何使用它?

在系统中需要某个类型的对象有且仅有一个时，**使用单例模式**。

单例是一个全局可访问的类，我们保证该类只创建一个实例。

就是这样。真的，仅此而已。

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

Code sample (so easy!)

```
class singleton
{
    public:
        static singleton& get_instance() //允许公开访问单例
        {
            static singleton instance; //静态变量确保仅存在一个单例实例 return instance; // 首次使
            用时实例化。
        }
    private:
        int test_value;
        singleton() {} //将构造函数设为私有以隐藏

    public:
        singleton(singleton const&)       = delete; //防止复制单例void operator=(singleton
        const&)  = delete; //防止赋值拷贝单例 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

应用程序 – 游戏画面管理

- 游戏有多个画面
  - 开始、游戏界面、游戏结束、商店等
- 不同的画面必须能够在代码的不同位置显示
  - 商店逻辑需要显示商店画面
  - 游戏逻辑需要显示开始/游戏/游戏结束画面
  - 设置逻辑希望显示设置界面
- 需要一个**中心位置**来按需调用和加载特定界面

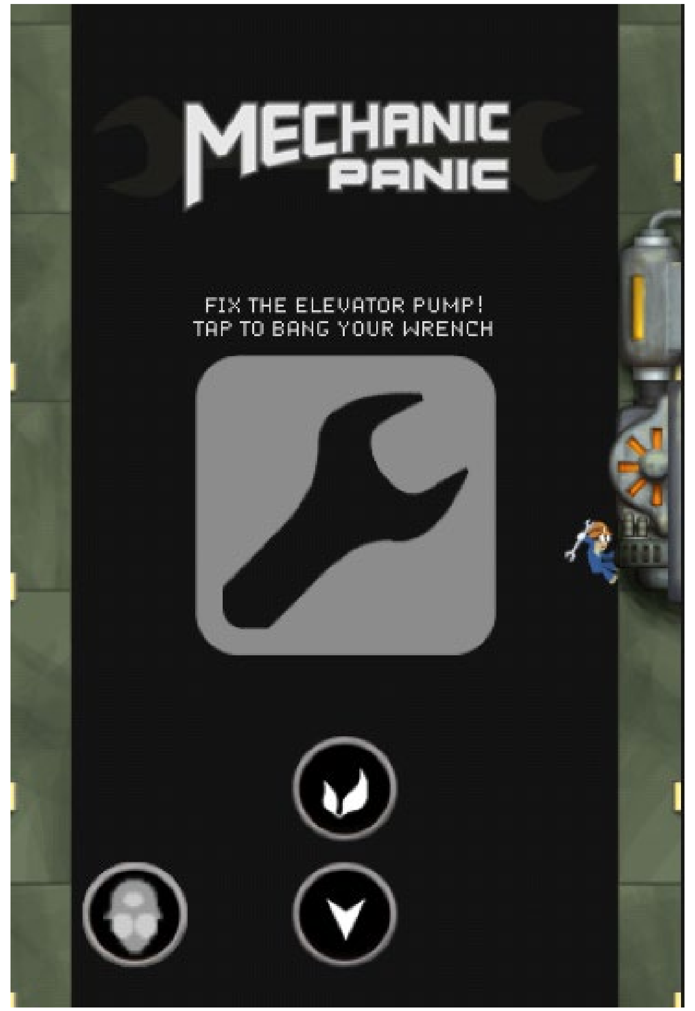
Application – Game screen management

- 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

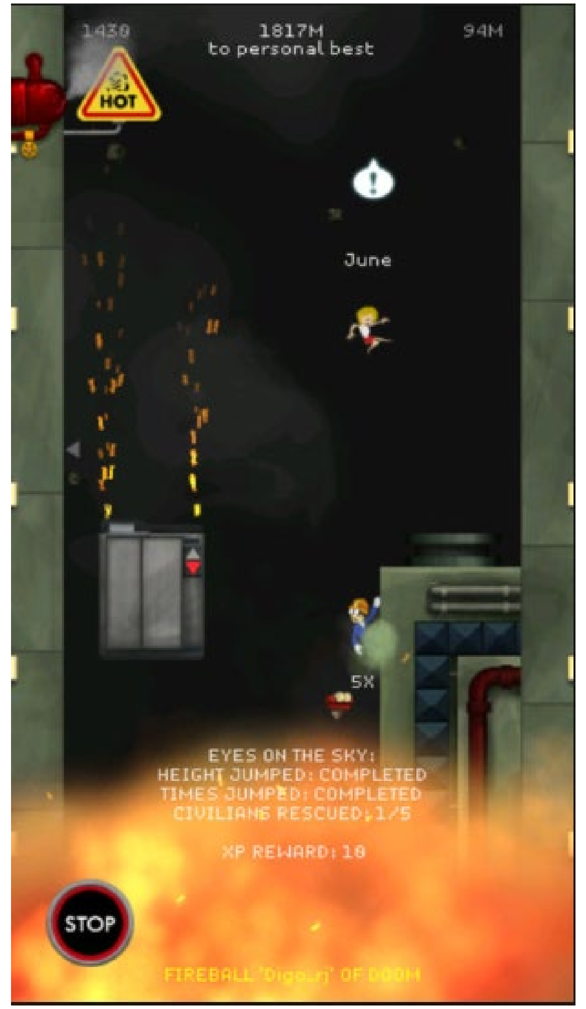
应用程序 – 游戏画面管理

