From Python to Java: Primitives, Objects, and References and the Java Memory Model

Heap Memory
Non-Heap Memory
other

Recall: Data Types We've Seen Thus Far

- `int` - an integer stored using 4 bytes

```
    int count = 0;
```

- `long` - an integer stored using 8 bytes

```
    long result = 1;
```

- `double` - a floating-point number (one with a decimal)

```
    double area = 125.5;
```

- `boolean` - either `true` or `false`

```
    boolean isPrime = false;
```

Primitive types

- `String` - a sequence of 0 or more characters

```
    String message = "Welcome to CS 112!";
```

- `Scanner` – an object for getting input from the user

```
    Scanner scan = new Scanner(System.in);
```

Recall: Data Types We've Seen Thus Far

- `int` - an integer stored using 4 bytes

  ```
  int count = 0;
  ```

- `long` - an integer stored using 8 bytes

  ```
  long result = 1;
  ```

- `double` - a floating-point number (one with a decimal)

  ```
  double area = 125.5;
  ```

- `boolean` - either `true` or `false`

  ```
  boolean isPrime = false;
  ```

- `String` - a sequence of 0 or more characters

  ```
  String message = "Welcome to CS 112!";
  ```

- `Scanner` – an object for getting input from the user

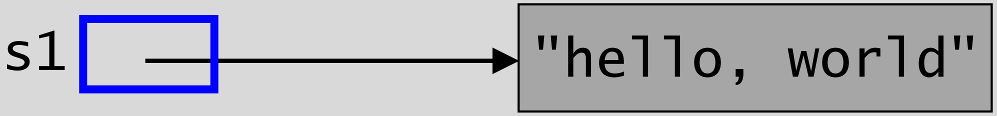
  ```
  Scanner scan = new Scanner(System.in);
  ```

Reference types

Reference Types

- Java stores ***objects*** the same way that Python does:

```
    String s1 = "hello, world";
```

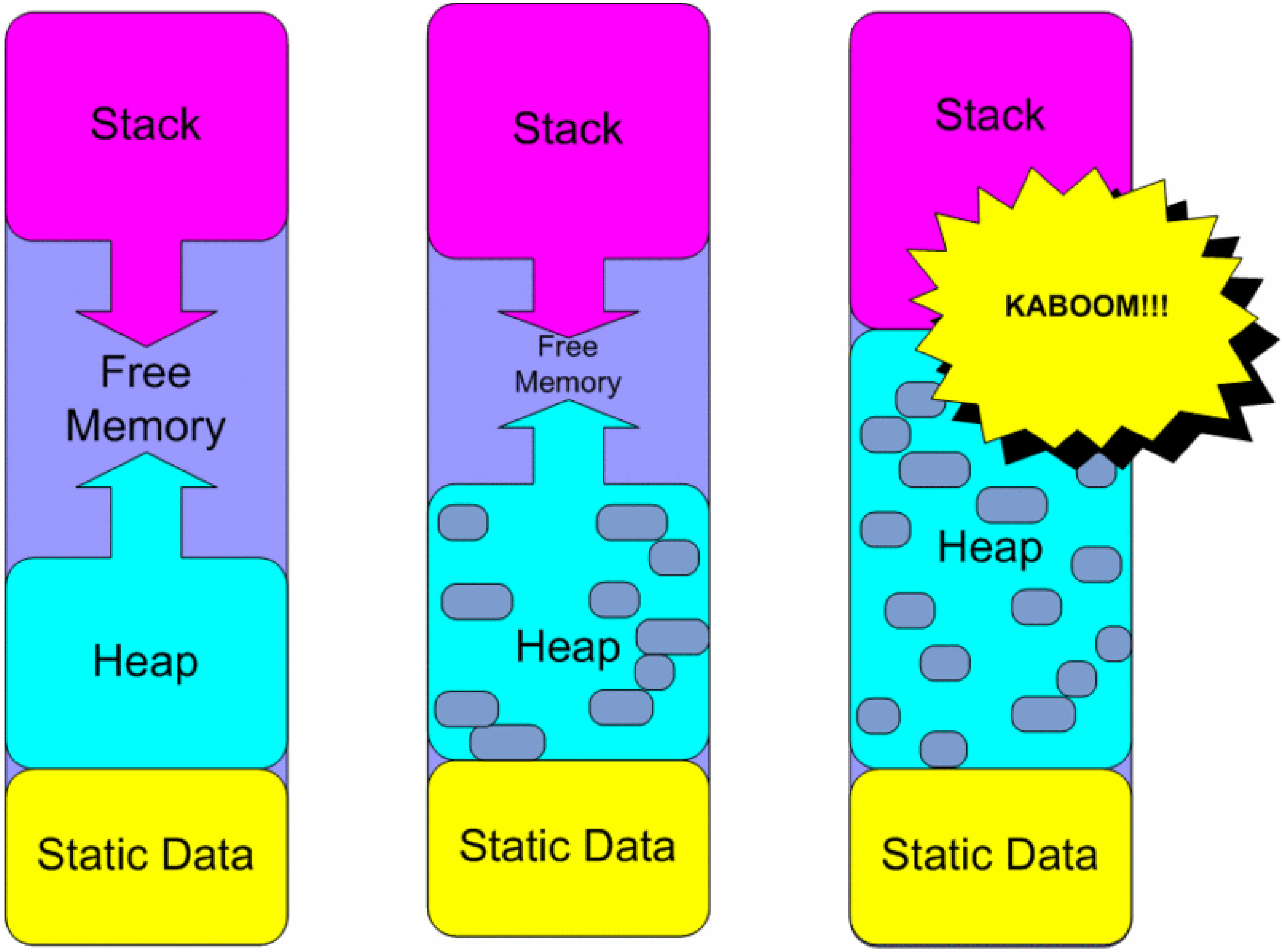


  - the object is located elsewhere in memory
  - the variable stores a reference to the object

- Data types that work this way are known as *reference types.*
  - variables of those types are *reference variables*

- We've worked with two *reference types* thus far:
  - `String`
  - `Scanner`

Java Memory Model



Normal SRAM Operation

Fragmented Heap

Stack Crash!

Memory Management: Looking Under the Hood

- There are three main types of memory allocation in Java.

- They correspond to three different regions of memory:

• Static	**class variables**
• Stack	**local variables, parameters**
• Heap	objects

Information known at compile time!

Memory Management: Looking Under the Hood

- There are three main types of memory allocation in Java.

- They correspond to three different regions of memory:

• Static	class variables
• Stack	local variables, parameters
• Heap	**objects**

Memory allocated during run time!

