Object Behavior

Constructors
Member functions (aka methods)
Overloading member functions
Static/class functions

Constructors

Constructors

Allow us to specify some of the attributes right away when we create the object.

A C++ Class with a Constructor

```
class ball
{
 public:
    int x;
    int y;
    int size;

    ball(int newX, int newY, int newSize)
    {
        this->x = newX;
        this->y = newY;
        this->size = newSize;
    }
};
```

A C++ Class with a Constructor

```
class ball
{
 public:
    int x;
    int y;
    int size;

    ball(int newX, int newY, int newSize)
    {
        this->x = newX;
        this->y = newY;
        this->size = newSize;
    }
};
```

A constructor is a function that is called automatically when an object instance is created

A C++ Class with a Constructor

```
class ball
{
 public:
    int x;
    int y;
    int size;

    ball(int newX, int newY, int newSize)
    {
        this->x = newX;
        this->y = newY;
        this->size = newSize;
    }
};
```

Parameters are usually used to set up initial values

A C++ Class with a Constructor

```
class ball
{
 public:
    int x;
    int y;
    int size;

    ll(int newX, int newY, int newSize)

    this->x = newX;
    this->y = newY;
    this->size = newSize;

};
```

You use the `this` keyword with an arrow to access an object's variables from a member function

A C++ Class with a Constructor

Member Functions

```
ball(int newX,
     int newY,
     int newSize)
{ … }
```

```
class ball { … }
```

A C++ Class with a Constructor

When a class is defined, its member functions (including the constructor) will be available in the code portion of the program's memory

Member Functions

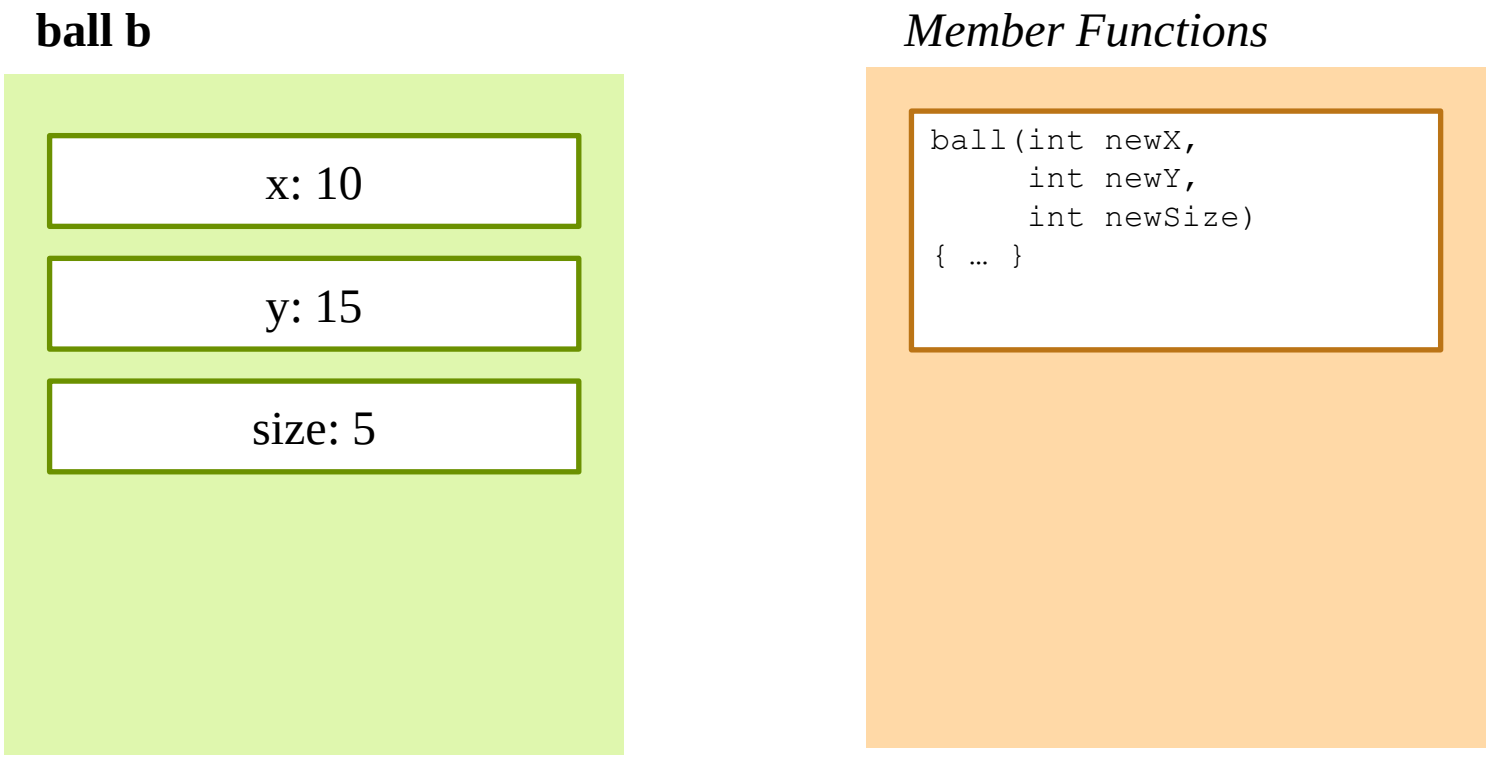
```
ball(int newX,
     int newY,
     int newSize)
{ … }
```

```
class ball { … }
```

A C++ Class with a Constructor

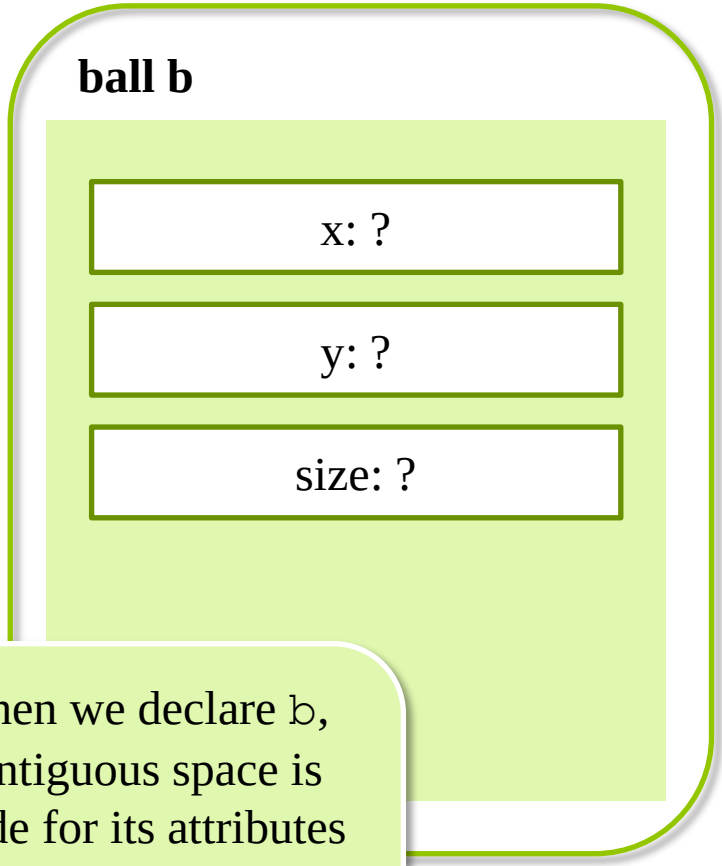
```
// Declare a new object of type ball
// using the parameters defined in the
// constructor

ball b(10, 15, 5);
```

A C++ Class with a Constructor




```
ball b(10, 15, 5);
```

A C++ Class with a Constructor



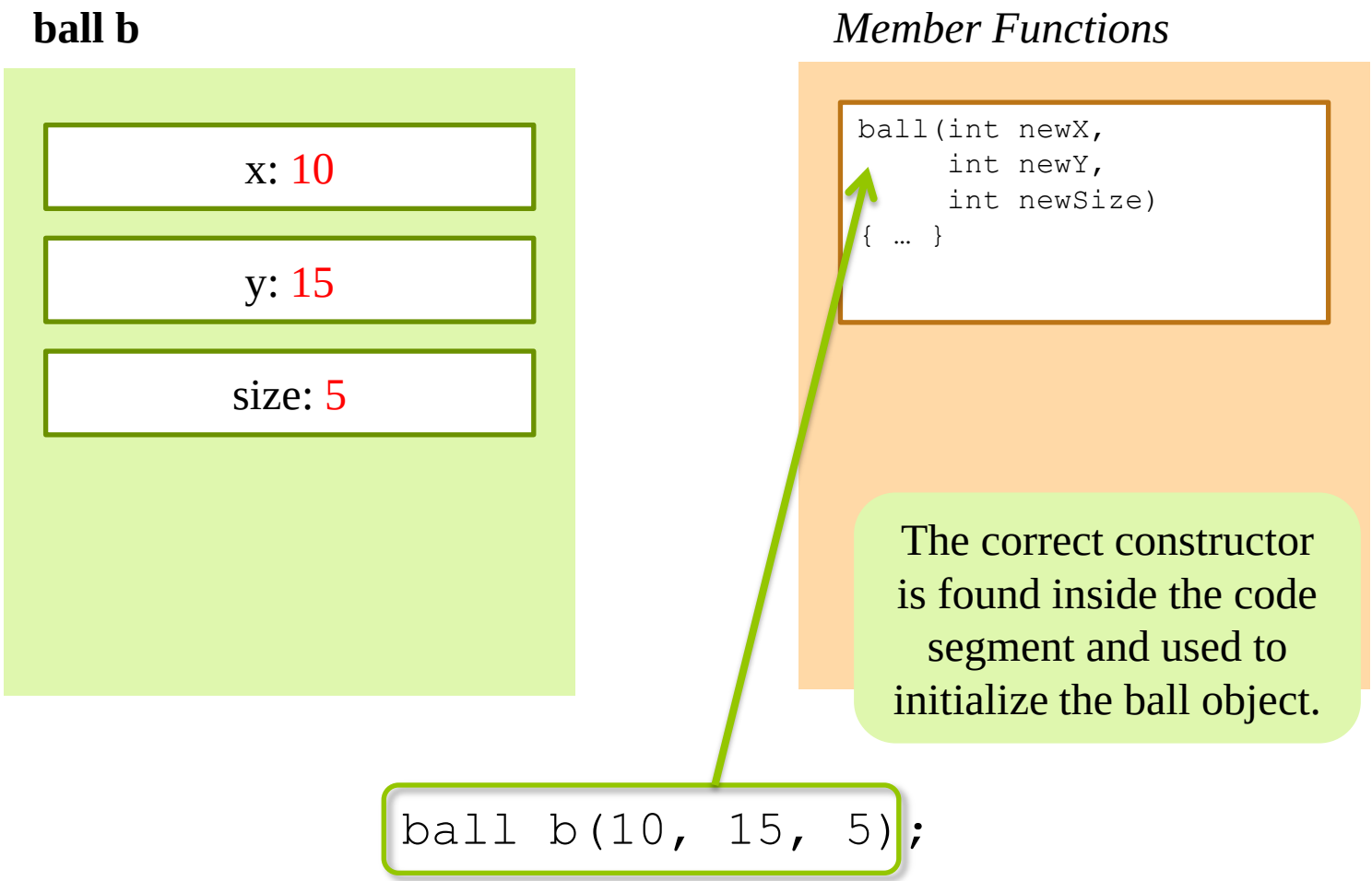
Member Functions

```
ball(int newX,
     int newY,
     int newSize)
{ … }
```

