COMP 1406: Java Basics

Java Basics

Arrays in Java

Java Memory Model

Comparison to C++

Java Basics

Variables

```
class TestProgram
{
    public static void main(String[] args)
    {
        int number1;
        float number2 = 1.4f;

        number1 = 10;

        System.out.println(number1 + ", " + number2);
    }
}
```

Data Types

Data Type	Info About Type
`byte`	8-bit whole number, -128 to 127
`short`	16-bit whole number, -32,768 to 32,767
`int`	32-bit whole number, -2^{31} to $2^{31}-1$
`long`	64-bit whole number, -2^{63} to $2^{63}-1$
`float`	floating point numbers with a fractional part
`double`	higher precision floating point numbers with a fractional part
`boolean`	true or false
`String`	object representing a string

Conditionals

```
class TestProgram
{
    public static void main(String[] args)
    {
        if (5 < 10)
        {
            System.out.println("Option 1");
        }
        else if (3 < 4)
        {
            System.out.println("Option 2");
        }
        else
        {
            System.out.println("Option 3");
        }
    }
}
```

Switch

```
class TestProgram
{
    public static void main(String[] args)
    {
        int x = 2;
        switch (x)
        {
            case 1:
                System.out.println("Option 1");
                break;
            case 2:
                System.out.println("Option 2");
                break;
            default:
                System.out.println("Default");
                break;
        }
    }
}
```

While Loops

```
class TestProgram
{
    public static void main(String[] args)
    {
        int x = 0;
        while (x < 10)
        {
            x++;
            System.out.println(x);
        }
    }
}
```

For Loops

```
class TestProgram
{
    public static void main(String[] args)
    {
        for (int x=0; x < 10; x++)
        {
            System.out.println(x);
        }
    }
}
```

Functions

```
class TestProgram
{
    static void printNumbers()
    {
        System.out.println("1, 2, 3, 4, 5, 6");
    }

    public static void main(String[] args)
    {
        printNumbers();
    }
}
```

Functions

```
class TestProgram
{
    static void printNumbers()
    {
        Console.WriteLine("1, 2, 3, 4, 5, 6");
    }
    public static void Main(String[] args)
    {
        printNumbers();
    }
}
```

If we want a function to act like those we have been writing in C++, we need to make them static (we'll learn the meaning later).

Functions

```
class TestProgram
{
    static void printNumbers()
    {
        System.out.println("1, 2, 3, 4, 5, 6");
    }

    public static void main(String[] args)
    {
        printNumbers();
    }
}
```

Note: You may see the word "method" used for all functions in Java – this is a special term for functions inside a class.

Functions

```
class TestProgram
{
    static boolean flipFlag(boolean flag)
    {
        return !flag;
    }

    public static void main(String[] args)
    {
        boolean flag = false;

        System.out.println(flag);
        flag = flipFlag(flag);
        System.out.println(flag);

    }
}
```

You can make a black box function the same was as in C++.

Pass by Reference?

```
class TestProgram
{
    static void add1(int num)
    {
        num++;
    }

    public static void main(String[] args)
    {
        int myNumber = 10;
        add1(myNumber);
        System.out.println(myNumber);
    }
}
```

What about references? Will this `add1()` work?

Pass by Reference?

Parameters in Java are automatically **pass-by-value** when they are **primitives** (`int`, `boolean`, etc).

Parameters are automatically **pass-by-reference** when they are **objects** (`String`, `Ball`, etc).

Java and C++ look very similar so far…

What are the differences?

Arrays in Java

Java Arrays Are Objects!

Declaring an array:

```
int[] numbers;
```

Java Arrays Are Objects!

Declaring an array:

```
int[] numbers;
```

Just like other objects, this does not make space for any `int`s in memory yet.

Java Arrays Are Objects!

Creating an array and assigning it to a variable:

```
int[] numbers = new int[3];
```

Java Arrays Are Objects!

Creating an array and assigning it to a variable:

```
int[] numbers = new int[3];
```

This is the code that actually makes space in memory.

Java Arrays Are Objects!

Assigning values to the array slots:

```
numbers[0] = 1;
numbers[1] = 3;
numbers[2] = 5;
```

Java Arrays Are Objects!

Accessing variables within the array object:

```
System.out.println(numbers.length);
```

Java Arrays Are Objects!

Accessing variables within the array object:

```
System.out.println(numbers.length);
```

In our example, this will be 3

Java Arrays Are Objects!

Java arrays will tell you if you go out of bounds:

```
int[] numbers = new int[3];
numbers[4] = -1;
```

Java Arrays Are Objects!