Memory Management: Looking Under the Hood

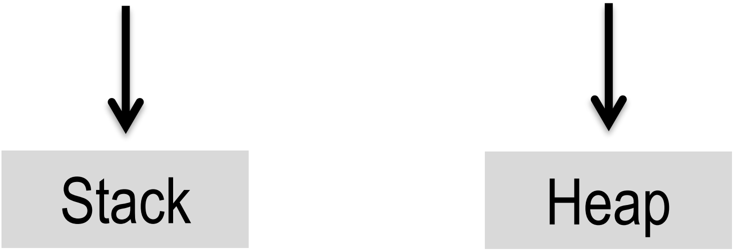
- There are three main types of memory allocation in Java.

- They correspond to three different regions of memory:

Static	class variables
Stack	local variables, parameters
Heap	**objects …**

Example: creating a Scanner object

```
Scanner scan = new Scanner(System.in);
```



Stack

Heap

Memory Management: Looking Under the Hood

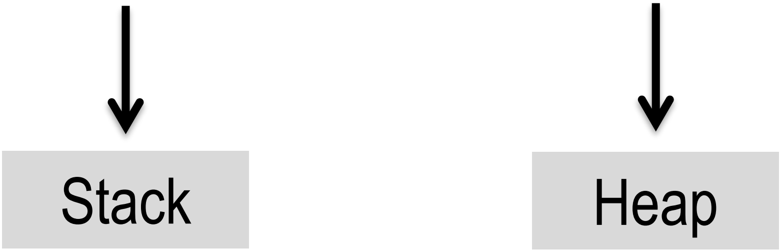
- There are three main types of memory allocation in Java.

- They correspond to three different regions of memory:

Static	class variables
Stack	local variables, parameters
Heap	**objects …**

Example: creating a String object

```
String str = new String("String");
```



Stack

Heap

Memory Management: Looking Under the Hood

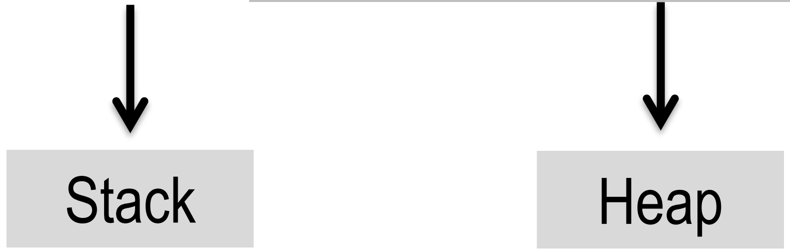
- There are three main types of memory allocation in Java.

- They correspond to three different regions of memory:

  - Static class variables
  - Stack local variables, parameters
  - Heap **objects …**

Example: creating a String object

```
String str = new String("String");
```



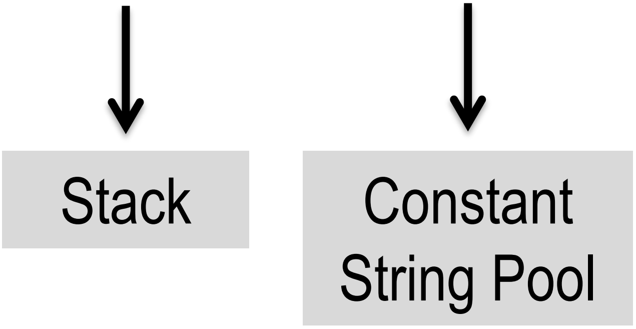
Stack

Heap

Memory Management: Looking Under the Hood

- There are three main types of memory allocation in Java.
- They correspond to three different regions of memory:
  - Static class variables
  - Stack local variables, parameters
  - Heap **objects …** … and The String Constant Pool for Java **literal** strings.

```
String str = "A String";
```



Stack

Constant String Pool

Memory Management: Looking Under the Hood

- There are three main types of memory allocation in Java.

- They correspond to three different regions of memory:

• Static	class variables
• Stack	local variables, parameters
• Heap	objects …

Memory Management, Type I: Static Storage

- Static storage is used in Java for *class variables*, which are declared using the keyword `static`:

```
public static double PI = 3.1495;
public static int numCompares;
```

- There is only one copy of each class variable; it is shared by all *instances* (i.e., all objects) and all *methods* of the class.

- The Java runtime system allocates memory for *class variables* *when the class is first encountered*.
  - this memory stays fixed for the duration of the program

Memory Management, Type I: Static Storage

- Static storage is used in Java for *class variables*, which are declared using the keyword `static`:

```
public static final double PI = 3.1495;
public static int numCompares;
```

- There is only one copy of each class variable; it is shared by all instances (i.e., all objects) and all methods of the class.

- The Java runtime system allocates memory for *class variables* when the class is first encountered.
  - this memory stays fixed for the duration of the program

- Keyword *final* makes the variable *read-only*. Once a variable declared as final is assigned a value, it cannot be re-assigned.

Memory Management, Type II: Stack Storage

- Method parameters and local variables are stored in a region of memory known as *the stack*.

- For each method call, a new *stack frame* is added to the top of the stack.

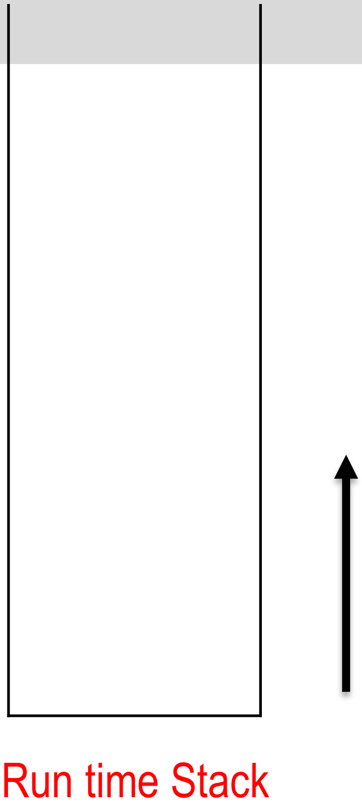
```
public class Foo {
  public static void x(int i) {
      int j = i - 2;

      if (i < 6)
          x(i + j);
  }

  public static void
    main(String[] args) {
      x(5);
  }
}
```

Memory Management, Type II: Stack Storage

- Method parameters and local variables are stored in a region of memory known as *the stack*.

- For each method call, a new *stack frame* is added to the top of the stack.

