

Records

Engineering 1D04, Teaching
Session 11

Records

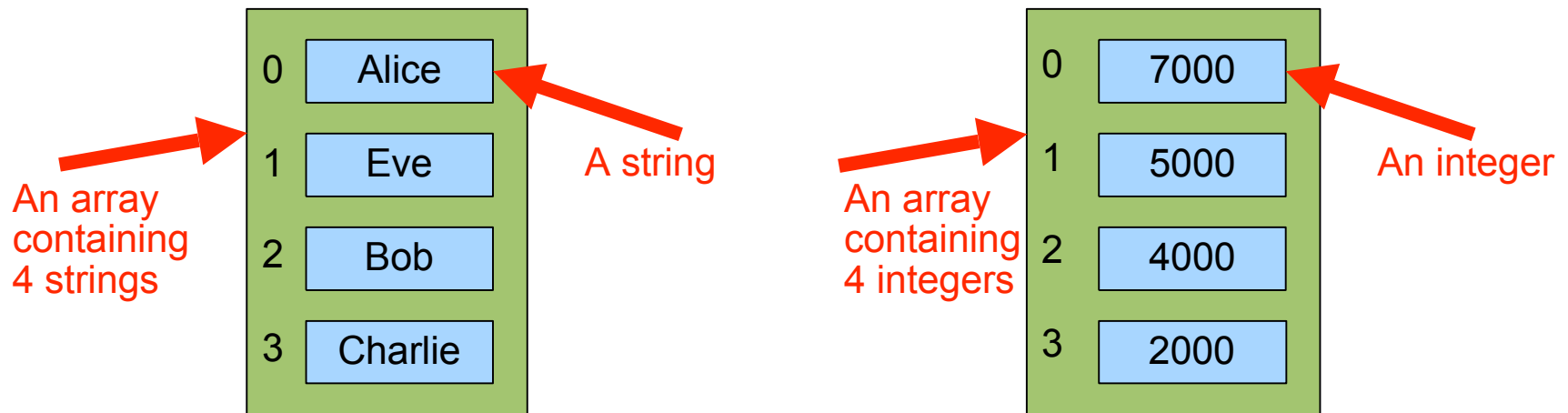
- So far we've seen a number of different types of variables:
 - Integers (int)
 - Strings (string)
 - Floating Point Numbers (float, double)
 - Boolean Values (bool)
 - Arrays of int, Arrays of double, etc.
 - 2D Arrays of int, 2D Arrays of double, etc.

Records

- In many cases, we have a need to represent data more complex than a single number or a text string.
- Arrays allow us to put related values of the same type together, but what if the types of the elements are different?

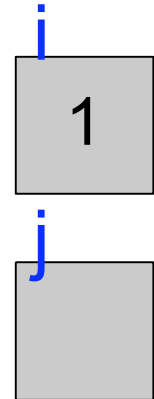
High Scores

- Recall the high score system.
 - We used two separate arrays to store the following items:
 - The top ten scores
 - The names associated with those scores



High Scores

```
for (j = names.Length - 1; j > i; j--)  
    names[j] = names[j-1];  
    scores[j] = scores[j-1];  
names[i] = name;  
scores[i] = score;
```



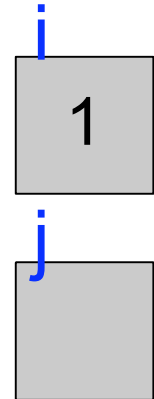
0	Alice	0	7000
1	Eve	1	5000
2	Bob	2	4000
3	Charlie	3	2000

Dave 6000

← New score to be inserted here

High Scores

```
for (j = names.Length - 1; j > i; j--)  
    names[j] = names[j-1];  
    scores[j] = scores[j-1];  
names[i] = name;  
scores[i] = score;
```



0	Alice	0	7000
1	Eve	1	5000
2	Bob	2	4000
3	Charlie	3	2000

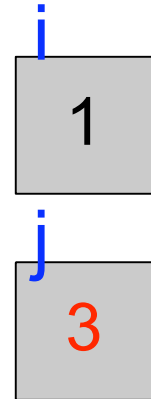
Dave 6000

← New score to be inserted here

Before starting, notice that
Eve's score is 5000.

High Scores

```
▶ for (j = names.Length - 1; j > i; j--)  
    names[j] = names[j-1];  
    scores[j] = scores[j-1];  
names[i] = name;  
scores[i] = score;
```



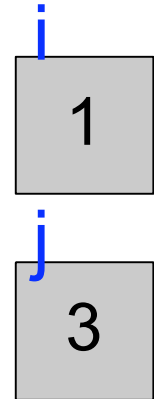
0	Alice	0	7000
1	Eve	1	5000
2	Bob	2	4000
3	Charlie	3	2000

Dave 6000

← New score to be inserted here

High Scores

```
for (j = names.Length - 1; j > i; j--)  
    names[j] = names[j-1];  
    scores[j] = scores[j-1];  
names[i] = name;  
scores[i] = score;
```



0	Alice
1	Eve
2	Bob
3	Bob

0	7000
1	5000
2	4000
3	2000

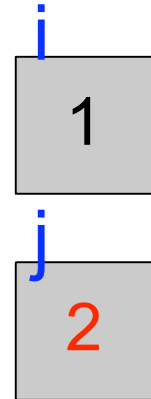
Dave 6000

← New score to be inserted here

Name moves

High Scores

```
▶ for (j = names.Length - 1; j > i; j--)  
    names[j] = names[j-1];  
    scores[j] = scores[j-1];  
names[i] = name;  
scores[i] = score;
```



0	Alice	0	7000
1	Eve	1	5000
2	Bob	2	4000
3	Bob	3	2000

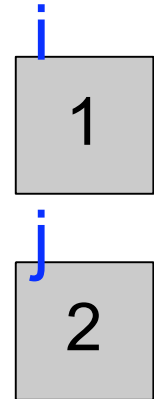
Dave 6000

← New score to be inserted here

Name moves
but score does not!

High Scores

```
for (j = names.Length - 1; j > i; j--)  
    names[j] = names[j-1];  
    scores[j] = scores[j-1];  
names[i] = name;  
scores[i] = score;
```



0	Alice
1	Eve
2	Eve
3	Bob

0	7000
1	5000
2	4000
3	2000

Dave 6000

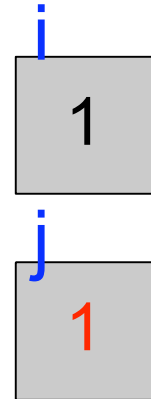
← New score to be inserted here

Name moves
but score does not!

High Scores

```
▶ for (j = names.Length - 1; j > i; j--)  
    names[j] = names[j-1];  
    scores[j] = scores[j-1];  
names[i] = name;  
scores[i] = score;
```

No longer true



0	Alice	0	7000
1	Eve	1	5000
2	Eve	2	4000
3	Bob	3	2000

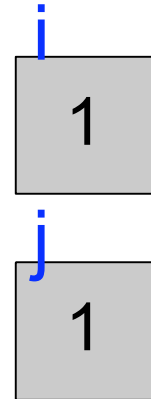
Dave 6000

← New score to be inserted here

High Scores

```
for (j = names.Length - 1; j > i; j--)  
    names[j] = names[j-1];  
    scores[j] = scores[j-1];  
names[i] = name;  
scores[i] = score;
```

No longer true



0	Alice
1	Eve
2	Eve
3	Bob

0	7000
1	7000
2	4000
3	2000

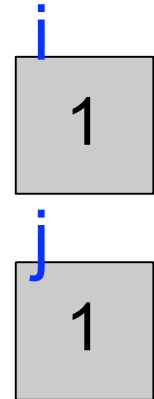
???

Dave 6000

New score to be inserted here

High Scores

```
for (j = names.Length - 1; j > i; j--)  
    names[j] = names[j-1];  
    scores[j] = scores[j-1];  
▶ names[i] = name;  
  scores[i] = score;
```



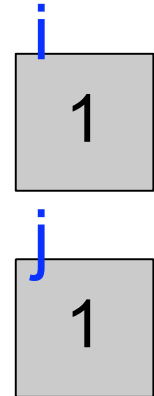
0	Alice	0	7000
1	Dave	1	7000
2	Eve	2	4000
3	Bob	3	2000

Dave 6000

← New score to be inserted here

High Scores

```
for (j = names.Length - 1; j > i; j--)  
    names[j] = names[j-1];  
    scores[j] = scores[j-1];  
names[i] = name;  
scores[i] = score;
```



0	Alice	0	7000
1	Dave	1	6000
2	Eve	2	4000
3	Bob	3	2000

Dave 6000

← New score to be inserted here

High Scores


```
for (j = names.Length - 1; j > i; j--)  
    names[j] = names[j-1];  
    scores[j] = scores[j-1];  
names[i] = name;  
scores[i] = score;
```

0	Alice	0	7000
1	Dave	1	6000
2	Eve	2	4000
3	Bob	3	2000


Eve's score is now 4000
(Bob's former score).

What happened?

High Scores



```
for (j = names.Length - 1; j > i; j--)  
{  
    names[j] = names[j-1];  
    scores[j] = scores[j-1];  
}  
names[i] = name;  
scores[i] = score;
```



0	Alice	0	7000
1	Dave	1	6000
2	Eve	2	4000
3	Bob	3	2000

Notice the lack of braces

Only the name gets shifted inside of the loop.

So the names and scores become mismatched!!

High Scores

- Keeping two separate arrays of associated data is potentially dangerous.
 - We have to ensure that the data in one array is always synchronised with the data in the other.
 - Keeping the two arrays synchronised requires additional code.
 - A bug like the one you just saw could easily result in names matching the wrong scores.
 - If we want to change the number of scores, we have to change the size of two arrays.