When we declare `b`, contiguous space is made for its attributes in memory, just like a `struct` (in this case, on the stack)

```
ball b(10, 15, 5);
```

A C++ Class with a Constructor



A C++ Class with a Constructor

```
class ball
{
 public:
    int x;
    int y;
    int size;

    ball(int newX, int newY, int newSize);
};

ball::ball(int newX, int newY, int newSize)
{
   this->x = newX;
   this->y = newY;
   this->size = newSize;
}
```

A C++ Class with a Constructor

```
class ball
{
 public:
    int x;
    int y;
    int size;

    ball(int newX, int newY, int newSize);
};

ball::ball(int newX, int newY, int newSize)
{
   this->x = newX;
   this->y = newY;
   this->size = newSize;
}
```

C++ lets you declare the constructor inside the class…

A C++ Class with a Constructor

```
class ball
{
 public:
    int x;
    int y;
    int size;

    ball(int newX, int newY, int newSize);
};
```

```
ball::ball(int newX, int newY, int newSize)
{
   this->x = newX;
   this->y = newY;
   this->size = newSize;
}
```

…and define it outside, to help with organization.

A C++ Class with a Constructor

```
class ball
{
 public:
    int x;
    int y;
    int size;

    ball(int newX, int newY, int newSize);
};

ball::ball(int newX, int newY, int newSize)
{
                                    wSize;
```

The class name and scope operator : : are used to tell C++ what class the function belongs to.

A Java Class with a Constructor

```
public class Ball
{
   int x;
   int y;
   int size;

   public Ball(int newX, int newY, int newSize)
   {
      this.x = newX;
      this.y = newY;
      this.size = newSize;
   }
}
```

A Java Class with a Constructor

```
public class Ball
{
   int x;
   int y;
   int size;

   public Ball(int newX, int newY, int newSize)
   {
      this.x = newX;
      this.y = newY;
      this.size = newSize;
   }
}
```

The constructor looks very similar to C++

A Java Class with a Constructor

```
public class Ball
{
   int x;
   int y;
   int size;

   ewX, int newY, int newSize)
   {
      this.x = newX;
      this.y = newY;
      this.size = newSize;
   }
}
```

The `this` keyword is used with a dot in Java

A Java Class with a Constructor

```
// Create an instance of Ball using
// the parameters defined in the
// constructor

Ball myNewBall = new Ball(10, 15, 5);
```

A Java Class with a Constructor

Ball myNewBall

x: 10

y: 15

size: 5

Methods

```
Ball(int newX,
     int newY,
     int newSize)
{ … }
```

```
Ball myNewBall = new Ball(10, 15, 5);
```

```
public class Person
{
    String          firstName;
    String          lastName;
    int             age;
    char            gender;
    boolean         retired;
}
```

Methods

```
Person()
{
}
```

```
public class Person
{
    String          firstName;
    String          lastName;
    int             age;
    char            gender;
    boolean         retired;
}
```

Methods

```
Person()
{
}
```

Notice that if we don't provide a constructor, we get one for free…

```
Person    p1;

p1 = new Person();

p1.firstName = "Bobby";
p1.lastName = "Socks";
p1.age = 24;
p1.gender = 'M';
p1.retired = false;
```

Person p1

firstName: "Bobby"

lastName: "Sox"

age: 24

gender: 'M'

retired: false

Methods

```
Person()
{
}
```

```
public Person(String f,
              String l,
              int a, char g,
              boolean r)
{
    firstName = f;
    lastName = l;
    age = a;
    gender = g;
    retired = r;
}
```

Methods

```
Person(String f,
       String l,
       int a, char g,
       boolean r)
{
   …
}
```

```
Person p1, p2, p3;

p1 = new Person("Bobby", "Socks", 24,
                'M', false);

p2 = new Person("Holly", "Day", 72,
                'F', true);

p3 = new Person("Hank", "Urchif", 19,
                'M', false);
```

```
Person p1, p2, p3;

p1 = new Person("Bobby", "Socks", 24,
                'M', false);

p2 = new Person("Holly", "
                'F', true);

p3 = new Person("Hank", "Urchif", 19,
                'M', false);
```

What happens if we don't know the age when making the object?

```
public Person(String f, String l,
              char g, boolean r)
{
    firstName = f;
    lastName = l;
    gender = g;
    retired = r;
    age = 0;
}
```

Just define a second constructor!

```
public Person(String f, String l,
              char g, boolean r)
{
    firstName = f;
    lastName = l;
    gender = g;
    retired = r;
    age = 0;
}
```

Be careful when choosing default values

Poll Everywhere Question

Given the following code, which poll option will work?

```
public class Person
{
  String   firstName;
  String   lastName;
  int      age;
  char     gender;
  boolean  retired;

  public Person(String f, String l,
                int a, char g,
                boolean r)
  {
    firstName = f;
    lastName = l;
    age = a;
    gender = g;
    retired = r;
   }
}
```

Text 37607

184214

```
Person p = new
Person("Jane",
 "Doe", 'F');
```

263645

```
Person p = new
 Person();
```

263646

None – there will be an error

Methods / Member Functions

Objects

Class Definition

Variables
Methods/Member function

Object Instance

Variables
Methods/Member function

Objects

Class Definition

Variables

Methods/Member function

Object Instance

Variables

Methods/Member function

These define the behaviour of the object

Adding Behavior in C++

```
class ball
{
 public:
    int x;
    int y;
    int size;

    ball(int newX, int newY, int newSize)
    {
        this->x = newX;
        this->y = newY;
        this->size = newSize;
    }
};

void printBall(ball b)
{
    cout << "Ball at " << b.x << ", " << b.y
         << " with size " << b.size << endl;
}
```

Adding Behavior in C++

```
class ball
{
 public:
    int x;
    int y;
    int size;

    ball(int newX, int newY, int newSize)
    {
        this->x = newX;
        this->y = newY;
        this->size = newSize;
    }
};
```

Procedural style:
So far we have written functions to *do things with* an object outside the class

```
void printBall(ball b)
{
    cout << "Ball at " << b.x << ", " << b.y
         << " with size " << b.size << endl;
}
```

Adding Behavior in C++

```
class ball
{
 public:
    int x;
    int y;
    int size;

    ball(int newX, int newY, int newSize)
    {
        this->x = newX;
        this->y = newY;
        this->size = newSize;
    }

    void print()
    {
        cout << "Ball at " << this->x << ", " << this->y
             << " with size " << this->size << endl;
    }
};
```

Adding Behavior in C++

```
class ball
{
 public:
    int x;
    int y;
    int size;

    ball(int newX, int newY, int newSize)
    {
        this->x = newX;
        this->y = newY;
        this->size = newSize;
    }

    void print()
    {
        cout << "Ball at " << this->x << ", " << this->y
             << " with size " << this->size << endl;
    }
};
```

Object-oriented style: Instead, make printing something a ball object can *do itself*

Adding Behavior in C++

```
class ball
{
 public:
    int x;
    int y;
    int size;

    ball(int newX, int newY, int newSize)
    {
        this->x = newX;
        this->y = newY;
        this->size = newSize;
    }

    void print()
    {
        cout << "Ball at " << this->x << ", " << this->y
             << " with size " << this->size << endl;
    }
};
```

Just like in the constructor, the object can access its attributes with `this`

Adding Behavior in C++

```
class ball
{
 public:
    int x;
    int y;
    int size;

    ball(int newX, int newY, int newSize)
    {
        x = newX;
        y = newY;
        size = newSize;
    }

    void print()
    {
        cout << "Ball at " << x << ", " << y
             << " with size " << size << endl;
    }
};
```

In fact, `this` is implied when writing constructors and methods and it is better style to drop it if there is no ambiguity