Java arrays will tell you if you go out of bounds:

```
int[] numbers = new int[3];
numbers[4] = -1;
```

Results in an error:
java.lang.ArrayIndexOutOfBoundsException

Poll Everywhere Question

What will the following code output, assuming the Java classes are in their own file?

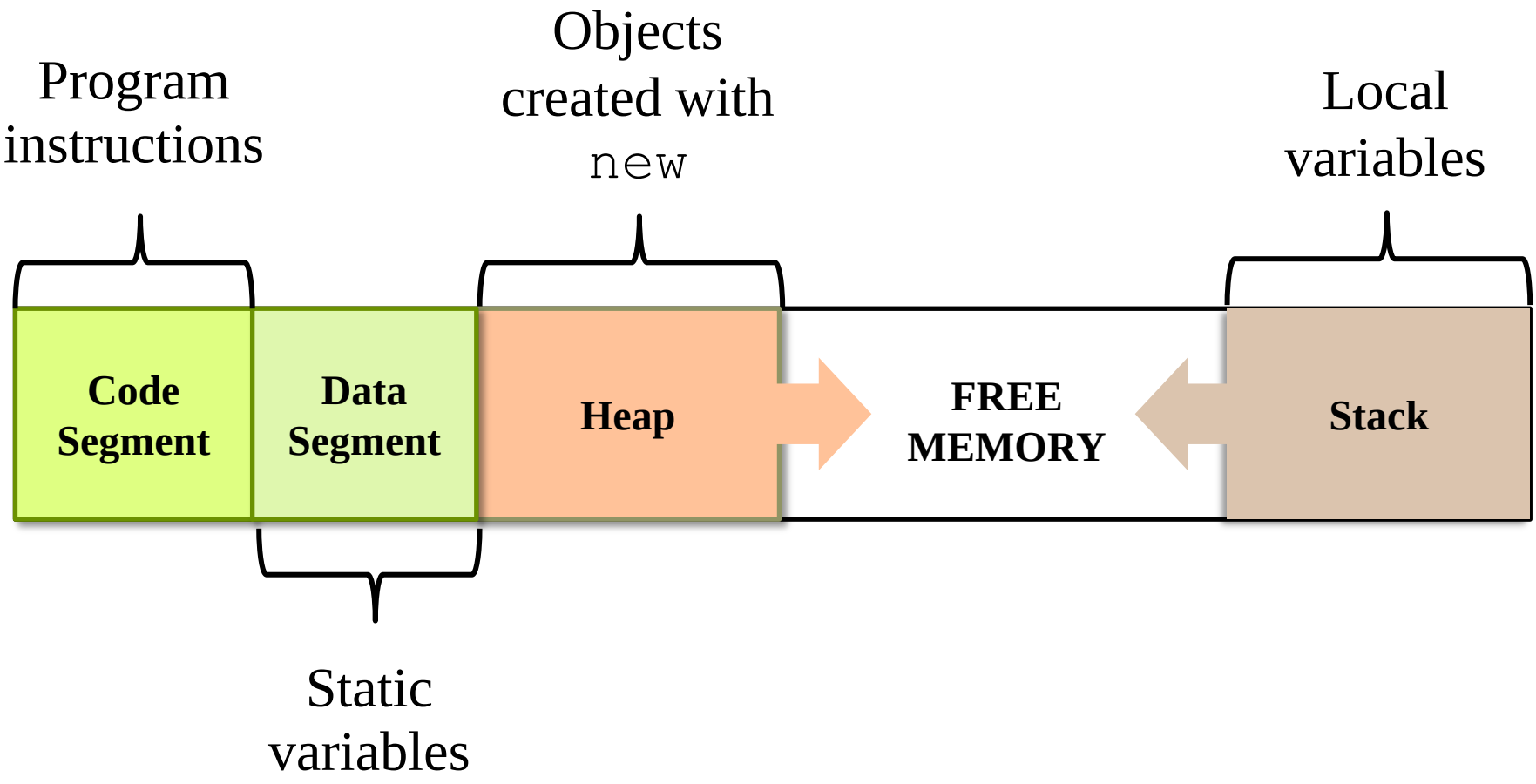
```
public class Ball
{
    int x;
    int y;
}

public class BallTester
{
    public static void main(String[] args)
    {
        Ball[] ballArray = new Ball[3];
        ballArray[0].x = 5;
        ballArray[0].y = 10;
        System.out.println(ballArray[0]
            + " " + ballArray[0]);
    }
}
```