High Scores

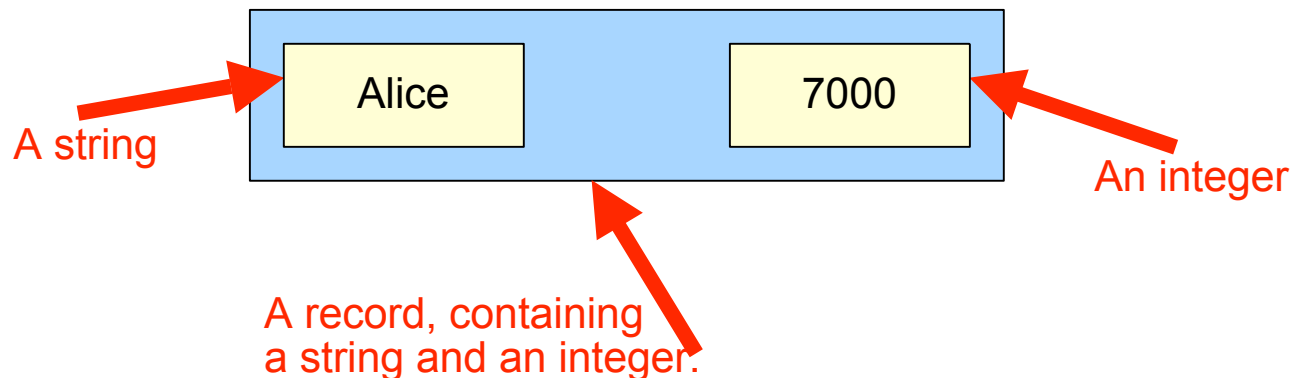
- We can only use a single array if we can store more complex data in an array.
 - Maybe we could store a string like
 - "Alice 7000 points"
 - It becomes inconvenient to extract the score portion of the string to compare it with other scores when inserting.
- We need an easy way to treat multiple pieces of data as a single cohesive object.

Records

- A common concept in programming languages is that of *records*.
- A *record* is several pieces of related data stored together in a single place.
 - name, address and phone number of a person
 - name and score of someone on a highscore list
 - title, track, album and artist of a music file

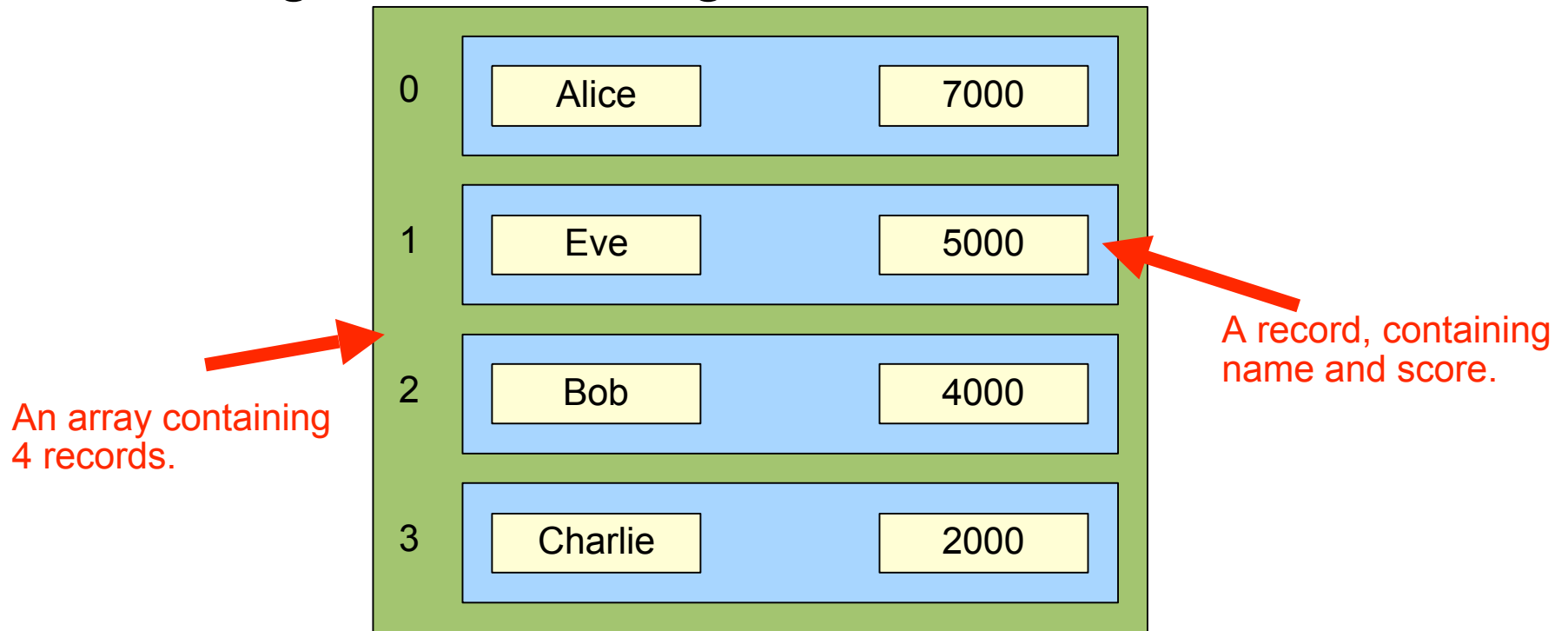
Records

- When dealing with records, all of the information in a record moves around together as a single unit.
- Consider a record containing a high score and the name of the person it belongs to.



Records

- We could then make a single array containing a number of these records.
- Our highscore list might then look like this



Records

- Because names and scores are bound together in a single package it's not possible for them to fall out of synchronisation with each other.
- The single packaging means that both the name and score can be moved with a single statement.
- This becomes increasingly useful as the number of items in the record increases from two to many more.

Records

- There are clearly many possibilities for combinations of different types of data to store in a record.
- Before storing data in a record you need to define its type.
- Once you have declared the type, you can create many records of this type (for example, one for each person in the high score list).

Records

- When declaring variables in any program you specify two things about each variable.
 - Its type
 - Its name

Examples:

- `int counter;`
 - type is integer, name is "counter".
- `string password;`
 - type is string, name is "password".

Records

- When declaring variables in a record you do exactly the same thing.
- You must specify the type and the name of each variable in the record. This defines the type of the whole record.

Records

- When declaring variables in a record you do exactly the same thing.
- You must specify the type and the name of each variable in the record. This defines the type of the whole record.
- Assuming a high score entry consists of a string for name and an integer for score, we want:
 - string name;
 - int score;

Records

- The idea of a record is to wrap up its components into a single unit.
- The string and the integer in the high score entry record exist as a pair.

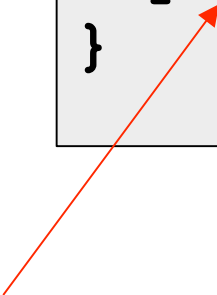
Records

- In C#, the *class* keyword is used for many things.
- In general, the *class* keyword defines a template for the creation of *objects*.
- A record is a specific sort of object but everything mentioned here about records will be applicable to objects in general.

Records

- One purpose of the class keyword is to define a record type.

```
class HighScoreEntry
{
    public string name;
    public int score;
}
```



public indicates we need to access these items from outside of the class - we still need to discuss public and private in more detail - stay tuned

Records

- After declaring a record type with a given name we use the *new* keyword to create a new instance of it.
- *new* returns a Reference to the new record.
 - The concept of references is very important!
- To create a new HighScoreEntry:

```
new HighScoreEntry() ;
```

Records

- Creating a new record instance

```
new HighScoreEntry();
```

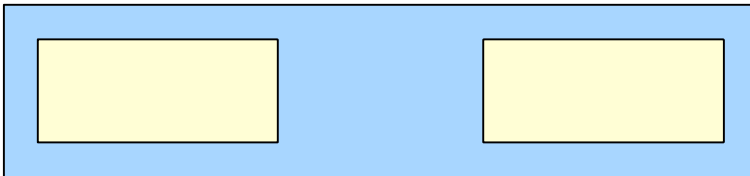
When the new statement runs a new record instance is created in memory.

Records

- Creating a new record instance

```
new HighScoreEntry();
```

When the new statement runs a new record instance is created in memory.

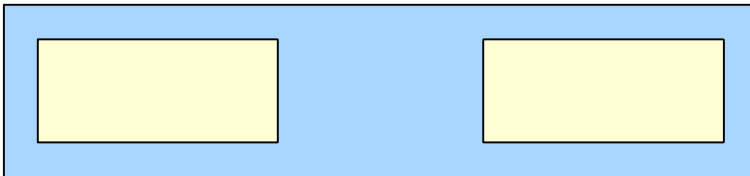


Records

- Creating a new record instance

```
new HighScoreEntry();
```

The new instance has no name. It just “floats” in memory.

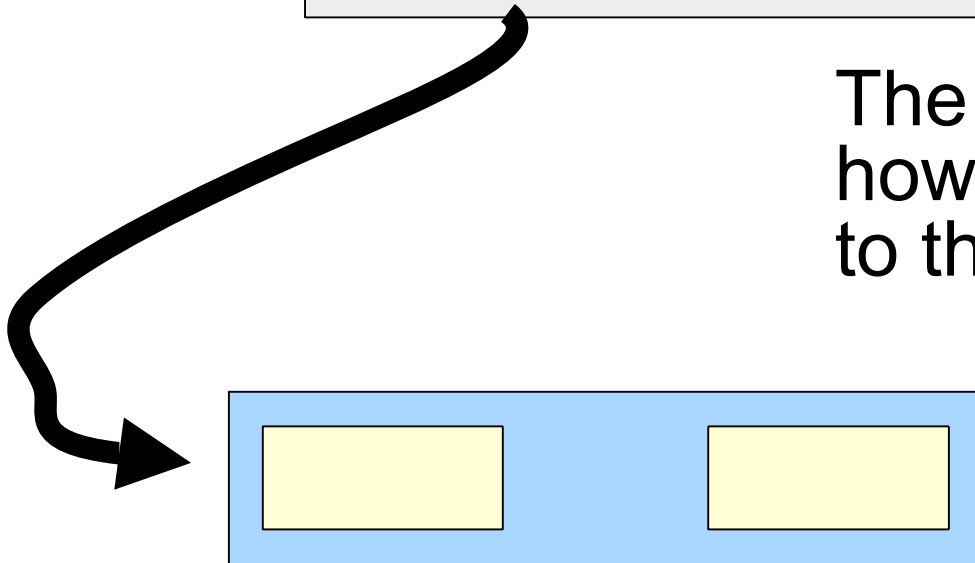


Records

- Creating a new record instance

```
new HighScoreEntry();
```

The *new* statement, however, returns a reference to the new object.

