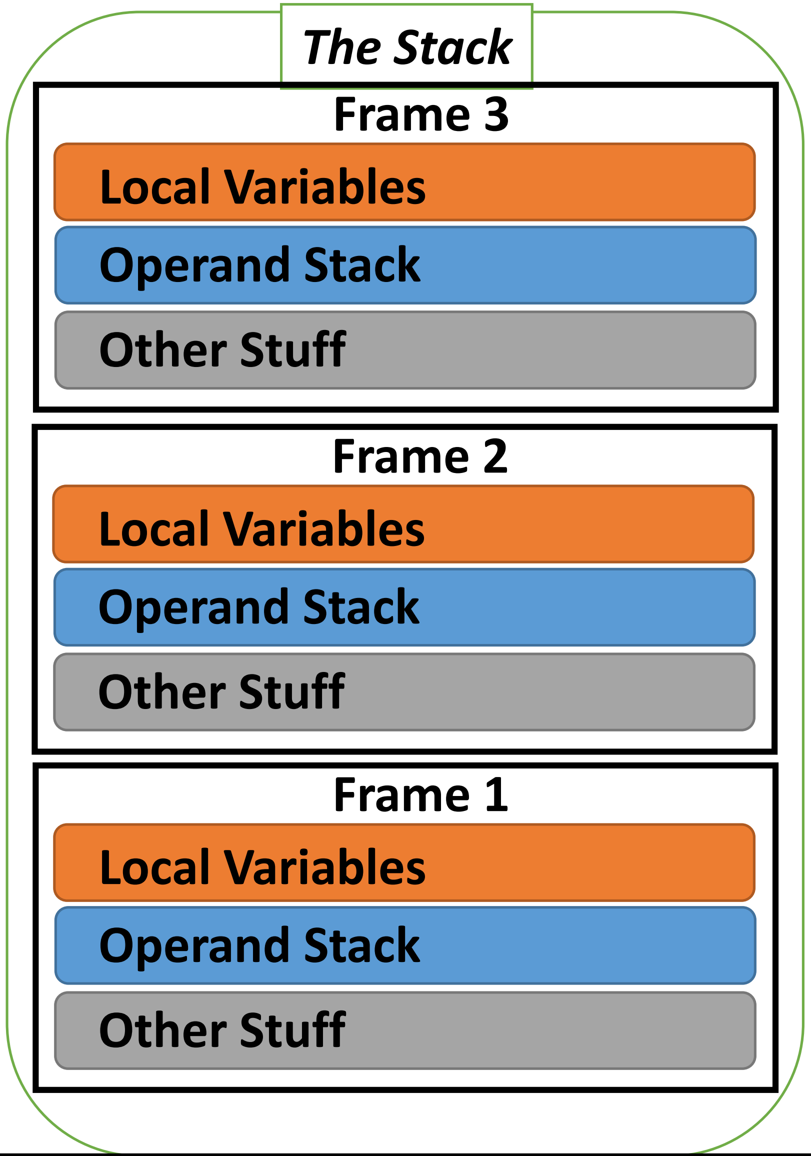
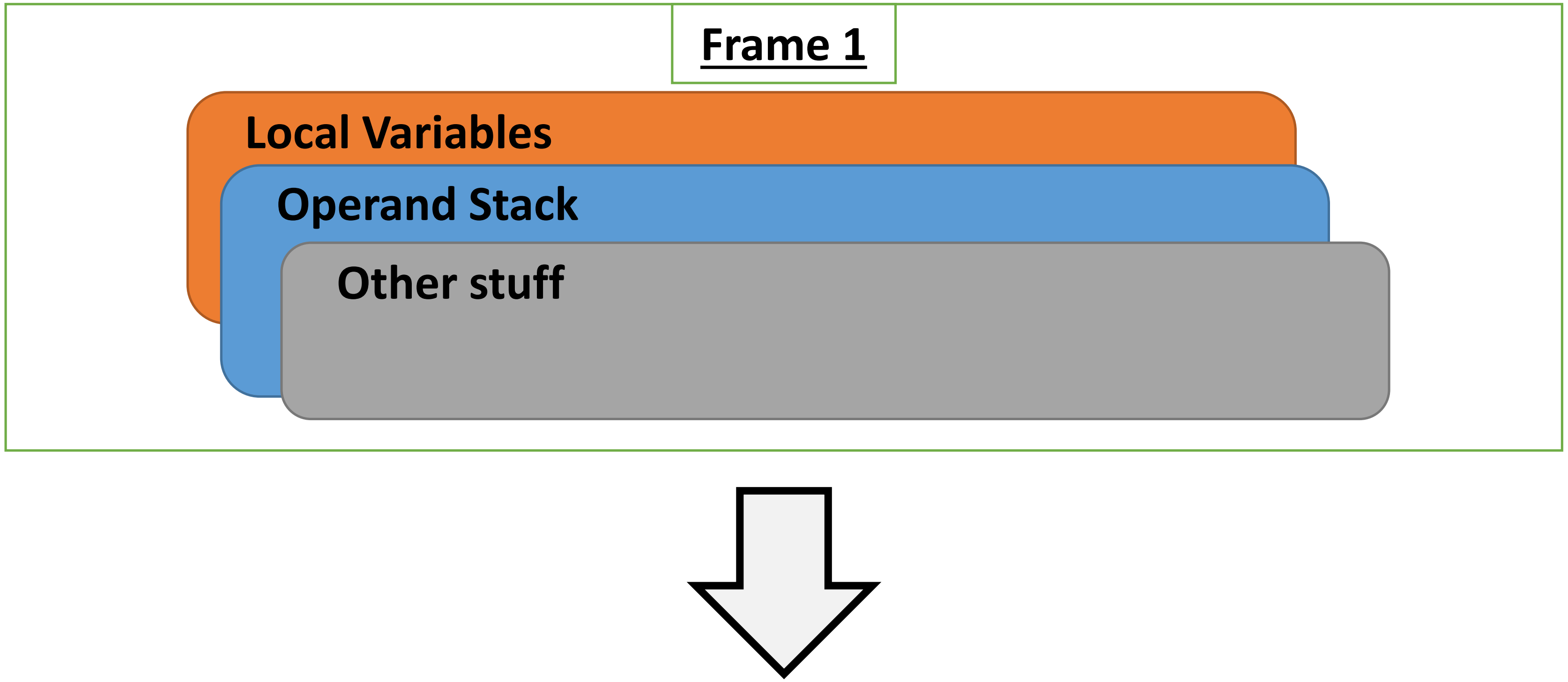
Computer Science II
Handout 4