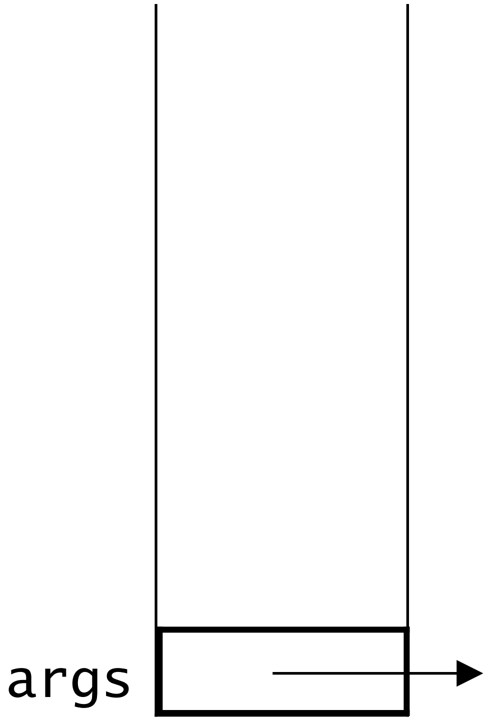
```
public class Foo {
 public static void x(int i) {
   int j = i - 2;

   if (i < 6)
     x(i + j);
 }

 public static void
  main(String[] args) {
   x(5);
 }
}
```



Memory Management, Type II: Stack Storage

- Method parameters and local variables are stored in a region of memory known as *the stack*.
- For each method call, a new *stack frame* is added to the top of the stack.

```
public class Foo {
  public static void x(int i) {
      int j = i - 2;

      if (i < 6)
          x(i + j);
  }

  public static void
    main(String[] args) {
      x(5);
  }
}
```



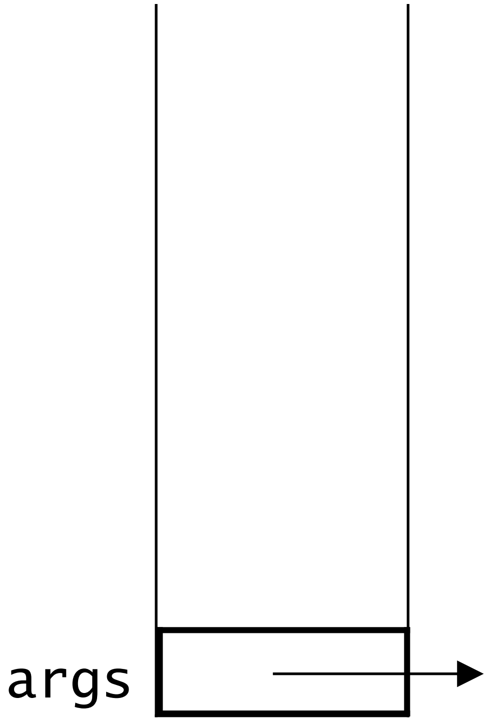
Run time Stack

Memory Management, Type II: Stack Storage

- Method parameters and local variables are stored in a region of memory known as *the stack*.

- For each method call, a new *stack frame* is added to the top of the stack.

```
public class Foo {
  public static void x(int i) {
      int j = i - 2;

      if (i < 6)
          x(i + j);
  }

  public static void
    main(String[] args) {
      x(5);
  }
}
```



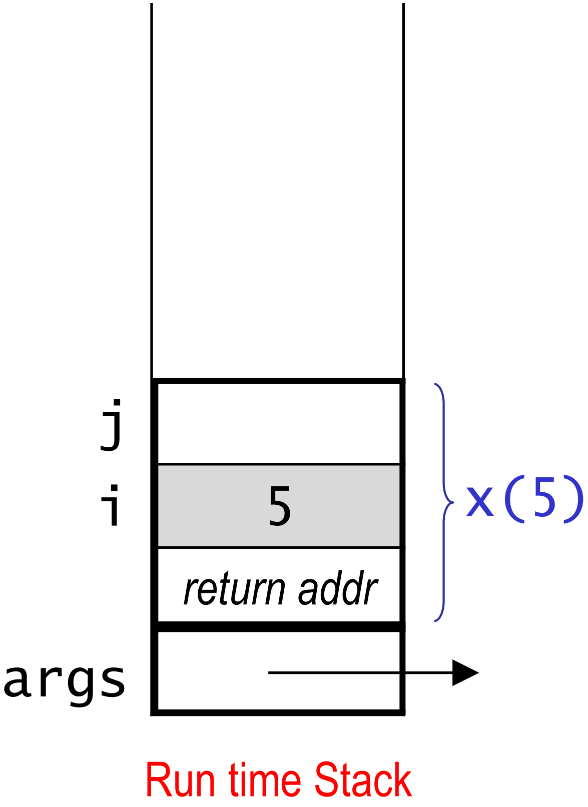
Run time Stack

Memory Management, Type II: Stack Storage

- Method parameters and local variables are stored in a region of memory known as *the stack*.

- For each method call, a new *stack frame* is added to the top of the stack.

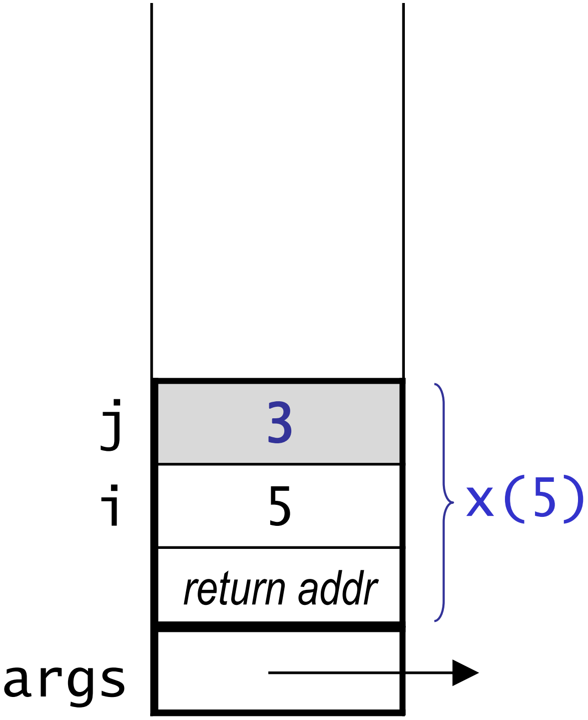
```
public class Foo {
  public static void x(int i) {
      int j = i - 2;

      if (i < 6)
          x(i + j);
  }

  public static void
    main(String[] args) {
      x(5);
  }
}
```



Memory Management, Type II: Stack Storage

- Method parameters and local variables are stored in a region of memory known as *the stack*.
- For each method call, a new *stack frame* is added to the top of the stack.

```
public class Foo {
  public static void x(int i) {
      int j = i - 2;

      if (i < 6)
          x(i + j);
  }

  public static void
    main(String[] args) {
      x(5);
  }
}
```