- 为每个游戏状态创建枚举： `MAIN_MENU`, `GAMEPLAY`, `GAME_OVER`
- 为不同画面创建 Screen 类：MainMenu、Gameplay、GameOver
- 创建一个 **ScreenManager 单例**
  - 包含：
    - GameState 枚举到 Screens 的映射表
    - 屏幕栈
  - 负责将屏幕压入或弹出屏幕栈
- 拥有一个函数 `show(GameState gs)` ，该函数接受一个 GameState 枚举，并将来自映射的屏幕推送到屏幕栈中

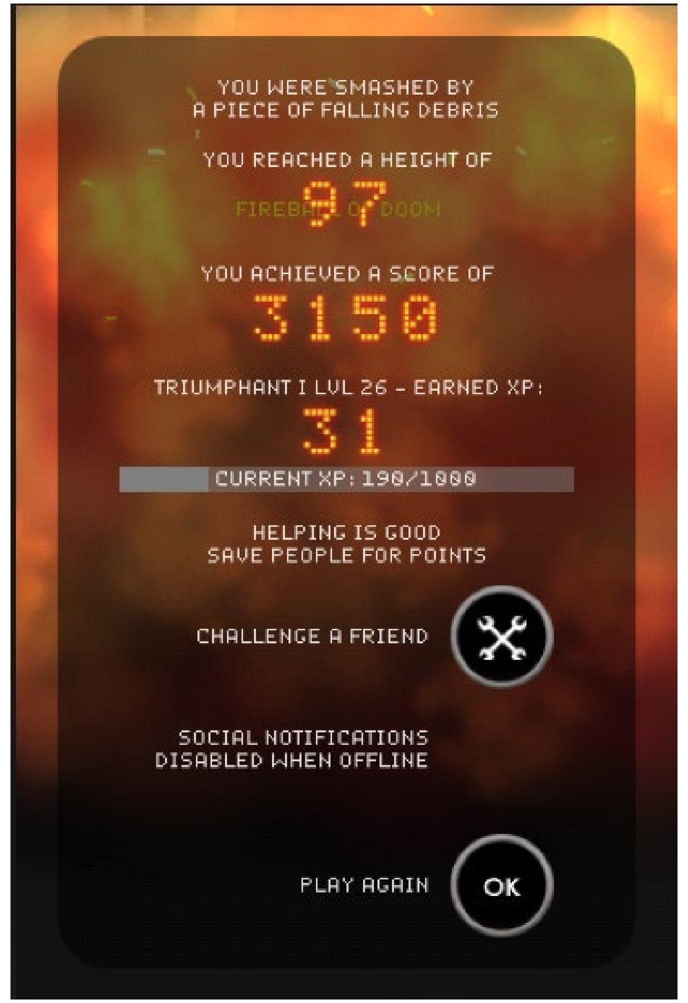
Mechanic Panic – Singleton screens example



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

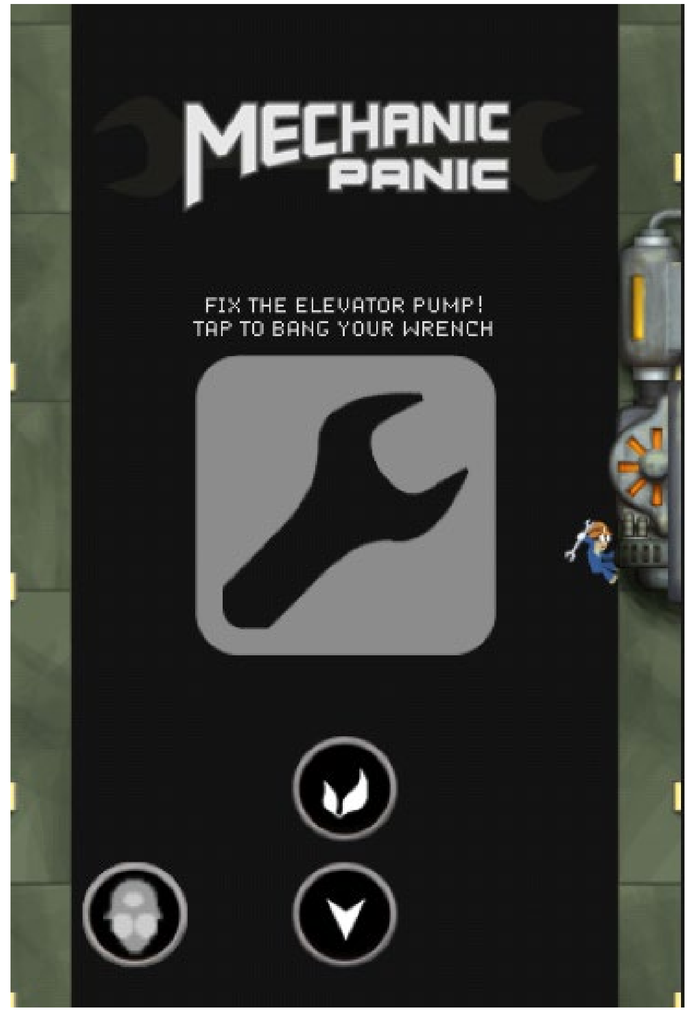


GameState enum: GAMEPLAY
ScreenManager::**getInstance()**.show(GAMEPLAY);

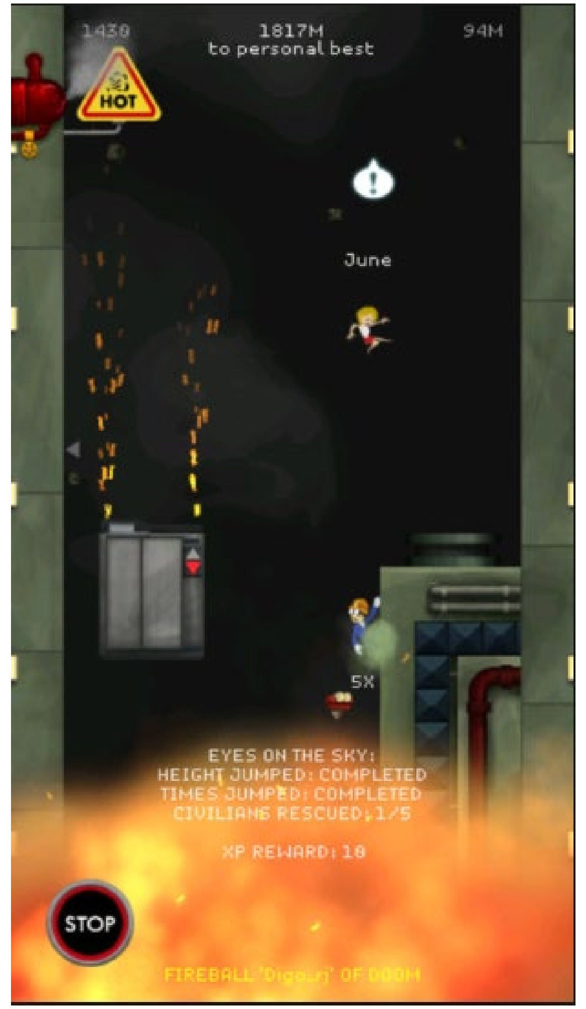


GameState enum: GAME_OVER
ScreenManager::**getInstance()**.show(GAME_OVER);

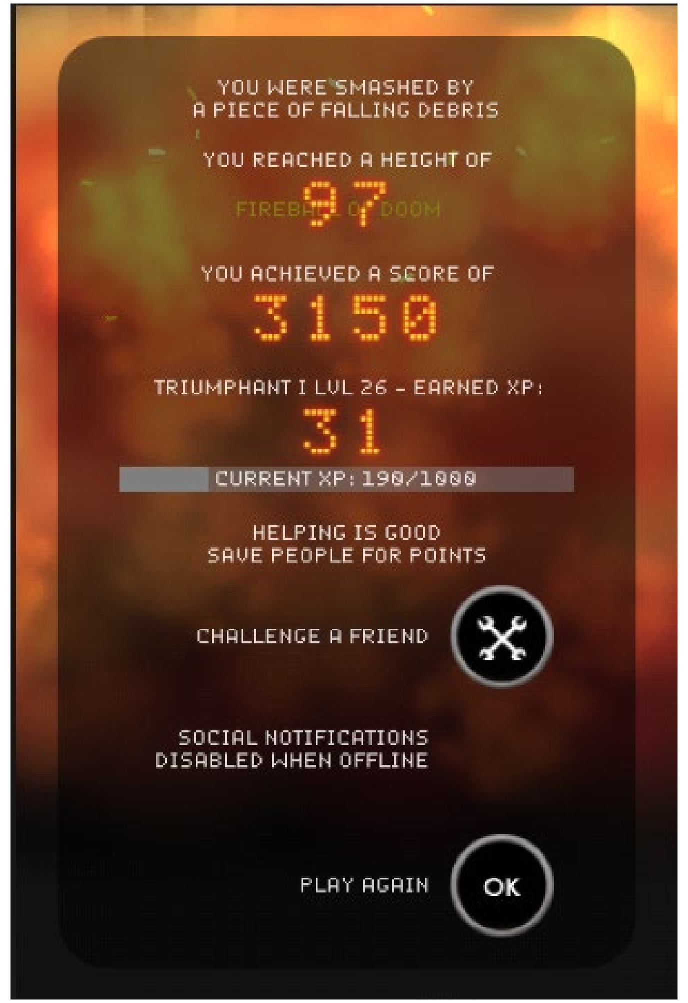
机械恐慌 – 单例屏幕示例



游戏状态枚举：　主菜单_
ScreenManager::**getInstance()**.显示(主_菜单);



游戏状态枚举：　游戏进行中
ScreenManager::**getInstance()**.显示(GAMEPLAY);



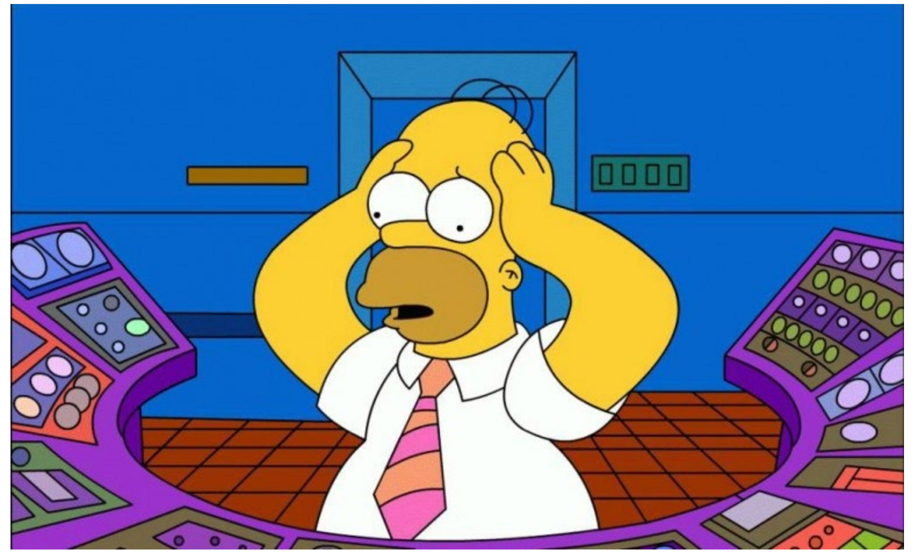
游戏状态枚举：　游戏结束_
ScreenManager::**getInstance()**.显示(游戏_结束);

OBSERVER

观察者

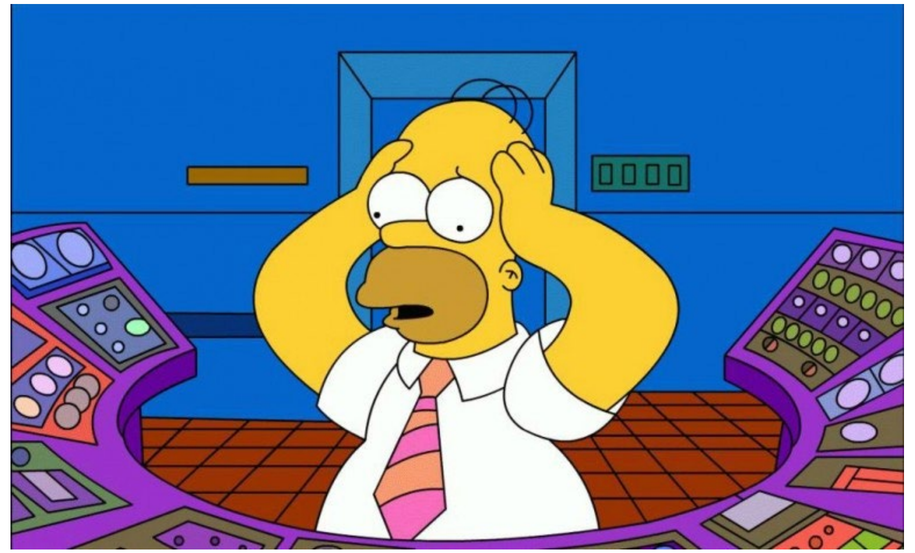