Adding Behavior in C++

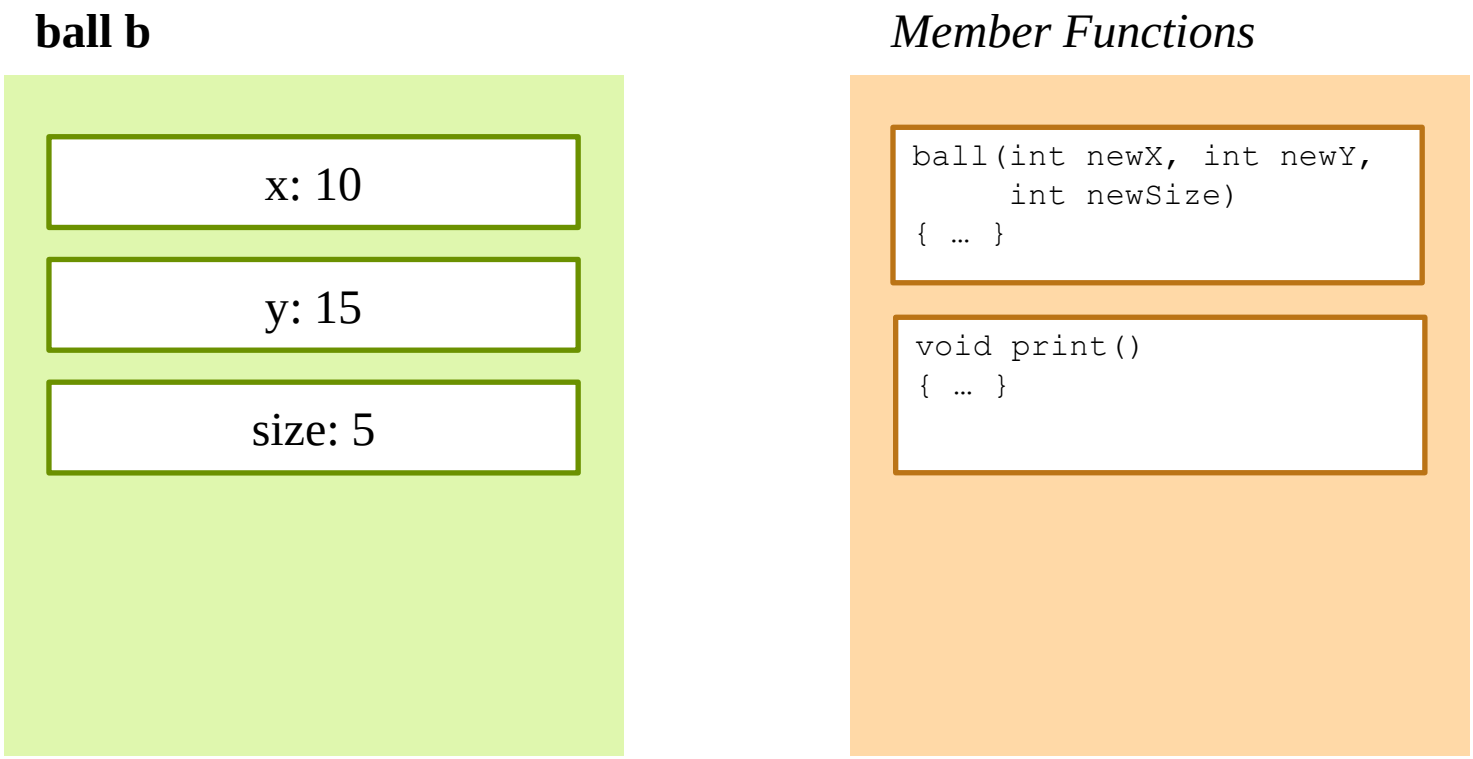
```
class ball
{
 public:
    int x;
    int y;
    int size;

    ball(int newX, int newY,
         int newSize)
    {
        x = newX;
        y = newY;
        size = newSize;
    }

    void print()
    {
        cout << "Ball at " << x << ", " << y
             << " with size " << size << endl;
    }
};
```

Member Functions

```
ball(int newX, int newY,
     int newSize)
{ … }
```

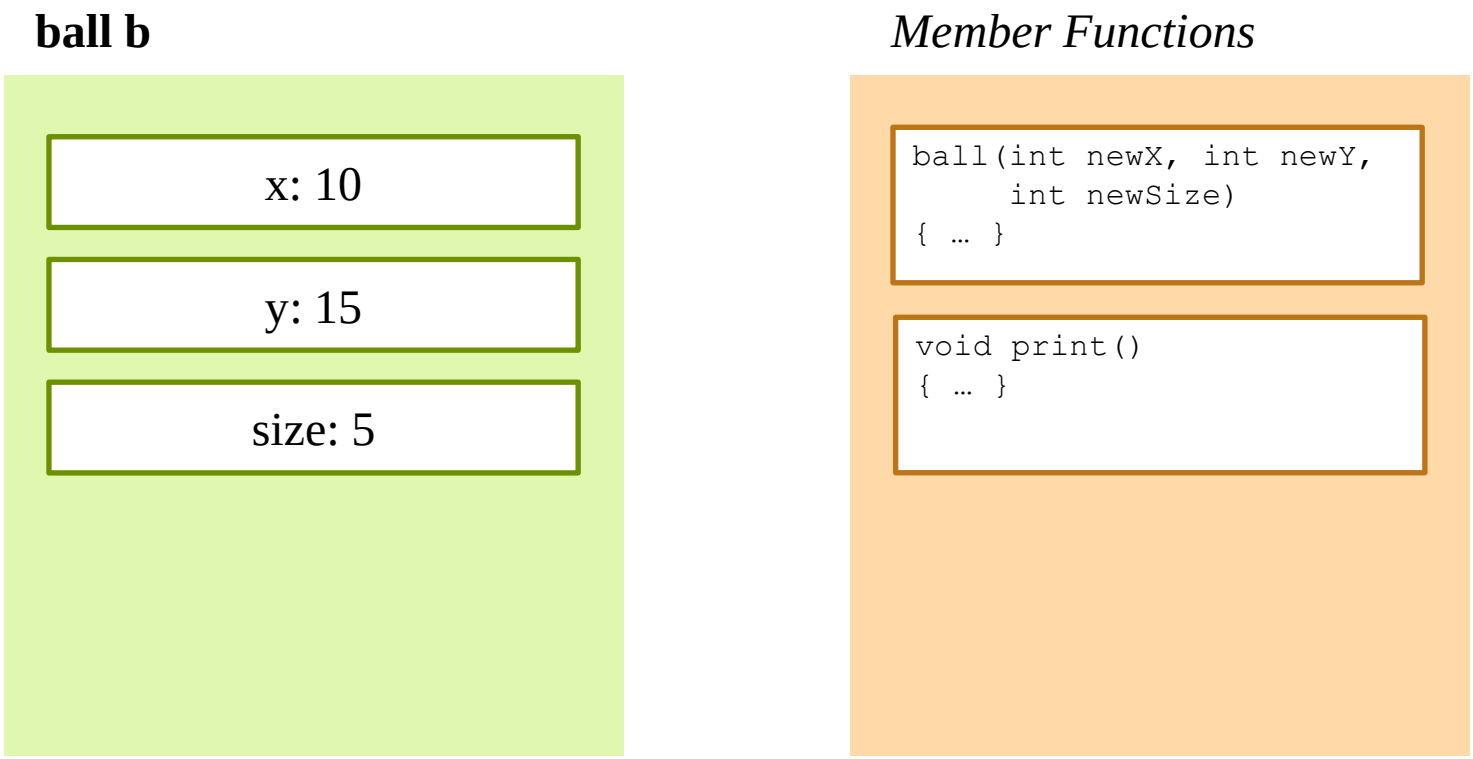
```
void print()
{ … }
```

Adding Behavior in C++



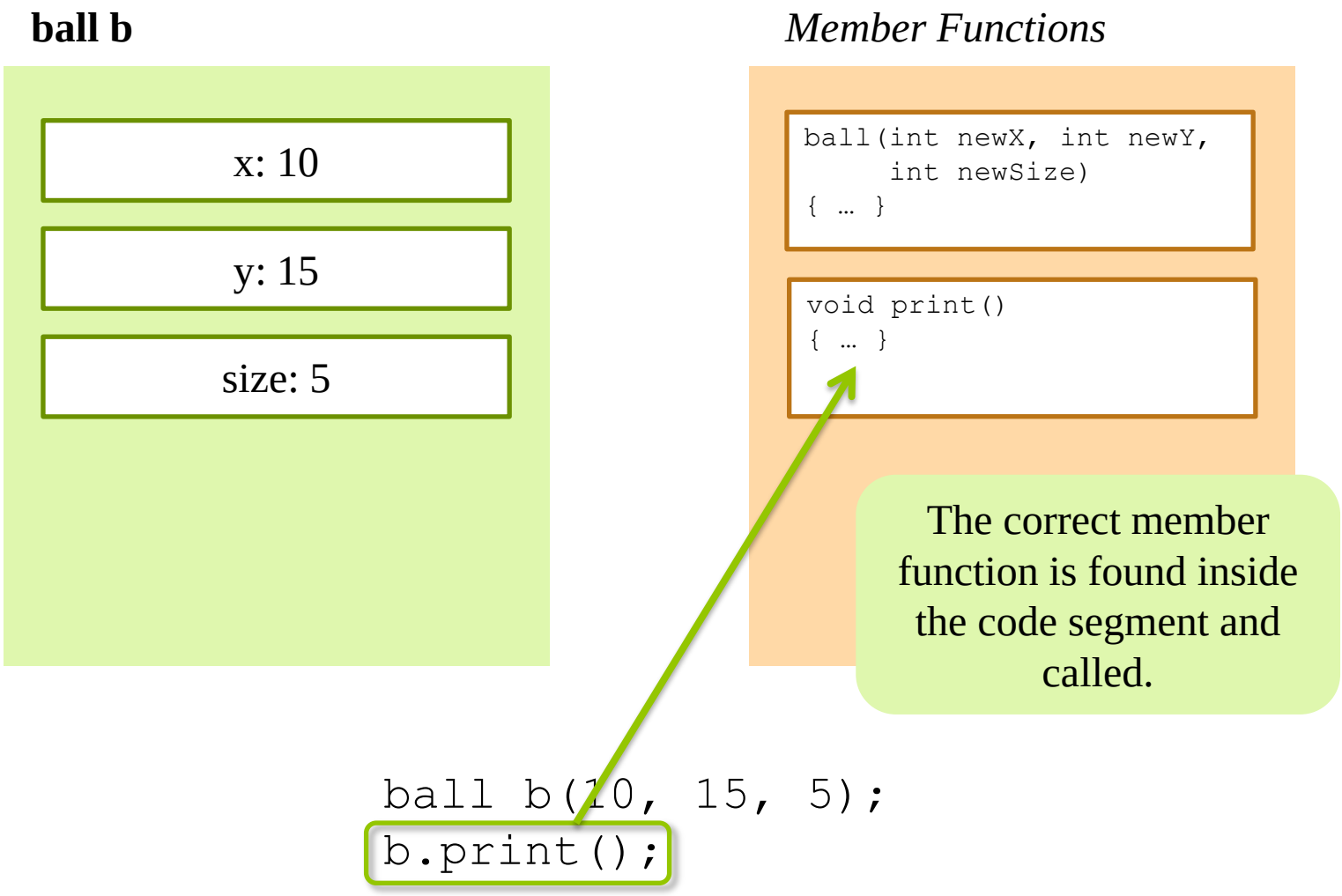

```
ball b(10, 15, 5);
```

Adding Behavior in C++




```
ball b(10, 15, 5);
b.print();
```

Adding Behavior in C++



Adding Behavior in C++

Just like constructors, methods can be defined outside the class definition using the scope operator : :

```
ball::ball(int newX, int newY,
           int newSize)
{
   x = newX;
   y = newY;
   size = newSize;
}

void ball::print()
{
   cout << "Ball at " << x << ", "
        << y << " with size "
        << size << endl;
}
```

```
class ball
{
 public:
    int x;
    int y;
    int size;

    ball(int newX, int newY, int newSize);
    void print();
};
```