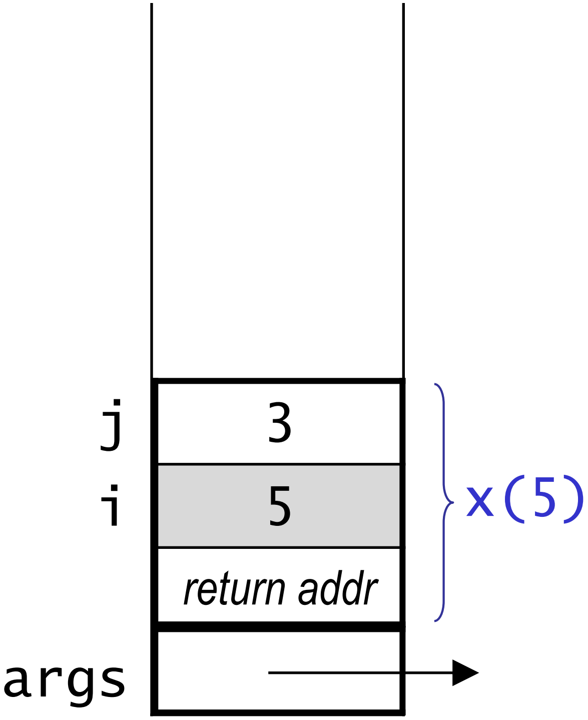


Run time Stack

Memory Management, Type II: Stack Storage

- Method parameters and local variables are stored in a region of memory known as *the stack*.
- For each method call, a new *stack frame* is added to the top of the stack.

```
public class Foo {
   public static void x(int i) {
         int j = i - 2;

         if (i < 6)
               x(i + j);
   }

   public static void
      main(String[] args) {
         x(5);
   }
}
```



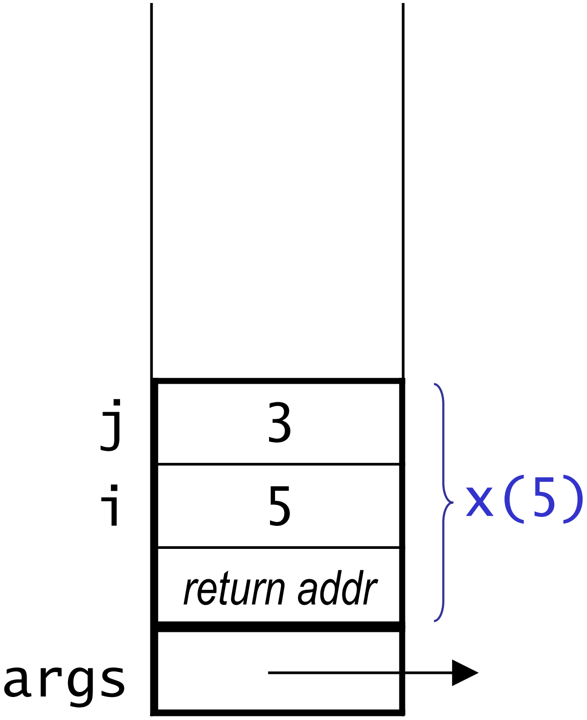
Run time Stack

Memory Management, Type II: Stack Storage

- Method parameters and local variables are stored in a region of memory known as *the stack*.

- For each method call, a new *stack frame* is added to the top of the stack.

```
public class Foo {
   public static void x(int i) {
         int j = i - 2;

         if (i < 6)
              x(i + j);
   }

   public static void
      main(String[] args) {
         x(5);
   }
}
```



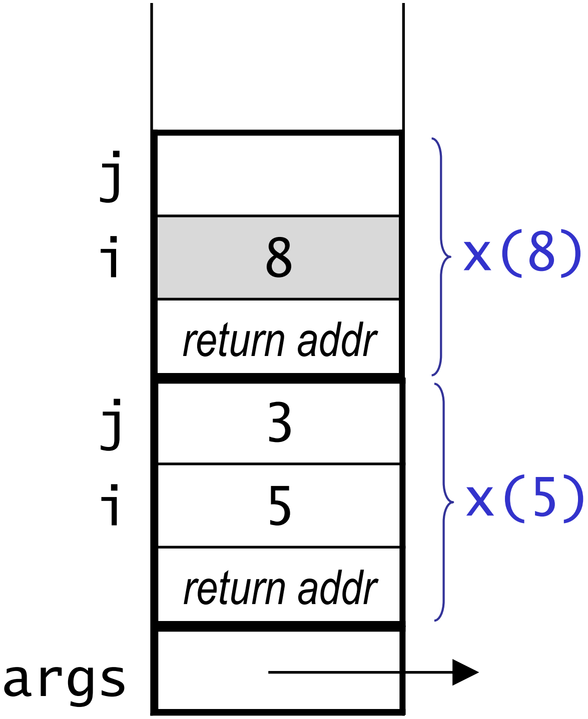
Run time Stack

Memory Management, Type II: Stack Storage

- Method parameters and local variables are stored in a region of memory known as *the stack*.

- For each method call, a new *stack frame* is added to the top of the stack.

```
public class Foo {
  public static void x(int i) {
      int j = i - 2;

      if (i < 6)
          x(i + j);
  }

  public static void
    main(String[] args) {
      x(5);
  }
}
```

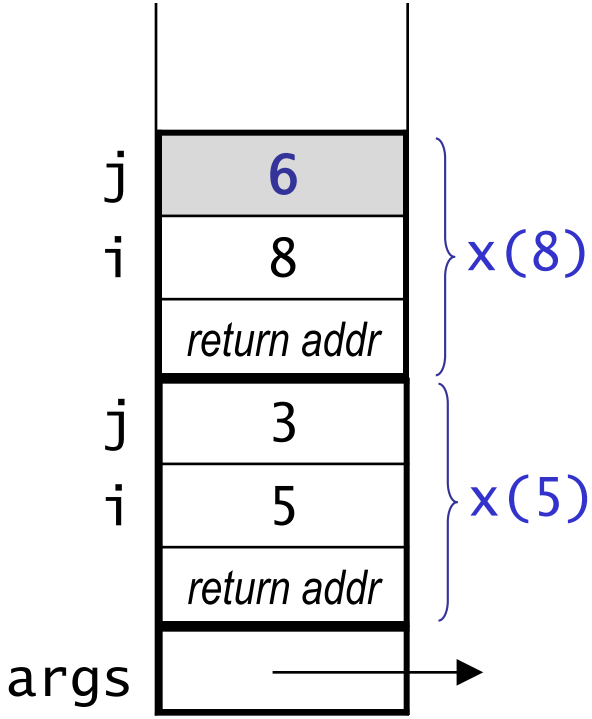


Run time Stack

Memory Management, Type II: Stack Storage

- Method parameters and local variables are stored in a region of memory known as *the stack*.
- For each method call, a new *stack frame* is added to the top of the stack.

```
public class Foo {
   public static void x(int i) {
         int j = i - 2;

         if (i < 6)
               x(i + j);
   }

   public static void
      main(String[] args) {
         x(5);
   }
}
```