Text 37607

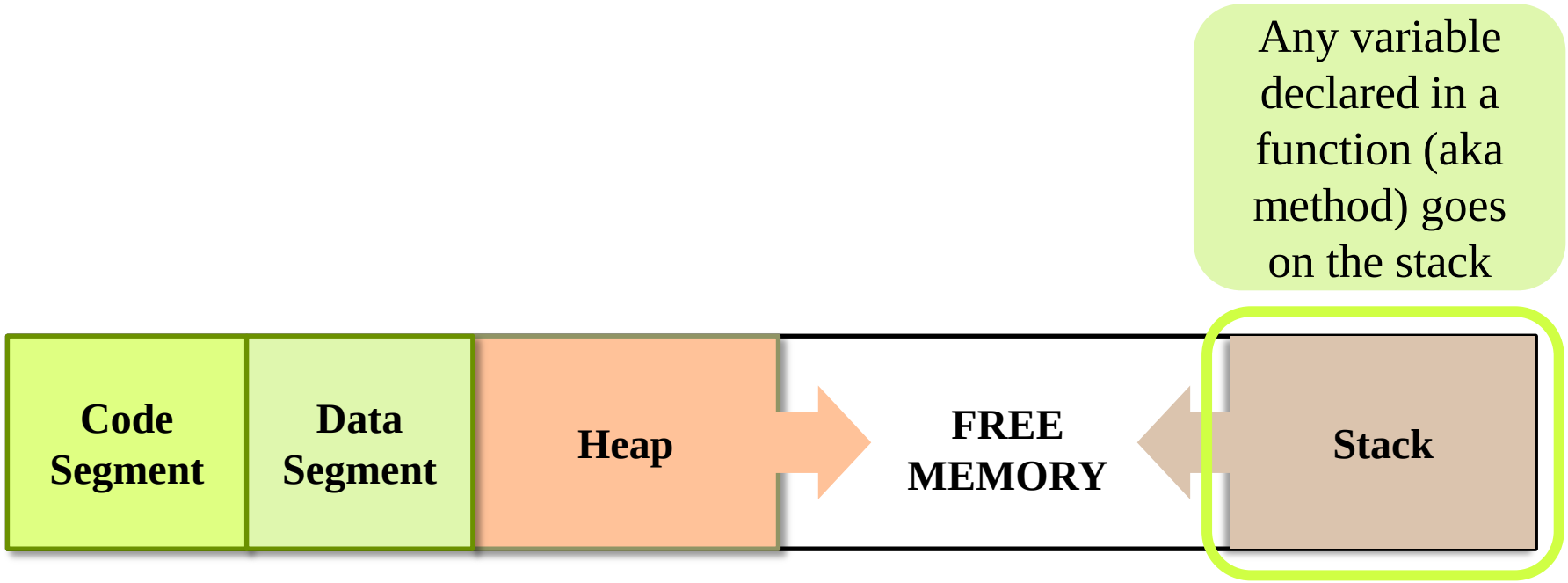
145574: 5, 10
145628: it won't compile
146157: there is a runtime error