Introduction

- 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?



简介

- 我想创建一个系统，其中 一个对象 可以 **向多个对象广播信息** 到 **多个对象**
- 如果系统的某一部分状态发生变化，我该如何通知许多不同类型的对象，而不会使该部分与其余部分发生**紧密耦合**?



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!

简介

- 我们倾向于将系统划分为协同工作的类
- 这些类共享信息
- 我们需要保持一致性
- 但我们不能将它们紧密耦合，因为这会降低灵活性
- 我们使用一种在编程中常见的惯用法，称为发布-订阅：
  - 该**主题**发布通知时并不知道谁是**观察者**
  - 任意数量的**观察者**都可以订阅以接收通知*

* 对我来说听起来有点像 Java GUI 监听器！

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

观察者设计模式

- **设计模式分类：行为型**
- **观察者模式**描述了如何建立这些关系
- 有两个关键对象：
  1. **主题** 可以有任意数量的依赖 观察者
  2. **观察者** 在主题状态发生改变时都会收到通知主题 状态发生变化时
- 每个 观察者 查询 主题 以同步它们的状态
- 该模式确保当一个 主题 改变状态时，其所有 观察者 都会自动收到通知

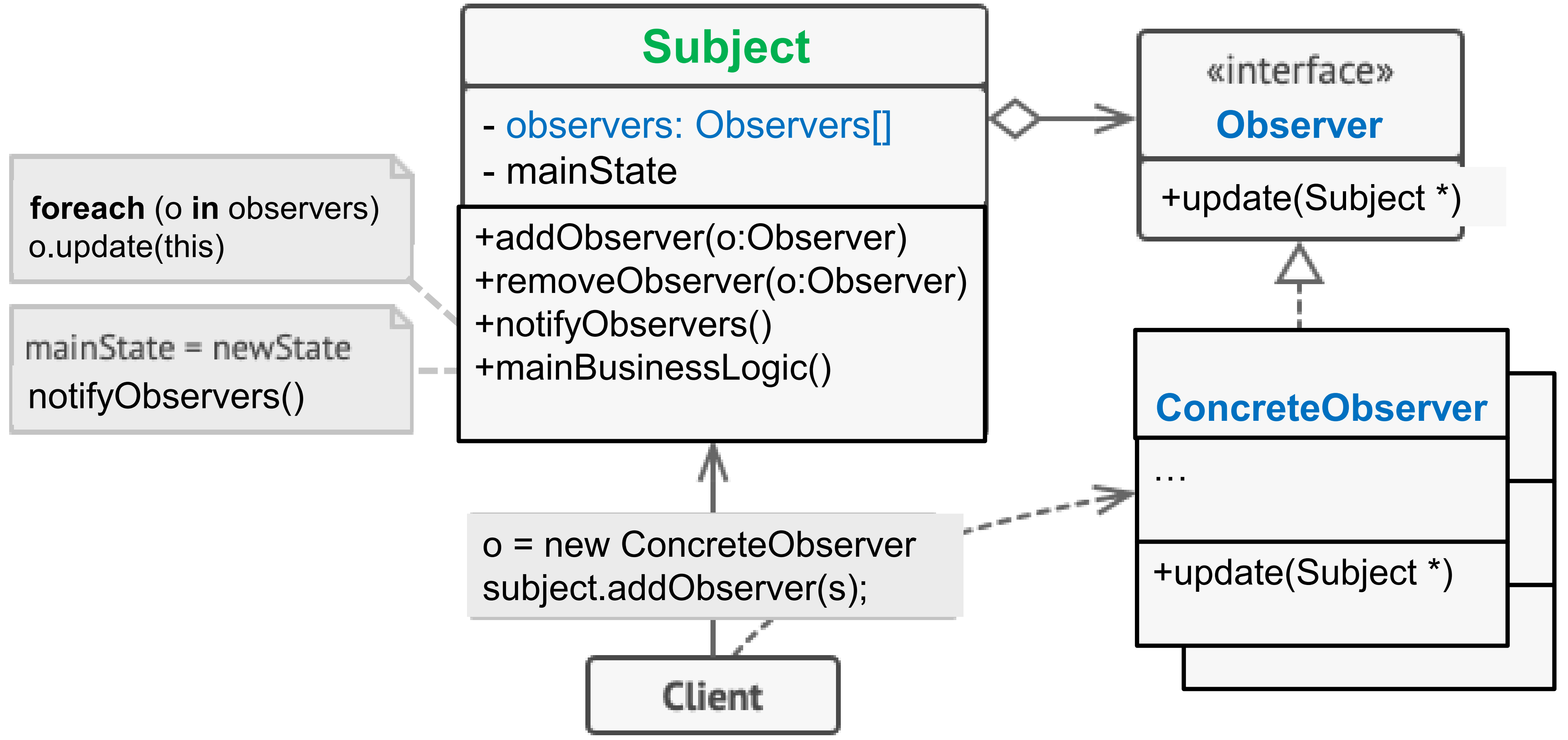
Observer design pattern

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

观察者设计模式

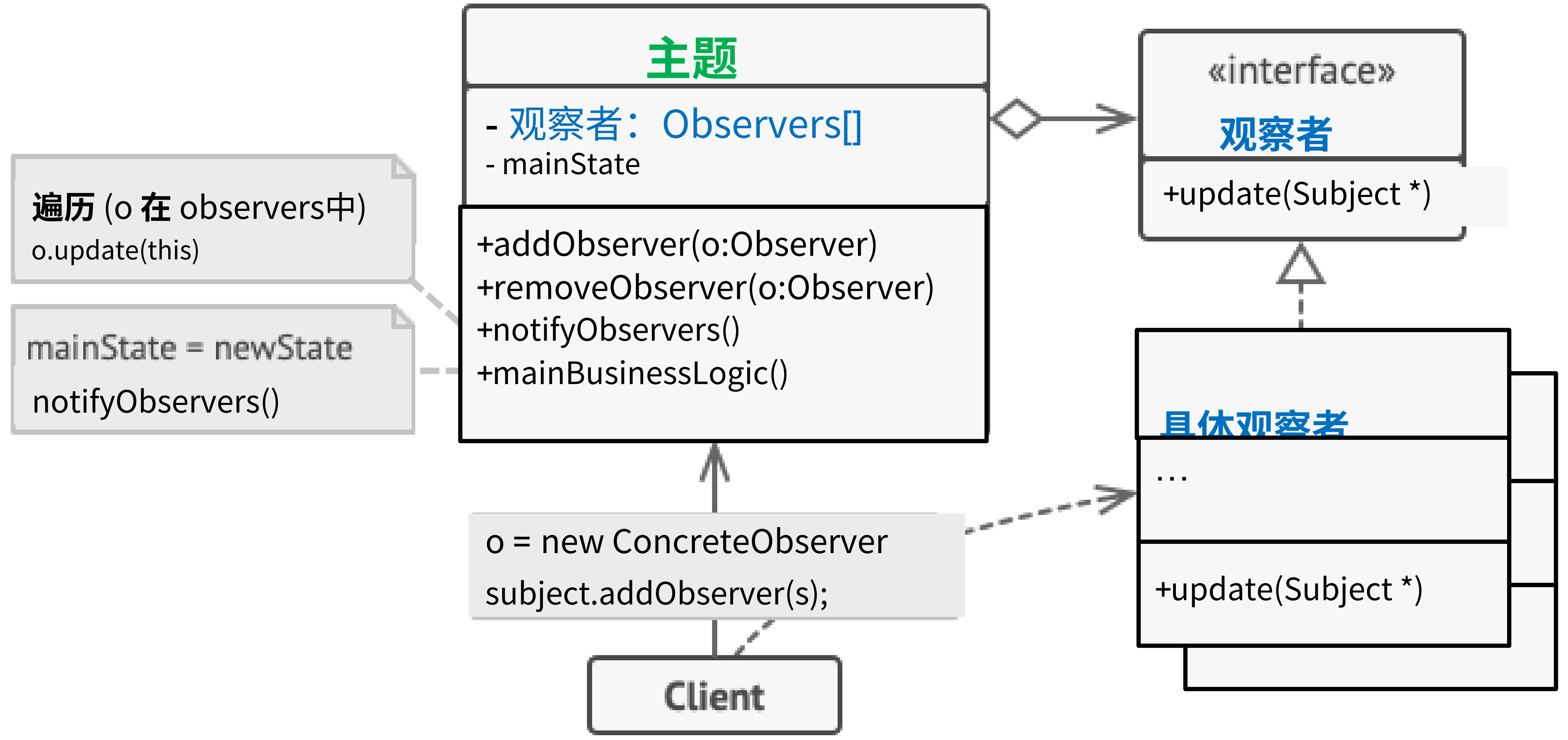
1. （抽象）**主题**
   - 知道其观察者
   - 任意数量的观察者可以观察一个主题
   - 提供一个用于添加和移除**观察者**的接口观察者
2. （抽象）**观察者**
   - 为应被通知主题变更的对象定义一个**更新接口**
