Methods: Be Careful about Re-declaration!

- There can be only one object with a given name in any scope.
- However, Java allows us to declare new local variables that hide ones with the same name that are declared in a larger scope.
- This is a major source of null pointer errors and bugs!

```
public class Driver {
    private Window win;
    public Driver() {
        Window win = new Window();
        addToWindow();
    }
    private void addToWindow() {
        win.add( new Oval(0, 0, 10, 10) );
    }
}
```

A reference to the global instance variable, which has never been initialized!

```
NullPointerException
```

Ensuring Objects Exist

- Another source of null pointer errors is when objects are initialized by one method, but only after another has already tried to use them.
- We must guarantee that all objects are initialized before they are used.
- This can be difficult, especially with interactive programs.

```
public class Driver {
    private Window win;
    private Oval o;
    public Driver() {
        win = new Window();
        o = new Oval( 0, 0, 10, 10 );
        win.add( o );
    }
    private void buttonOnePressed() {
        o.setBackground( Color.red );
    }
    private void buttonTwoPressed() {
        o.setBackground( Color.red );
    }
}
```

If `buttonTwoPressed()` is ever called before `buttonOnePressed()`: **NullPointerException**

```
public class Driver {
    private Window win;
    private Oval o;
    public Driver() {
        win = new Window();
        o = new Oval( 0, 0, 10, 10 );
    }
    private void buttonOnePressed() {
        win.add( o );
    }
    private void buttonTwoPressed() {
        if ( o != null )
            o.setBackground( Color.red );
    }
}
```

One solution: initialize objects in constructor (which is always guaranteed to run before any buttons can be pressed), even if you don’t use them there.

```
public class Driver {
    private Window win;
    private Oval o;
    public Driver() {
        win = new Window();
        o = new Oval( 0, 0, 10, 10 );
        win.add( o );
    }
    private void buttonOnePressed() {
        o.setBackground( Color.red );
    }
    private void buttonTwoPressed() {
        o.setBackground( Color.red );
    }
}
```

Another (not so great) solution: use conditional to check that all objects exist first.
Ensuring Objects Exist

There are many solutions to the problem, but they all boil down to the same thing: make sure all used objects are initialized.

```
public class Driver {
    private Window win = new Window();
    private Oval o = new Oval(0, 0, 10, 10);

    public Driver() {
        // public Driver()
    }

    private void buttonOnePressed() { 
        win.add(o);
    }

    private void buttonTwoPressed() {
        o.setBackground(Color.red);
    }
}
```

Another way: initialize objects when they are declared globally. They are again made at time of construction, without needing to put code in constructor itself.

```
Note: in such cases, when the constructor effectively does nothing (except for its default behavior of instantiating an object in memory), it can in fact be deleted.

The compiler will use a default constructor, with the same name as the class, and with no input parameters.
```

Communication in Methods

The main way of passing control in programs is via method calling.

“One-way communication”

```
public class Driver {
    // the main constructor
    public Driver() {
        method(6);
    }
}
```

May use parameter input, passing along needed objects and values to the called method.

```
// a helper method
private void method( int i ) {
    // do something with
    // input parameter i
}
```

How can we return information back?

“Two-way communication”

```
public class Driver {
    // the main constructor
    public Driver() {
        sumUp(3, 6);
        o1 = new Oval(10, 10, sum, sum);
    }

    // adds two inputs
    private void sumUp( int i, int j ) {
        sum = i + j;
    }
}
```

Using Instance Variables

We could use global instance variables to record any needed information:

1. Driver() calls sumUp().
2. Inputs added, recorded to the global sum variable.
3. Driver() draws circle with diameter == sum.

What disadvantages can this approach have?

```
public class Driver {
    private int sum;
    private Oval o1, o2;

    // the main constructor
    public Driver() {
        sumUp(3, 6);
        sumUp(4, 10);
        o1 = new Oval(10, 10, sum, sum);
        o2 = new Oval(20, 20, sum, sum);
    }

    // adds two inputs
    private void sumUp( int i, int j ) {
        sum = i + j;
    }
}
```

Using Instance Variables

One problem: multiple calls to the private method can overwrite sum variable.

