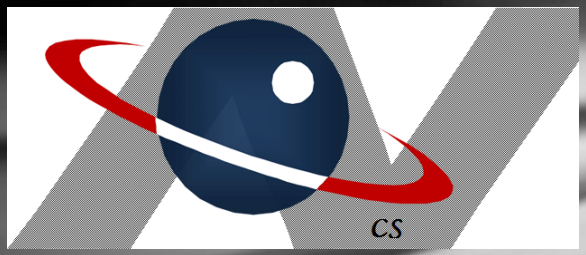
O(N) CS LESSONS

Lesson 10A – OOP Fundamentals



By John B. Owen

Table of Contents

Objectives

- In this lesson you'll learn how to use Java Objects.
- Objects are the fundamental building blocks for OOP, or Object Oriented Programming
- Mastering the use of Objects is essential to learning OOP, a fairly new programming paradigm that originated in C and was further developed in JAVA.

Definition

- An object in Java, or any other programming language, is simply an item of some sort that has **data characteristics**, as well as **methods** that belong to it, all encapsulated into its definition
- This definition is called a class

Definition

- The class is the description or blueprint of the object *(the idea, abstract, not yet real)*, just like the plans for a house, or recipe for a meal.
- The object is an instance of the class *(the actual thing)*, like the actual house, or the actual meal.

Definition - class

- The plans for a house include a list of materials, and the blueprints for assembling those materials.
- The recipe for a meal contains the list of ingredients, and the steps for mixing and cooking the meal.

Definition - class

- Likewise, a class has instance variables (the ingredients, materials) and methods (the processes that construct the object, initialize the instance variables, and that manipulate the data contained in the instance variables.)

Pointers vs containers

- Objects are *referenced*, or pointed to by object variables or object references.
- This is a **key difference** from how primitive variables work.

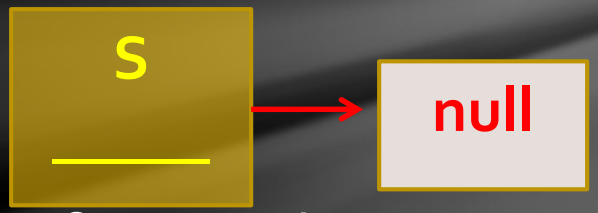
Pointers vs containers

- In a primitive, the variable actually "contains" the value.
- In an object, the variable "points to" or "references" the object that contains the value by holding its memory address.

Object vs primitives

- Let's explore this "pointer" idea …
  - Here is an example of an object variable for the String class

String s;



- s is an object variable or ***object reference*** that ***can*** point to or *reference* a String, but isn't doing so yet. Instead, it points to nothing, or null, a special place reserved in memory that an object reference points to when it has no object to point to. It must point somewhere!

Object vs primitives

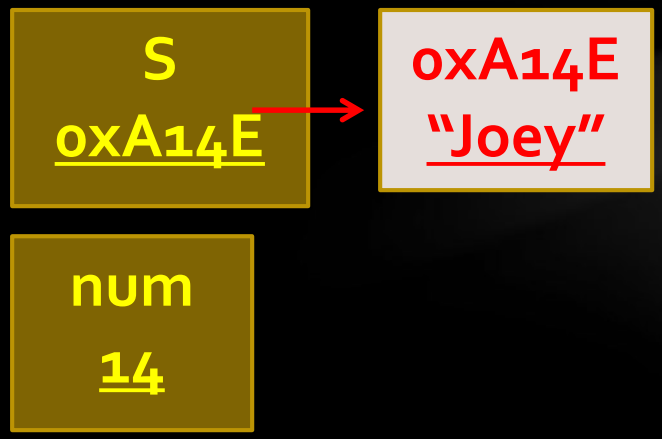
- int num; is a primitive variable that can ***contain*** (not "*point to*" or "*reference*") a primitive integer value, but doesn't yet. It "contains" nothing (but not null! Only objects can do that).

Object vs primitives

- Later, we'll discuss in more depth the fact that an int automatically contains zero in some cases, but only when it belongs to an object or a class.
- Likewise for a String…it only points to null automatically when it is a member of an object or a class.