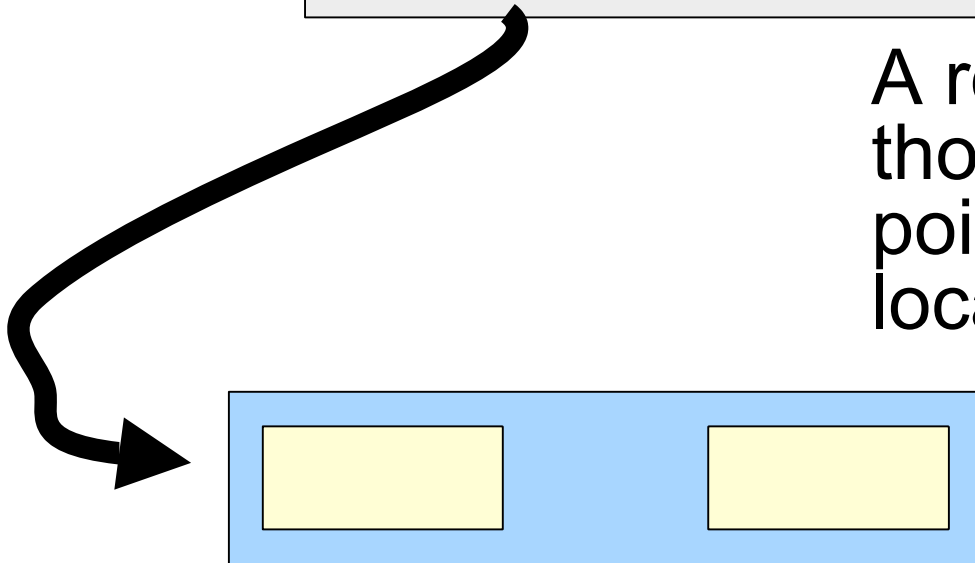


Records

- Creating a new record instance

```
new HighScoreEntry();
```

A reference can be thought of as an arrow or a pointer indicating the location of an object.



Records

- *new* returns a reference to the new record.
- We can follow this reference to find the record.
- A reference is its own type of value (typically the memory address of the record).

Records

- We can create variables to store references.

```
HighScoreEntry myRef;
```

- This creates a variable called *myRef* that is capable of storing a reference to a *HighScoreEntry* record.
- This variable can not store references to other types of records.

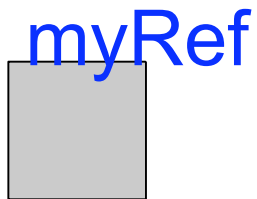
Records

- Naturally, a reference variable can be used to store the reference that is returned by *new*.

```
HighScoreEntry myRef;  
myRef = new HighScoreEntry();
```

Creating New Records – Demo

```
HighScoreEntry myRef;  
myRef = new HighScoreEntry();
```



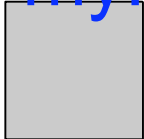
Just like with any other variable declaration a new named box is created in memory. This box is able to store HighScoreEntry references.

Creating New Records – Demo

```
HighScoreEntry myRef;  
myRef = new HighScoreEntry();
```



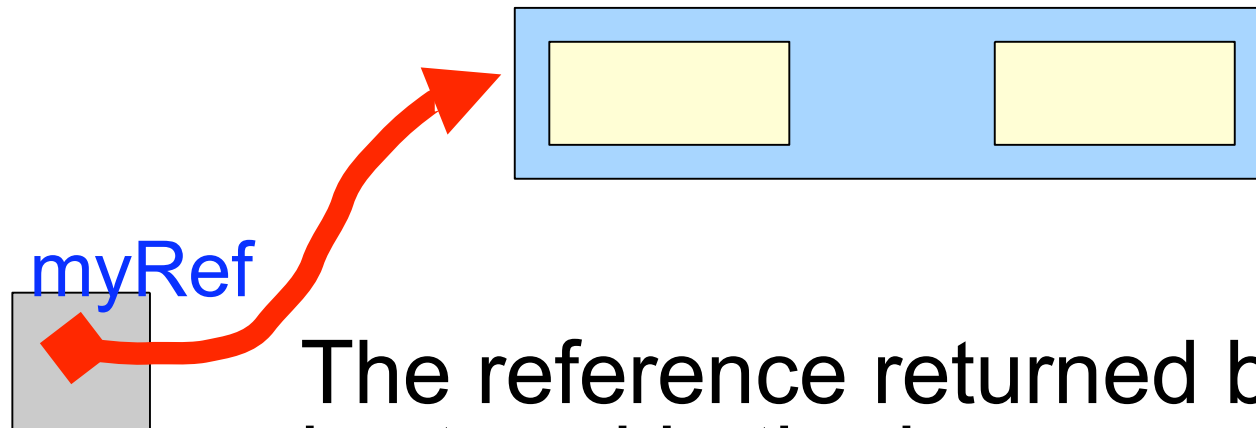
myRef



A new HighScoreEntry record is created (floating in memory, without a name).

Creating New Records – Demo

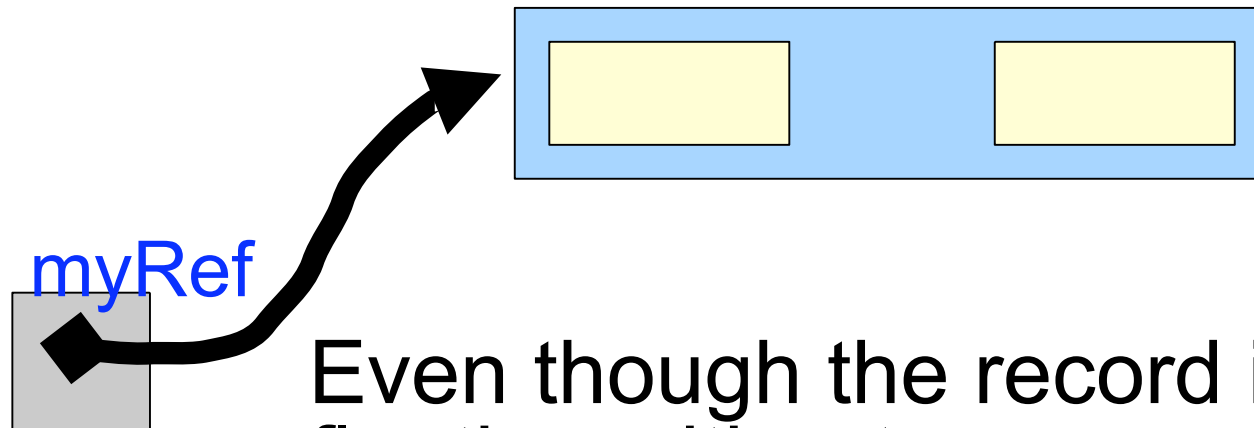
```
HighScoreEntry myRef;  
myRef = new HighScoreEntry();
```



The reference returned by `new` is stored in the box named *myRef*.

Creating New Records – Demo

```
HighScoreEntry myRef;  
myRef = new HighScoreEntry();
```



Even though the record itself is floating without a name, we can follow the reference to access it.

Records

- For records to be useful we need to be able to inspect and modify (access) their contents.

Records

- For records to be useful we need to be able to inspect and modify (access) their contents.
- For this purpose we use the . (dot) operator.

Records

- Recall the format of the record definition.

```
class HighScoreEntry
{
    public string name;
    public int score;
}
```

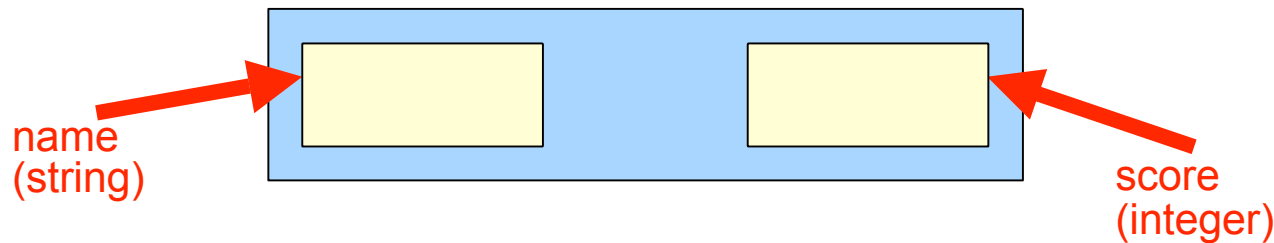
Records

- Recall the format of the record definition.

```
class HighScoreEntry  
{  
    public string name;  
    public int score;  
}
```

Our record has
two elements

name (a string)
score (an integer)

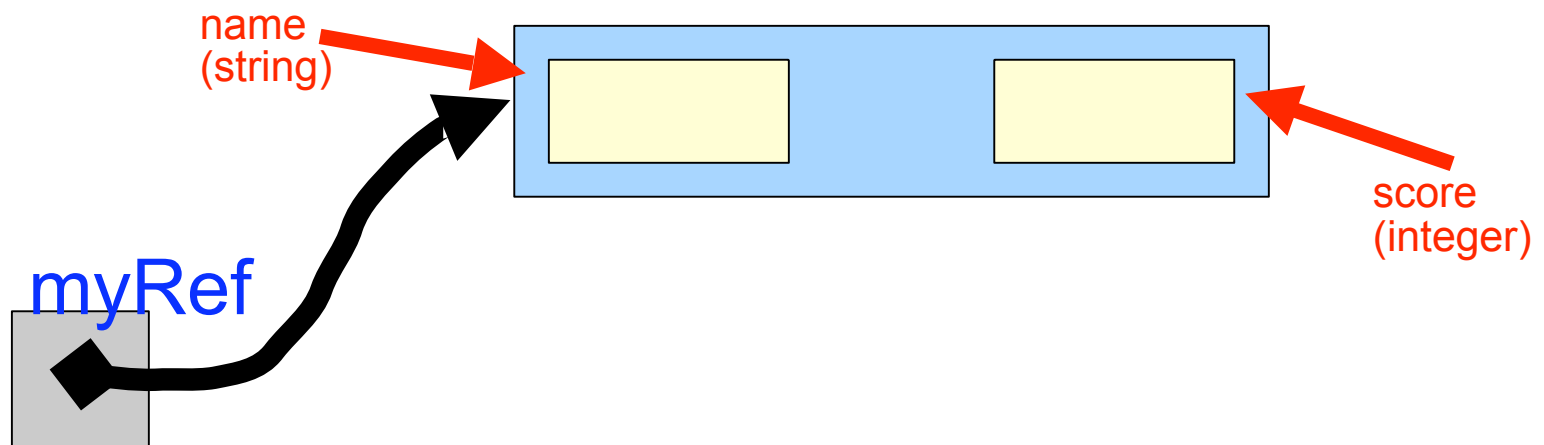


Records

- Given a reference to a record, we can access each of its individual elements (name or score, for example) by using the dot syntax.