3. **具体主题**
   - 向其观察者发送其状态变更的**通知**
4. **具体观察者**
   - 维护对一个ConcreteSubject对象的引用
   - 存储需要与主题的状态保持一致的信息
   - 实现 Observer 更新接口

Observer design pattern: Class diagram



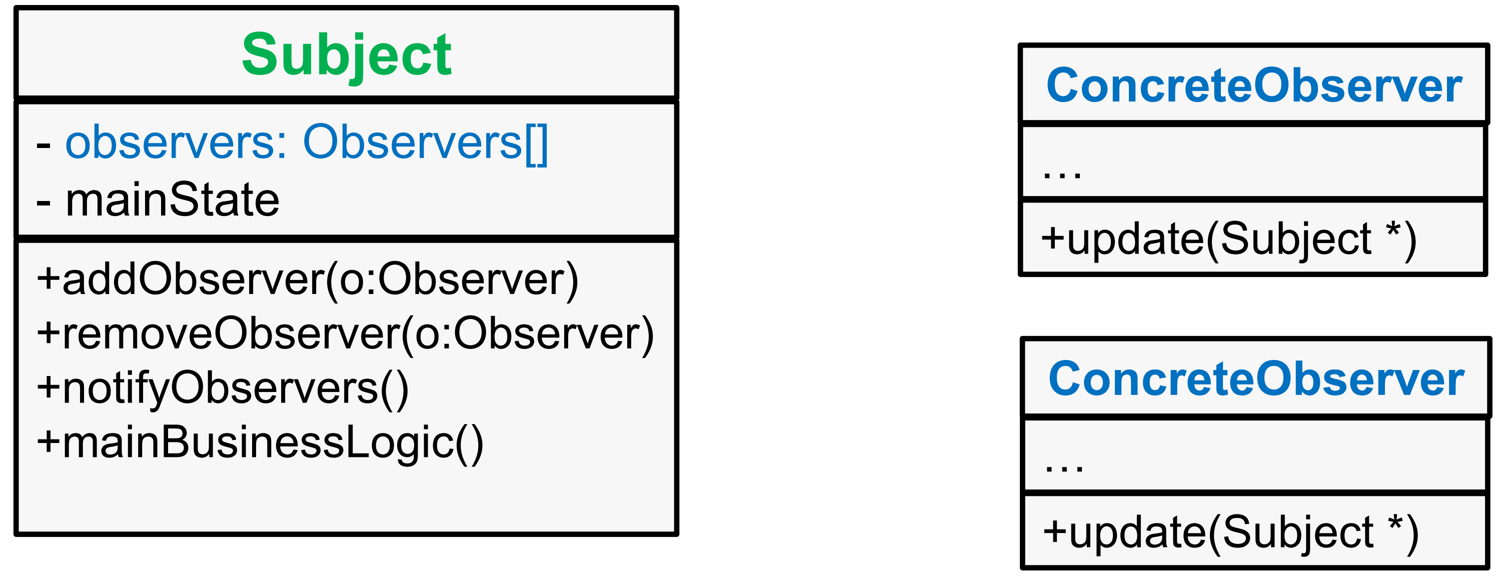
https://refactoring.guru/design-patterns/observer

观察者设计模式：类图



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

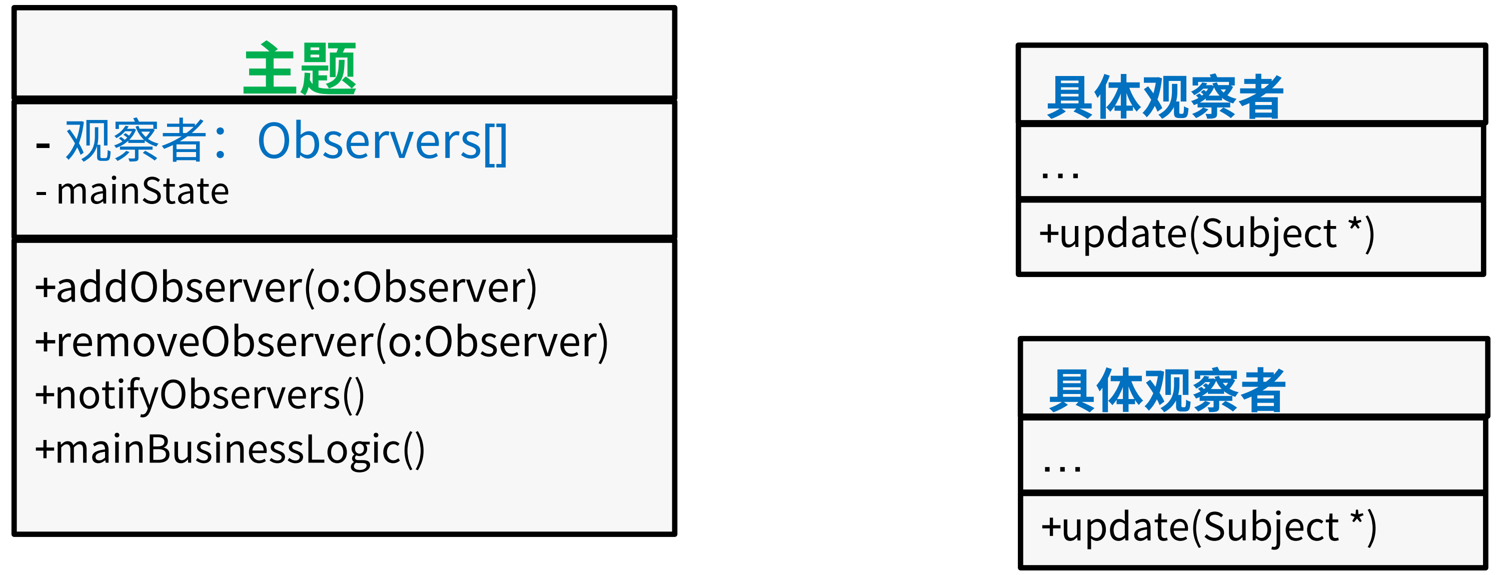
1. Create subject and observers



Instantiate subject, and 2 observer objects

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

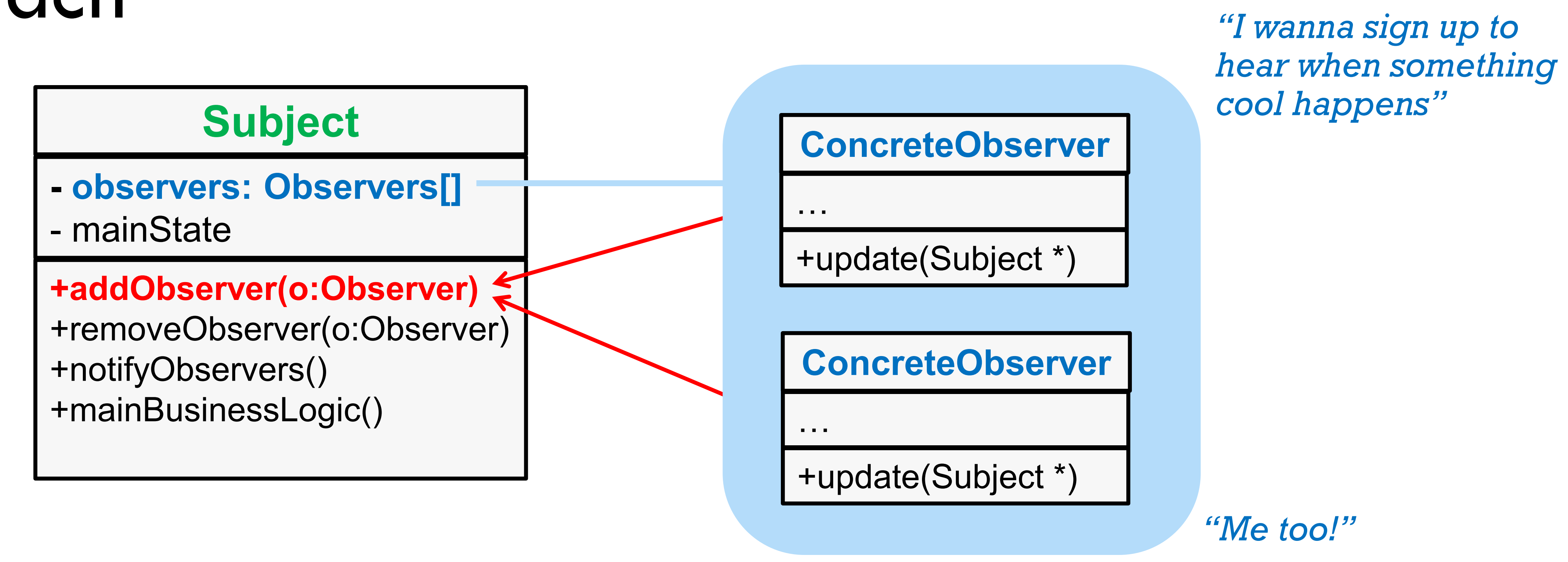
1. 创建主题和观察者



实例化 主题， 以及 2 个观察者对象

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

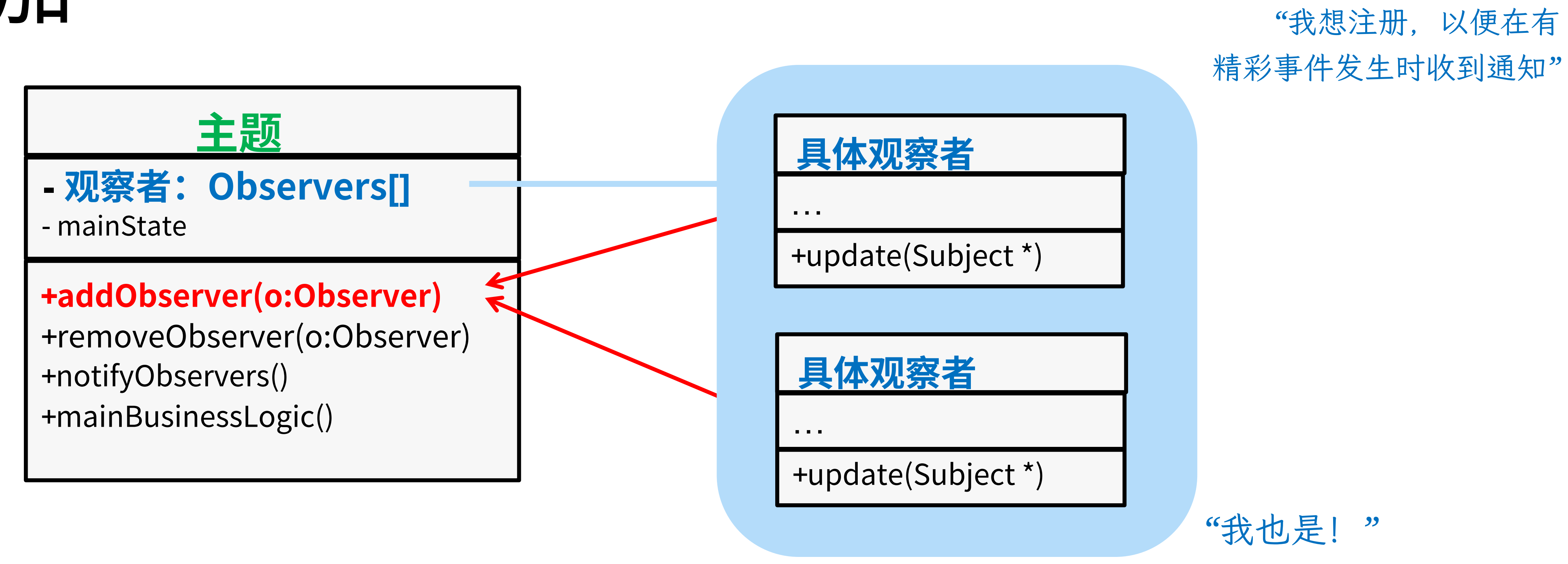
2. Attach



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

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

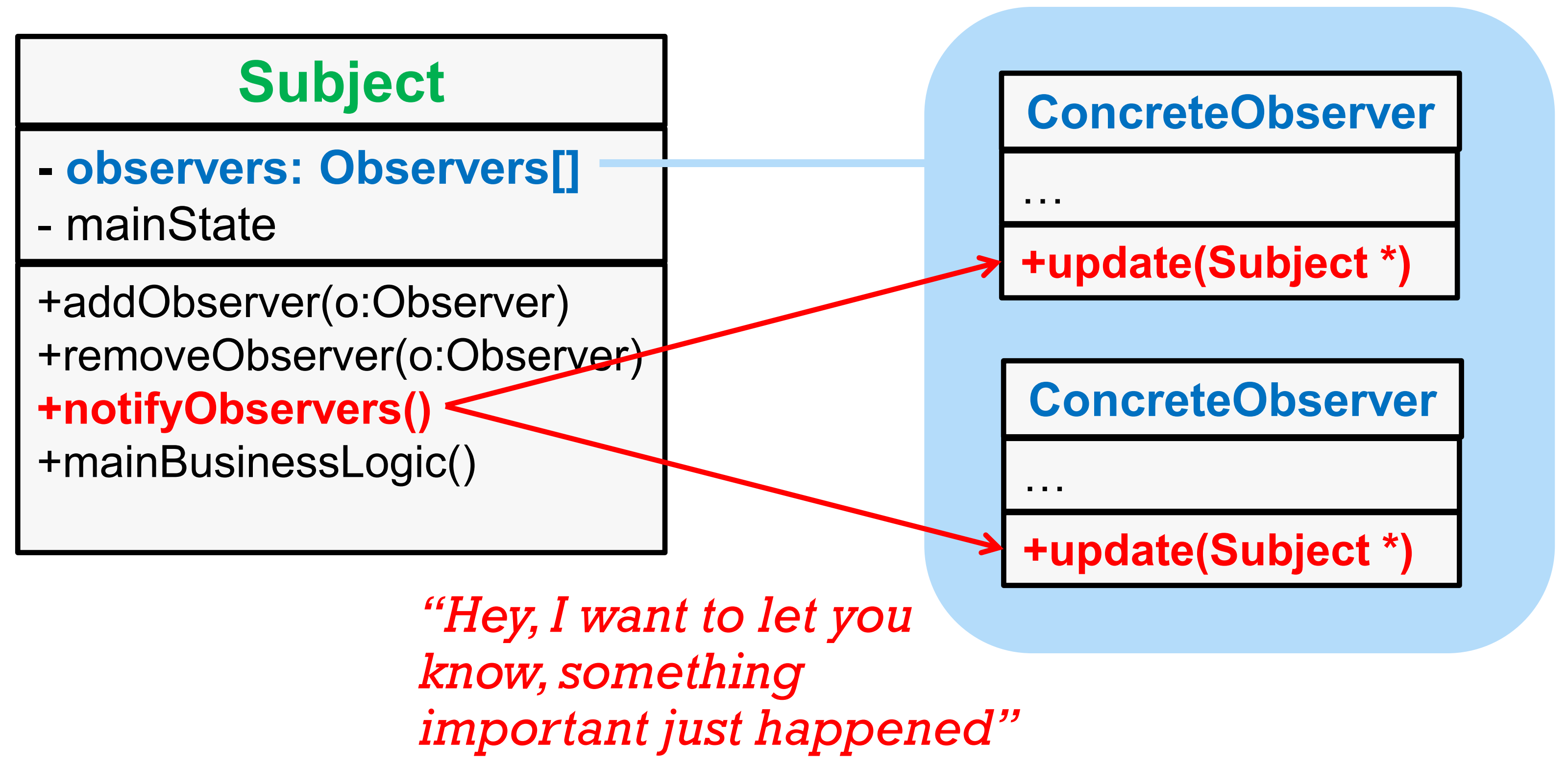
2. 附加



观察者对象 可以被 添加 到 主题 中以"监听"重要事件

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

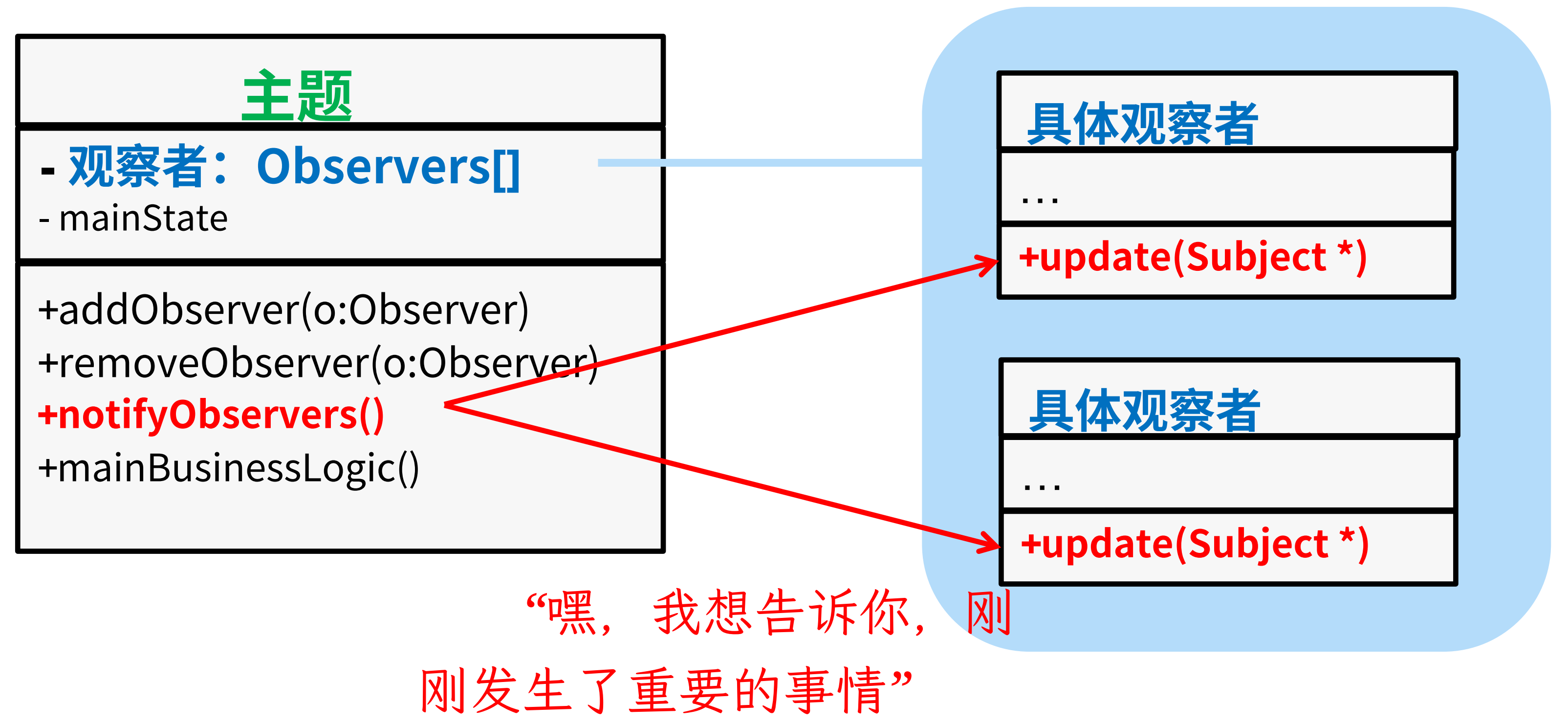