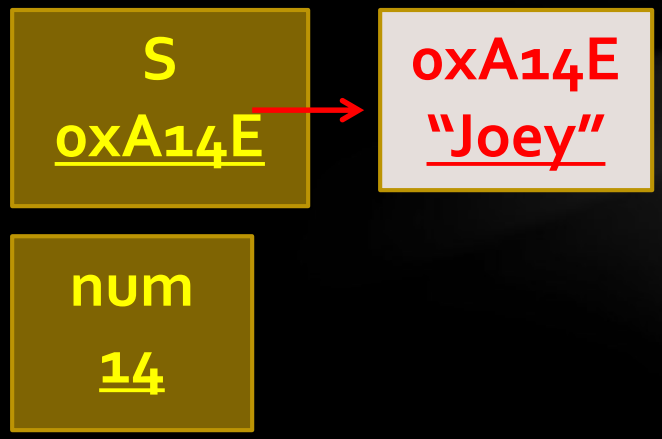
Object vs primitives

- Now we give each one something…
- s = new String (“Joey”);
- num = 14;



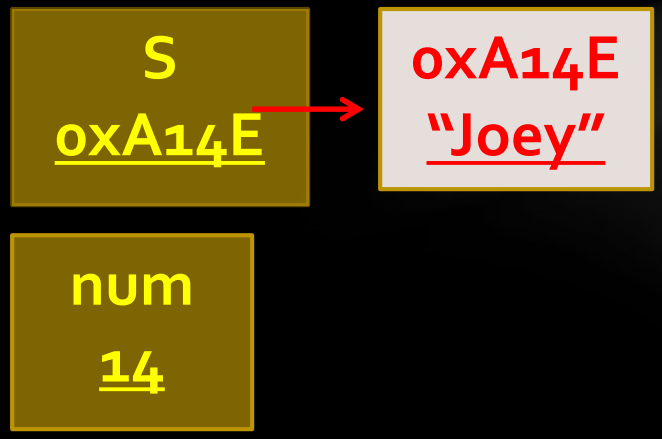
Object vs primitives

- s now references a new String object, and num contains an integer value.



Object vs primitives

- Notice the primary difference…
  - s contains the address to another memory location, which contains the value
  - num actually contains the value.



Class constructor

- To *point to* or *reference* the String variable s to a String object, one must be constructed using the *new* operator, like this:
- s = new String("Joey");
- **All objects** must be constructed according to their class definition using the *class constructor.*

Class constructor

- When the new command is invoked by the class constructor, new memory in RAM according to the required size of the object is allocated for that object, and values are placed into that memory location.

Primitive assignment

- On the other hand, primitives are NOT constructed, but merely assigned.
- Primitives are a predetermined memory size, and don't need this "new" process, another key difference between them and objects. Review Lesson 2A for a summary of all the different primitive memory requirements.

String class constructors

```
s = new String("Joey");
```

- This is only one way to construct a String.
- There are actually FIFTEEN different ways a String can be constructed, as defined in the String class, but we'll just deal with this one for now. *(See the Java API for more details on this.)*