Understanding Objects: memory in Java

- Recall the *frame* which stores local variables, etc.
  - One for each method that executes
  - This is an abstraction of how Java handles memory
- Frames are stored together, and Java organizes them in a *stack*
  - The collection of frames is often called *<u>the</u> stack*
- Much of the data we use simply lives on *the stack* in memory
  - Such as primitive data type variables defined within a method



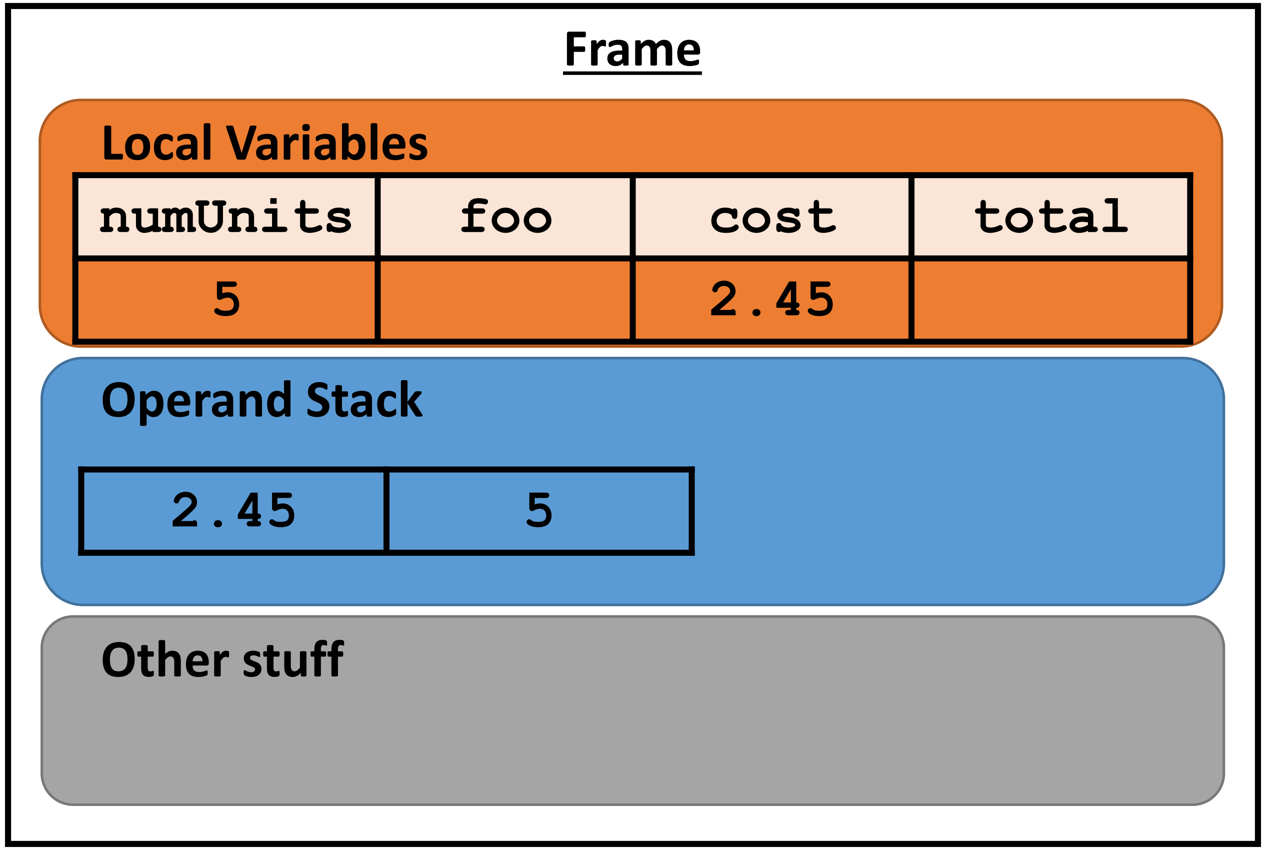
Understanding Objects: memory in Java



1	2	3	4	5	6	7	....
10023	10024	10025	10026	10027	10028	10029	....

Understanding Objects: memory in Java

```
int numUnits = 5;
int foo;
double cost = 2.45;
double total = cost * numUnits;
```