Records

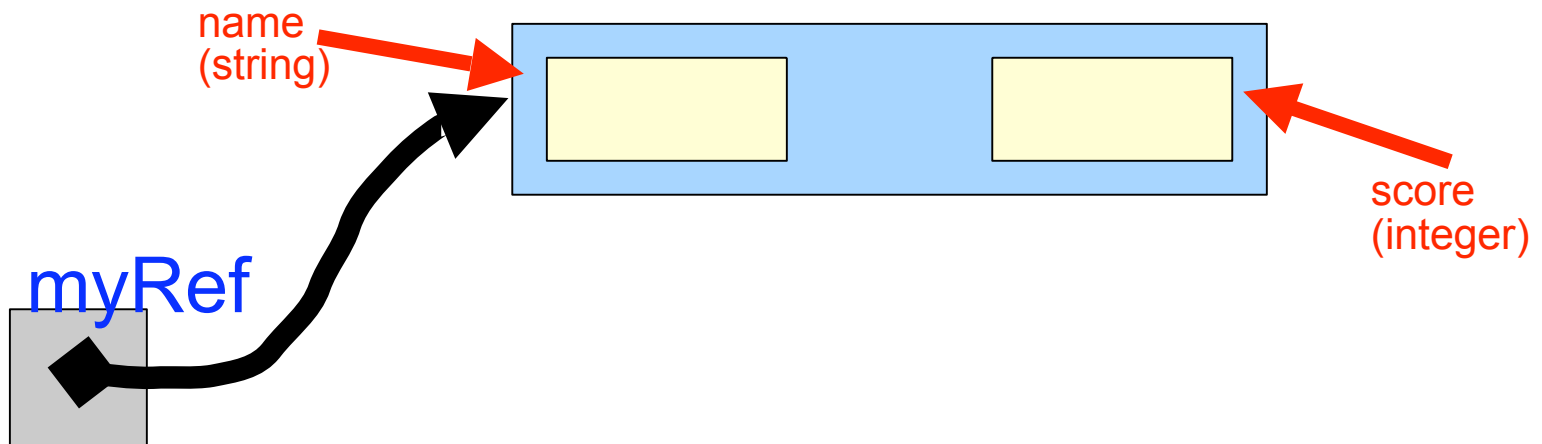
- Given our new record from before...



Records

- Given our new record from before...

```
myRef.name = "Alice";  
myRef.score = 7000;
```

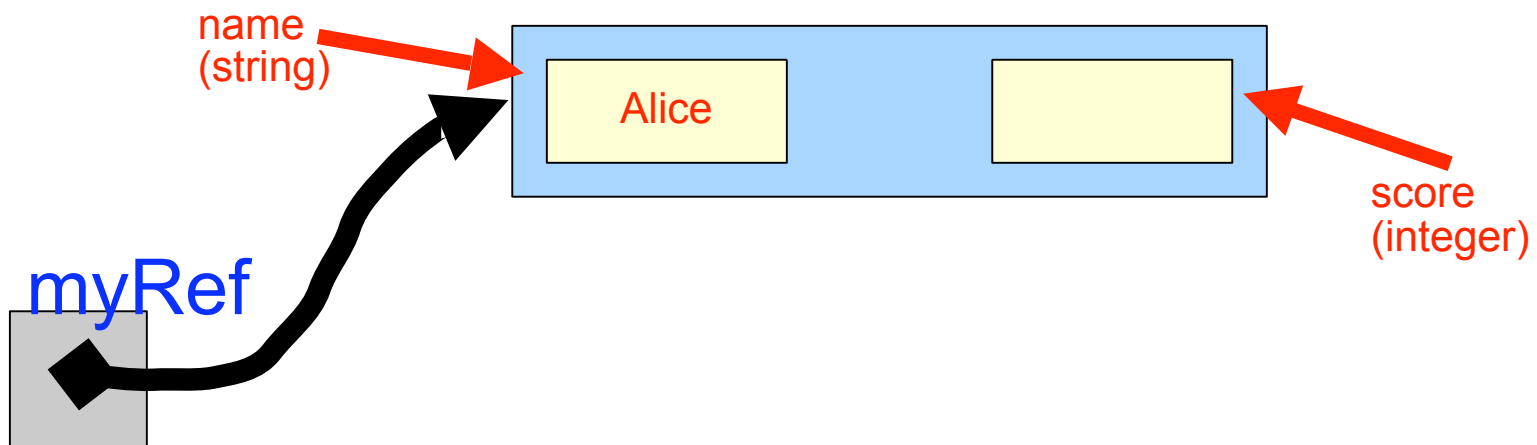


Records

- Given our new record from before...



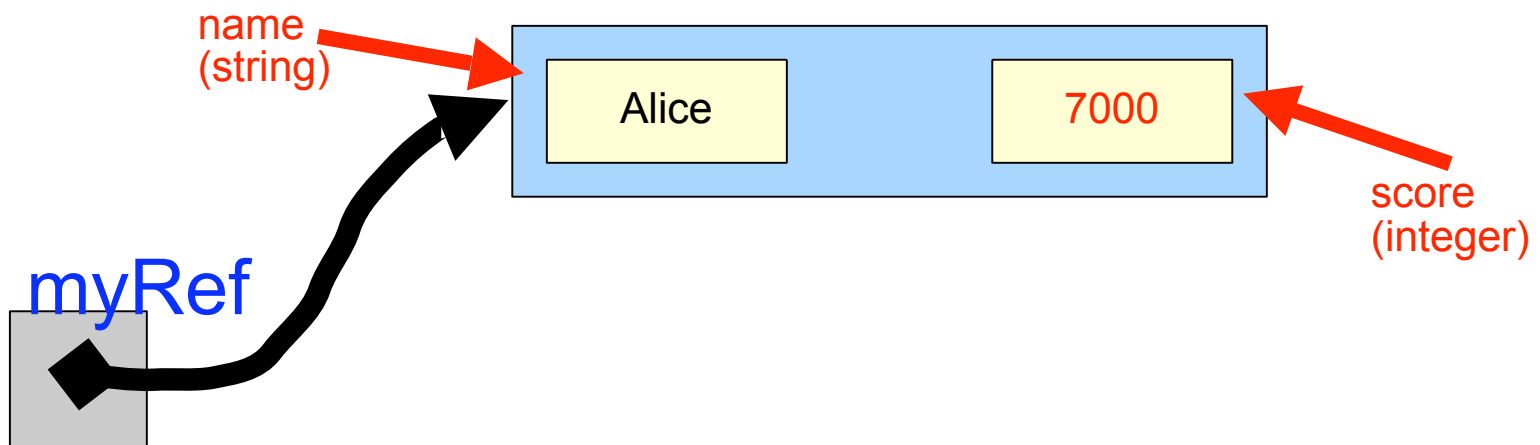
```
myRef.name = "Alice";  
myRef.score = 7000;
```



Records

- Given our new record from before...

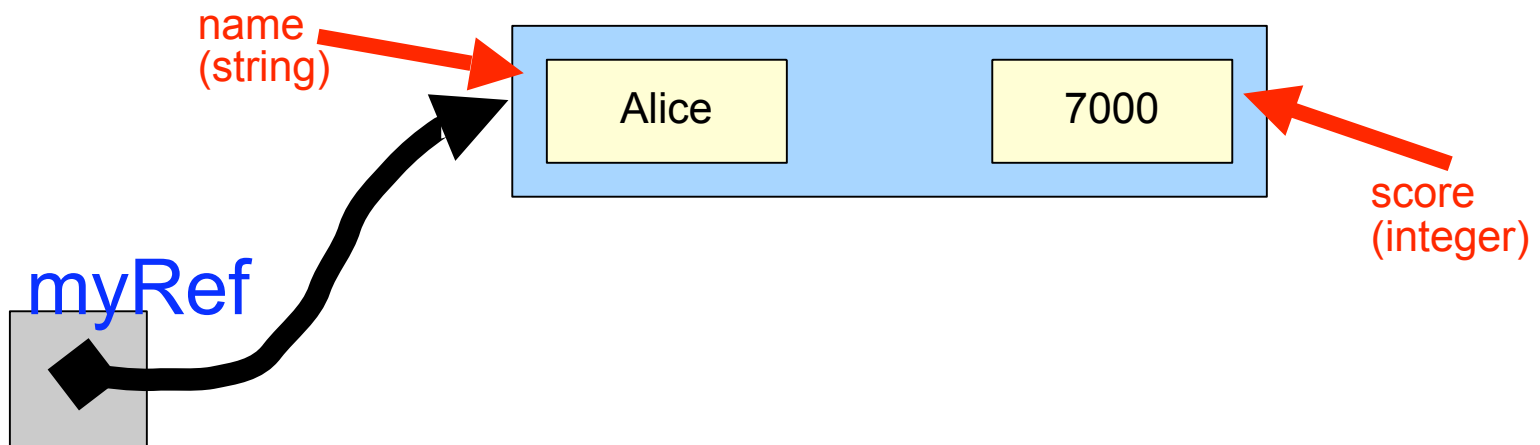
▶ `myRef.name = "Alice";`
`myRef.score = 7000;`



Records

- Given our new record from before...

```
myRef.name = "Alice";  
myRef.score = 7000;
```



A Quick Summary

A Quick Summary

- The *class* keyword defines the layout of a record type.
- The word "class" is used like "category" to define a group of similar objects.
- A class is a kind of template for creating new objects of that type.
- Objects of a specific type are said to belong to that class.

A Quick Summary

- Records are data types that can contain several different elements.
- We can make new records (object instances) using the *new* keyword.
- A record is a specific kind of object.
- A variable (named memory location) can contain a reference to an object. An object itself has no name.

A Quick Summary

- But seriously, what's the deal with references?
- Isn't this “new” business just a really strange way of declaring a variable?

Object References

Object References

- Recall the introduction to variables.
- Each variable is a specific, named place in memory where a value is stored.

```
int a, b, c;
```

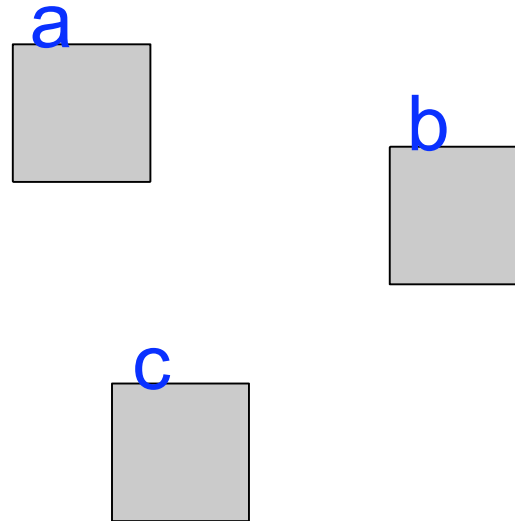
```
a = 5;
```

```
b = a;
```

```
c = 3;
```

```
a = 7;
```

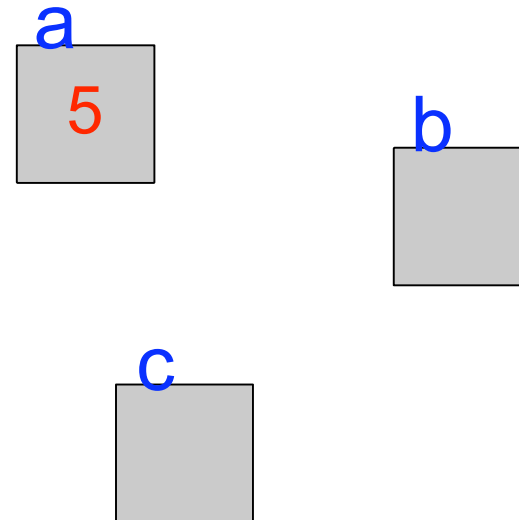
```
c = b + a;
```



Object References


- Recall introduction to variables.
- Each variable is a specific, named place in memory where a value is stored.