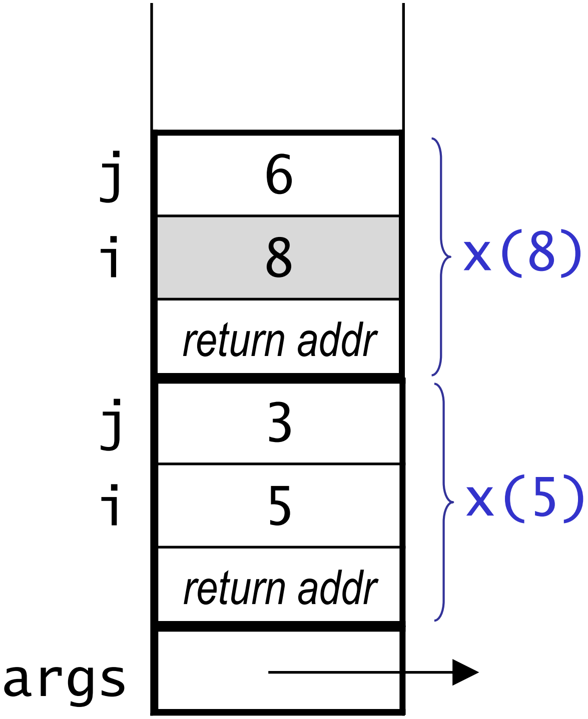


Memory Management, Type II: Stack Storage

- Method parameters and local variables are stored in a region of memory known as *the stack*.

- For each method call, a new *stack frame* is added to the top of the stack.

```
public class Foo {
  public static void x(int i) {
      int j = i - 2;

      if (i < 6)
          x(i + j);
  }

  public static void
    main(String[] args) {
      x(5);
  }
}
```

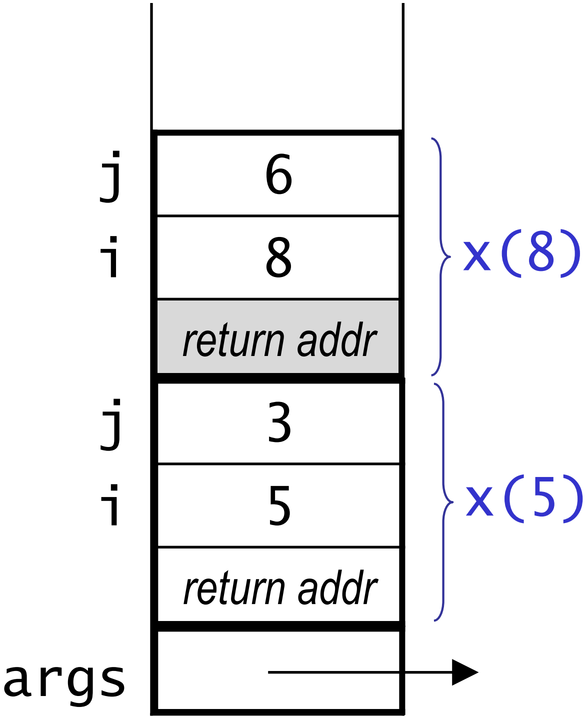


Memory Management, Type II: Stack Storage

- Method parameters and local variables are stored in a region of memory known as *the stack*.

- For each method call, a new *stack frame* is added to the top of the stack.

```
public class Foo {
  public static void x(int i) {
      int j = i - 2;

      if (i < 6)
          x(i + j);
  }

  public static void
    main(String[] args) {
      x(5);
  }
}
```

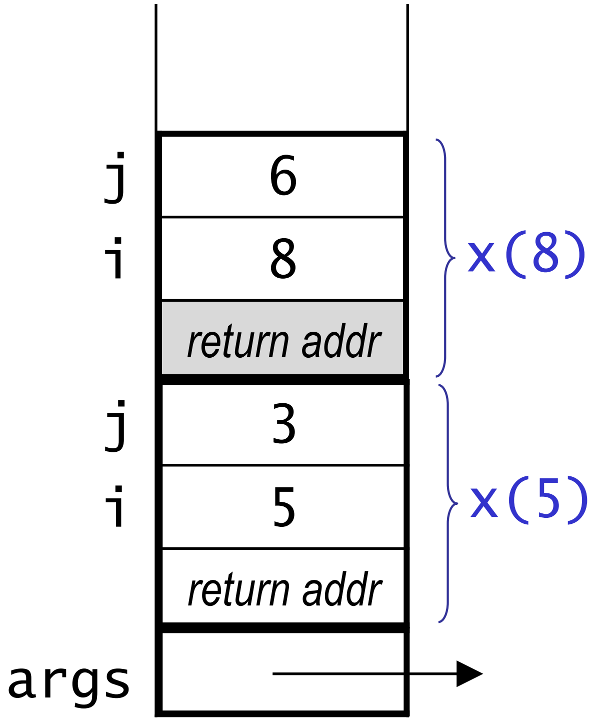


Memory Management, Type II: Stack Storage

- Method parameters and local variables are stored in a region of memory known as *the stack*.

- For each method call, a new *stack frame* is added to the top of the stack.

```
public class Foo {
  public static void x(int i) {
      int j = i - 2;

      if (i < 6)
          x(i + j);
  }

  public static void
    main(String[] args) {
      x(5);
  }
}
```



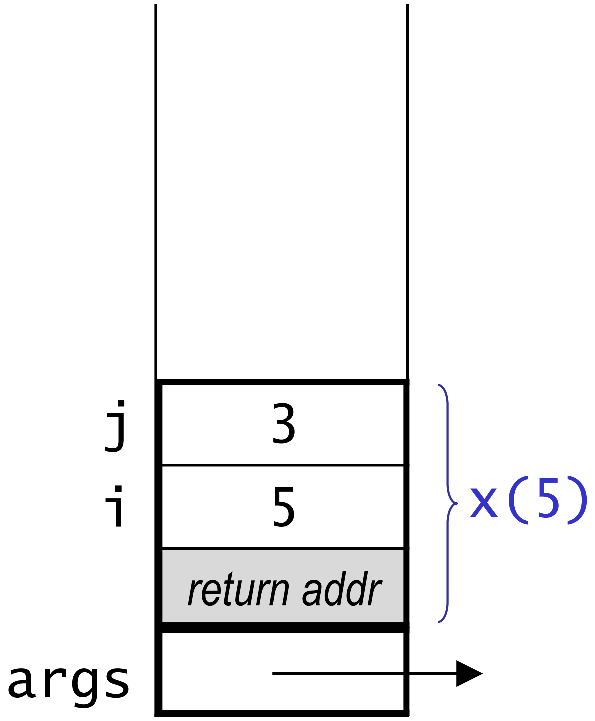
- When a method completes, its stack frame is removed.

Memory Management, Type II: Stack Storage

- Method parameters and local variables are stored in a region of memory known as *the stack*.