3. Update



Subject notifies observer objects when important event happens

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

3. 更新



主题 在发生重要事件时通知 观察者对象当重要事件发生时

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

使用场景

- 在以下情况下使用观察者模式：
  - 一个 **被观察对象的改变** **主体** 需要 **更改** **其他对象**，但你不知道有多少 对象 需要被修改
  - 一个 **对象** 需要 **通知** **其他对象** 而不对这些对象的具体类型做任何假设（松耦合！）
- 我需要 教授 在 **收到通知** 当有 学生 加入其课程时
- 我希望 显示内容 能够 **更新** 当 窗口 的大小发生变化时
- 我需要 日程视图 在 **更新** 当 数据库 发生更改时

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

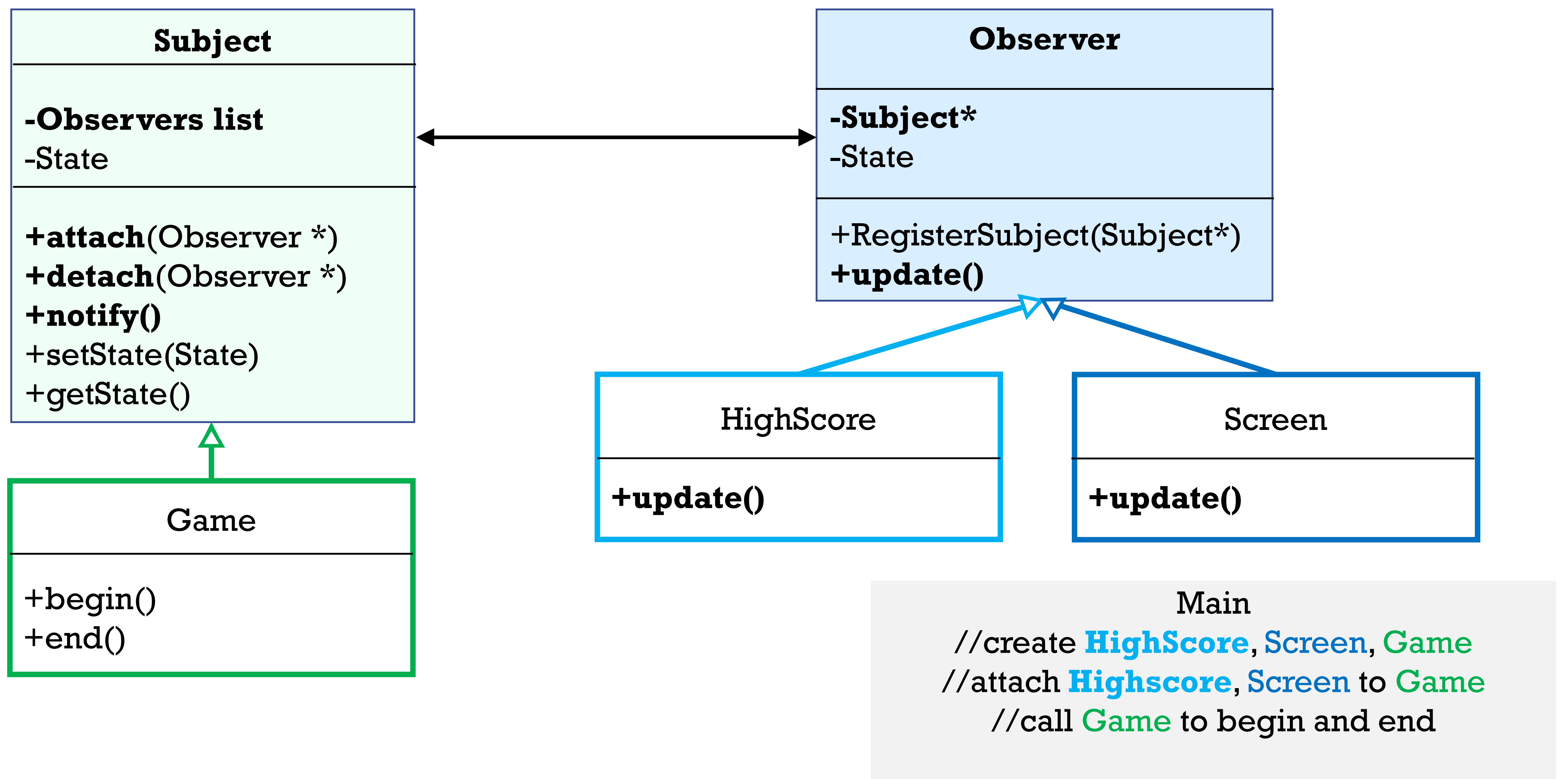
- **Game** class is a **Subject**
  - Subject contains a vector of observer pointers
  - Subject's **notify()** will call all observers' **update()**
- HighScore and Screen class are **Observers**
  - All observers have an **update()**
  - Perform own logic when this **update()** called by **Subject**

游戏示例

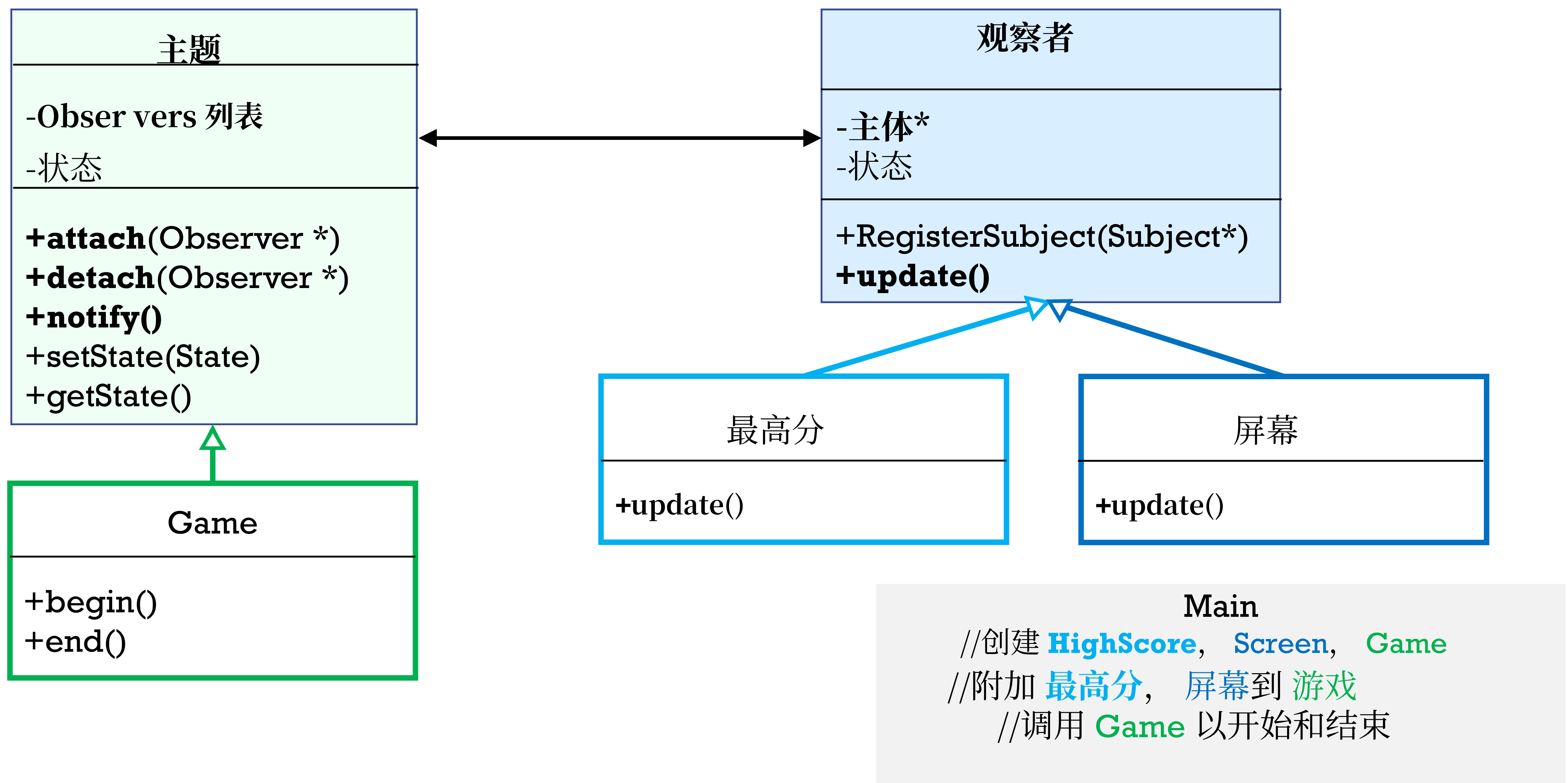
- 游戏类将在游戏开始和结束时通知观察者
  - 游戏开始 – 高分榜关闭，游戏开始界面显示
  - 游戏结束 – 高分榜显示，游戏结束界面显示