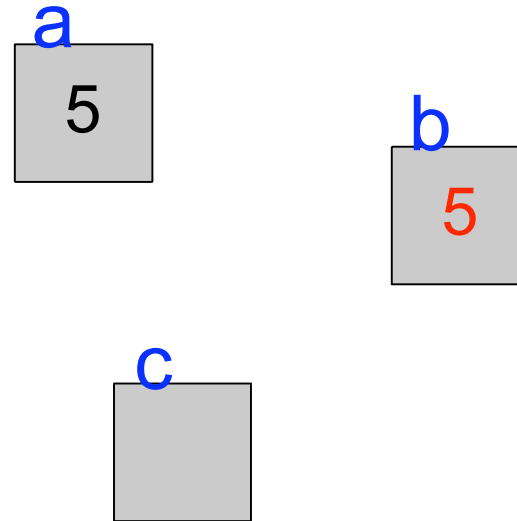
```
int a, b, c;  
  
a = 5; ←  
b = a;  
c = 3;  
a = 7;  
c = b + a;
```



Object References

- Recall introduction to variables.
- Each variable is a specific, named place in memory where a value is stored.

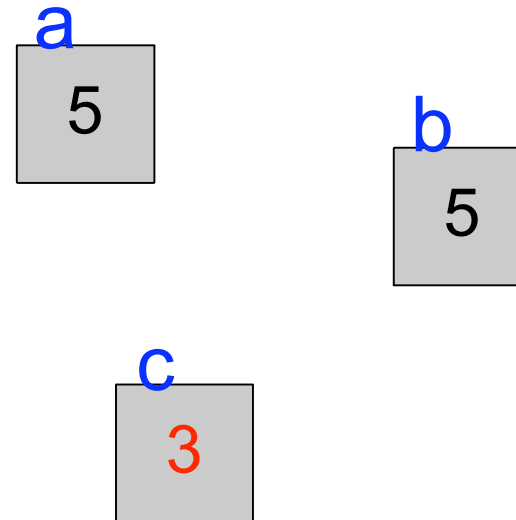
```
int a, b, c;  
  
a = 5;  
b = a;   
c = 3;  
a = 7;  
c = b + a;
```



Object References

- Recall introduction to variables.
- Each variable is a specific, named place in memory where a value is stored.

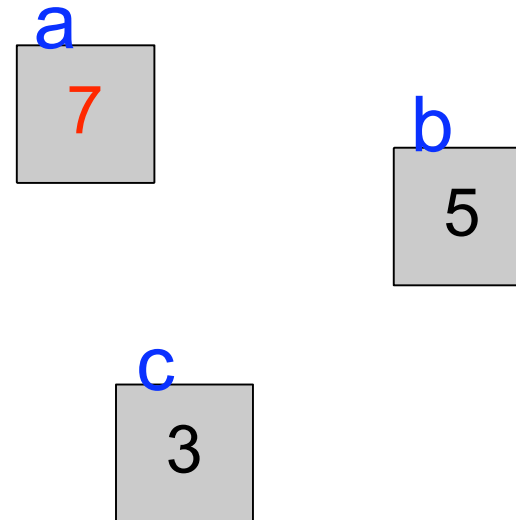
```
int a, b, c;  
  
a = 5;  
b = a;  
c = 3; ←  
a = 7;  
c = b + a;
```



Object References

- Recall introduction to variables.
- Each variable is a specific, named place in memory where a value is stored.

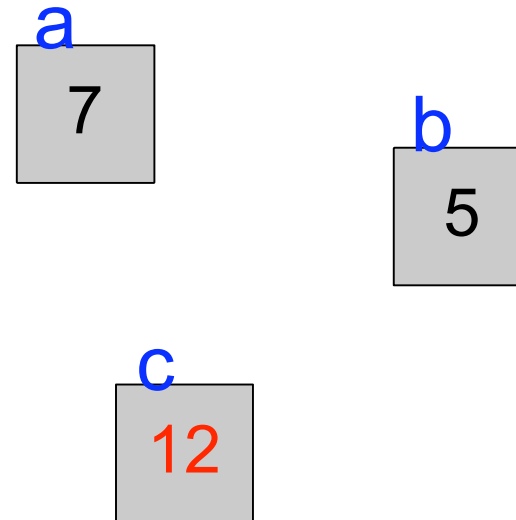
```
int a, b, c;  
  
a = 5;  
b = a;  
c = 3;  
a = 7; ←  
c = b + a;
```



Object References

- Recall introduction to variables.
- Each variable is a specific, named place in memory where a value is stored.

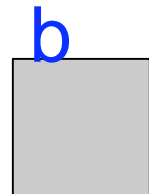
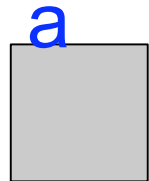
```
int a, b, c;  
  
a = 5;  
b = a;  
c = 3;  
a = 7;  
c = b + a; ←
```



Object References

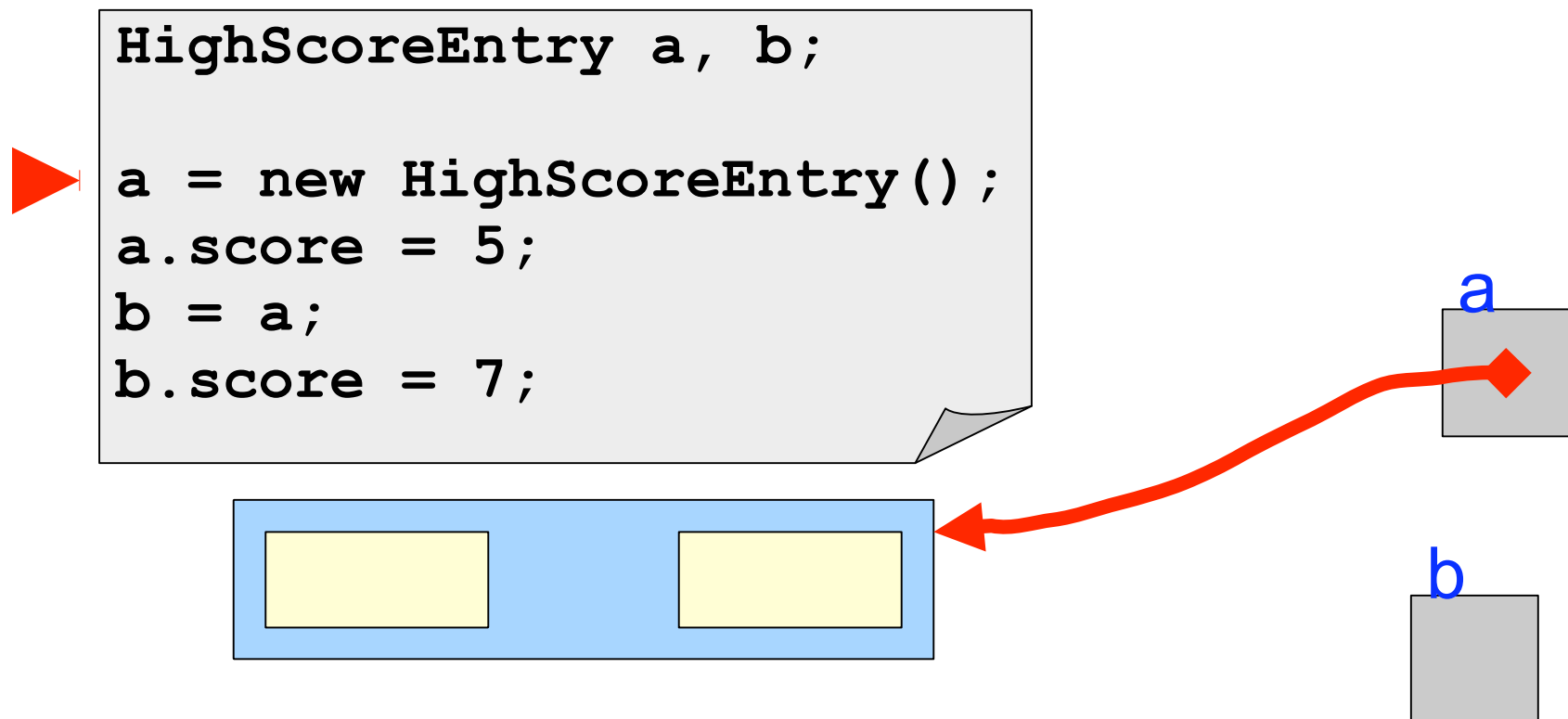
- We can draw a similar diagram for reference types.

```
HighScoreEntry a, b;  
  
a = new HighScoreEntry();  
a.score = 5;  
b = a;  
b.score = 7;
```