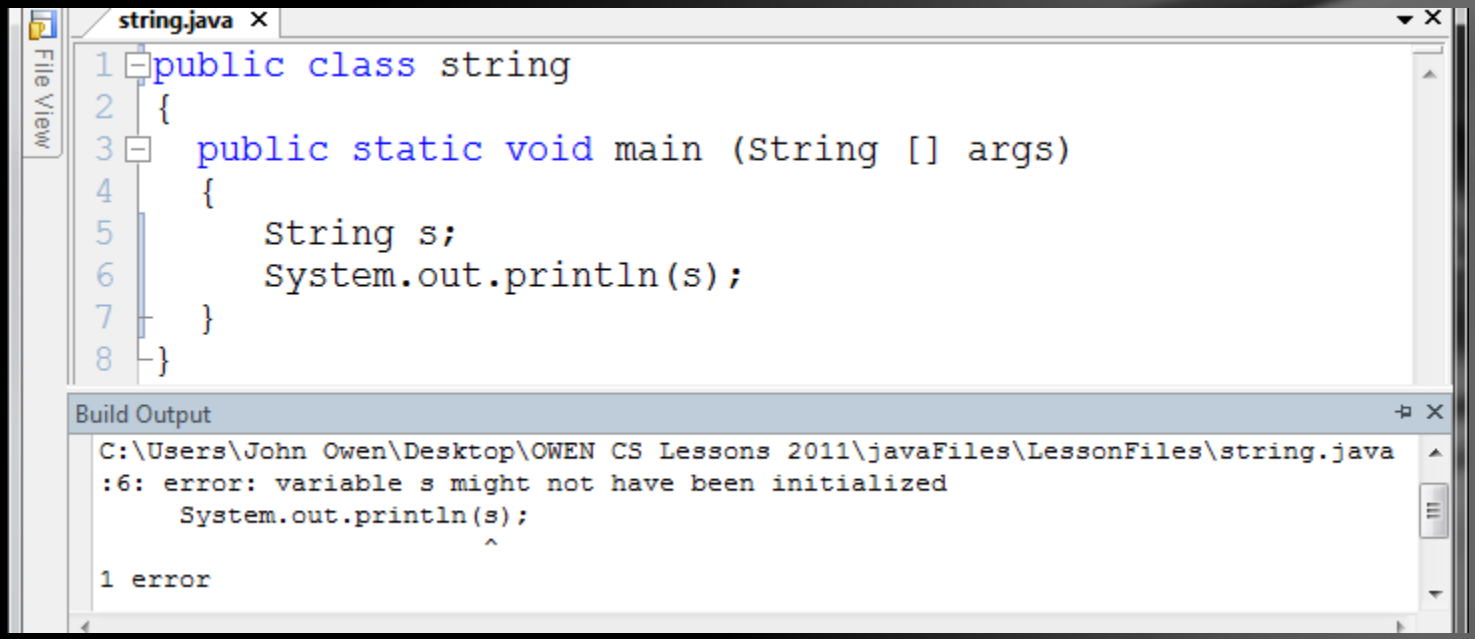
Methods

- Most objects also have methods that "belong" to them.
- You have used the String.length() method several times, which calculates and returns the length of the String to which it belongs.
- The String class has several other methods as well.

String output

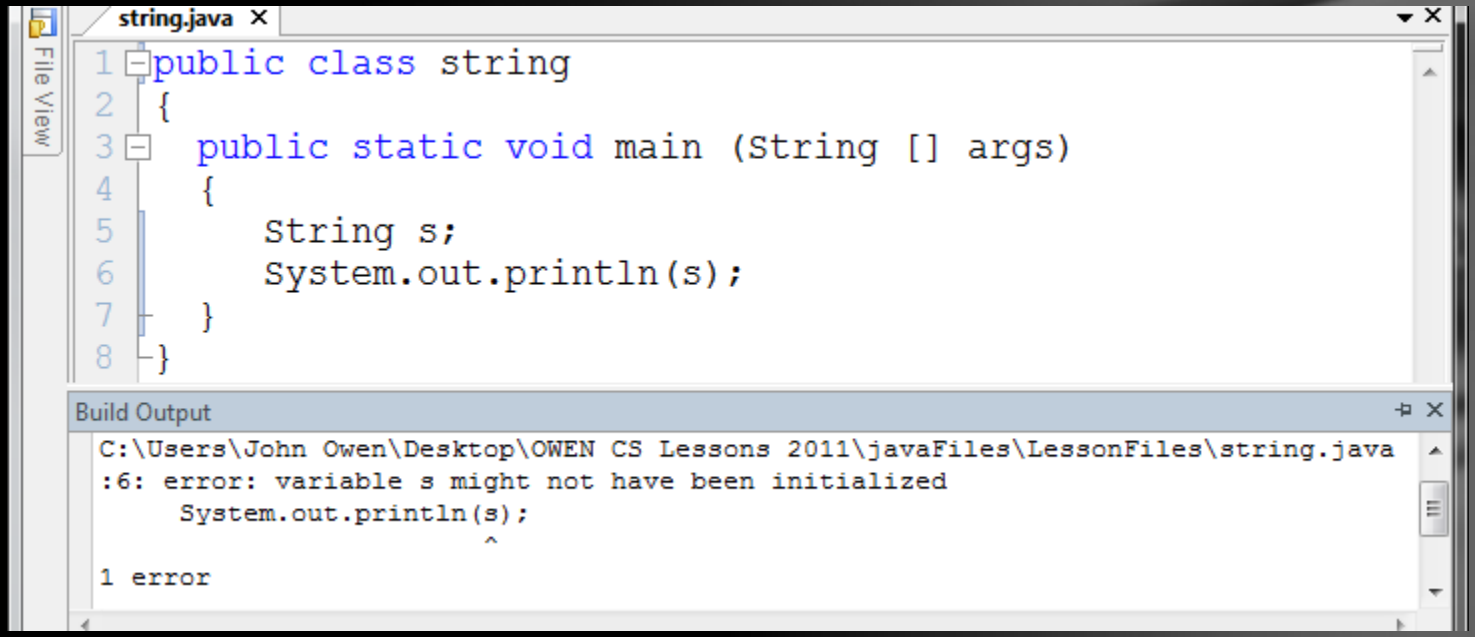
Let's explore some method examples to clarify.