Adding Behavior in Java

```
public class Ball
{
  int x;
  int y;
  int size;

  public Ball(int newX, int newY, int newSize)
  {
     this.x = newX;
     this.y = newY;
     this.size = newSize;
  }

  public void print()
  {
      System.out.println("Ball at " + this.x + ", " + this.y +
                         " with size " + this.size);
  }
}
```

The same ideas apply to Java classes as well.

Adding Behavior in Java

```
public class Ball
{
  int x;
  int y;
  int size;

  public Ball(int newX, int newY, int newSize)
  {
     this.x = newX;
     this.y = newY;
     this.size = newSize
  }

  public void print()
  {
      System.out.println("Ball at " + this.x + ", " + this.y +
                         " with size " + this.size);
  }
}
```

Look! No static! That's because our method is part of an object's behaviour now.

Adding Behavior in Java

```
public class Ball
{
  int x;
  int y;
  int size;

  public Ball(int newX, int newY, int newSize)
  {
     this.x = newX;
     this.y = newY;
     this.size = newSize;
  }

  public void print()
  {
      System.out.println("Ball at " + this.x + ", " + this.y +
                         " with size " + this.size);
  }
}
```

Just like in C++, we can drop `this` because it's implied

Adding Behavior in Java

```
public class Ball
{
  int x;
  int y;
  int size;

  public Ball(int newX, int newY, int newSize)
  {
     x = newX;
     y = newY;
     size = newSize;
  }

  public void print()
  {
      System.out.println("Ball at " + x + ", " + y +
                         " with size " + size);
  }
}
```

Adding Behavior in Java

```
public static void main(String[] args)
{
   Ball myNewBall = new Ball(10,15,5);
   myNewBall.print();
}
```

```
Ball at 10, 15 with size 5
```

Adding Behavior in Java

Suppose we had a function to determine what discount a Person object is eligible for…

```
public static int computeDiscount(Person  p)
{
    if ((p.gender == 'F')
         && (p.age < 13 || p.retired))
    {
        return 50;
    }
    else
    {
        return 0;
    }
}
```

This is the **procedural style** of working with object instances

Adding Behavior in Java

…we can make it a method and move it into the `Person` class.

```
public class Person
{
    // Define attributes first
    ...
    // Now define the constructors
    ...

    // Finally, write your methods here
    public int computeDiscount()
    {
        if ((this.gender == 'F') &&
            (this.age < 13 || this.retired))
        {
            return 50;
        }
        return 0;
    }
}
```

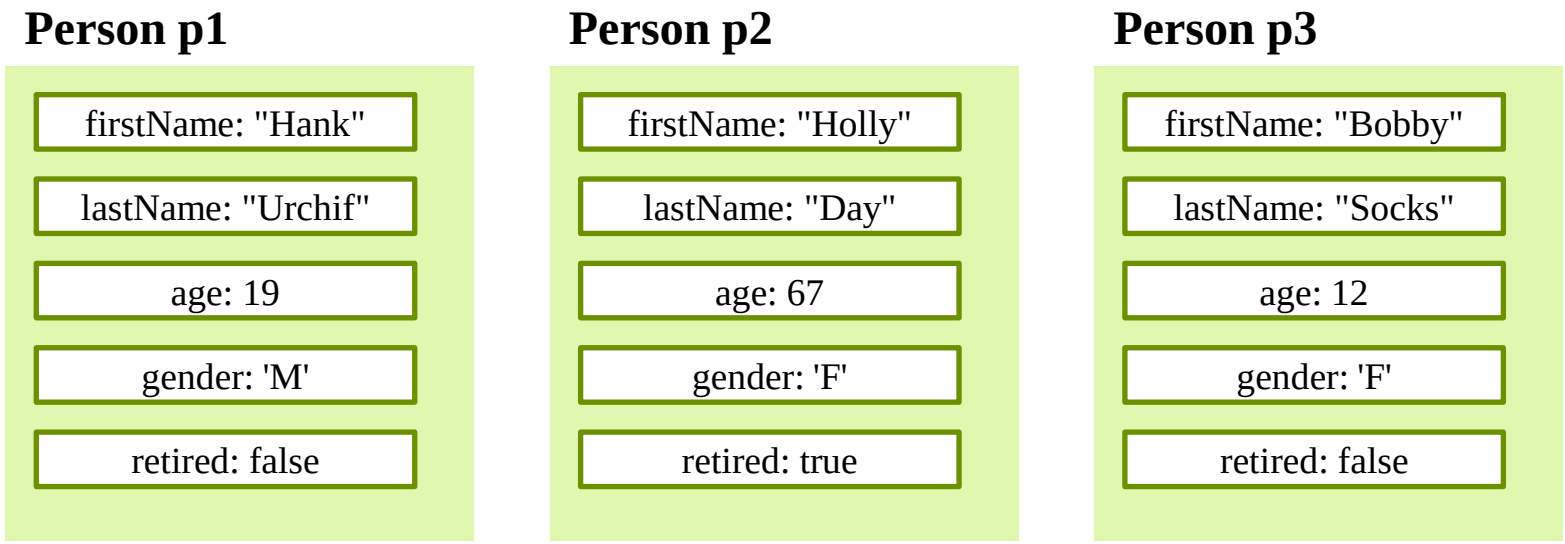
This is the **object-oriented style** of working with object instances

Adding Behavior in Java

…we can make it a method and move it into the `Person` class.

```
public class Person
{
    // Define attributes first
    ...
    // Now define the constructors
    ...

    // Finally, write your methods here
    public int computeDiscount()
    {
        if ((this.gender == 'F')
            (this.age < 13 || thi
        {
            return 50;
        }
        return 0;
    }
}
```

We no longer need a parameter, since we will have access to the object through `this`.




```
Person p1 = new Person("Hank", "Urchif", 19, 'M');
Person p2 = new Person("Holly", "Day", 67, 'F', true);
Person p3 = new Person("Bobby", "Socks", 12, 'F');

System.out.println(
   "p1's discount = " + p1.computeDiscount());

System.out.println(
   "p2's discount = " + p2.computeDiscount());

System.out.println(
   "p3's discount = " + p3.computeDiscount());
```

Person p1

- firstName: "Hank"
- lastName: "Urchif"
- age: 19
- gender: 'M'
- retired: false

Person p2

- firstName: "Holly"
- lastName: "Day"
- age: 67
- gender: 'F'
- retired: true

Person p3

- firstName: "Bobby"
- lastName: "Socks"
- age: 12
- gender: 'F'
- retired: false

```
public int computeDiscount()
{
   if ((this.gender == 'F') &&
       (this.age < 13 ||
           this.retired))
   {
      return 50;
   }
   return 0;
 }
```

```
System.out.println(
   "p1's discount = " + p1.computeDiscount());
```

Person p1

- firstName: "Hank"
- lastName: "Urchif"
- age: 19
- gender: 'M'
- retired: false

Person p2

- firstName: "Holly"
- lastName: "Day"
- age: 67
- gender: 'F'
- retired: true

Person p3

- firstName: "Bobby"
- lastName: "Socks"
- age: 12
- gender: 'F'
- retired: false

```
public int computeDiscount()
{
   if ((this.gender == 'F') &&
       (this.age < 13 ||
           this.retired))
   {
      return 50;
   }
   return 0;
 }
```

We are calling the method with `p1`...

```
System.out.println(
   "p1's discount = " + p1.computeDiscount());
```

Person p1

- firstName: "Hank"
- lastName: "Urchif"
- age: 19
- gender: 'M'
- retired: false