Java Memory Model

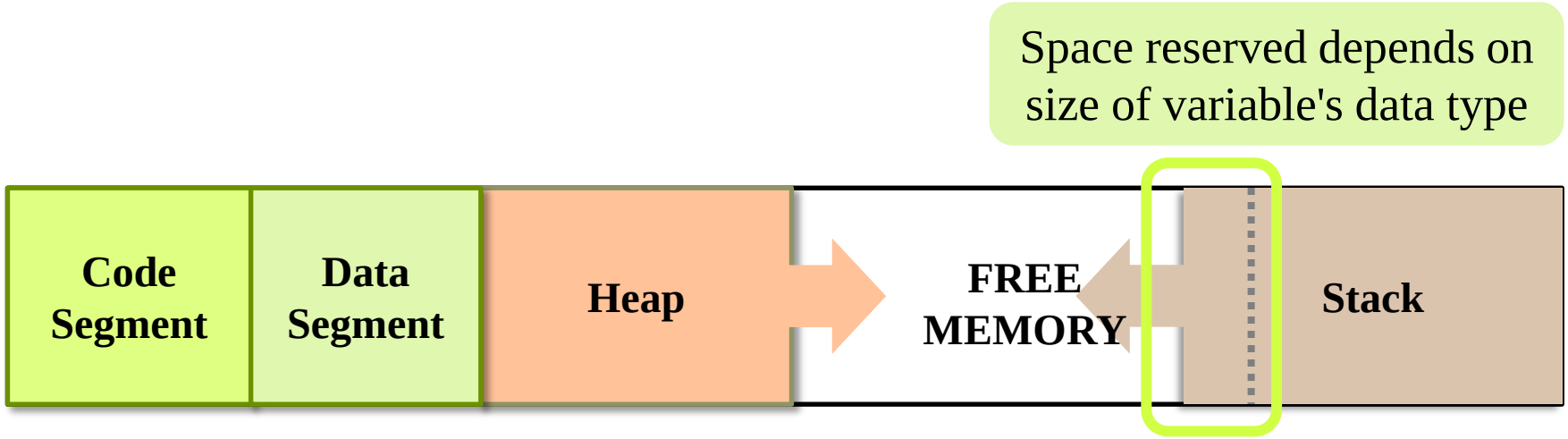
Java Memory Model



Primitives in Memory



Primitives in Memory

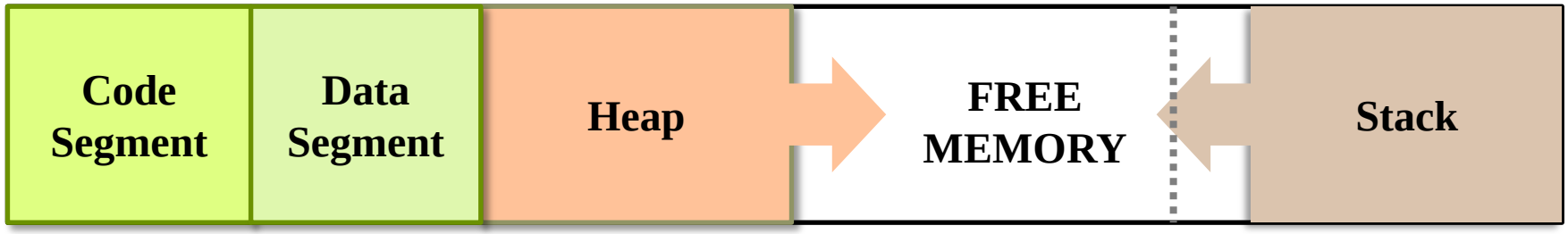



```
int newNumber;
```

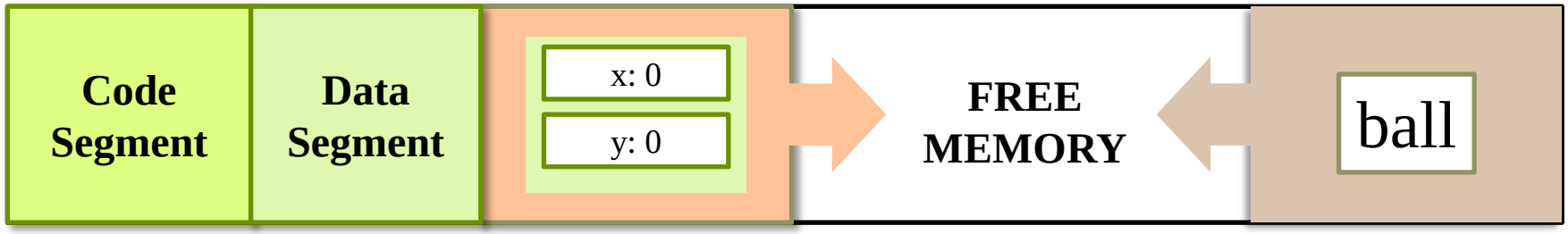
Data Type	Bytes Used	Values
byte	1	-128 to +127
double	8	-10^{308} to $+10^{308}$
float	4	-10^{38} to $+10^{38}$
short	2	-32,768 to +32,767
int	4	-2,147,483,648 to +2,147,483,647
long	8	-9,223,372,036,854,775,808 to +9,223,372,036,854,775,807
char	2	ASCII or Unicode character
boolean	1	true/false

Primitives in Memory

Space is freed up again when a variable goes out of scope



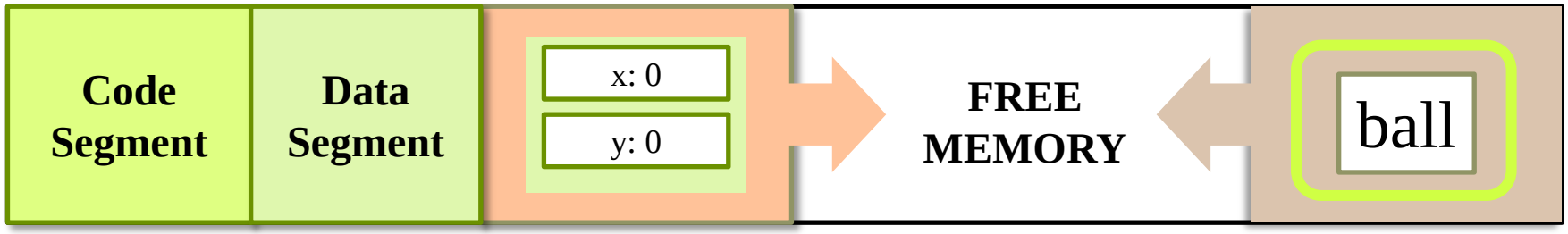
Objects in Memory




```
Ball ball = new Ball();
```