In this program, the String object variable has not been initialized *(given a beginning value)*, and so the compiler reports an error.



String output

You might think it should output "null", since that was mentioned earlier, but it doesn't in this case. Here's why…the variable is declared inside the main method, and does not belong to an object or a class, in which case null is NOT automatically assigned.

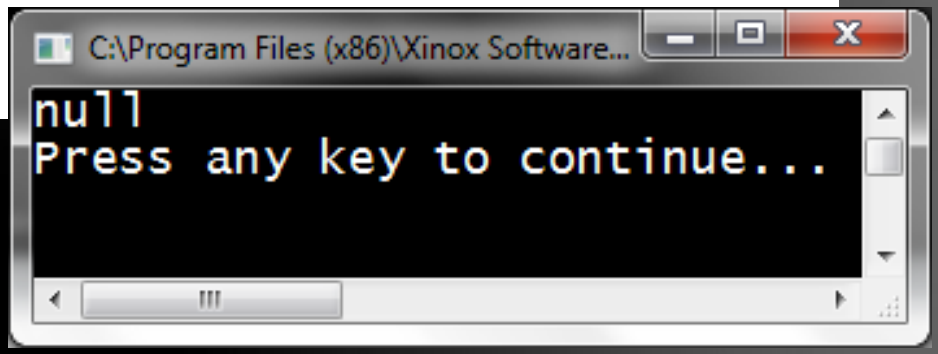


String output

However, if String s is declared outside the method but still inside the class as a class variable, it works! Note the word "static" in front…this is necessary to make it "belong" to the class, but not to any particular object.

string.java

```
1  public class string
2  {
3     static String s;
4     public static void main (String [] args)
5     {
6         System.out.println(s);
7     }
8  }
```

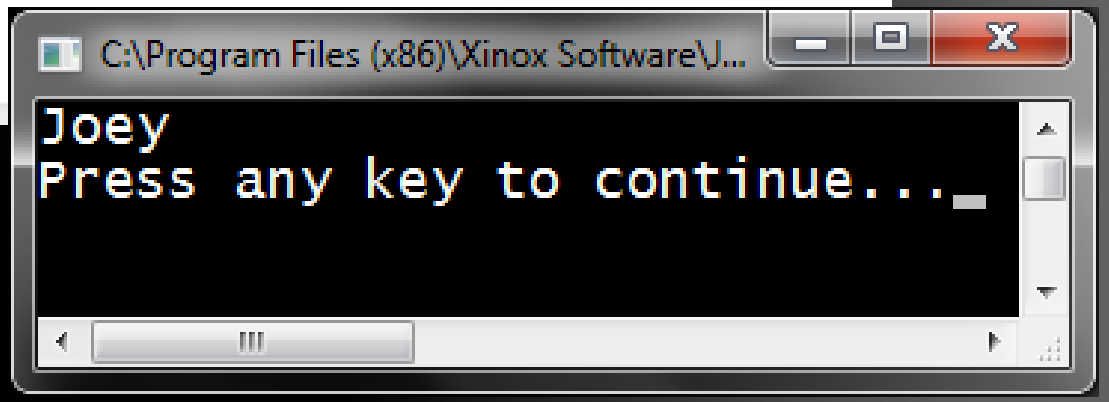


String output

Now we construct and output a String object!