Frame

Local Variables

numUnits	foo	cost	total
5		2.45	

Operand Stack

2.45	5

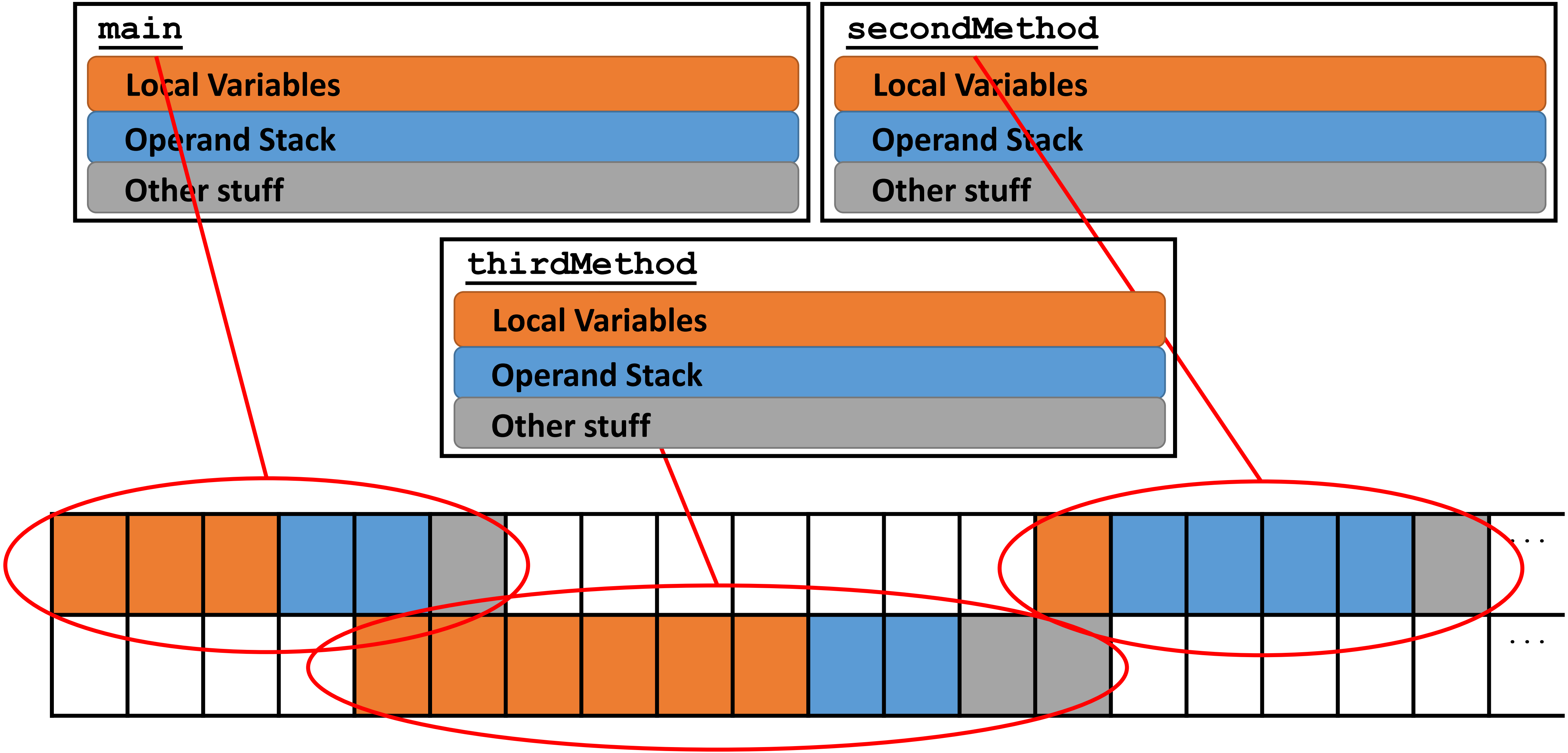
Other stuff

Understanding Objects: memory in Java

Recall:

- Each method has its own frame
- Java executes one command at a time, so only one method is active at a time
    - And **only one frame** is active
- At compile-time, Java gathers together all variables and operands for *each* method
    - Each executed command may then refer to these values within the *active* frame