Objects in Memory

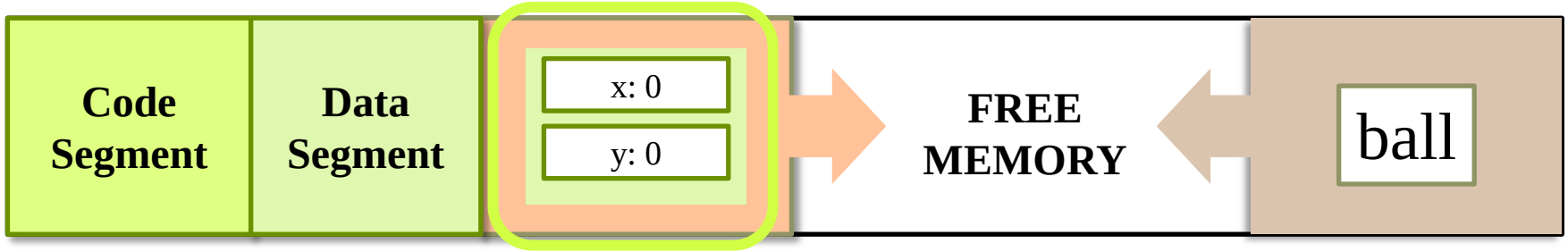
The variable `ball` goes on the stack, but is just a reference to the actual object




```
Ball ball = new Ball();
```

Objects in Memory

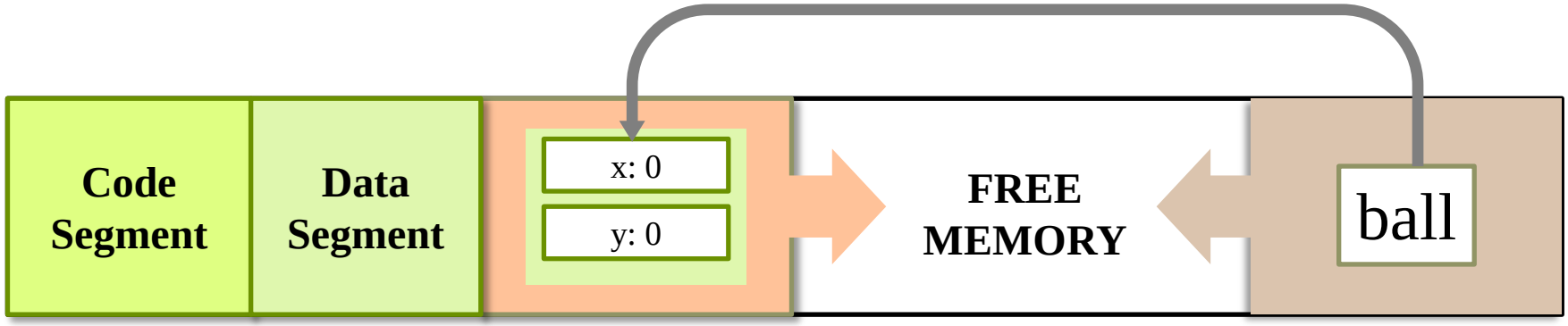
The `new` keyword creates a new object instance, putting its variables on the heap




```
Ball ball = new Ball();
```

Objects in Memory

We save a reference from the new `Ball` on the heap to the `ball` variable, essentially setting up a pointer




```
Ball ball = new Ball();
```

Poll Everywhere Question

Which of the following statements is true?

Text 37607

160526: In Java, a reference to an object can never appear on the heap.

160725: Java does not make use of pointers in any way.

160728: You can pass by reference in Java in some cases.

Comparison to C++

In **C++**, you get to choose whether you declare objects statically (on the stack) or dynamically (on the heap).

In **Java**, objects are always created dynamically. Only references to objects can be on the stack.

In **C++**, you can choose to pass objects by reference, with a pointer, or by value. You must make your choice explicit in the code.

In **Java**, you only get to use pass by reference, and this fact is not made explicit in the code.

In **C++**, you must manually delete dynamically allocated memory yourself.

In **Java**, the garbage collector determines whether an object is being referred to anymore, and if not, the object is deleted automatically.

In **C++**, you have a great deal of control over your memory and how you access it.

In **Java**, everything is hidden and taken care of for you.