- For each method call, a new *stack frame* is added to the top of the stack.

```
public class Foo {
  public static void x(int i) {
      int j = i - 2;

      if (i < 6)
          x(i + j);
  }

  public static void
    main(String[] args) {
      x(5);
  }
}
```



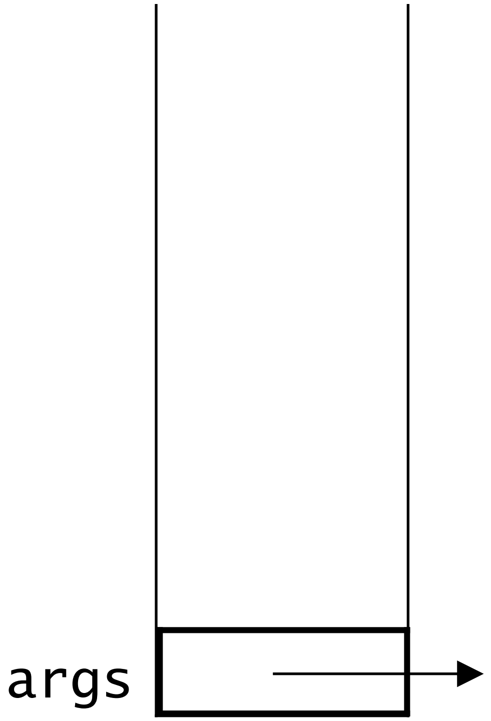
- When a method completes, its stack frame is removed.

Memory Management, Type II: Stack Storage

- Method parameters and local variables are stored in a region of memory known as *the stack*.

- For each method call, a new *stack frame* is added to the top of the stack.

```
public class Foo {
  public static void x(int i) {
      int j = i - 2;

      if (i < 6)
          x(i + j);
  }

  public static void
    main(String[] args) {
      x(5);
  }
}
```



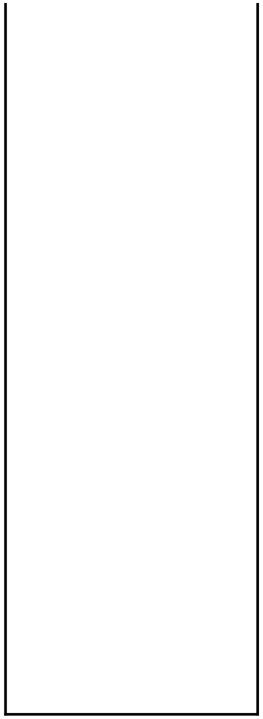
- When a method completes, its stack frame is removed.

Memory Management, Type II: Stack Storage

- Method parameters and local variables are stored in a region of memory known as *the stack*.

- For each method call, a new *stack frame* is added to the top of the stack.

```
public class Foo {
  public static void x(int i) {
      int j = i - 2;

      if (i < 6)
          x(i + j);
  }

  public static void
    main(String[] args) {
      x(5);
  }
}
```



- When a method completes, its stack frame is removed.

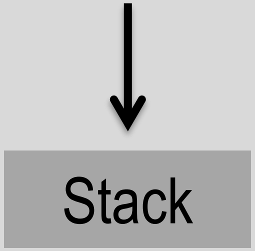
Primitive vs. Reference types

- Static variables are stored in *Static memory…*
- Objects are stored on *the Heap*…
- Primitive variables are stored on *the Stack*…

Primitive vs. Reference types

- Static variables are stored in *Static memory…*
- Objects are stored on *the Heap*…
- Primitive variables are stored on *the Stack*…

```
int p_var = 5;      // primitive variable
```



Stack

Primitive vs. Reference types

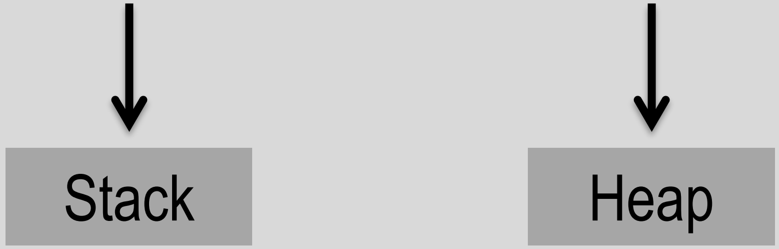
- Static variables are stored in *Static memory…*
- Objects are stored on *the Heap*…
- Primitive variables are stored on *the Stack*…

```
int p_var = 5;       // primitive variable

// Java Wrapper Classes for primitive types
```

Primitive vs. Reference types

- Static variables are stored in *Static memory…*
- Objects are stored on *the Heap*…
- Primitive variables are stored on *the Stack*…

```
int p_var = 5;          // primitive variable

Integer i_ref = new Integer(5);
```



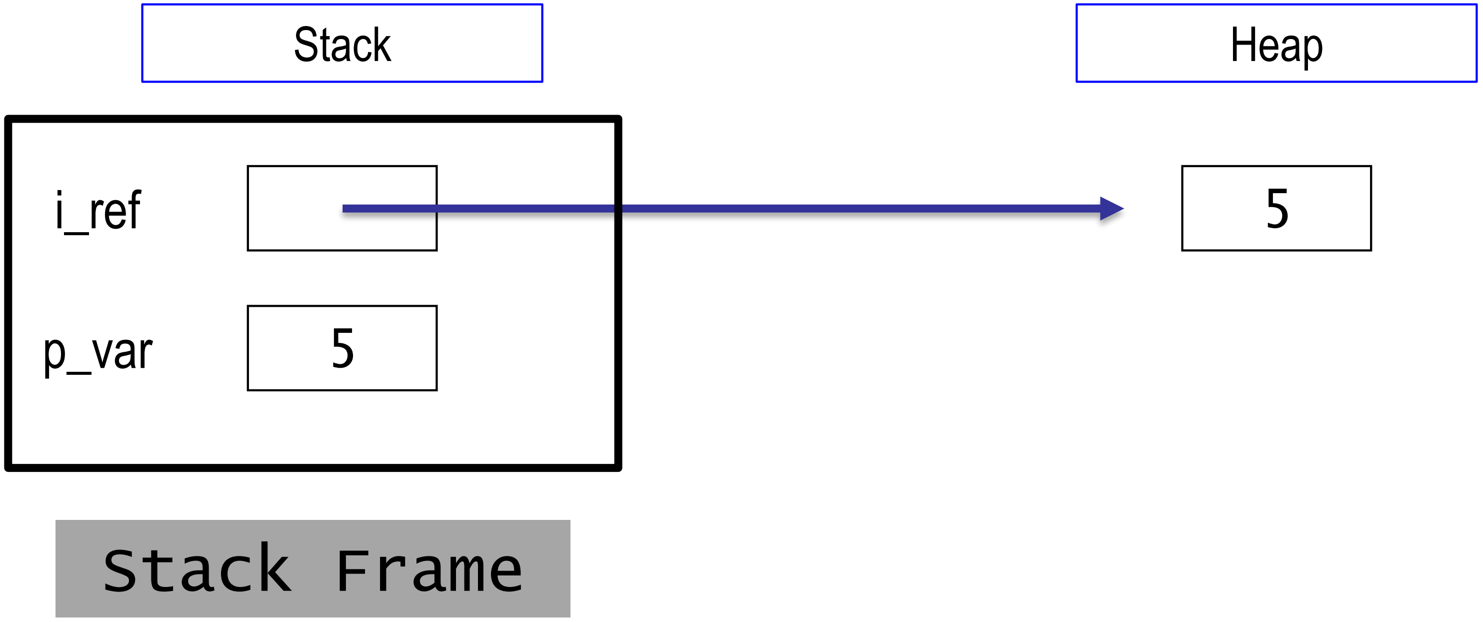
Primitive vs. Reference types

- Static variables are stored in *Static memory…*
- Objects are stored on *the Heap*…
- Primitive variables are stored on *the Stack*…

```
int p_var = 5;      // primitive variable

Integer i_ref = new Integer(5);

Character c_ref = new Character( 'c' );
Double d_ref = new Character(5.555555);
Float f_ref = new Float(5.5);
Boolean b_ref = new Boolean( true );
```