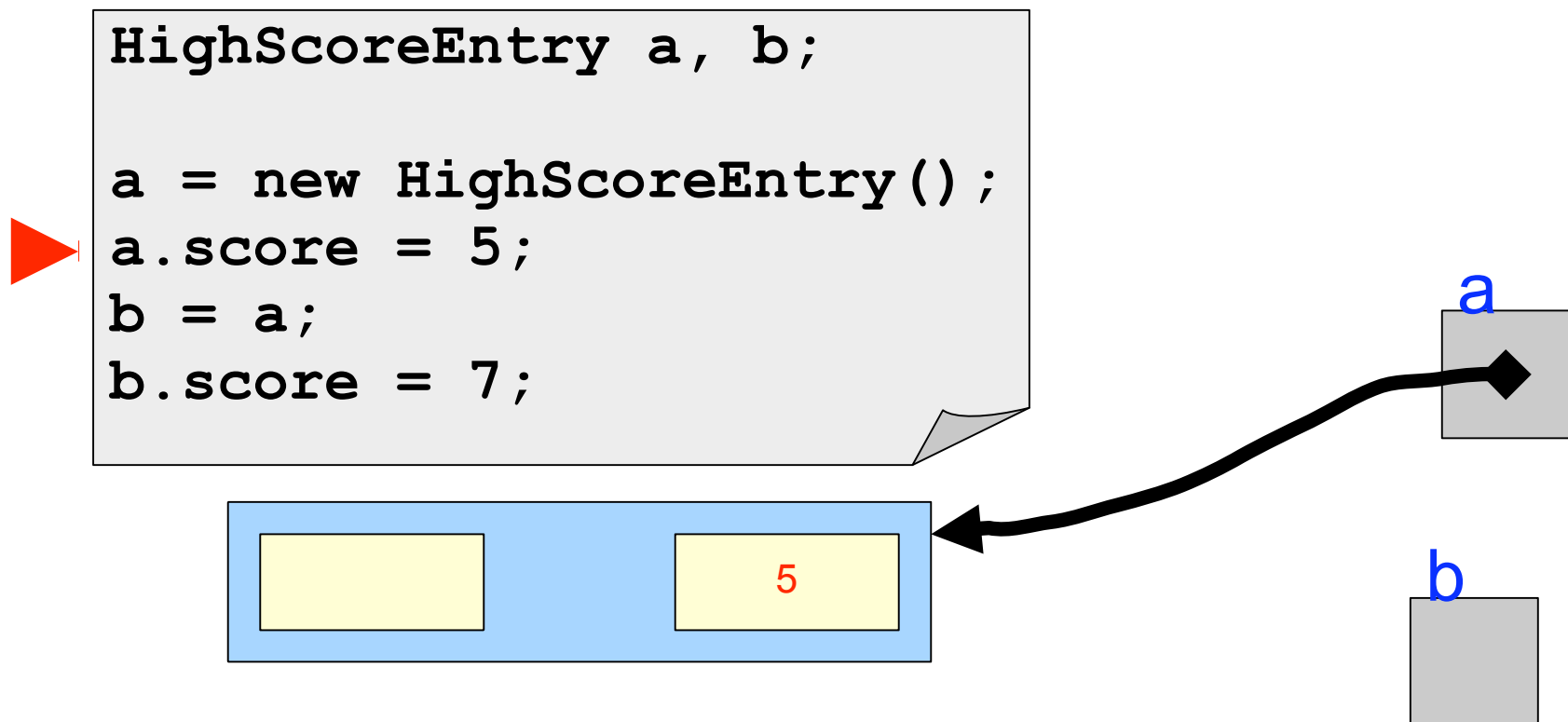
Object References

- We can draw a similar diagram for reference types.



Object References

- We can draw a similar diagram for reference types.

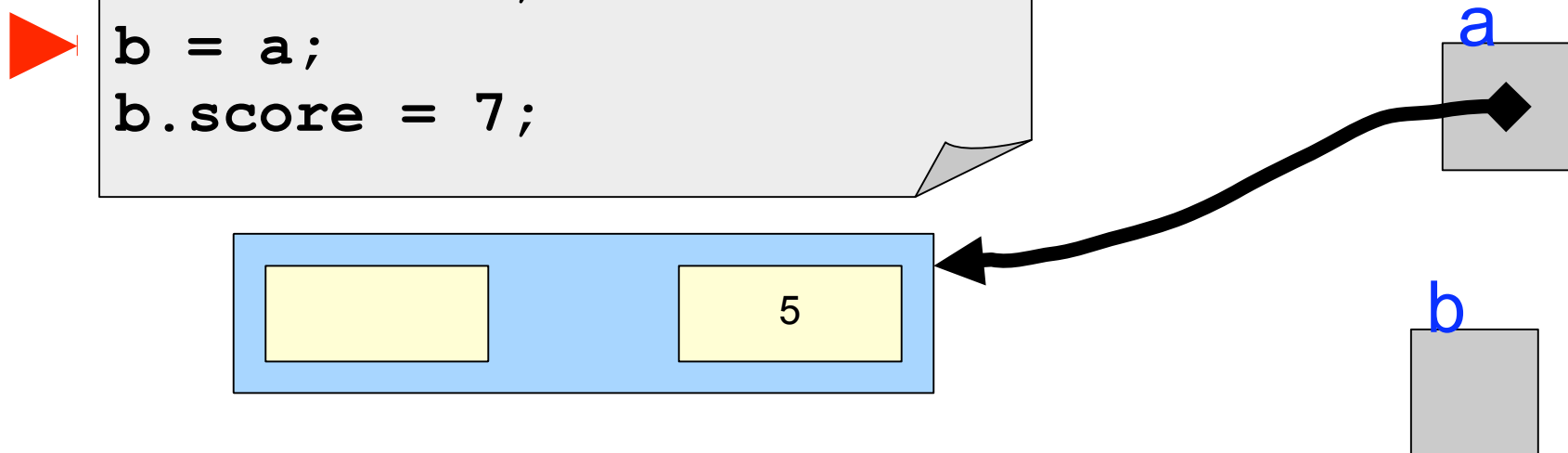


Object References

- We can draw a similar diagram for reference types.

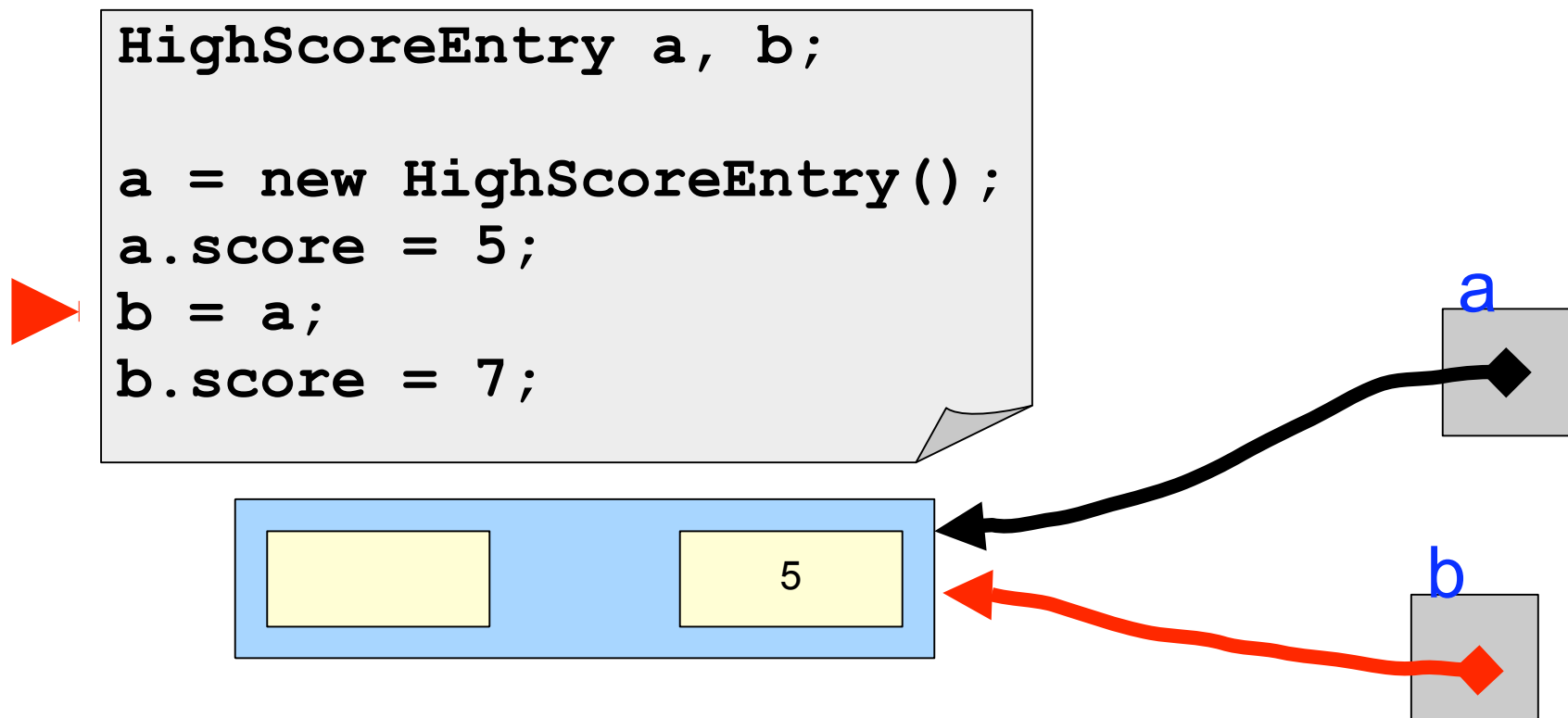
```
HighScoreEntry a, b;  
  
a = new HighScoreEntry();  
a.score = 5;  
b = a;  
b.score = 7;
```

What happens next?



Object References

- We can draw a similar diagram for reference types.

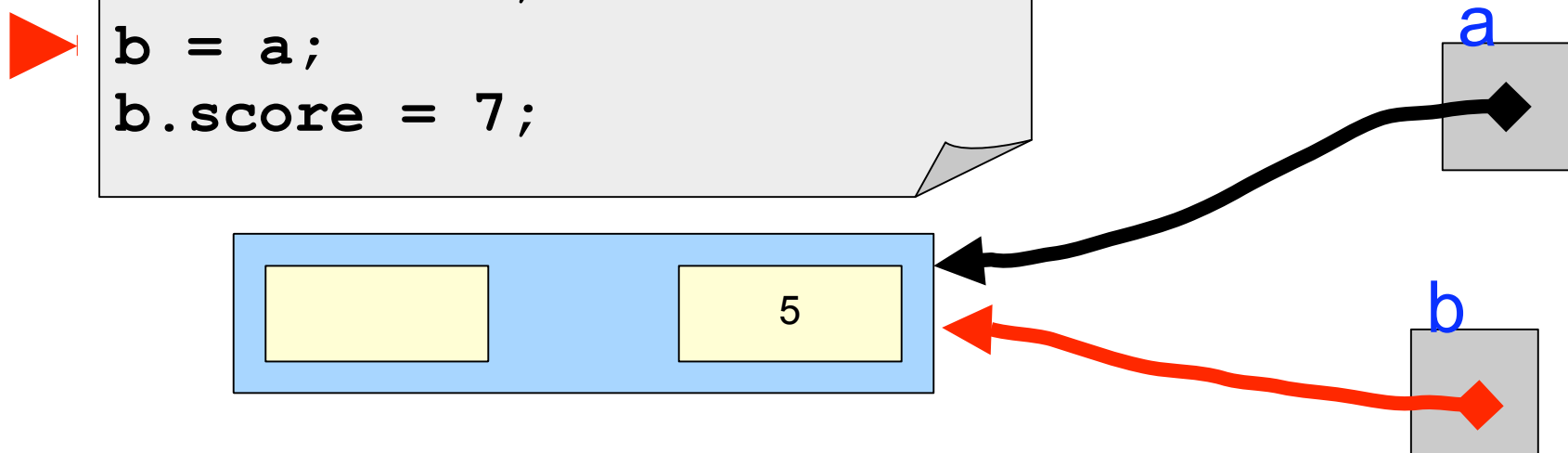


Object References

- We can draw a similar diagram for reference types.

```
HighScoreEntry a, b;  
  
a = new HighScoreEntry();  
a.score = 5;  
b = a;  
b.score = 7;
```