Person p2

- firstName: "Holly"
- lastName: "Day"
- age: 67
- gender: 'F'
- retired: true

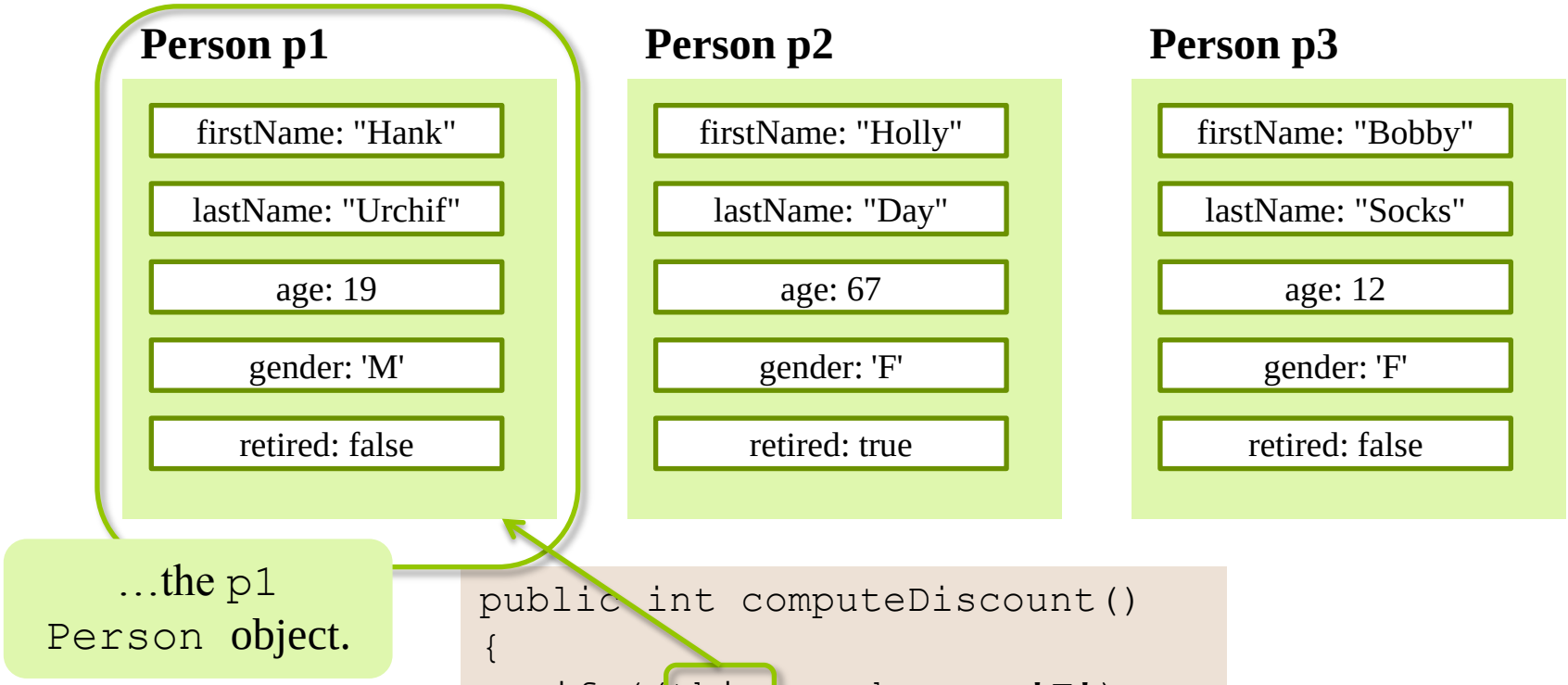
Person p3

- firstName: "Bobby"
- lastName: "Socks"
- age: 12
- gender: 'F'
- retired: false

```
public int computeDiscount()
{
   if ((this.gender == 'F') &&
       (this.age < 13 ||
                        ed))
       return 50;
   }
   return 0;
 }
```

…so `this` refers to…

```
System.out.println(
   "p1's discount = " + p1.computeDiscount());
```




```
public int computeDiscount()
{
   if ((this.gender == 'F') &&
       (this.age < 13 ||
           this.retired))
   {
      return 50;
   }
   return 0;
 }
```

```
System.out.println(
   "p1's discount = " + p1.computeDiscount());
```

Person p1

firstName: "Hank"

lastName: "Urchif"

age: 19

gender: 'M'

retired: false

Person p2

firstName: "Holly"

lastName: "Day"

age: 67

gender: 'F'

retired: true

Person p3

firstName: "Bobby"

lastName: "Socks"

age: 12

gender: 'F'

retired: false

```
public int computeDiscount()
{
   if ((gender == 'F') &&
       (age < 13 || retired))
   {
      return 50;
   }
   return 0;
 }
```

And of course it still works without `this` because it is implied.

```
System.out.println(
   "p1's discount = " + p1.computeDiscount());
```

Adding Behavior in Java

Methods can have parameters that are an object of the same class as the method belongs to.

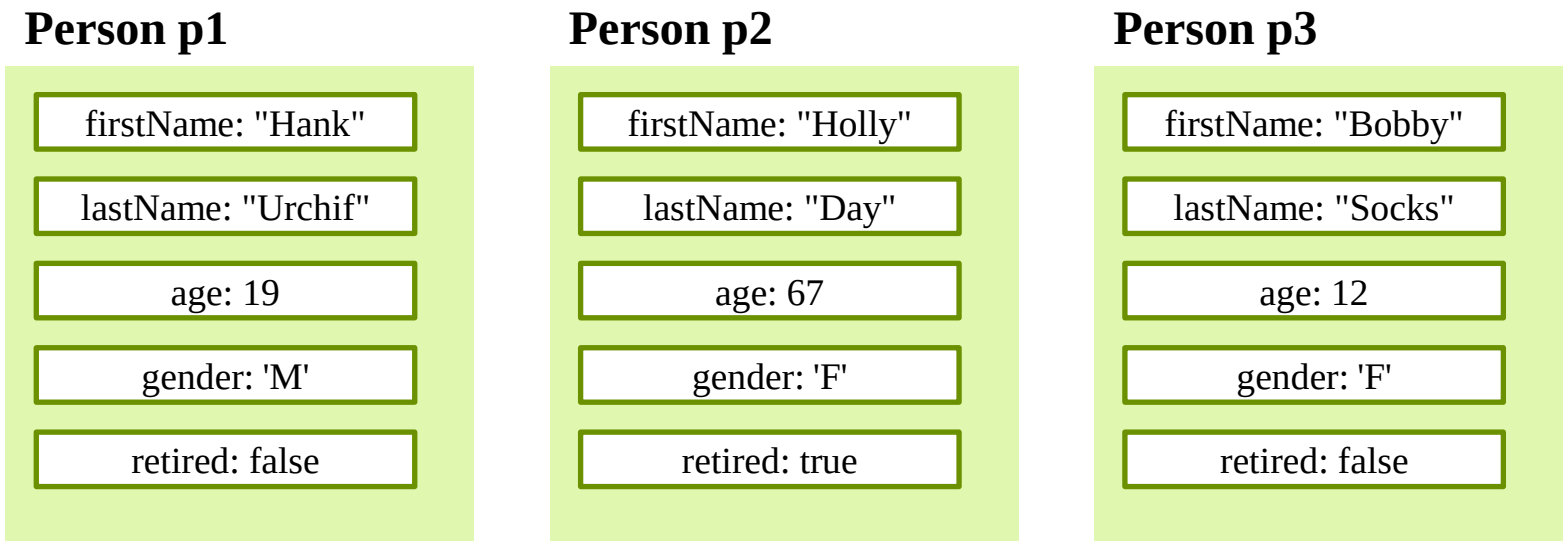
```
public class Person
{
    // Define attributes first
    ...
    // Now define the constructors
    ...

    // Finally, write your methods here
    public boolean isOlderThan(Person  x)
    {
        return (this.age > x.age);
    }
}
```

Adding Behavior in Java

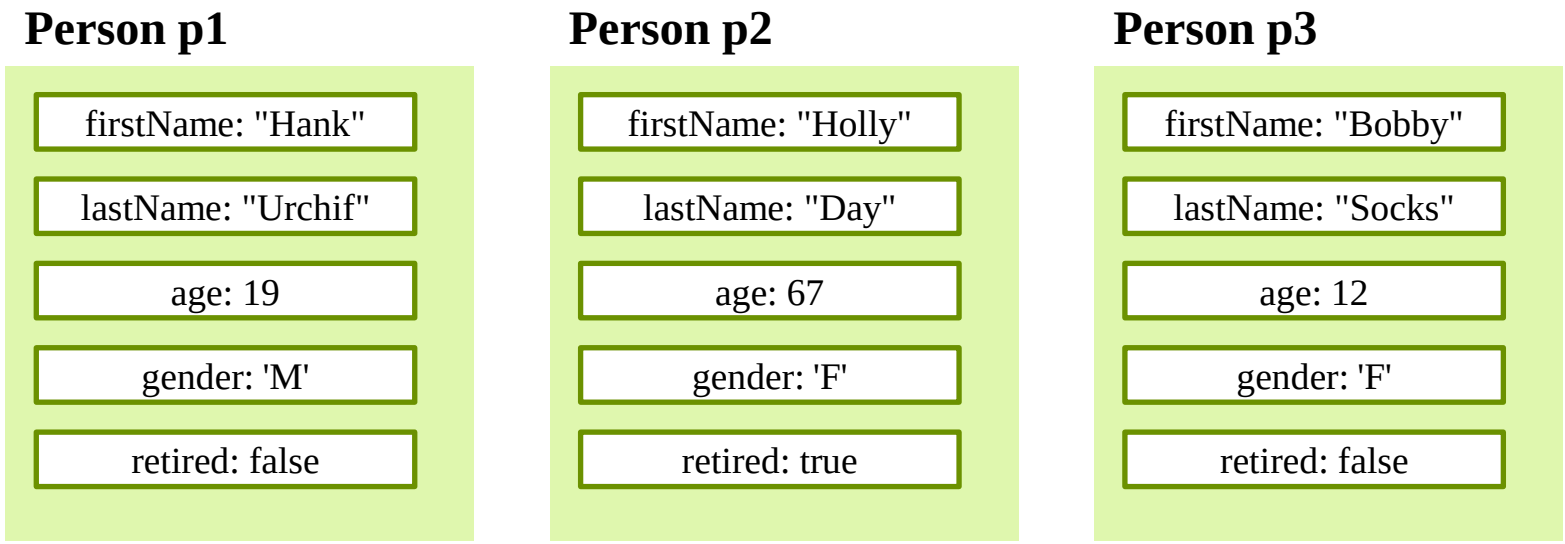
Methods can have parameters that are an object of the same class as the method belongs to.

```
public class Person
{
    // Define attributes first
    ...
    // Now define the constructors
    ...

    // Finally, write your methods here
    public boolean isOlderThan(Person  x)
    {
        return (this.age > x.age);
    }
}
```