- **游戏**类是一个**主题**
  - 主题包含一个观察者指针的向量
  - 被观察对象的 **notify()** 将调用所有 观察者' **update()**
- HighScore 和 Screen 类都是 **观察者**
  - 所有 观察者 都有一个 **update()**
  - 当此 **update()** 被 **被观察对象** 调用时执行自身的逻辑

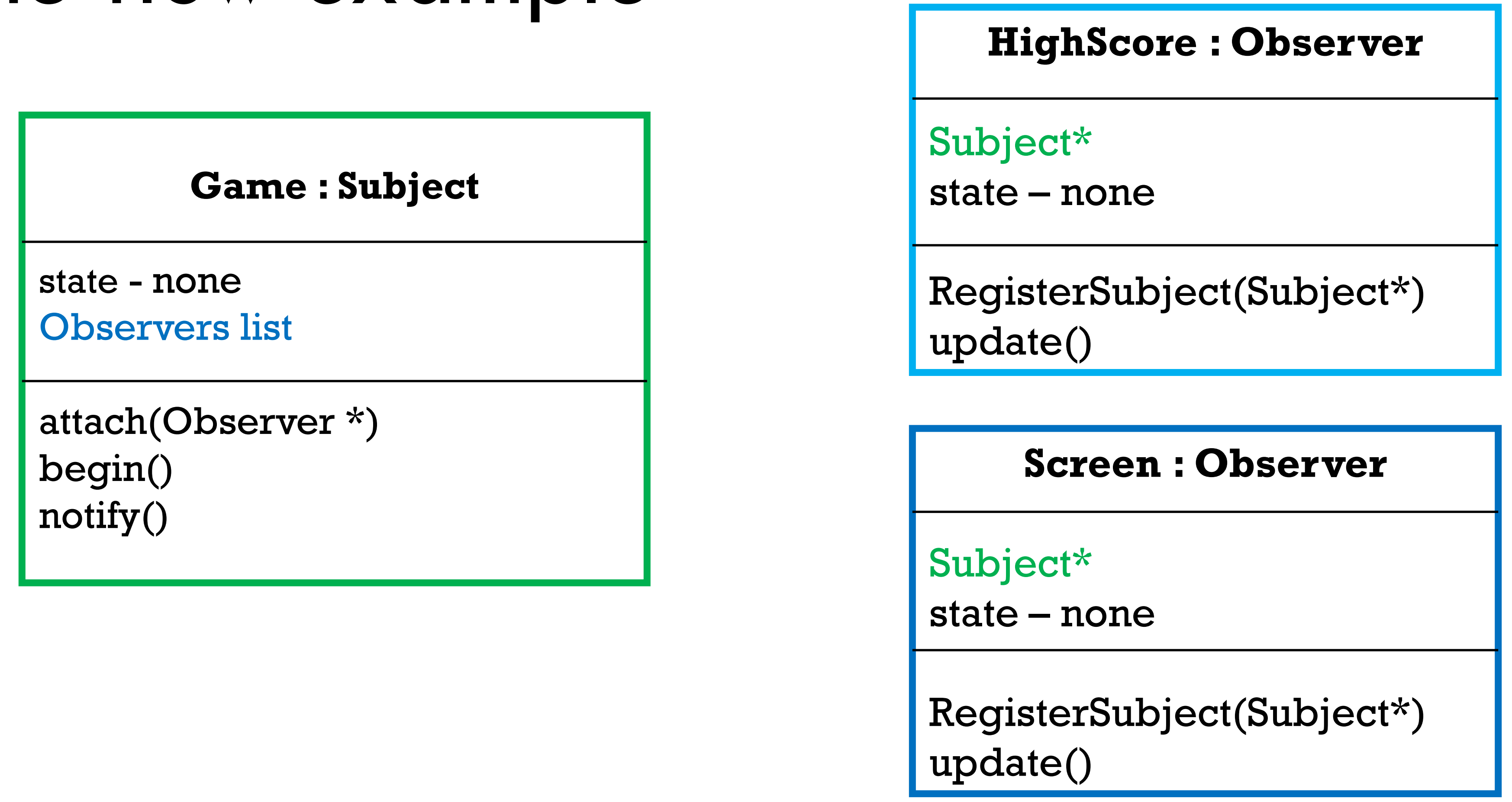
Game example



游戏示例

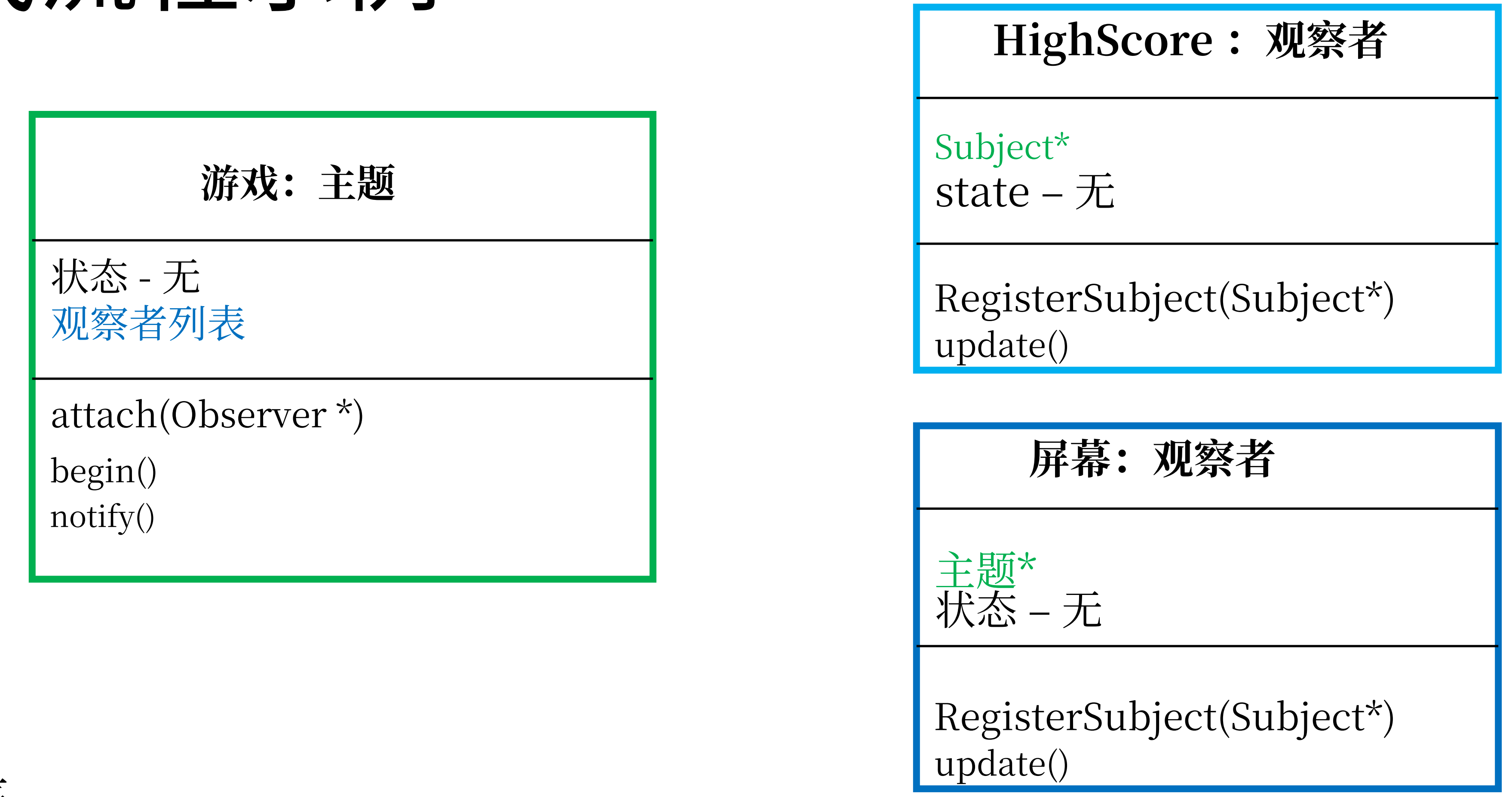


Game flow example



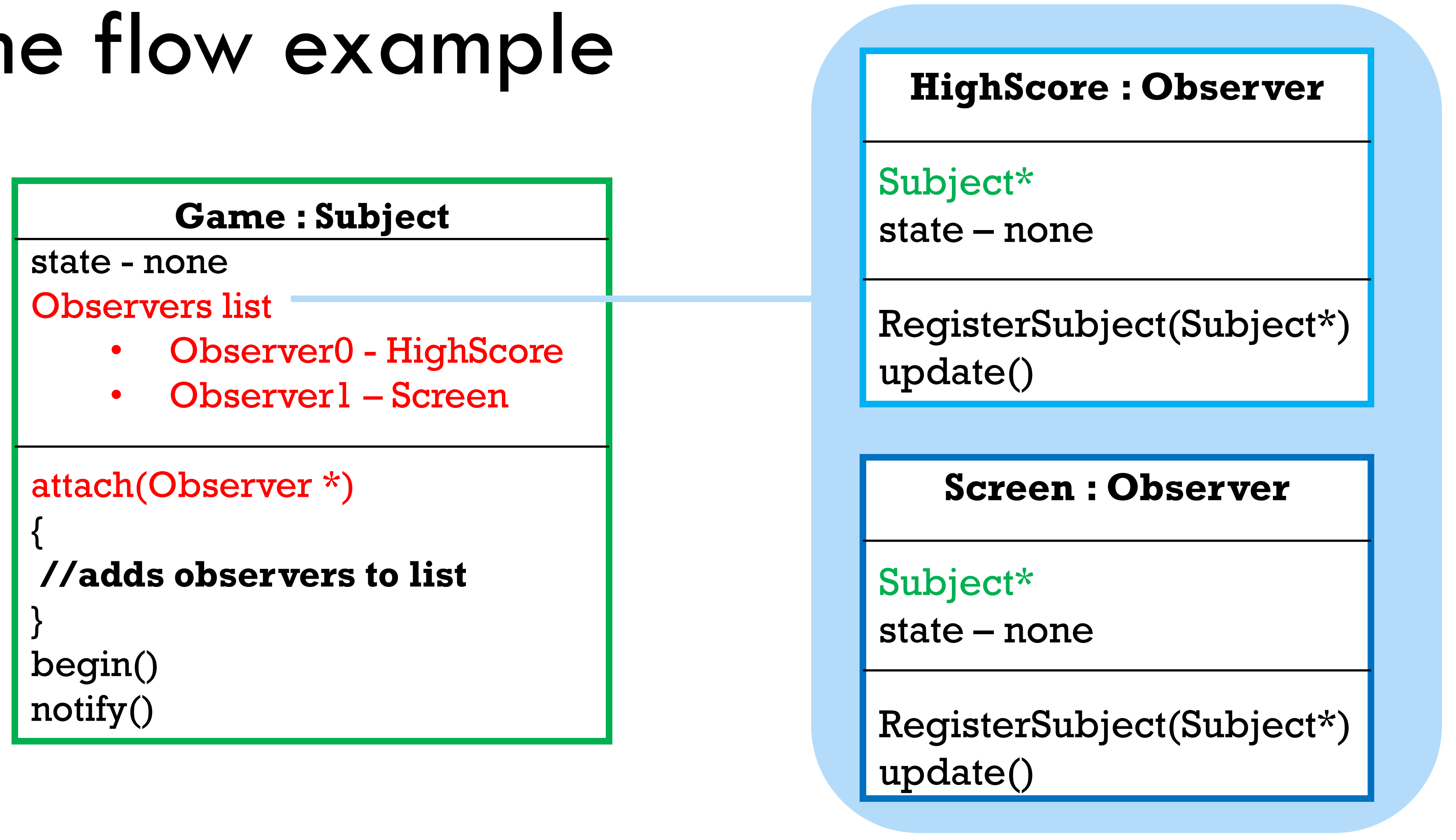
- Main
    - Create **Game**, **HighScore**, and **Screen** objects

游戏流程示例



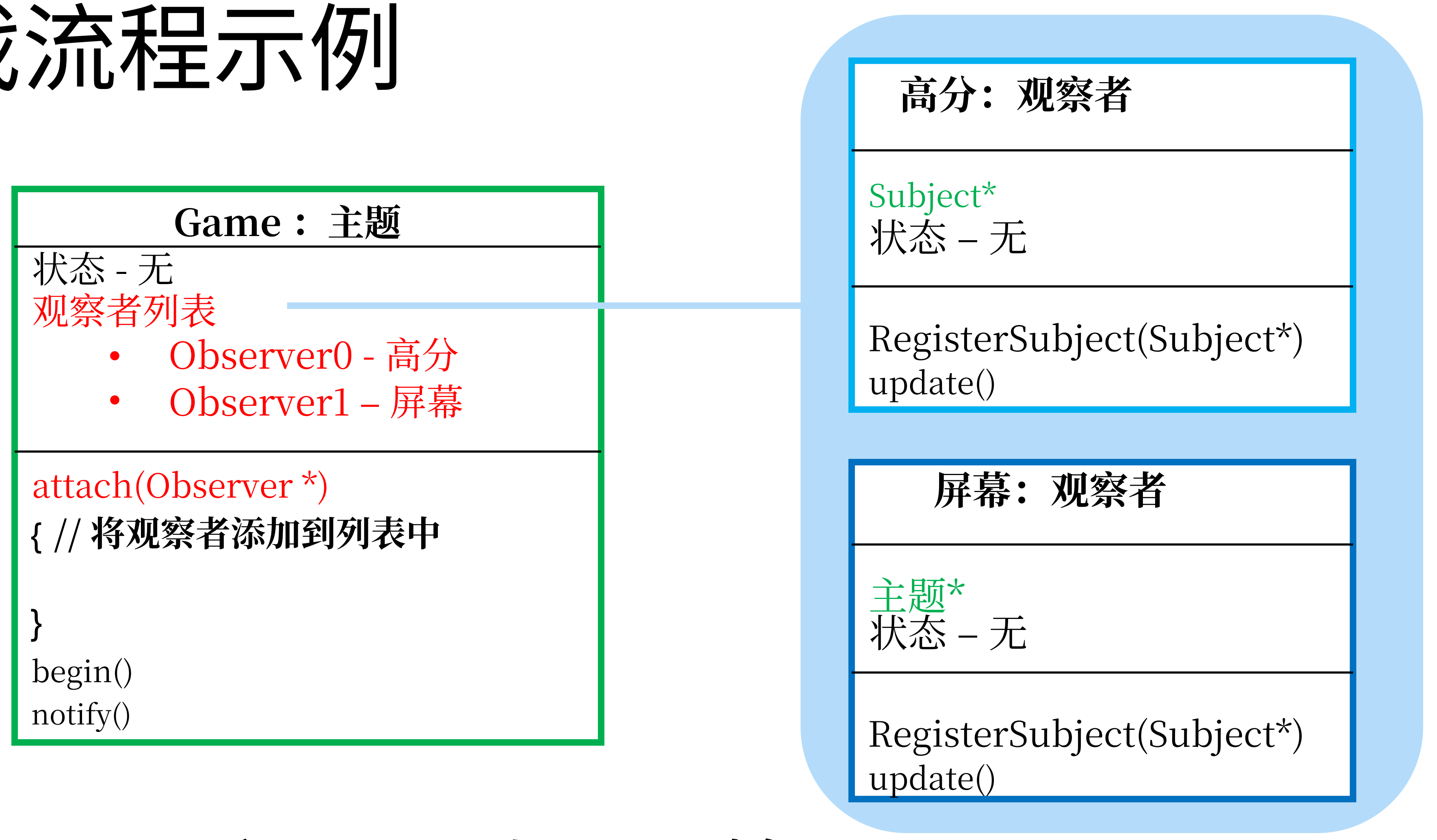
- 主程序
    - 创建 **游戏**、 **高分** 和 **屏幕** 对象

Game flow example



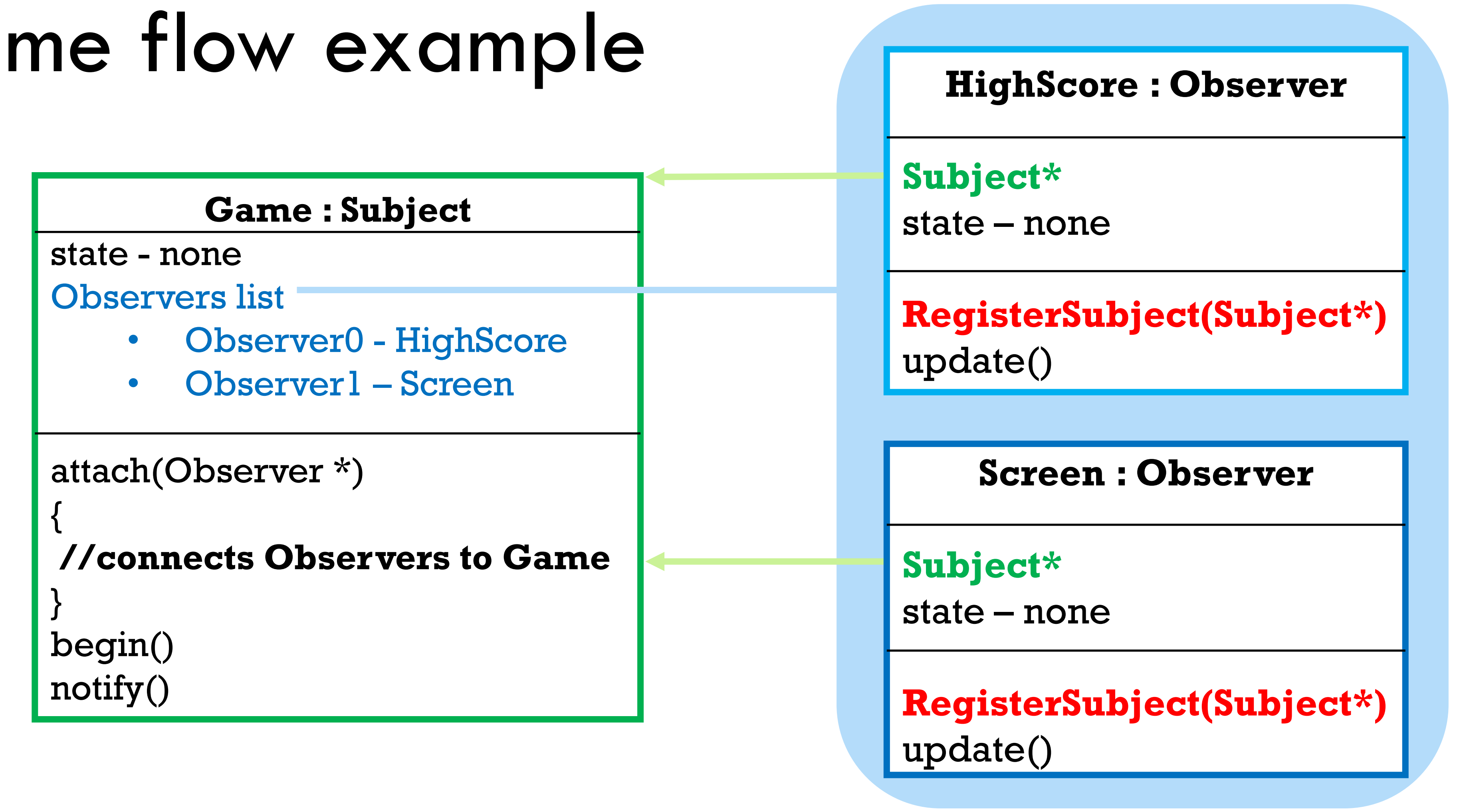
- Attach **HighScore** and **Screen** to **Game** object
  - Pass HighScore and Screen to **attach(Observer*)**
  - Game doesn't know they're HighScore and Screen objects
  - Game sees them as the **abstract Observer type**

游戏流程示例



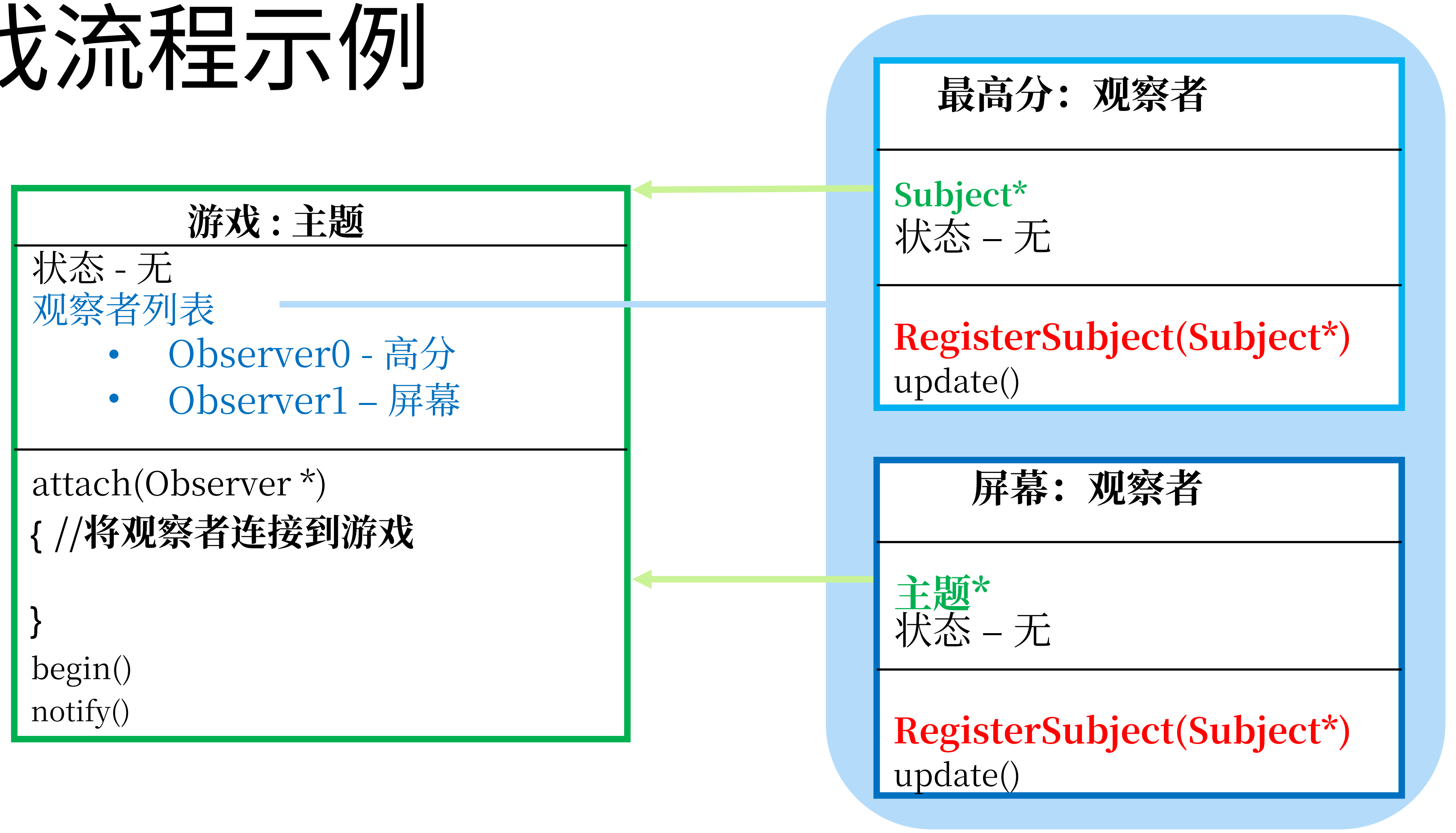
- 绑定 **HighScore** 和**Screen**到**Game**对象
  - 传递HighScore和Screen到**attach(Observer*)**
  - Game 不知道它们是 HighScore 和 Screen 对象