**a and b now
both refer to
the same
record!**

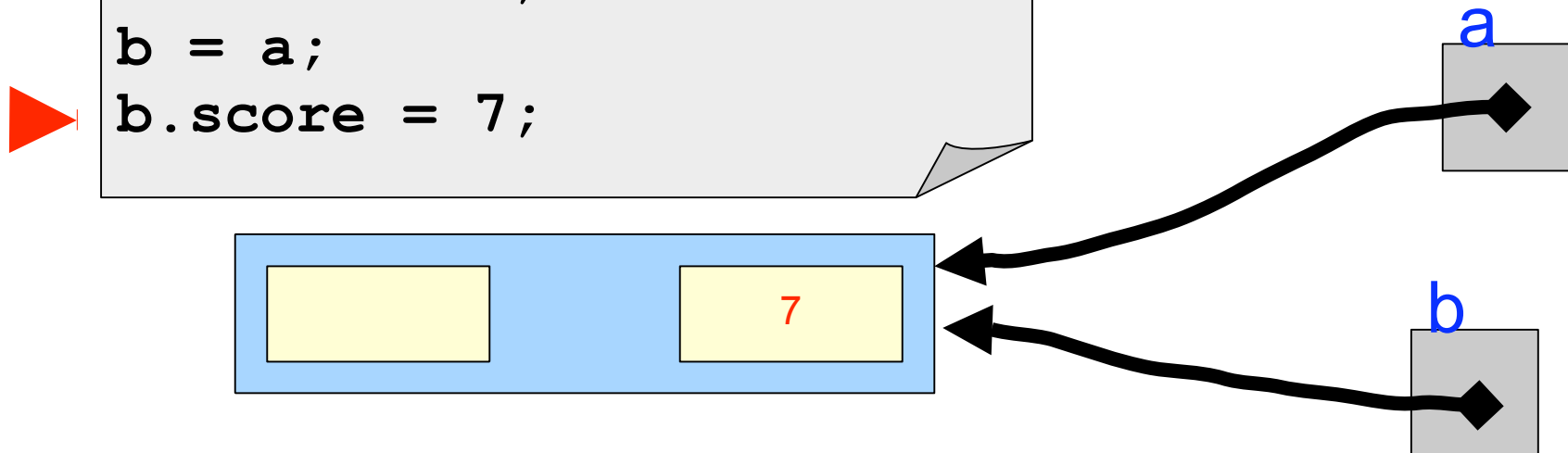


Object References

- We can draw a similar diagram for reference types.

```
HighScoreEntry a, b;  
  
a = new HighScoreEntry();  
a.score = 5;  
b = a;  
b.score = 7;
```

**a and b really
do refer to the
same record!**



Object References

- With references it's possible to have two reference variables referring to the same object.
- In the example, **a** and **b** both refer to the same object.
- Any changes made via **a** (**a.score**, **a.name**) will change **b** as well. Any change made via **b** will change **a** as well.

Objects – Things to Consider

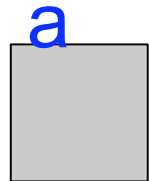
Objects – Things to Consider

- What happens if we have no references to an object?

Objects – Things to Consider

- Take this example

```
HighScoreEntry a;  
  
a = new HighScoreEntry();  
a.score = 5;  
a = new HighScoreEntry();
```



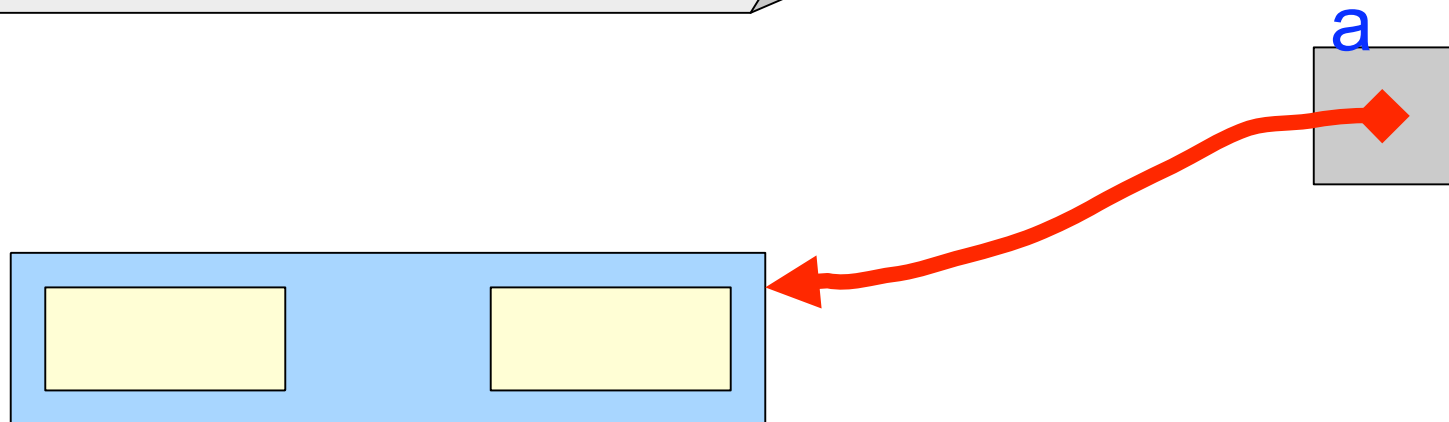
Objects – Things to Consider

- Take this example

```
HighScoreEntry a;
```



```
a = new HighScoreEntry();  
a.score = 5;  
a = new HighScoreEntry();
```



Objects – Things to Consider

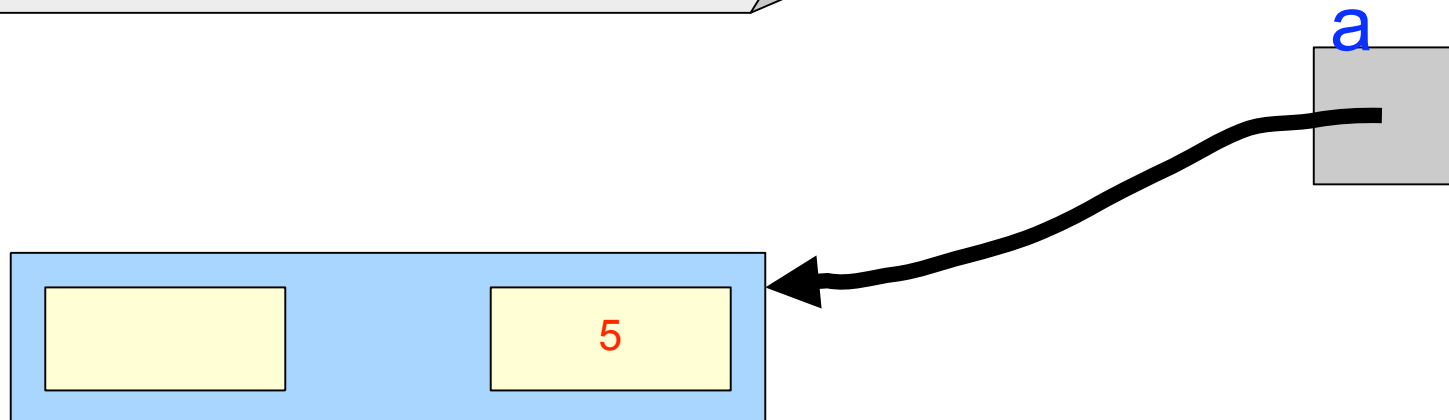
- Take this example

```
HighScoreEntry a;
```

```
a = new HighScoreEntry();
```

```
a.score = 5;
```

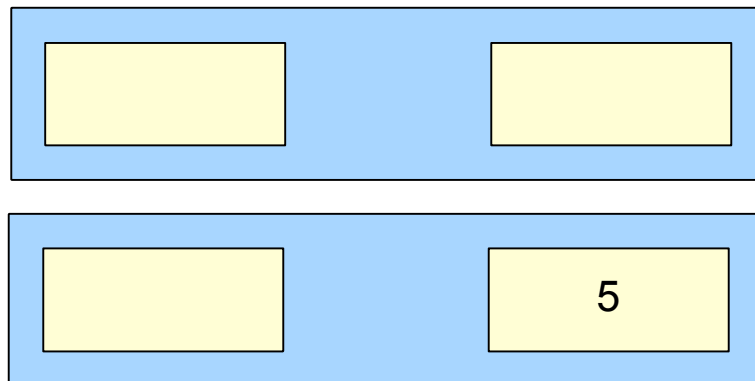
```
a = new HighScoreEntry();
```



Objects – Things to Consider

- Take this example

```
HighScoreEntry a;  
  
a = new HighScoreEntry();  
a.score = 5;  
a = new HighScoreEntry();
```

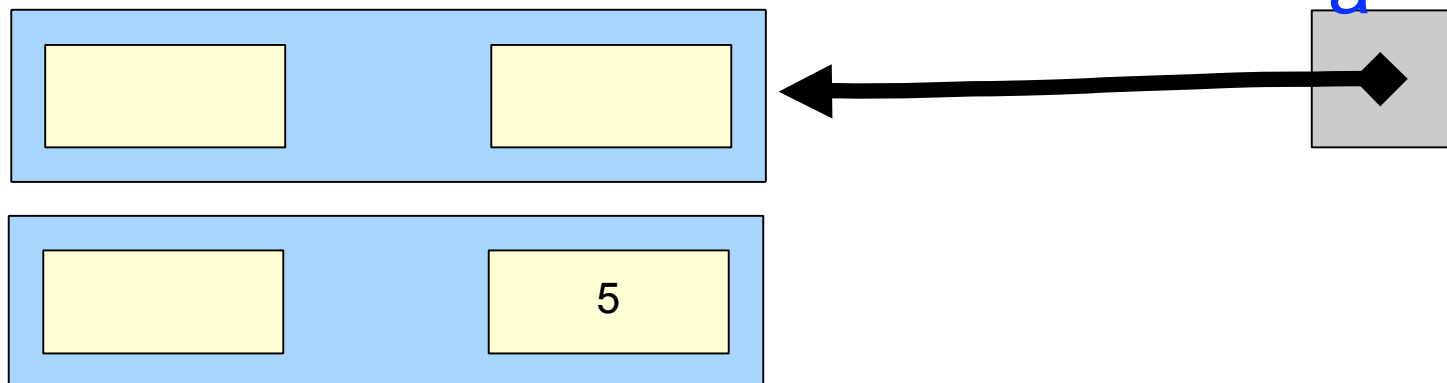


Objects – Things to Consider

- Take this example

```
HighScoreEntry a;  
  
a = new HighScoreEntry();  
a.score = 5;  
a = new HighScoreEntry();
```