Understanding Objects: memory in Java



main

Local Variables

Operand Stack

Other stuff

secondMethod

Local Variables

Operand Stack

Other stuff

thirdMethod

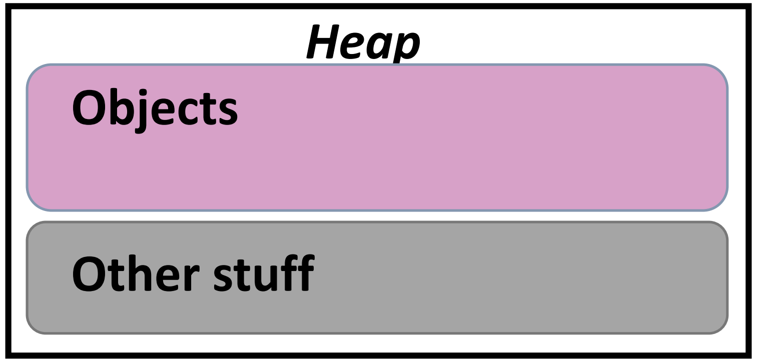
Local Variables

Operand Stack

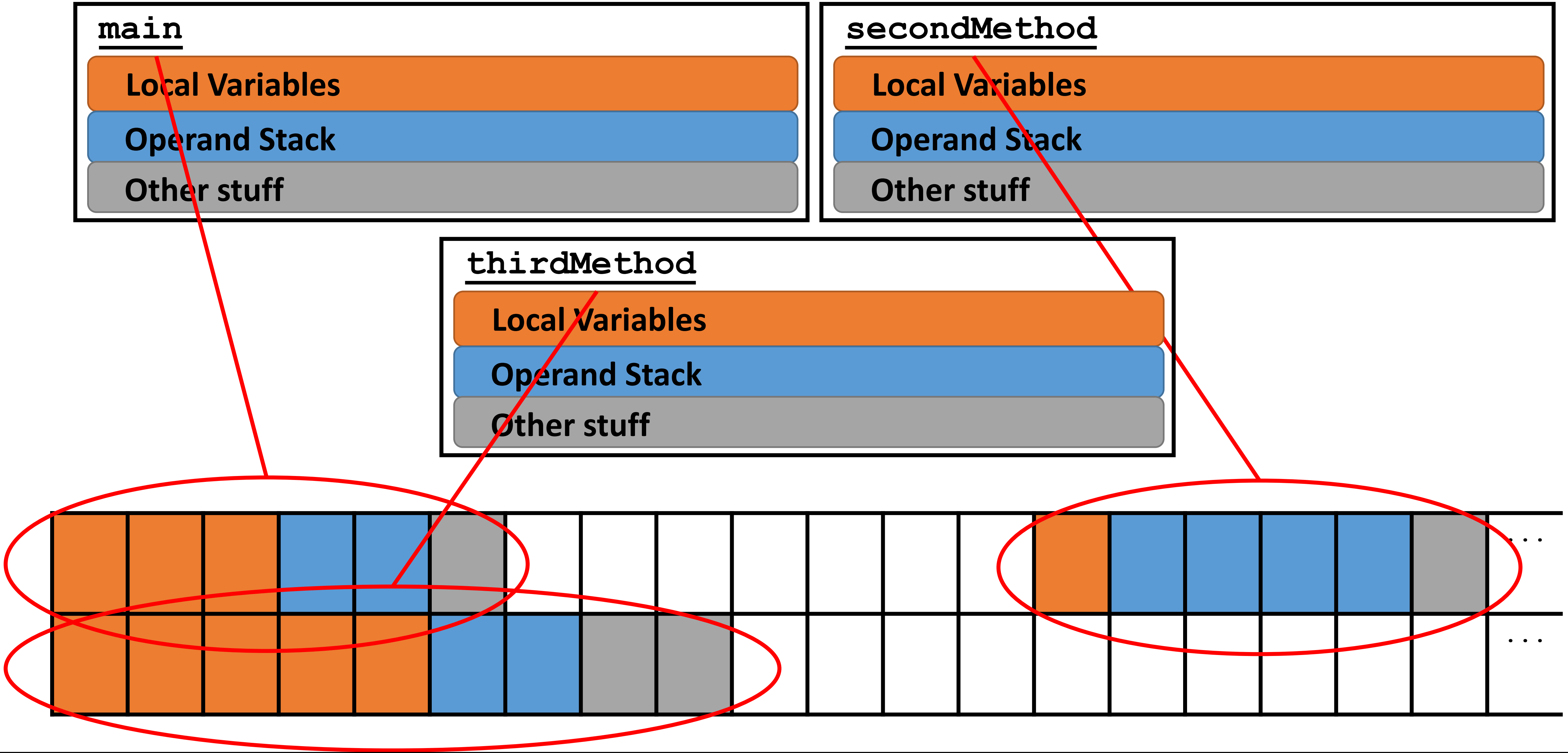
Other stuff

Understanding Objects: memory in Java

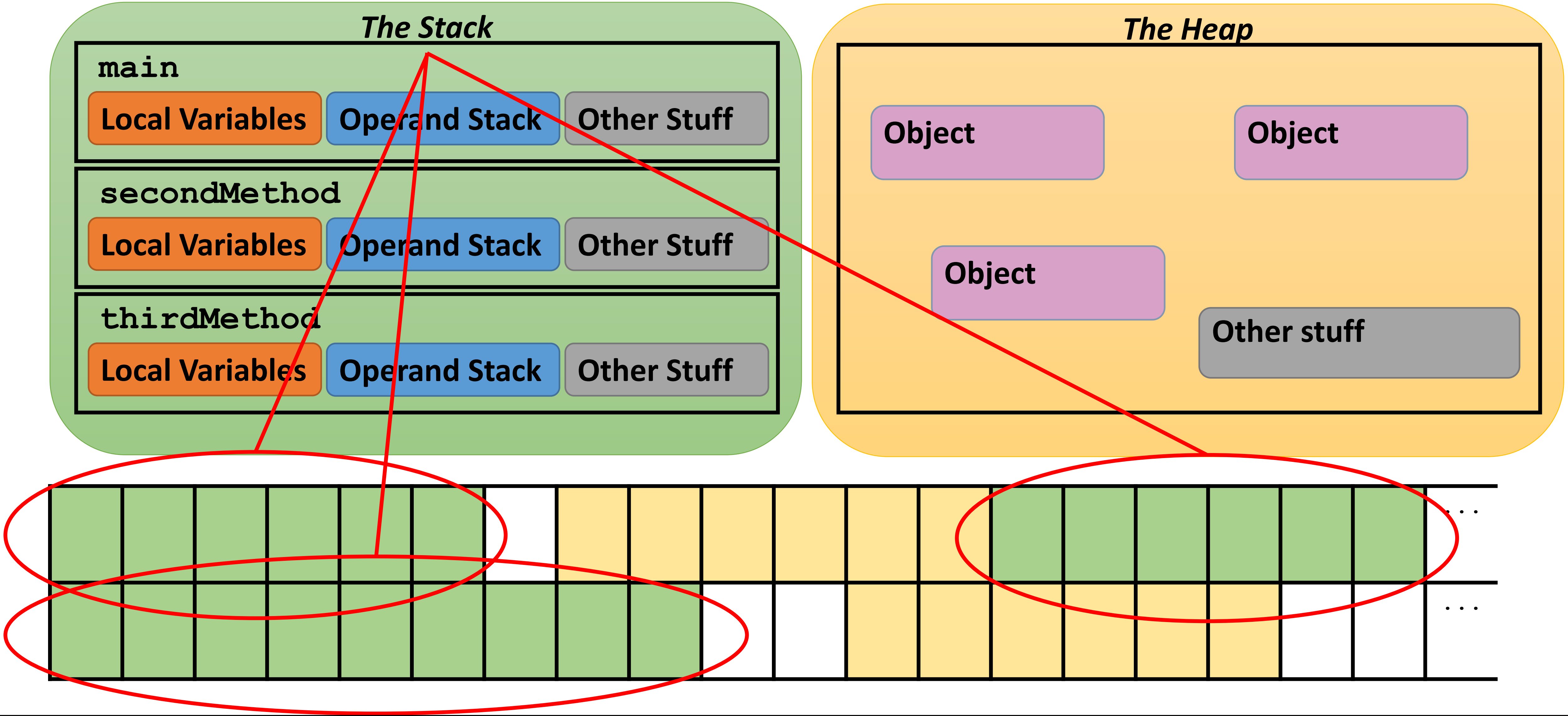
- The stack is not the only place data can live!
- *The heap* is a place to store Objects and other stuff
- Method definitions and variables within an Object are stored on the heap