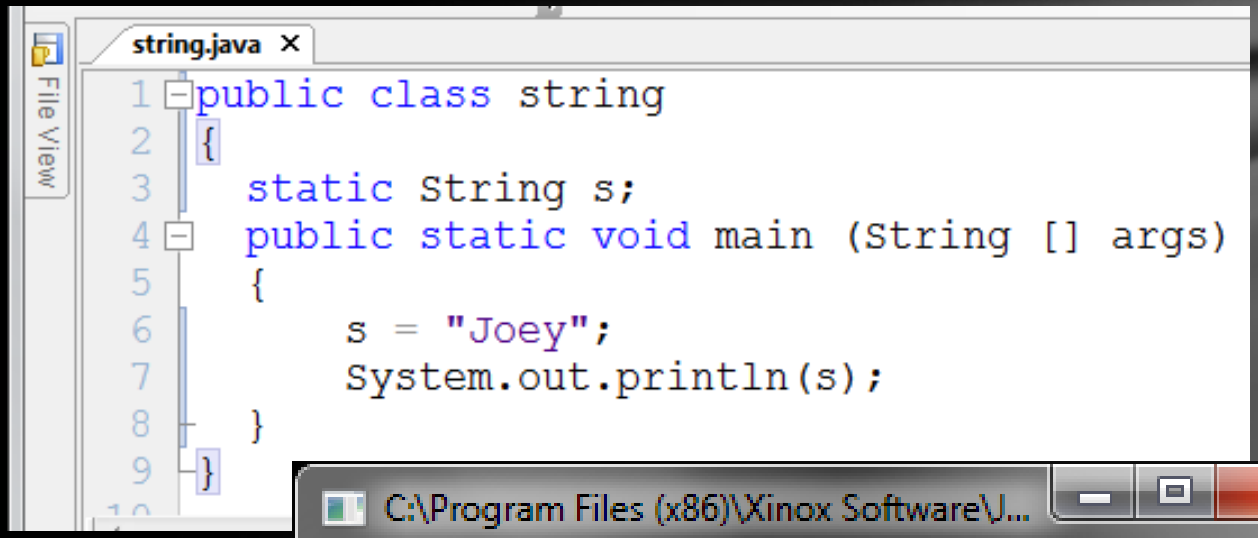
string.java

```
1  public class string
2  {
3     static String s;
4     public static void main (String [] args)
5     {
6          s = new String("Joey");
7          System.out.println(s);
8     }
9  }
```



String autoboxing

Here is a different way, which also works, but only for certain types of data…



String autoboxing

Here's why…since Strings are used so much in Java programming, the construction process is ***automatic*** when using this technique (s = "Joey";), often called "autoboxing".

This also works for Wrapper objects, which we'll cover later on.

String autoboxing

It's handy because it simplifies the coding process, not requiring the use of the "new" operator every time.

It also brings up a curious anomaly when using the "==" operator with Strings, which is important to know about.

Using "==" with Strings

With primitives, the "==" operator actually checks to see if two values are equal.

With objects, it checks to see if the two objects exist in the same memory location, in essence checking to see if two object variables are pointing to the same object.

Using "==" with Strings

Let's examine this anomaly in a program.

Below you see a predictable output…

string.java

```
1  public class string
2  {
3     static String s,t;
4     public static void main (String [] args)
5     {
6          s = "Joey";
7          t = new String("Joey");
8          System.out.println(s+" "+t+" "+s.equals(t)+" "+(s==t));
9     }
10 }
```

```
C:\Program Files (x86)\Xinox Software\JCreatorV5LE\...
Joey Joey true false
Press any key to continue...
```

Using "==" with Strings

The equals method returns true since the contents of each String is the same.

string.java

```
1  public class string
2  {
3     static String s,t;
4     public static void main (String [] args)
5     {
6          s = "Joey";
7          t = new String("Joey");
8          System.out.println(s+" "+t+" "+s.equals(t)+" "+(s==t));
9     }
10 }
```

```
C:\Program Files (x86)\Xinox Software\JCreatorV5LE\...
Joey Joey true false
Press any key to continue...
```