  - Game 将它们视为 **抽象的 Obser ver 类型**

Game flow example



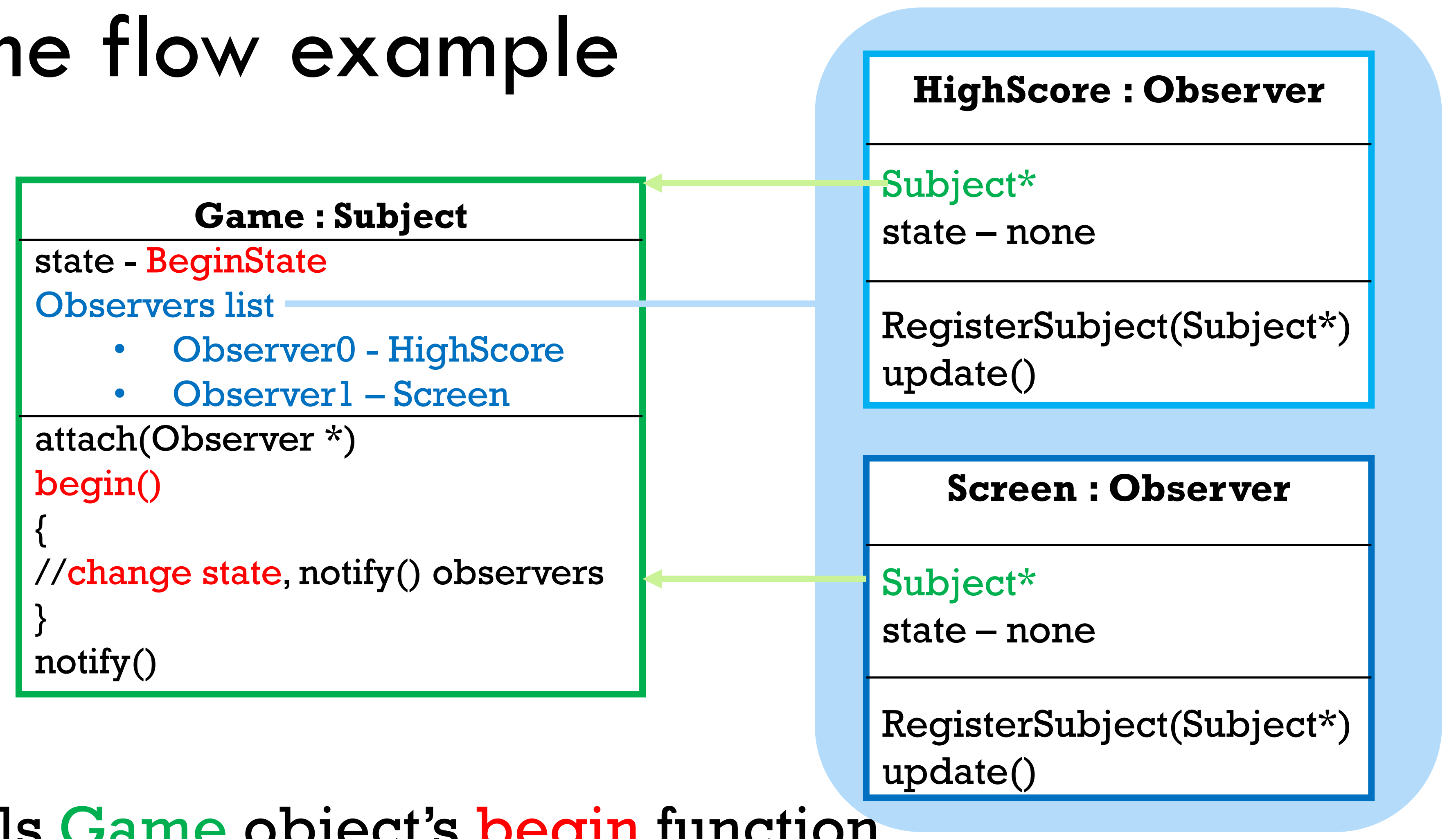
- **Game** now connected to **HighScore** and **Screen**
- Need to connect **HighScore** and **Screen** to **Game**
  - Pass **Game** to **RegisterSubject(Subject*)** in each Observer

游戏流程示例



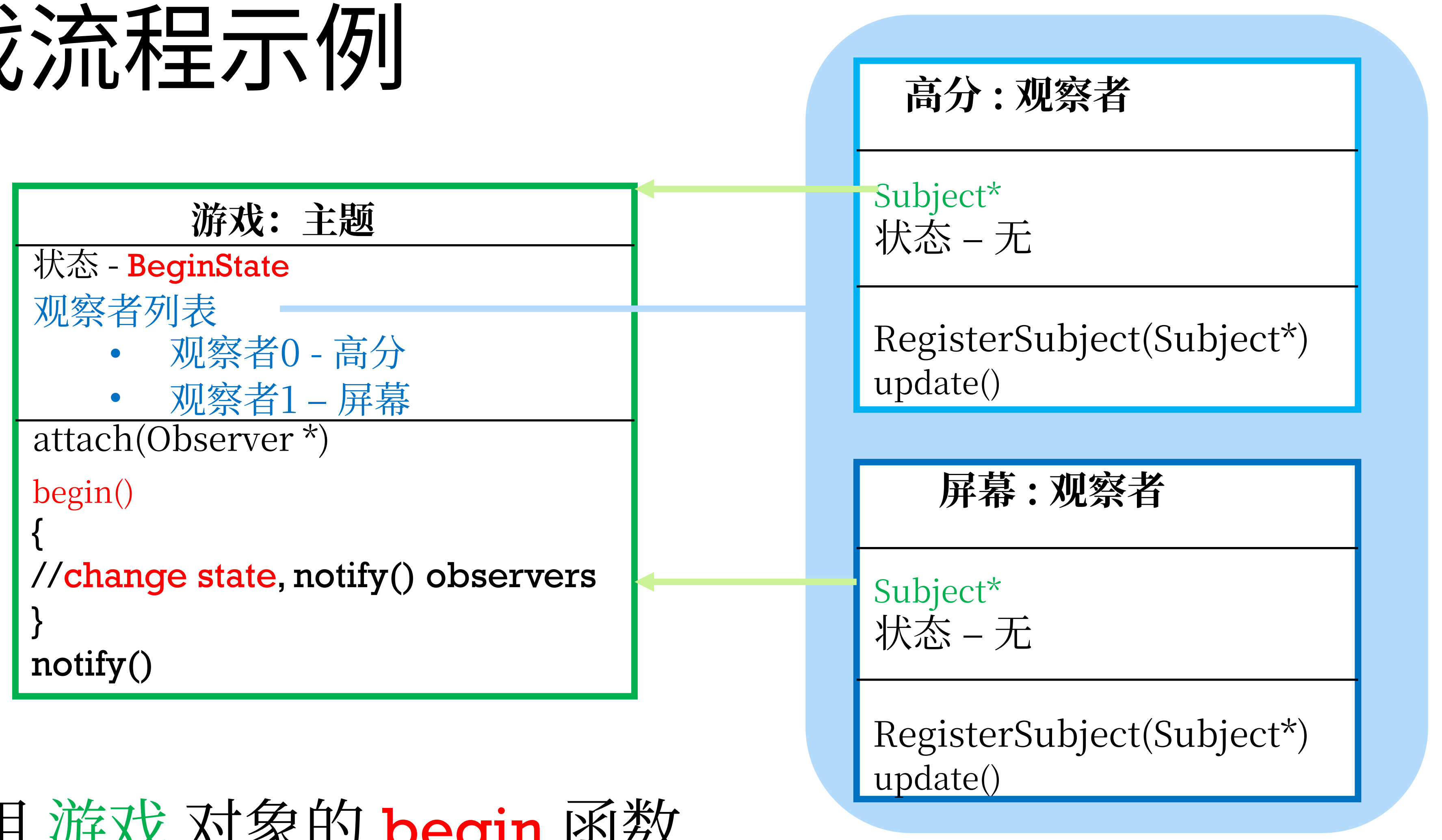
- **游戏**现已连接至 **高分** 和 **屏幕**
- 需要将 **高分** 和 **屏幕** 连接到 **游戏**
  - 在每个观察者中将 **游戏** 传递给 **RegisterSubject(Subject*)** 方法

Game flow example



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

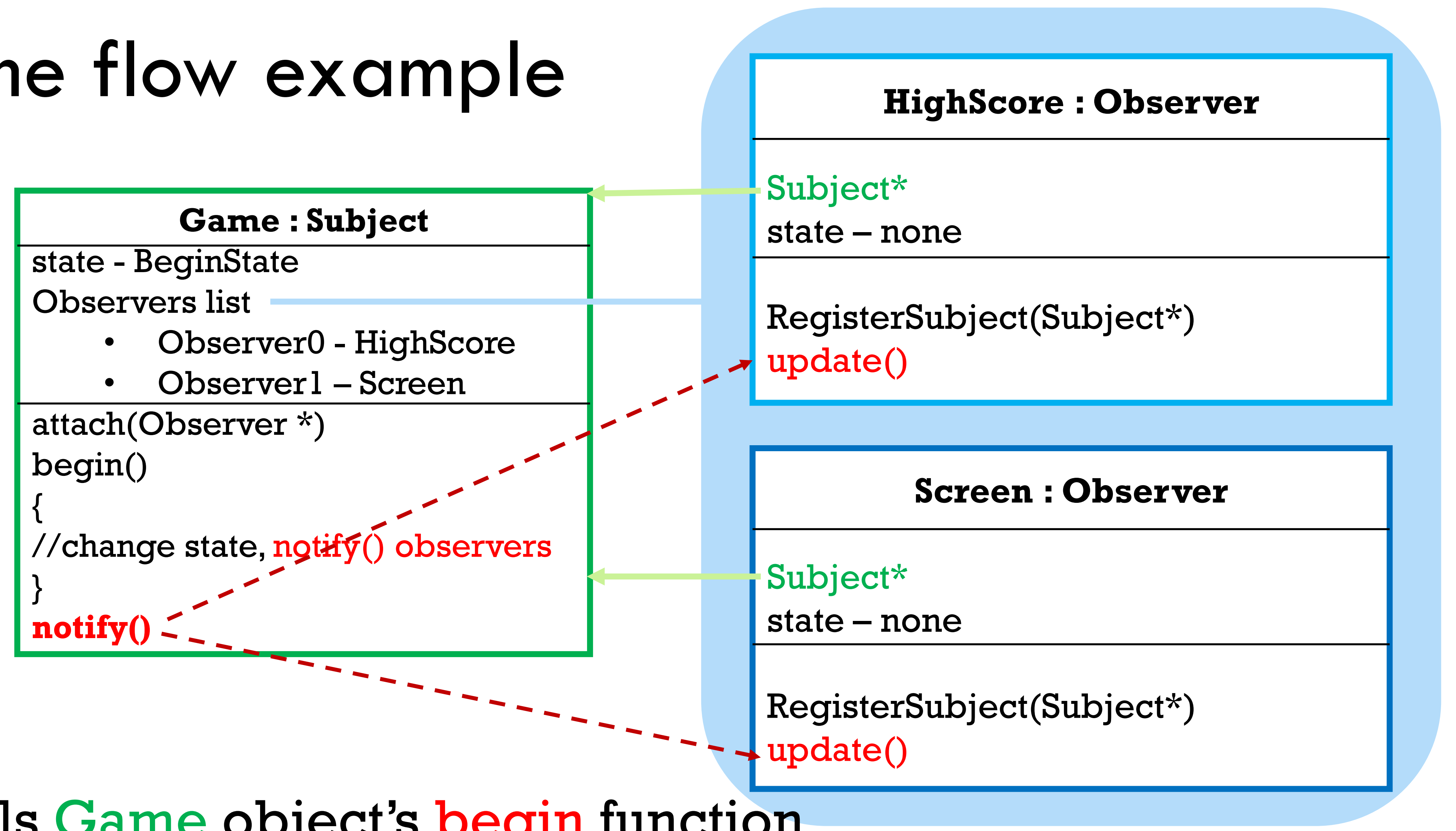
游戏流程示例



- 客户端调用 游戏 对象的 begin 函数
  - 将游戏状态更改为 BeginState

SubjectObserver.cpp

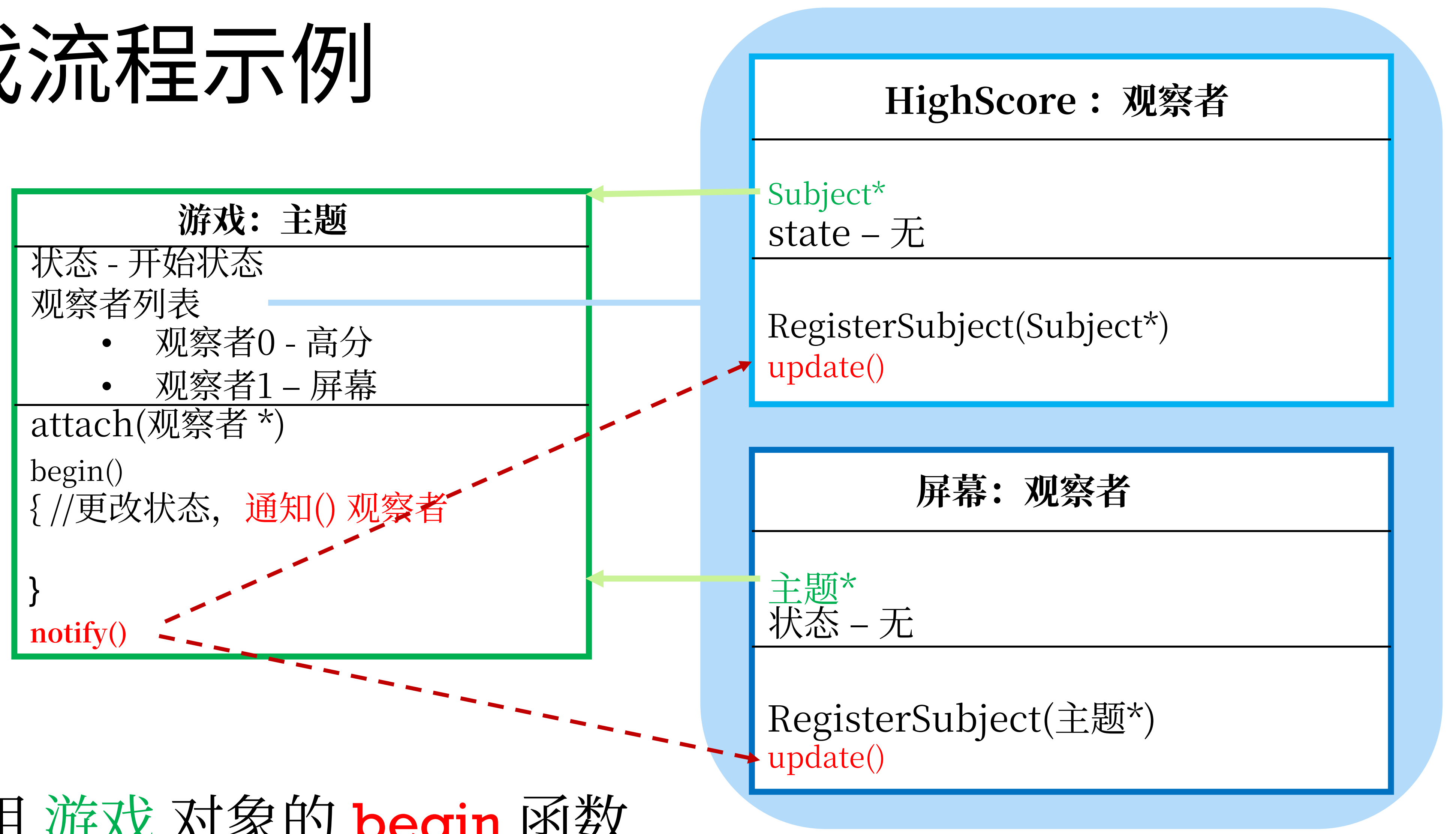
Game flow example



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

SubjectObserver.cpp

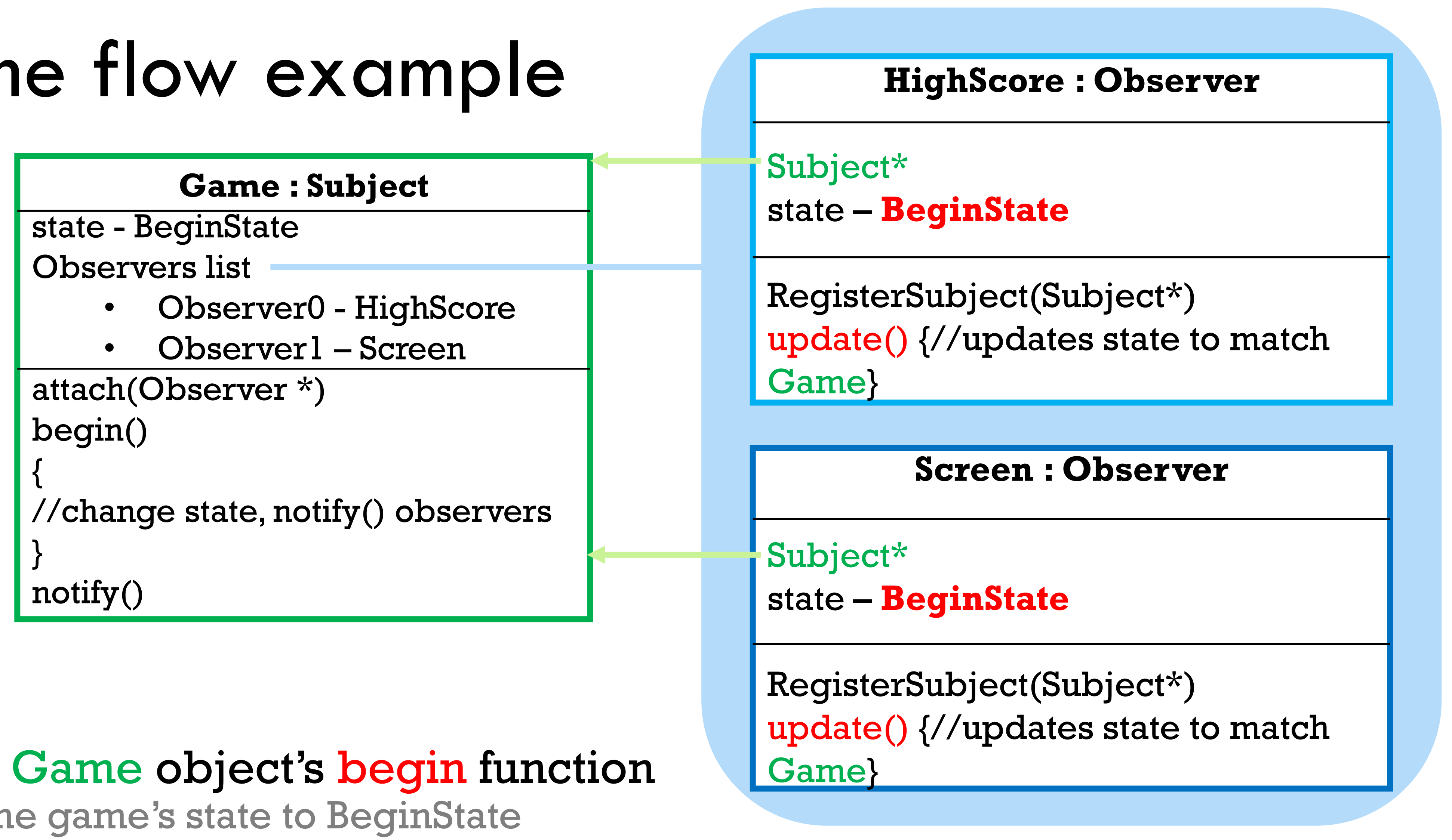
游戏流程示例



- 客户端调用 游戏 对象的 begin 函数
  - 将游戏状态更改为 BeginState
  - 通知列表中的所有观察者 某些内容已更改

SubjectObserver.cpp

Game flow example



```