Understanding Objects: memory in Java



Understanding Objects: memory in Java

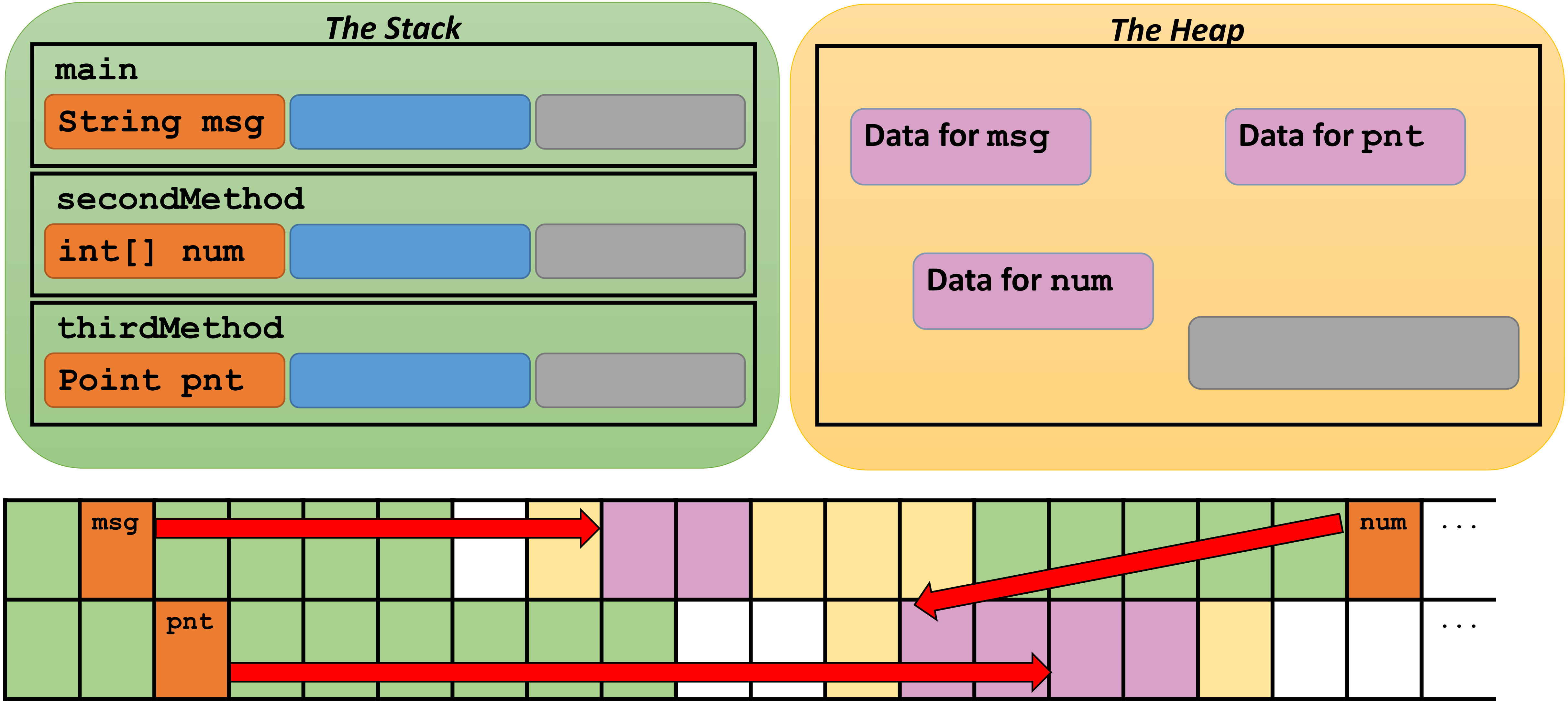


Understanding Objects: memory in Java

- Objects always live on *the heap*
  - This is shared memory, so you need to know where to look

- The where is given by an Object *reference*
  - References to Objects can be stored on the stack
  - Objects themselves (their attributes and operations) are still stored on the heap

Understanding Objects: memory in Java

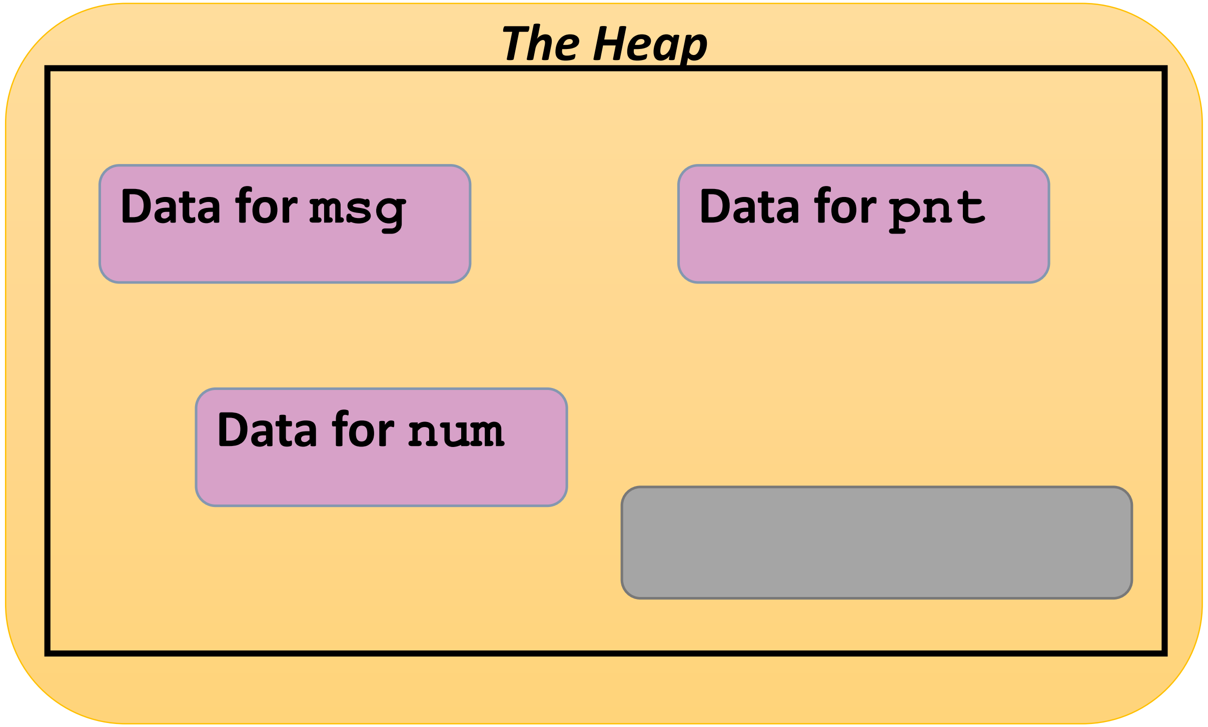


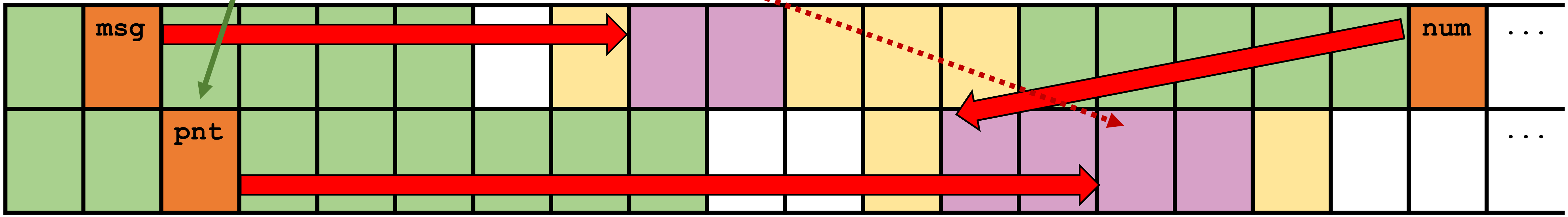
Understanding Objects: memory in Java

- Values passed between methods are always passed using the stack
  - Object references (values) are moved back and forth, not the Objects themselves
- Making space for Objects on the heap requires a special step
  - Use the **`new`** keyword for Objects we design