Using "==" with Strings

However, the "==" operator shows false because the objects are in different places in memory.

string.java

```
1  public class string
2  {
3     static String s,t;
4     public static void main (String [] args)
5     {
6          s = "Joey";
7          t = new String("Joey");
8          System.out.println(s+" "+t+" "+s.equals(t)+" "+(s==t));
9     }
10 }
```

```
C:\Program Files (x86)\Xinox Software\JCreatorV5LE\...
Joey Joey true false
Press any key to continue...
```

Using "==" with Strings

Here's the anomaly…observe. Can you figure it out? How does the "==" show true?

string.java

```
1  public class string
2  {
3     static String s,t;
4     public static void main (String [] args)
5     {
6          s = "Joey";
7          t = "Joey";
8          System.out.println(s+" "+t+" "+s.equals(t)+" "+(s==t));
9     }
10 }
```

```
C:\Program Files (x86)\Xinox Software\JCreatorV5LE\...
Joey Joey true true
Press any key to continue...
```

Using "==" with Strings

Here's how…there is a special place in memory for Strings, called common memory. When Strings are constructed using the "autoboxing" process, the values are placed in this area, and any duplicates simply reference the same word in that area of memory, thus having the same address, causing the "==" to be true.

string.java

```
1  public class string
2  {
3     static String s,t;
4     public static void main (String [] args)
5     {
6          s = "Joey";
7          t = "Joey";
8          System.out.println(s+" "+t+" "+s.equals(t)+" "+(s==t));
9     }
10 }
```

```
C:\Program Files (x86)\Xinox Software\JCreatorV5LE\...
Joey Joey true true
Press any key to continue...
```

Object class…back to the beginning

- Before moving on, we have to go back to the beginning, back to the motherland, the origin of all objects, the **Object class**!
- The class defined as Object is the most basic class there is.
- All other classes are based on this.

Object class

- An instance of the String class is also an instance of the Object class
- This may sound confusing, but you'll get used to it.
- All Java class definitions originate from the Object class

Object class

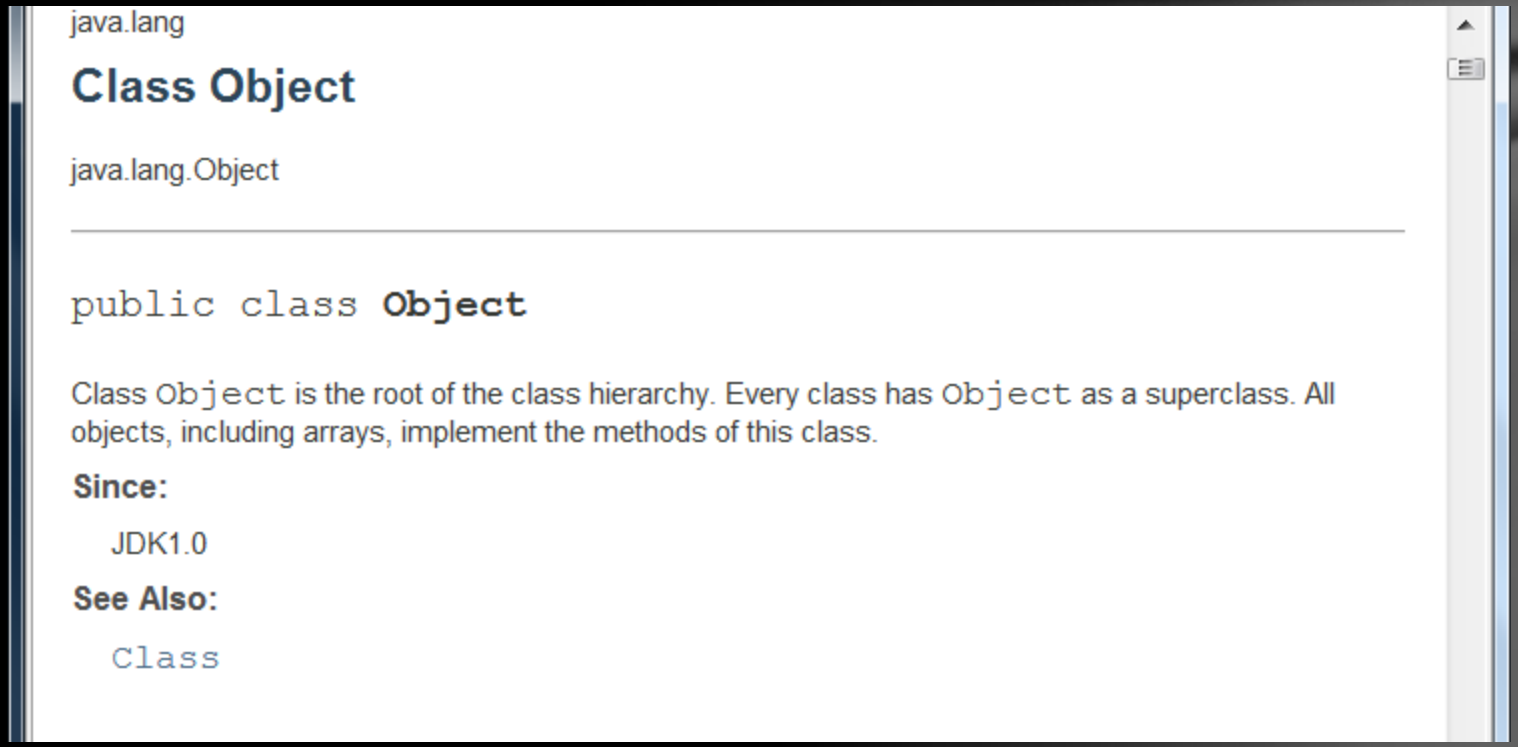
- Here is the basic class definition for the Object class, as described in the Java documentation