Thus, these two methods do different things. (Can you see why?)

```
public class Driver {
    private int sum;
    private Oval o1, o2;

    // the main constructor
    public Driver() {
        sumUp(3, 6);
        sumUp(4, 10);
        o1 = new Oval(10, 10, sum, sum);
        o2 = new Oval(20, 20, sum, sum);
    }

    // adds two inputs
    private void sumUp( int i, int j ) {
        sum = i + j;
    }
}
```

```
public class Driver {
    private int sum;
    private Oval o1, o2;

    // the main constructor
    public Driver() {
        sumUp(3, 6);
        sumUp(4, 10);
        o1 = new Oval(10, 10, sum, sum);
        o2 = new Oval(20, 20, sum, sum);
    }

    // adds two inputs
    private void sumUp( int i, int j ) {
        sum = i + j;
    }
}
```
The return Statement

- We can mark the end of a method using simple code:

  ```java
  return;
  ```

- This will return control to the location in the code where the method was originally called.

- For simple `void` methods, we often do not bother.

  The method will automatically return control when it reaches the end of its code-block.

- Sometimes, however, we will want to explicitly return control, particularly when we want to end the method early.

  You did this, for example in a programming assignment, where any improper input caused the program to end immediately.

Coding a non-void Method

- We write a non-void method just like any other, except for 4 things:
  1. Method signature has some other type instead of `void`.
  2. The `return` statement must exist (optional in `void` methods).
  3. The `return` statement must be followed by some expression of the same type as given in the signature.
  4. The `return` statement must be the last thing the method does on any given execution run.

  ```java
  private Color getRandomColor() {
    int r = (int)(Math.random() * 256);
    int g = (int)(Math.random() * 256);
    int b = (int)(Math.random() * 256);
    Color col = new Color(r, g, b);
    return col;
  }
  ```

If the `return` statement is missing, is in the wrong place, or does not have the right type, the code will not compile.

Non-void Methods

- Another possibility is to use non-void methods.

  ```java
  private Color getRandomColor() {
    int r = (int)(Math.random() * 256);
    int g = (int)(Math.random() * 256);
    int b = (int)(Math.random() * 256);
    Color col = new Color(r, g, b);
    return col;
  }
  ```

  ```java
  public Driver() {  
    Oval o = new Oval(0, 0, 50, 50);  
    Color c = getRandomColor();  
    o.setBackground(c);  
  }
  ```

  If a method returns output of type X, we can call it wherever we can use a value of type X.

  E.g., if our code wants an `int`, we can use any method that returns an `int`: if it wants a `Color`, we use a method that returns a `Color`: etc.

  We “plug in” the method call, and after it runs, the returned value gets substituted in its place.
Using non-void Methods

- If a method returns output of type X, we can call it wherever we can use a value of type X.

```java
public Driver()
{
    Oval o = new Oval( 0, 0, 50, 50 );
    o.setBackground( getRandomColor() );
}

private Color getRandomColor()
{
    int r = (int)( Math.random() * 256 );
    int g = (int)( Math.random() * 256 );
    int b = (int)( Math.random() * 256 );
    Color col = new Color( r, g, b );
    return col;
}
```

Using non-void Methods

- If a method returns output of type X, we can call it wherever we can use a value of type X.

```java
public class Driver
{
    private int sum;
    private Oval o;
    // the main constructor
    public Driver()
    {
        int sum1 = sumUp( 3, 6 );
        o = new Oval( 10, 10, sum1, sum1 );
        int sum2 = sumUp( 4, 10 );
        o = new Oval( 10, 10, sum2, sum2 );
    }
    // adds two inputs
    private int sumUp( int i, int j )
    {
        return i + j;
    }
}
```

Using non-void Methods

- Notice that order doesn't matter.

```java
private int getRed()
{
    return new Color( r, g, b ).getRed();
}
```

Using non-void Methods

- Notice that order doesn't matter.

```java
public class Driver
{
    private Oval o;
    // the main constructor
    public Driver()
    {
        int sum1 = sumUp( 3, 6 );
        int sum2 = sumUp( 4, 10 );
        o = new Oval( 10, 10, sum1, sum1 );
        o = new Oval( 10, 10, sum2, sum2 );
    }
    // adds two inputs
    private int sumUp( int i, int j )
    {
        return i + j;
    }
}
```
This Week & Next

› **Meetings this week:**
  › Monday/Wednesday/Friday: regular classroom
  › Tuesday: in the CS Lab (16 Wing)

› **Reading 06:** Ch. 6 due Friday, Nov 8 at 4:00 PM

› **Program 06:** due 11:59 PM, Thursday November 7

› **Midterm 2:** Monday November 11

› **Office Hours:** Wing 212
  › Monday/Friday: 2:15 PM–3:15 PM
  › Tuesday: 2:30 PM–1:30 PM
  › Wednesday: 12:05 PM–1:00 PM