Understanding Objects: memory in Java

```
String msg = “Hello”;

int[] num = new int[4];

Point pnt = new Point();
```





Understanding Objects: memory in Java

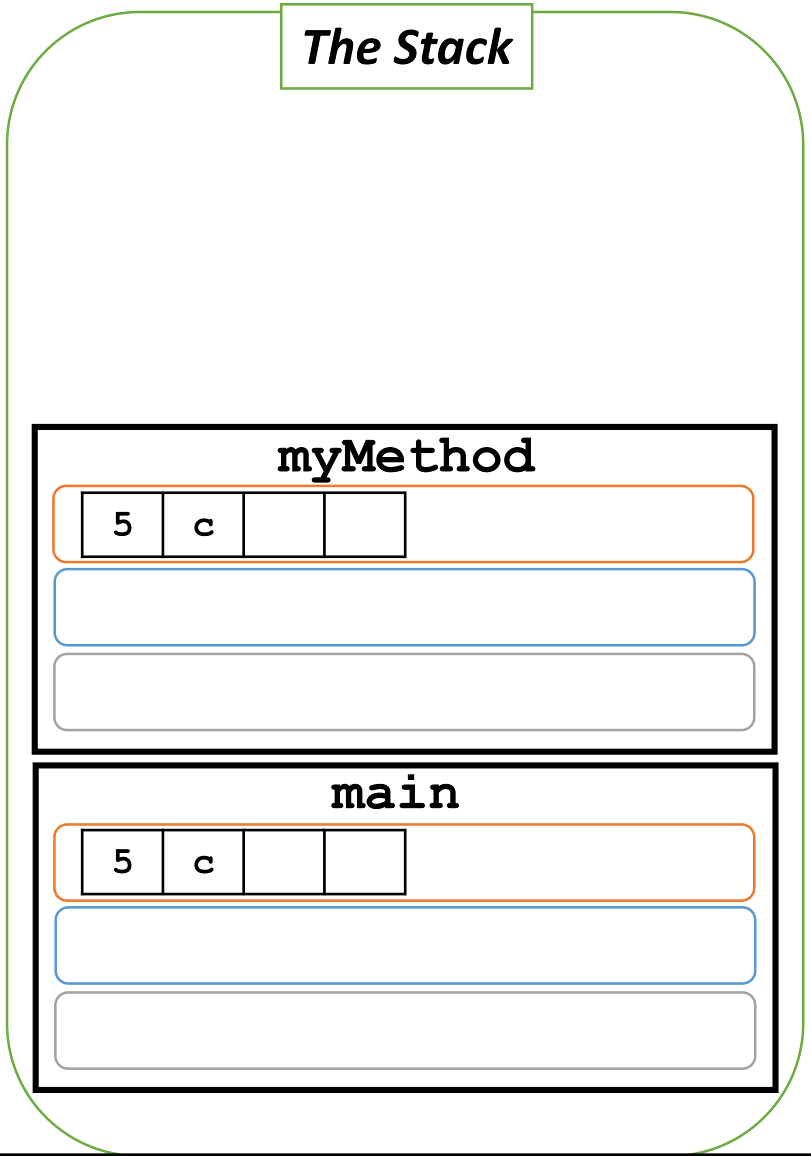
```
String msg = “Hello”;

int[] num = new int[4];

Point pnt = new Point();
```

- Objects are always given a reference that lives on the **stack**, while the actual data lives on the **heap**
  - We will later see examples of Object references also living on the heap!

Understanding Objects: memory in Java

Calling methods using primitive types on the stack:

```
public static void main(String[] args) {
        int a = 5;
        char z = ‘c’;
        myMethod(a, z);
        // a == ?, c == ?
}
public static void myMethod(int a, char z) {
        a++;
        z = ‘Q’;
}
```