This works!




```
Person p1 = new Person("Hank", "Urchif", 19, 'M');
Person p2 = new Person("Holly", "Day", 67, 'F', true);
Person p3 = new Person("Bobby", "Socks", 12, 'F');

if (p1.isOlderThan(p2) && p1.isOlderThan(p3))
    oldest = p1;
else if (p2.isOlderThan(p1) && p2.isOlderThan(p3))
    oldest = p2;
else
    oldest = p3;
```




```
public boolean isOlderThan(Person  x)
{
   return (this.age > x.age);
}
```

```
if (p1.isOlderThan(p2) && p1.isOlderThan(p3))
    oldest = p1;
else if (p2.isOlderThan(p1) && p2.isOlderThan(p3))
    oldest = p2;
else
    oldest = p3;
```

Person p1

firstName: "Hank"
lastName: "Urchif"
age: 19
gender: 'M'
retired: false

Person p2

firstName: "Holly"
lastName: "Day"
age: 67
gender: 'F'
retired: true

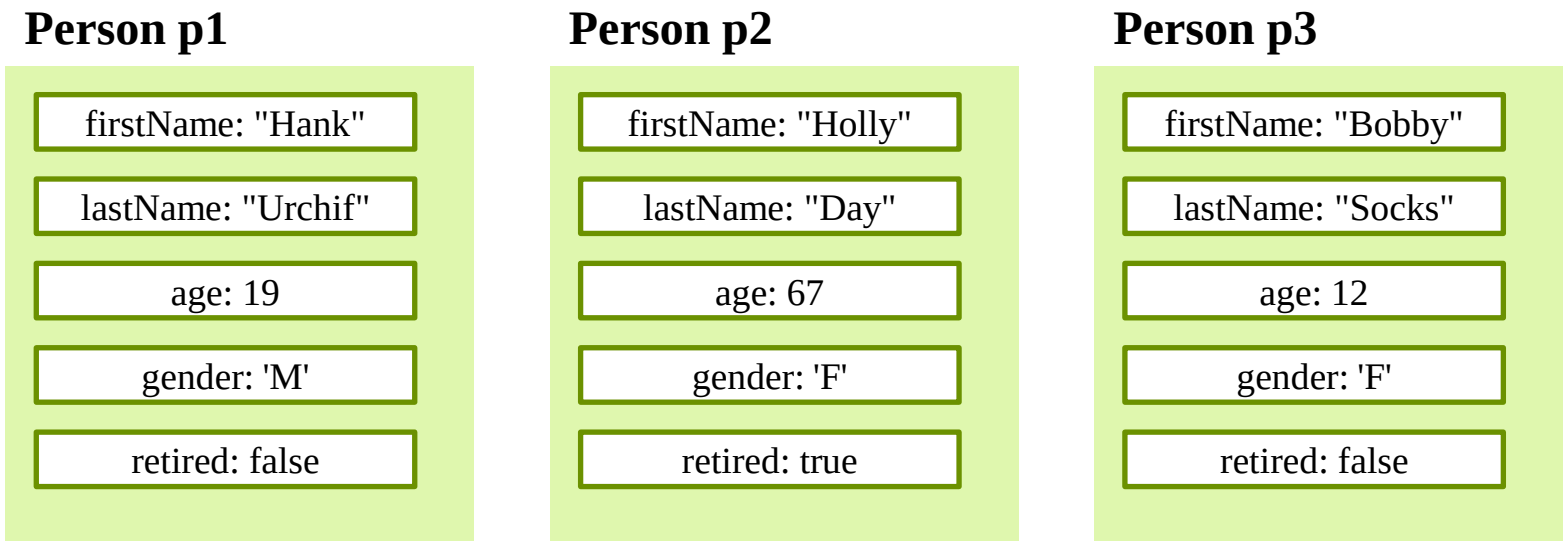
Person p3

firstName: "Bobby"
lastName: "Socks"
age: 12
gender: 'F'
retired: false

```
public boolean isOlderThan(Person  x)
{
    return (this.age > x.age);
}
```

```
if (p1.isOlderThan(p2) && p1.isOlderThan(p3))
    oldest = p1;
else if (p2.isOlderThan(p1) && p2.isOlderThan(p3))
    oldest = p2;
else
    oldest = p3;
```

The object calling the method is `p1`…




```
public boolean isOlderThan(Person  x)
{
    return (this.age > x.age);
}
```

…so `this` is referring to `p1`

```
if (p1.isOld                    OlderThan(p3))
    oldest = p1;
else if (p2.isOlderThan(p1) && p2.isOlderThan(p3))
    oldest = p2;
else
    oldest = p3;
```

Person p1

- firstName: "Hank"
- lastName: "Urchif"
- age: 19
- gender: 'M'
- retired: false

Person p2

- firstName: "Holly"
- lastName: "Day"
- age: 67
- gender: 'F'
- retired: true

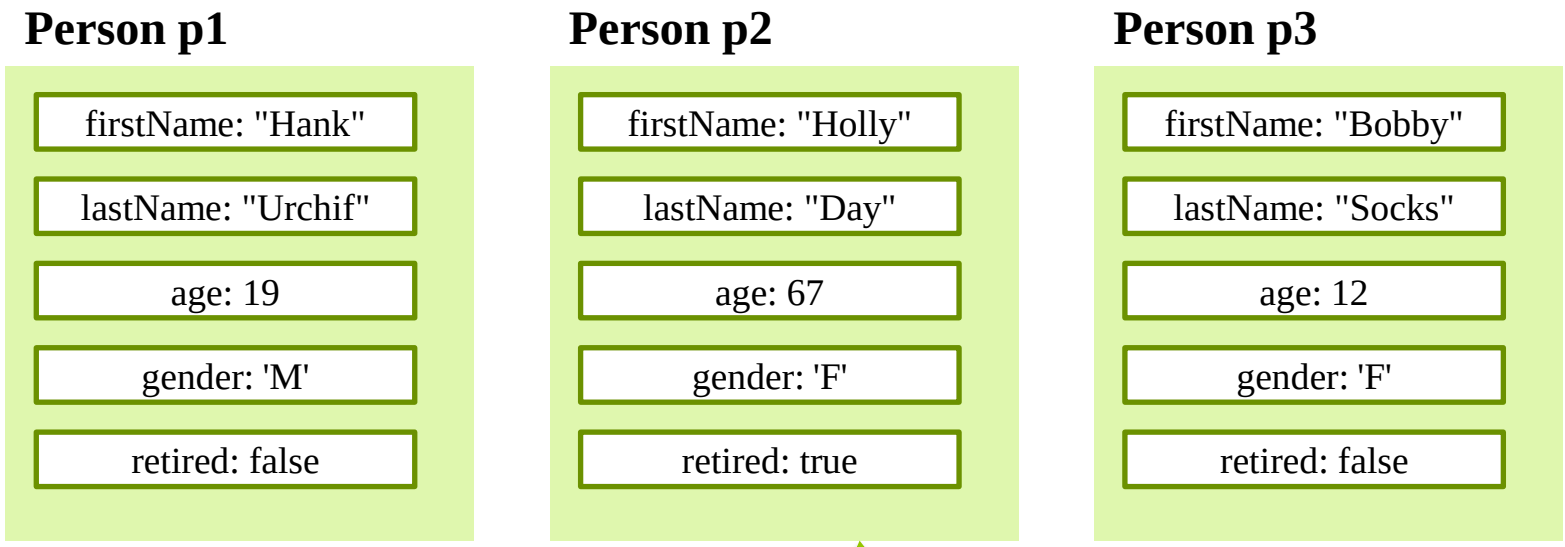
Person p3

- firstName: "Bobby"
- lastName: "Socks"
- age: 12
- gender: 'F'
- retired: false

```
public boolean isOlderThan(Person  x)
{
    return (this.age > x.age);
}
```

```
if (p1.isOlderThan(p2) && p1.isOlderThan(p3))
    oldest = p1;
else if (p2.isOlderThan(p3))
    oldest = p2;
else
    oldest = p3;
```

The parameter passed to the method is `p2`…




```
public boolean isOlderThan(Person  x)
{
    return (this.age > x.age);
}
```

…so x is referring to p2

```
if (p1.isOlderThan(p2)                          p3))
    oldest = p1;
else if (p2.isOlderThan(p1) && p2.isOlderThan(p3))
    oldest = p2;
else
    oldest = p3;
```

Adding Behavior in Java

Some methods modify the object.

```
public class Person
{
    // Define attributes first
    ...
    // Now define the constructors
    ...

    // Finally, write your methods here
    public void retire()
    {
        retired = true;
    }
}
```

Adding Behavior in Java

Some methods modify the object.

```
public class Person
{
    // Define attributes first
    ...
    // Now define the constructors
    ...

    // Finally, write your methods here
    public void retire()
    {
        retired = true;
    }
}
```

Modifies the calling object's `retired` attribute

Adding Behavior in Java

Some methods modify both the object and the parameter.

```
public void swapNameWith(Person x)
{
    String tempName;

    // Swap the first names
    tempName = firstName;
    firstName = x.firstName;
    x.firstName = tempName;

    // Swap the last names
    tempName = lastName;
    lastName = x.lastName;
    x.lastName = tempName;
}
```

Adding Behavior in Java

Some methods modify both the object and the parameter.

Modifies the calling object's `firstName` attribute

```
public void swapNameWith(Person x)
{
    String tempName;

    // Swap the first names
    tempName = firstName;
    firstName = x.firstName;
    x.firstName = tempName;

    // Swap the last names
    tempName = lastName;
    lastName = x.lastName;
    x.lastName = tempName;
}
```

Adding Behavior in Java

Some methods modify both the object and the parameter.

```
public void swapNameWith(Person x)
{
    String tempName;

    // Swap the first names
    tempName = firstName;
    firstName = x.firstName;
    x.firstName = tempName;

    // Swap the last names
    tempName = lastName;
    lastName = x.lastName;
    x.lastName = tempName;
}
```

Modifies the x's `firstName` attribute

Adding Behavior in C++

Everything described for Java applies to C++ as well *except* that not all objects are passed by reference.

```
void swapNameWith(person x)
{
    string tempName;

    // Swap the first names
    tempName = firstName;
    firstName = x.firstName;
    x.firstName = tempName;

    // Swap the last names
    tempName = lastName;
    lastName = x.lastName;
    x.lastName = tempName;
}
```

Adding Behavior in C++

Everything described for Java applies to C++ as well *except* that not all objects are passed by reference.

```
void swapNameWith(person x)
{
    string tempName;

    // Swap the first names
    tempName = firstName;
    firstName = x.firstName;
    x.firstName = tempName;

    // Swap the last names
    tempName = lastName;
    lastName = x.lastName;
    x.lastName = tempName;
}
```

This member function won't work as expected unless `x` is made a reference!