Game : Subject
state - BeginState
Observers list
    •  Observer0 - HighScore
    •  Observer1 – Screen
attach(Observer *)
begin()
{
//change state, notify() observers
}
notify()
```

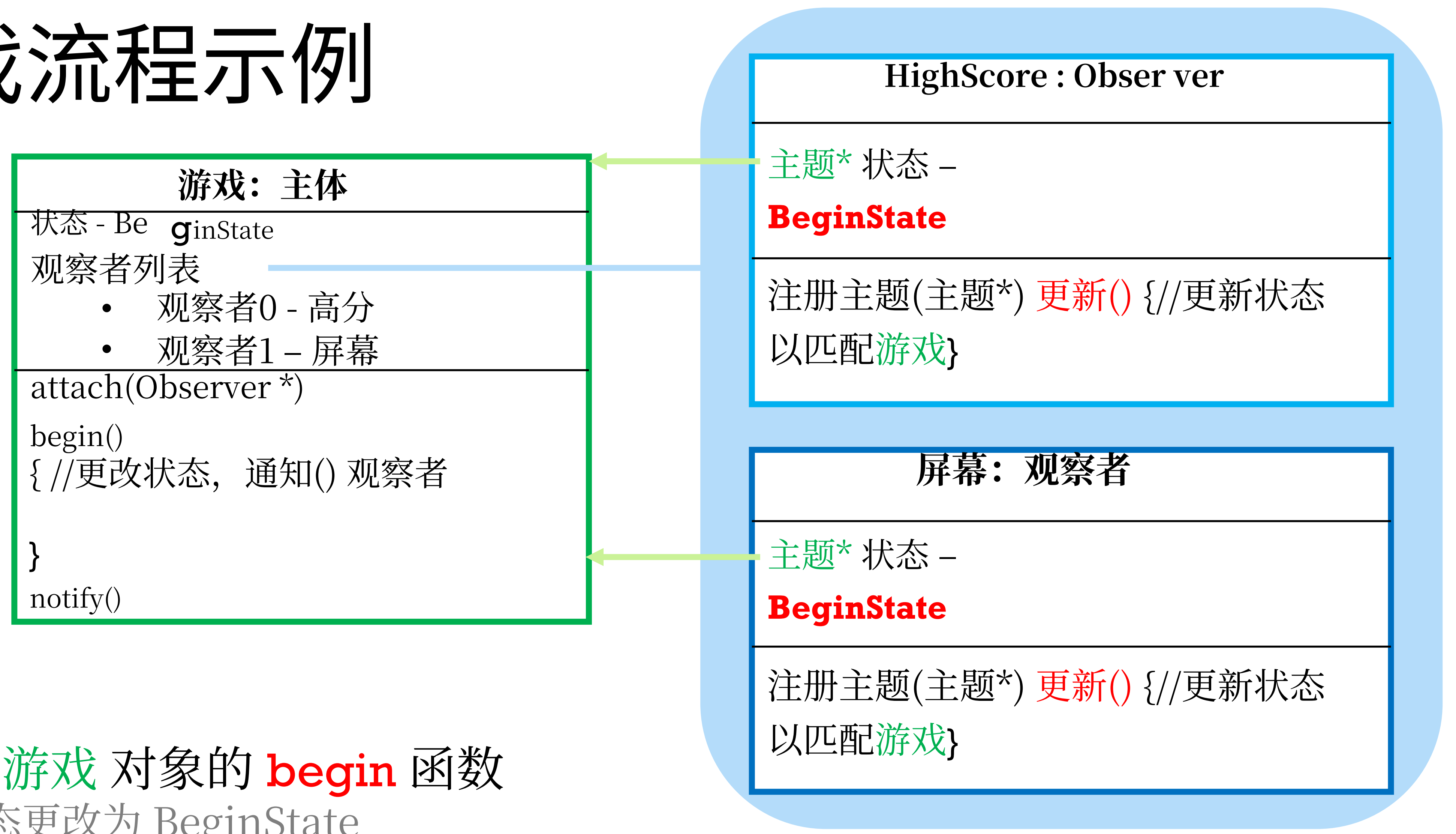
```
HighScore : Observer
Subject*
state – BeginState
RegisterSubject(Subject*)
update() {//updates state to match
Game}
```

```
Screen : Observer
Subject*
state – BeginState
RegisterSubject(Subject*)
update() {//updates state to match
Game}
```

- 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

SubjectObserver.cpp

游戏流程示例



```
游戏：主体
状态 - Be  ginState
观察者列表
    •  观察者0 - 高分
    •  观察者1 – 屏幕
attach(Observer *)
begin()
{ //更改状态，通知() 观察者
}
notify()
```

```
HighScore : Obser ver
主题* 状态 –
BeginState
注册主题(主题*) 更新() {//更新状态
以匹配游戏}
```

```
屏幕：观察者
主题* 状态 –
BeginState
注册主题(主题*) 更新() {//更新状态
以匹配游戏}
```

- 客户端调用 游戏 对象的 begin 函数
  - 将游戏状态更改为 BeginState
  - 通知列表中的所有观察者，某些内容已发生变化
  - 观察者查询 Game 的 继承 getState() 以获取其当前状态
  - 将内部状态设置为与 游戏的 BeginState 一致