The Stack

myMethod

5	c		

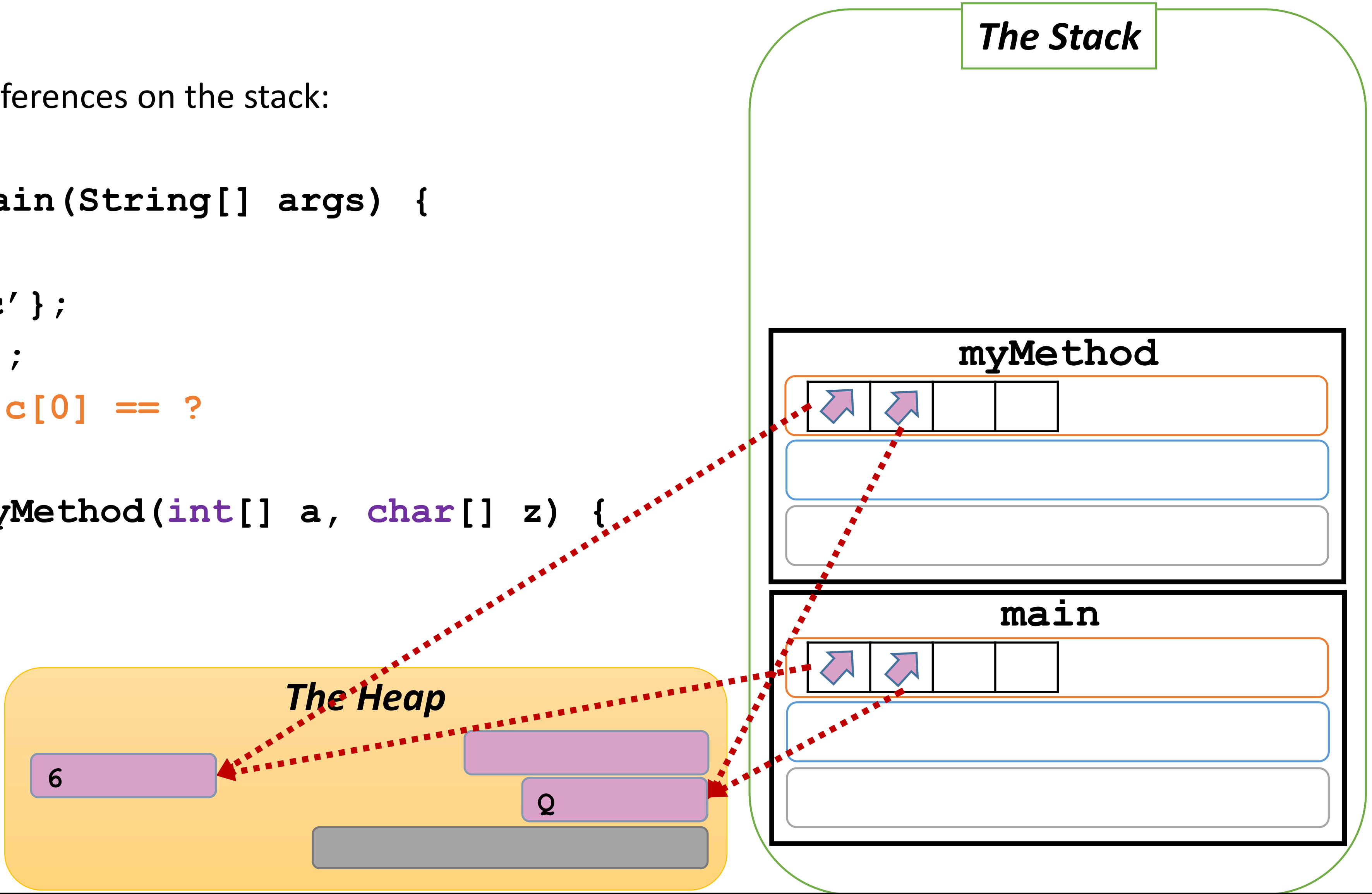
main

5	c		

Understanding Objects: memory in Java

Calling methods using array references on the stack:

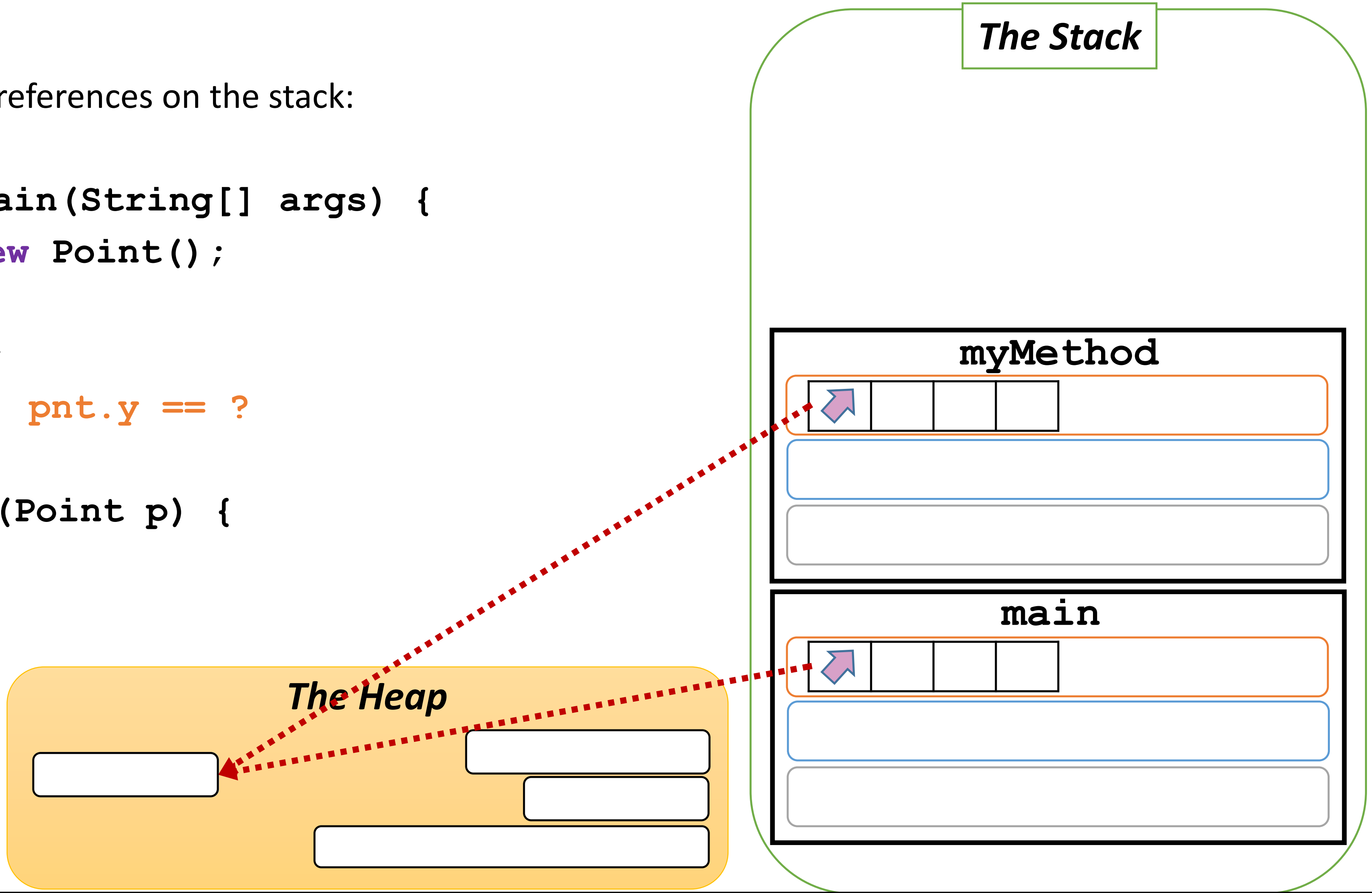
```
public static void main(String[] args) {
        int[] a = {5};
        char[] z = {'c'};
        myMethod(a, z);
        // a[0] == ?, c[0] == ?
}
public static void myMethod(int[] a, char[] z) {
        a[0]++;
        z[0] = 'Q';
}
```