int Primitive vs. Integer Object

```
Integer i_ref = new Integer(5);  // an integer object
int p_var = 5;                   // primitive variable
```



Testing for Equivalent *Primitive* Values

- The `==` and `!=` operators are used to compare primitives.
  - `int`, `double`, `char`, etc.

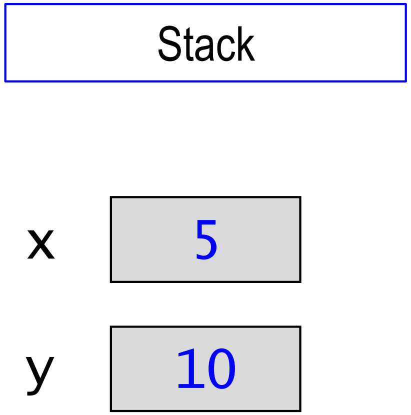
What is being compared
is the value stored
at a specific
address location.

Testing for Equivalent *Primitive* Values

- The `==` and `!=` operators are used to compare primitives.
  - `int`, `double`, `char`, etc.

```
int x = 5;
int y = 10;
if ( x == y ) {


}
```

Comparing
the actual values
5 and 10.

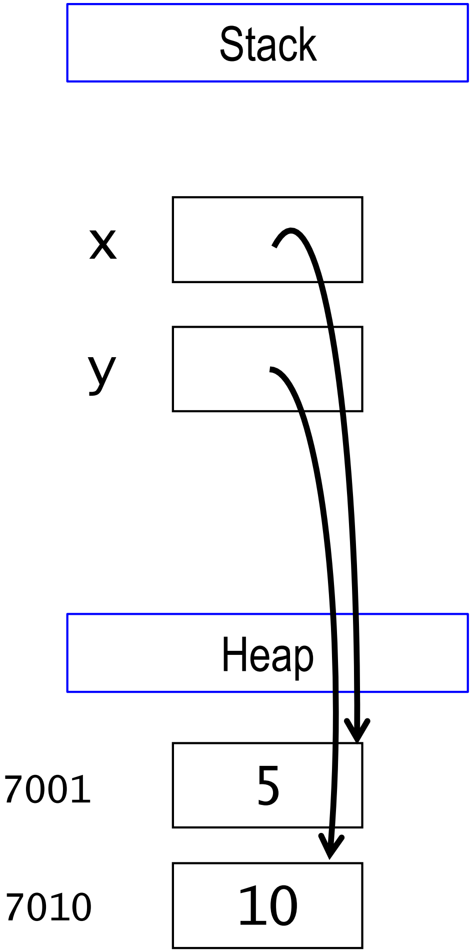


Testing for Equivalent *Objects:*

Numeric Wrapper Classes

- The `==` and `!=` operators do *not* typically work when comparing *objects*

```
Integer x = new Integer(5);
Integer y = new Integer(10);
if ( x == y ) {


}
```



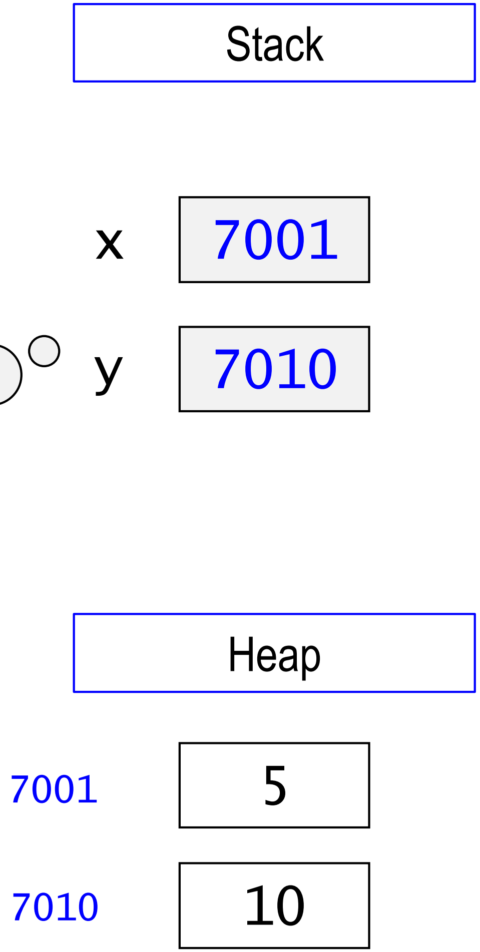
Testing for Equivalent *Objects:*

Numeric Wrapper Classes

- The `==` and `!=` operators do *not* typically work when comparing *objects*

```
Integer x = new Integer(5);
Integer y = new Integer(10);
if ( x == y ) {


}
```

The value stored in the variables are references!



Stack

x 7001

y 7010

Heap

7001 5

7010 10

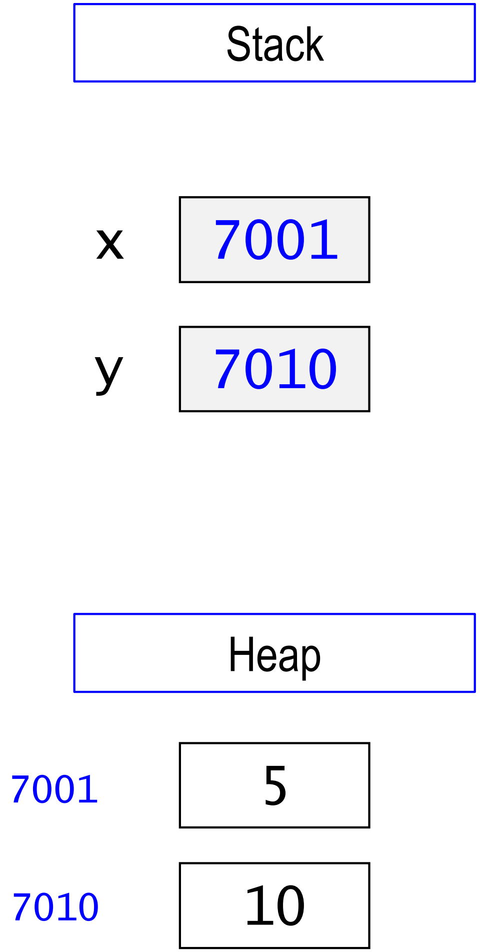
Testing for Equivalent *Objects:*

Numeric Wrapper Classes

- The `==` and `!=` operators do *not* typically work when comparing *objects*

```
Integer x = new Integer(5);
Integer y = new Integer(10);
if ( x == y ) {


}
```

Comparing the address locations of the Integer objects!