The reference to the old entry is replaced by a reference to the new entry



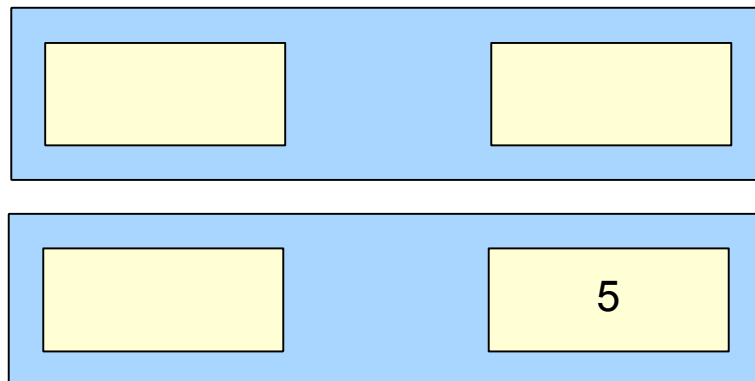
Objects – Things to Consider

- Take this example

```
HighScoreEntry a;  
  
a = new HighScoreEntry();  
a.score = 5;  
a = new HighScoreEntry();
```

We have no way to reference the entry with the 5 in it

What happens to it?



Objects – Things to Consider

- Left to itself, the object would just sit around consuming memory.
- But fortunately, C# has a language feature called *Garbage Collection*.
- When records are no longer accessible they are *garbage collected* (since nobody will notice that they are missing anyway).
- All memory associated with the record is freed for other uses.

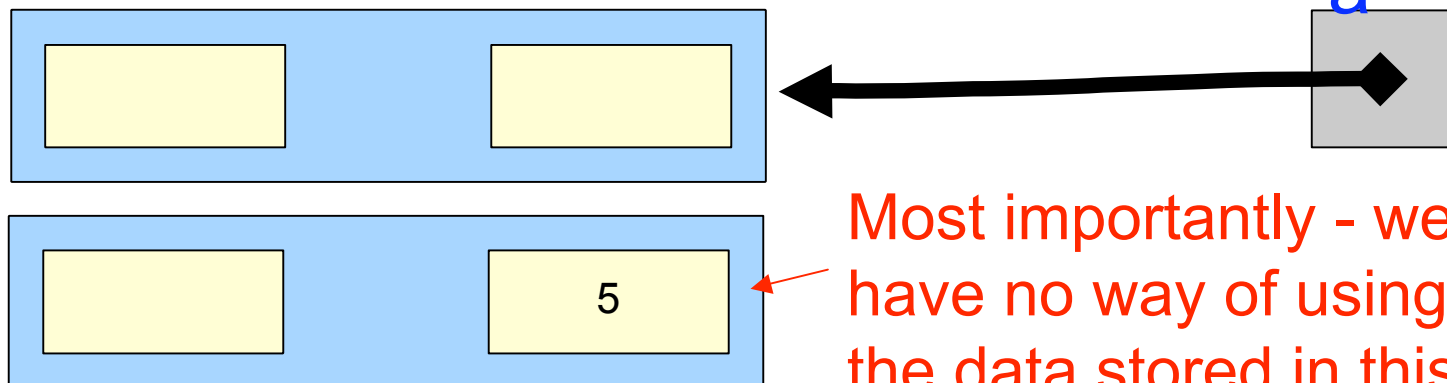
Objects – Things to Consider

- Take this example

```
HighScoreEntry a;  
  
a = new HighScoreEntry();  
a.score = 5;  
a = new HighScoreEntry();
```

We have no way to reference the entry with the 5 in it

It gets garbage collected.



High Scores Yet Again

```
void addScore(int newScore, string newName)
```

```
{
    int mark;    //index of identified element
    int j;       //index used in moving entries down in array
    HSRecord newRecord;

    newRecord = new HSRecord();
    newRecord.score = newScore;
    newRecord.name = newName;
    if (INhighScore < maxElements)
    {
        highScore[INhighScore] = newRecord;
        INhighScore++;
    }
    mark = 0;
    while (mark <= INhighScore - 1 &&
           highScore[mark].score >= newScore)
        mark++;
    if (mark <= INhighScore - 1)
    {
        for (j = INhighScore - 1; j > mark; j--)
        {
            highScore[j] = highScore[j - 1];
        }
        highScore[mark] = newRecord;
    }
}
```

DECLARATIONS REQUIRED

```
class HSRecord
{
    public string name;
    public int score;
}
const int maxElements = 6;
HSRecord[] highScore =
    new HSRecord[maxElements];
int INhighScore = 0;    //Num elements in
                        //highScore
```

High Scores Yet Again

- AddScores with objects

```
mark = 0;
while (mark <= INhighScore - 1 &&
      highScore[mark].score >= newScore)
    mark++;

if (mark <= INhighScore - 1)
{
    for (j = INhighScore - 1; j > mark; j--)
    {
        highScore[j] = highScore[j - 1];
    }
    highScore[mark] = newRecord;
}
```

moves both score
and name in a
single instruction

note: it is
moving the
references to the score and name!

careful not to “lose”
the reference to the
new record

Objects – More Things to Consider

- What is contained in a reference that doesn't (yet) point to a valid record?
- What happens if you try to access the non-existent record pointed to by such a reference?

Objects – More Things to Consider

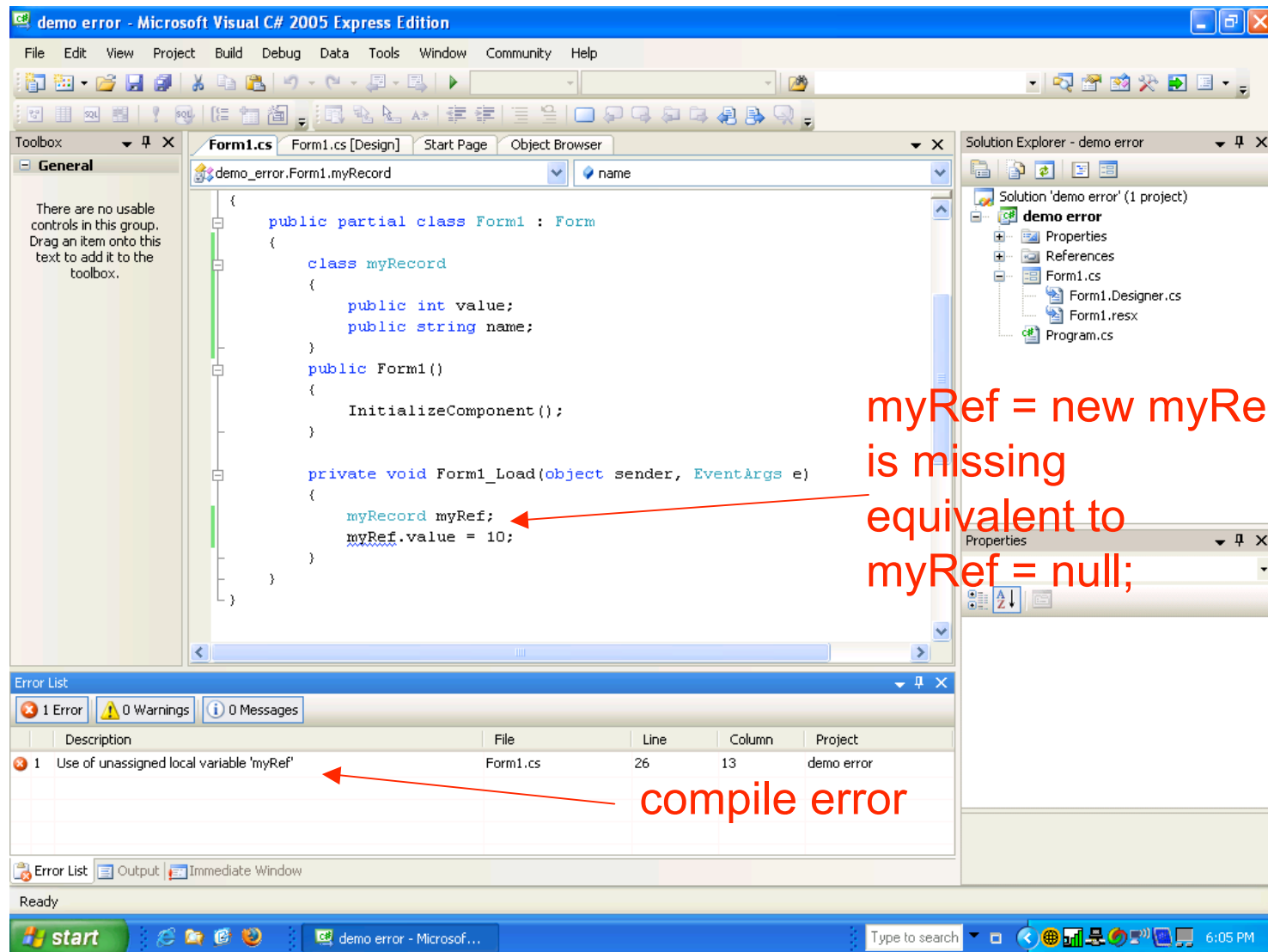
- New references are initialised to null until you assign a new value to them.
- "null" means that the reference variable refers to no object.

Objects – More Things to Consider

- But what happens if you try to access a null reference?

```
MyClass myRef;  
  
myRef = null;  
myRef.value = 10;
```

Objects – More Things to Consider



Objects – More Things to Consider

- It is not always possible to detect null references at compile time.
- After a null reference exception occurs your program crashes and stops running.
- Sometimes we won't know if a reference variable is null or not, so how do we handle that?

Objects – More Things to Consider

- Where necessary, we can avoid crashes by checking our reference variables to ensure that they are non-null before accessing them.

```
MyClass myRef;  
  
<... myRef might be null ...>  
  
if (myRef != null)  
    myRef.value = 10;
```