Understanding Objects: memory in Java

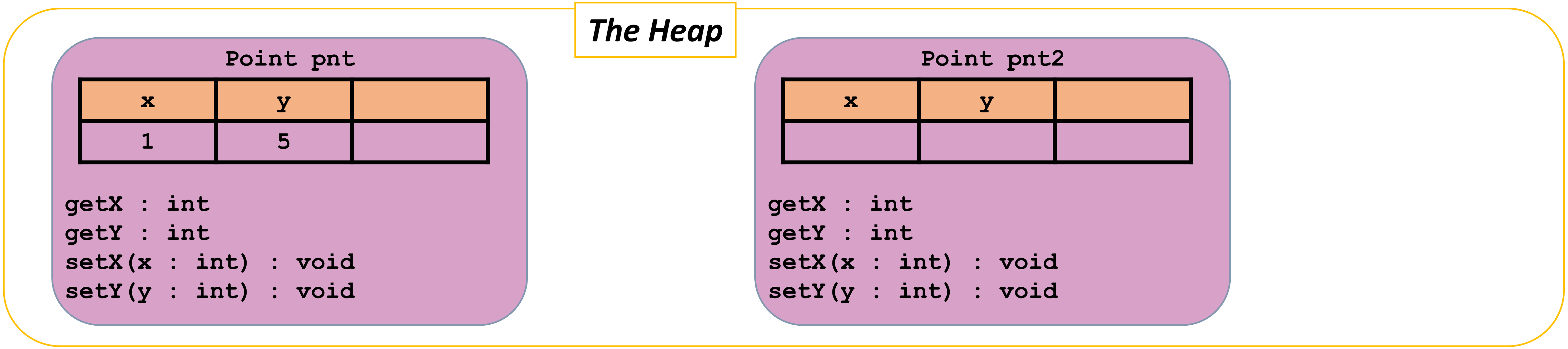
Calling methods using Object references on the stack:

```
public static void main(String[] args) {
        Point pnt = new Point();

        myMethod(pnt);
        // pnt.x == ?, pnt.y == ?
}
public void myMethod(Point p) {
        p.setX(1);
        p.setY(5);
}
```



Understanding Objects: memory in Java

- Objects on the heap contain both *instance variables* and instructions for creating *method frames*



The Heap

Point pnt

x	y	
1	5	

```
getX : int
getY : int
setX(x : int) : void
setY(y : int) : void
```

Point pnt2

x	y	

```
getX : int
getY : int
setX(x : int) : void
setY(y : int) : void
```

Understanding Objects: memory in Java

- Objects are *instances* of classes
  - They follow the same outline: their operations and attribute types are the same, but their attribute values and method executions may differ

- These are referred to as *instance variables* and *instance methods*

- But it is possible to use classes without using actual instances!

Static variables and methods

- The `static` keyword indicates that a variable (or method!) belongs to a *class*, not to an *instance*
  - So the variable/method does not belong to any specific Object

- Static variables:
  - Do *not* belong to an instance of the class
  - Are *not* stored within an instance of the class
  - Do *not* rely on an instance even existing!

- Static methods:
  - Do *not* belong to an instance of the class
  - Can *not* operate on instance variables
  - Do *not* rely on an instance even existing!

Static variables and methods

- **Static variables** only store *one* value for the entire class
  - This one value is shared by all instances of the class, if they exist
  - It exists even if no instances of the class have been created

Static variables and methods – example

```
public class Countable {
    private static int instanceCount = 0;

    public Countable() {
        instanceCount++;
    }

    public int getInstanceCount() {
        return instanceCount;
    }
}
```

This constructor increases the instance count each time a new instance is created

```java
public class Point {
        private static int pointCount = 0;  // static variable
        private int x;  // instance variable
        private int y;  // instance variable

        public Point() {
                 pointCount++;
        }

         public Point(int x, int y) {
                pointCount++;
                this.x = x;
                this.y = y;
        }

        public int getCount() {
                return pointCount;  // refers to the static variable
        }

        // continued ...

}
```

```
public class PointDemo {

    public static void main(String[] args) {
        Point p1, p2;

        p1 = new Point(5, 15);
        System.out.println(p1.getCount());

        p2 = new Point();
        System.out.println(p2.getCount());

        p2.setX(1);
        p2.setY(1);

        System.out.println(p2.getCount());
    }

}
```

Static variables and methods

- Static variables are useful when storing information that is constant across a class

- For example:
  - To create incrementing student numbers
  - To create unique license plates
  - For values that rarely (or never) change, like tax rates or a conversion rate between miles and kilometres

Static variables and methods

- **Static methods** are shared methods that do not operate on values particular to a given Object
  - Unlike with variables, there is no issue of wasted memory
  - Static methods are instead motivated by design: these are operations that belong to the class, not the Object

Static variables and methods

Instance methods

- Operations that must be performed on a specified Object
- May use the Object's attributes
- Result of the operation is seen by the Object

vs.

Static methods

- Operations that do not need a particular Object
- Do not use any Object's attributes
- Written within the class, but may not be related to any instance of that class

Static variables and methods

- Static methods can be called directly from the class when needed
  - An Object instance is not required!

- These are useful for *utility* methods that perform operations on parameters, but do not store or require other data

- Static methods **can not** refer to non-static members (*methods* or *variables*)
  - They *may* call other static methods
  - They *may* use static variables

```
public class Converter {
      private static double ratio = 0.621371;
      // no. of miles in 1 km

      public static double miToKm(double mi) {
            return (1.0 / ratio) * mi;
            // refers to a static variable
      }

      public static double kmToMi(double km) {
            return ratio * km;
            // refers to a static variable
      }

}
```

```
public class ConverterDemo {
    public static void main(String[] args) {
        double k = Converter.miToKm(100.0);
        // 160.934km in 100.0mi

        double m = Converter.kmToMi(50.0);
        // 31.0686mi in 50.0km
    }

}
```

```
public class Employee {
        private static String companyName = “Widgets Inc.”;
        private String name;
        private int hours;
        private double rate;

        public Employee() {     }

        public static String getCompanyName() {
                return companyName;  // static method refers to a static variable
        }

        public int getHours() {
                return hours;f
        }

        public static double getRate() {
                return rate;
        }
}
```