Overloading

Overloading

Allows us to define multiple methods of the same name, but with different parameter types.

Don't confuse this with overriding, which we'll learn about when studying advanced OOP.

Overloading in Java

Allowed:

```
public void eat(Apple x) { … }
public void eat(Orange x) { … }
public void eat(Banana x, Banana y) { … }
```

Not allowed:

```
public double calculatePayment(BankAccount account){ … }
public double calculatePayment(BankAccount x){ … }
```

Overloading in Java

Allowed:

```
public int computeHealthRisk(int age, int weight, boolean smoker) { ... }
public int computeHealthRisk(boolean smoker, int age, int weight) { ... }
public int computeHealthRisk(int weight, boolean smoker, int age) { ... }
```

Not allowed:

```
public int computeHealthRisk(int age, int weight, boolean smoker) { ... }
public int computeHealthRisk(int weight, int age, boolean smoker) { ... }
```

Overloading in C++

Same rules, plus this is not allowed:

```
double calculatePayment(bankAccount account){ … }
double calculatePayment(bankAccount &account){ … }
```

Static Members

Static Members

When an attribute or member function/method is static, it belongs to the class instead of each individual object. There will be exactly one copy.

Static Members in C++

```
class ball
{
 public:
    int x;
    int y;
    int size;
    int id;

    static int lastID;

    ball(int newX, int newY, int newSize)
    {
        x = newX;
        y = newY;
        size = newSize;
        id = lastID++;
    }
};
```

Static Members in C++

```
class ball
{
 public:
    int x;
    int y;
    int size;
    int id;

    static int lastID;

    ball(int newX, int newY, int newSize)
    {
        x = newX;
        y = newY;
        size = newSize;
        id = lastID++;
    }
};
```

There will be exactly one copy of this attribute shared by all `ball` objects

Static Members in C++

```
class ball
{
 public:
    int x;
    int y;
    int size;
    int id;

    static int lastID;

    ball(int newX, int newY,
         int newSize)
    {
        x = newX;
        y = newY;
        size = newSize;
        id = lastID++;
    }
};
```

Member Functions

```
ball(int newX, int newY,
     int newSize)
{ … }
```

Static Attributes

lastID:

Static Members in C++

```
class ball
{
 public:
    int x;
    int y;
    int size;
    int id;

    static int lastID;

    ball(int newX, int newY, int newSize)
    {
        x = newX;
        y = newY;
        size = newSize;
        id = lastID++;
    }
};
```

Static Members in C++

```
class ball
{
 public:
    int x;
    int y;
    int size;
    int id;

    static int lastID;

    ball(int newX, int newY, int newSize)
    {
        x = newX;
        y = newY;
        size = newSize;
        id = lastID++;
    }
};
```

If the value is constant, it can be initialized here…

Static Members in C++

```
class ball
{
 public:
    int x;
    int y;
    int size;
    int id;

    static int lastID;

    ball(int newX, int newY, int newSize)
    {
        x = newX;
        y = newY;
        size = newSize;
        id = lastID++;
    }
};
```

```
int ball::lastID = 0;
```

…otherwise, we need to add an initializer outside the class definition.

Static Members in C++

Member Functions

```
ball(int newX, int newY,
     int newSize)
{ … }
```

Static Attributes

```
int ball::lastID = 0;
```


lastID: 0

Static Members in C++

```
class ball
{
 public:
    int x;
    int y;
    int size;
    int id;

    static int lastID;

    ball(int newX, int newY, int newSize)
    {
        x = newX;
        y = newY;
        size = newSize;
        id = lastID++;
    }
};

int ball::lastID = 0;
```

`lastID` will have been initialized by the time it is used here, so the first `id` of a ball will be 1.

Static Members in C++

```
class ball
{
 public:
    int x;
    int y;
    int size;
    int id;

    static int lastID;

    ball(int newX, int newY, int newSize)
    {
        x = newX;
        y = newY;
        size = newSize;
        id = lastID++;
    }
};

int ball::lastID = 0;
```

Because there is only one copy for all `balls`, the next `id` will end up being 2, and so on.

Member Functions

```
ball(int newX, int newY,
     int newSize)
{ … }
```

Static Attributes

lastID: 0

```
ball b(10,15,5);
```

ball b

x: ?

y: ?

size: ?

id: ?

Member Functions

```
ball(int newX, int newY,
     int newSize)
{ … }
```

Static Attributes

lastID: 0

```
ball b(10,15,5);
```

Makes enough space for the attributes in memory…

ball b

x: ?

y: ?

size: ?

id: ?