Stack

x	7001
y	7010

Heap

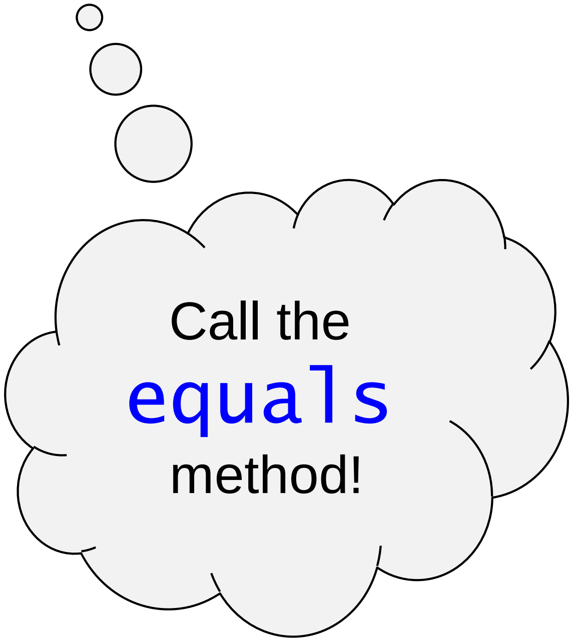
7001	5
7010	10

Testing for Equivalent *Objects:*

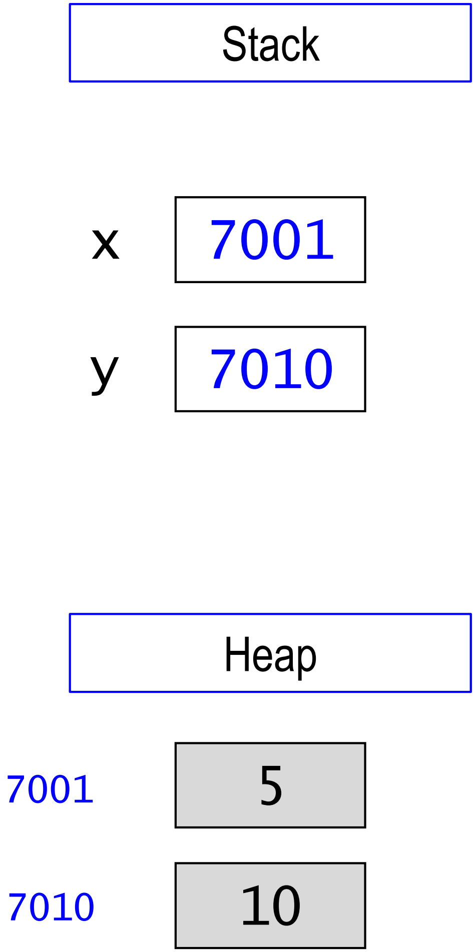
Numeric Wrapper Classes

- The `==` and `!=` operators do *not* typically work when comparing *objects*

```
Integer x = new Integer(5);
Integer y = new Integer(10);
if ( x.equals(y) ) {


}
```



Call the `equals` method!



Stack

x	7001
y	7010

Heap

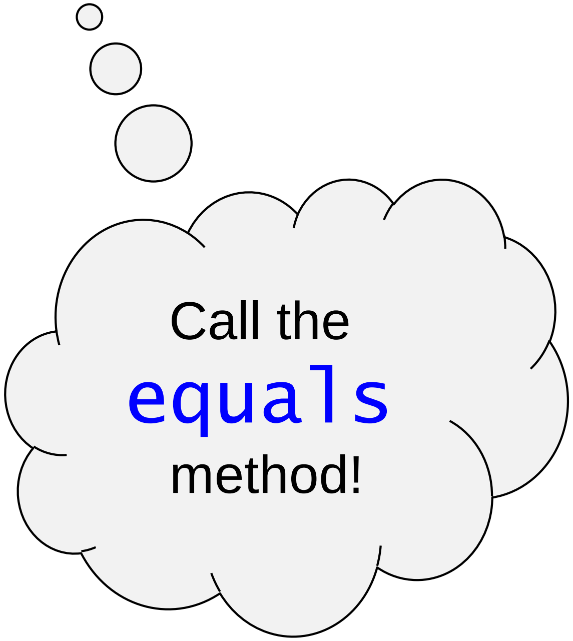
7001	5
7010	10

Testing for Equivalent *Objects:*

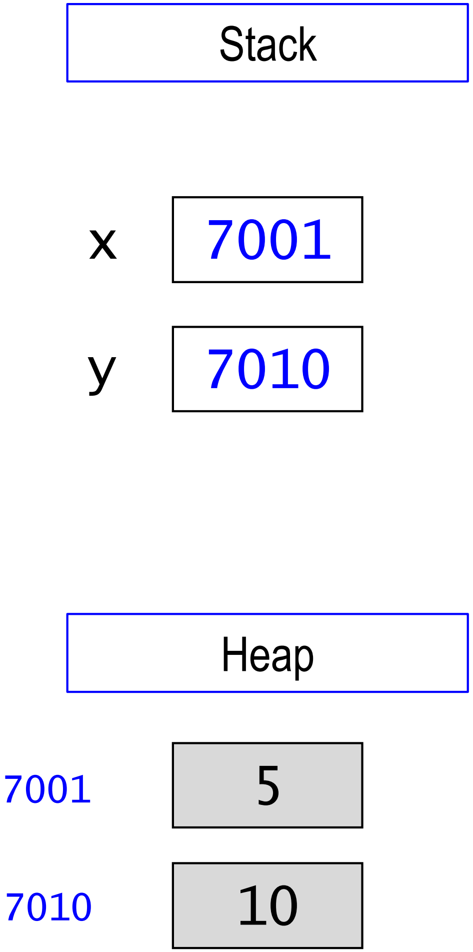
Numeric Wrapper Classes

- The == and != operators do *not* typically work when comparing *objects*

```
Integer x = new Integer(5);
Integer y = new Integer(10);
if ( y.equals(x) ) {


}
```



Call the equals method!



Stack

x	7001
y	7010

Heap

7001	5
7010	10