Static Attributes

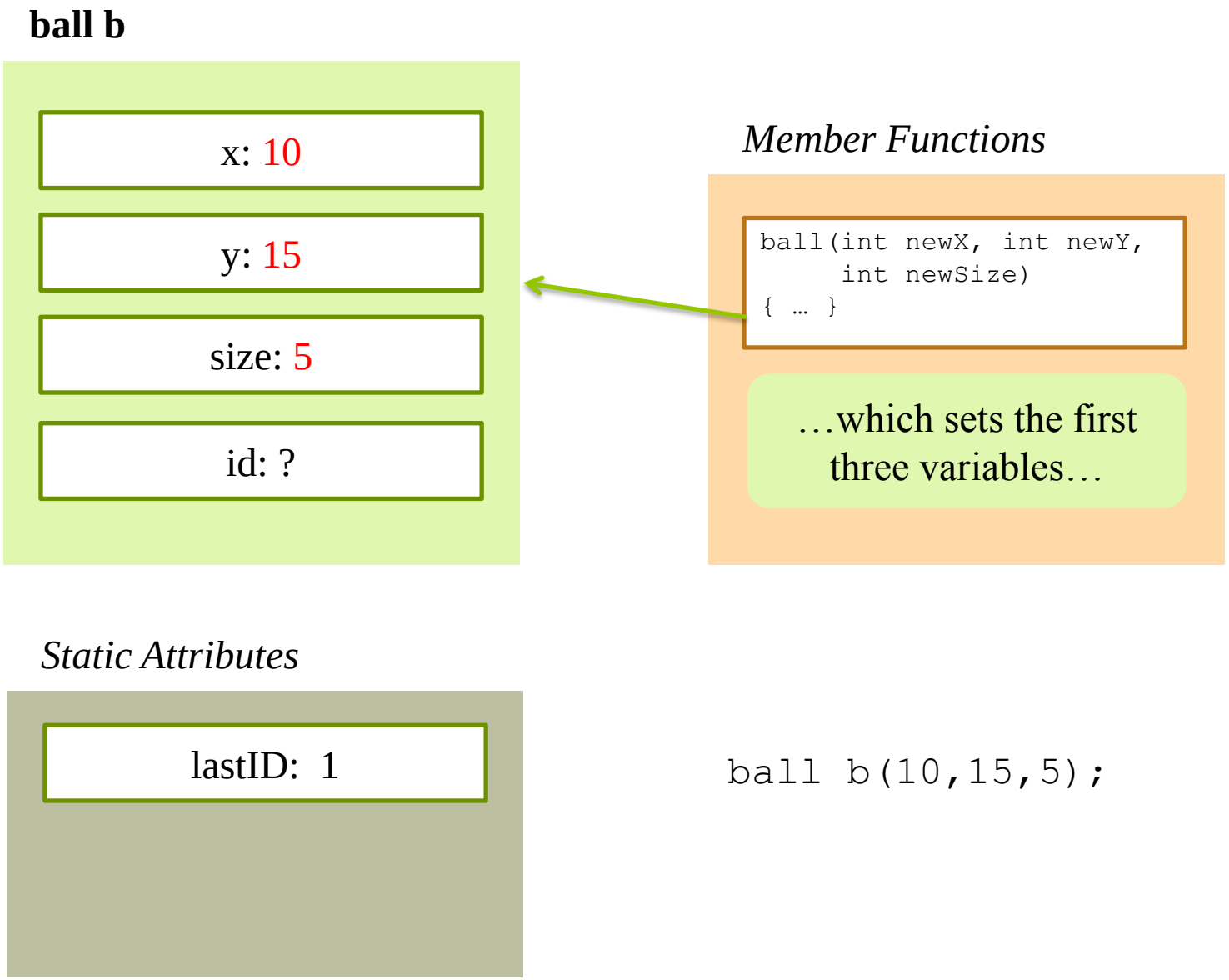
lastID: 0

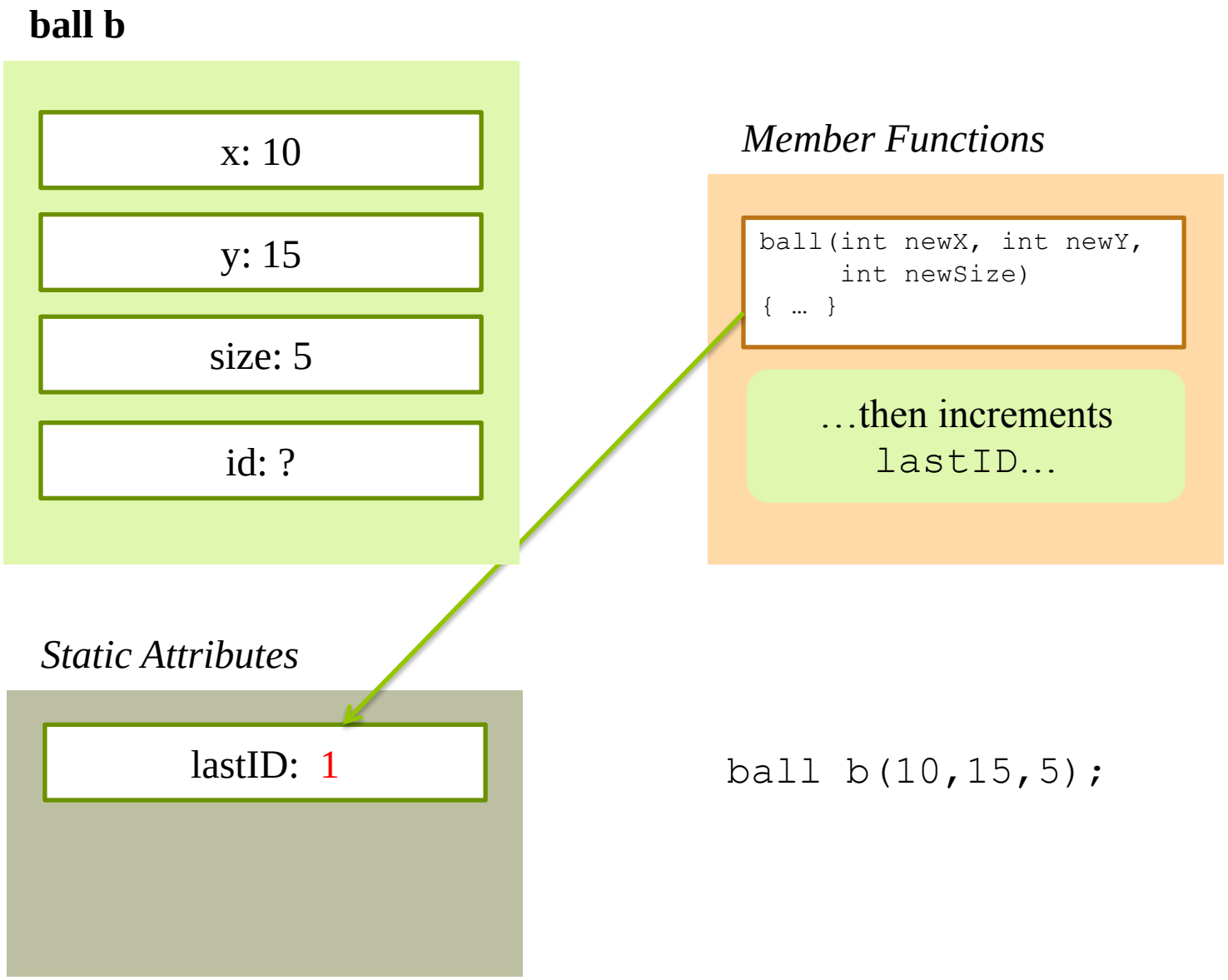
Member Functions

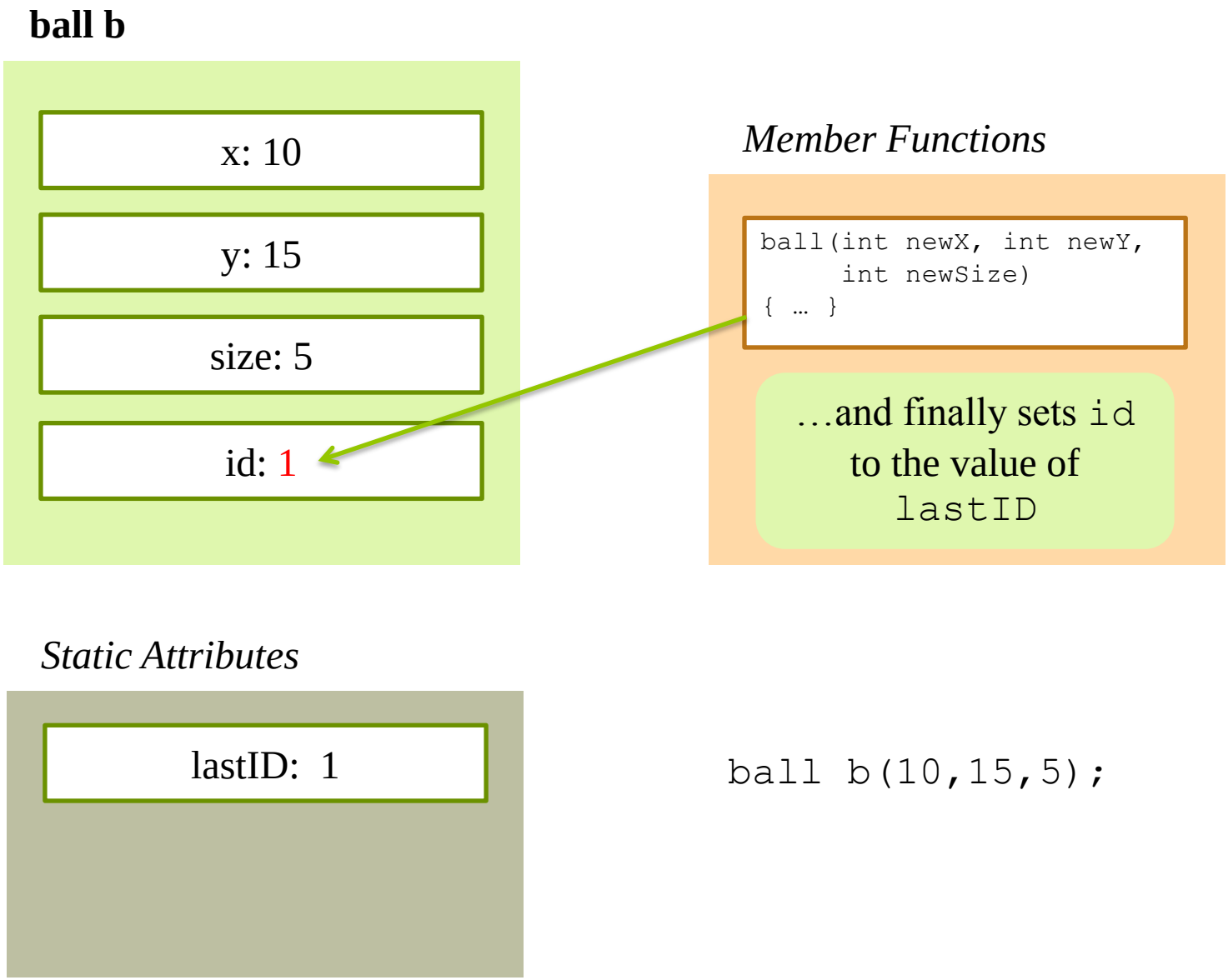
```
ball(int newX, int newY,
     int newSize)
{ … }
```

```
ball b(10,15,5);
```

…then calls the appropriate `ball` constructor…


ball b
x: 10
y: 15
size: 5
id: ?
Member Functions
ball(int newX, int newY,
int newSize)
{ … }
…which sets the first three variables…
Static Attributes
lastID: 1
ball b(10,15,5);


ball b
x: 10
y: 15
size: 5
id: ?
Member Functions
ball(int newX, int newY,
int newSize)
{ … }
…then increments
lastID…
Static Attributes
lastID: 1
ball b(10,15,5);


ball b
x: 10
y: 15
size: 5
id: 1
Member Functions
ball(int newX, int newY,
int newSize)
{ … }
…and finally sets id
to the value of
lastID
Static Attributes
lastID: 1
ball b(10,15,5);

```
class ball
{
 public:
    int x, y, size;
    int id;

    static int lastID;

    ball(int newX, int newY, int newSize)
    {
        x = newX;
        y = newY;
        size = newSize;
        id = generateID();
    }

    static int generateID()
    {
        return lastID++;
    }
};
```

```
class ball
{
 public:
    int x, y, size;
    int id;

    static int lastID;

    ball(int newX, int newY, int newSize)
    {
        x = newX;
        y = newY;
        size = newSize;
        id = generateID();
    }

    static int generateID()
    {
        return lastID++;
    }
};
```

A static member function no longer has access to `this` (and therefore cannot access member attributes)

```
class ball
{
 public:
    int x, y, size;
    int id;

    static int lastID;

    ball(int newX, int newY, int newSize)
    {
        x = newX;
        y = newY;
        size = newSize;
        id = generateID();
    }

    static int generateID()
    {
        return lastID++;
    }
};
```

But even without `this`, the function can access the static attribute `lastID`.

```
class ball
{
 public:
    int x, y, size;
    int id;

    static int lastID;

    ball(int newX, int newY, int newSize)
    {
        x = newX;
        y = newY;
        size = newSize;
        id = generateID();
    }

    static int generateID()
    {
        return lastID++;
    }
};

int ball::lastID = 0;
```

Member Functions

```
ball(int newX, int newY,
     int newSize)
{ … }
```

Static Attributes

lastID: 0

Static Functions

```
int generateID()
{ … }
```

Static Members in Java

```
public class Ball
{
    int x, y, size;
    int id;
    static int nextID = 0;

    public Ball(int newX, int newY, int newSize)
    {
        x = newX;
        y = newY;
        size = newSize;
        id = generateNextID();
    }

    public static int generateNextID()
    {
        return nextID++;
    }
}
```

Static Members in Java

```
public class Ball
{
    int x, y, size;
    int id;
    static int nextID = 0;

    public Ball(int newX, int newY, int newSize)
    {
        x = newX;
        y = newY;
        size = newSize;
        id = generateNextID();
    }

    public static int generateNextID()
    {
        return nextID++;
    }
}
```

We can always initialize a static attribute right away in Java

Static Members in Java

```
public class Ball
{
    int x, y, size;
    int id;
    static int nextID = 0;

    public Ball(int newX, int newY, int newSize)
    {
        x = newX;
        y = newY;
        size = newSize;
        id = generateNextID();
    }

    public static int generateNextID()
    {
        return nextID++;
    }
}
```

Static methods work the same way as static member functions in C++

Static Members in Java

Why did we use static when making C++-like functions in our testing classes early on?