*public class **Object***

*Class Object is the root of the class hierarchy. Every class has Object as a **superclass**. All objects, including arrays, implement the methods of this class.*

Object class

- Here is the API for the Object class



java.lang

Class Object

java.lang.Object

`public class` **`Object`**

Class `Object` is the root of the class hierarchy. Every class has `Object` as a superclass. All objects, including arrays, implement the methods of this class.

Since:

JDK1.0

See Also:

`Class`

Object construction

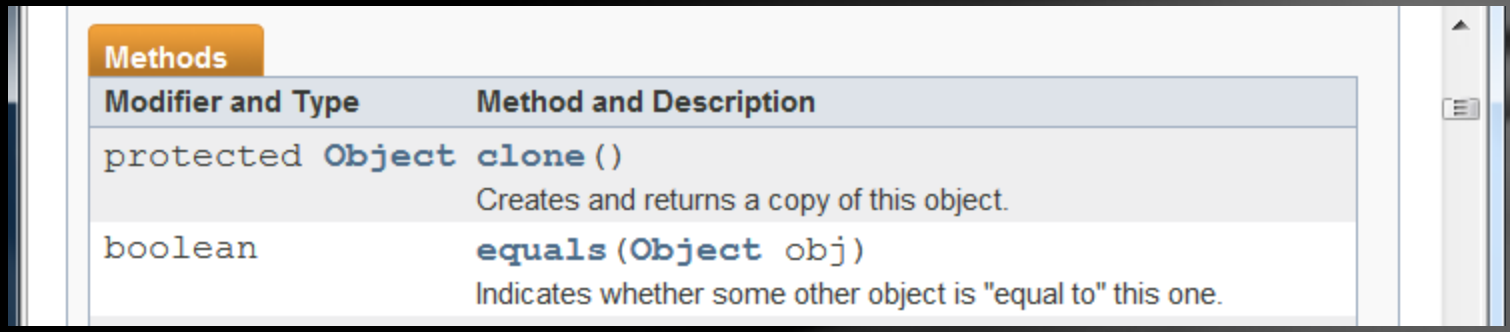
- The Object class has only one constructor

  Object()

- It has no actual data to speak of, which may seem strange, but there is a good reason which you'll learn about later

Object methods

- It does "own" several methods, among which are the following:

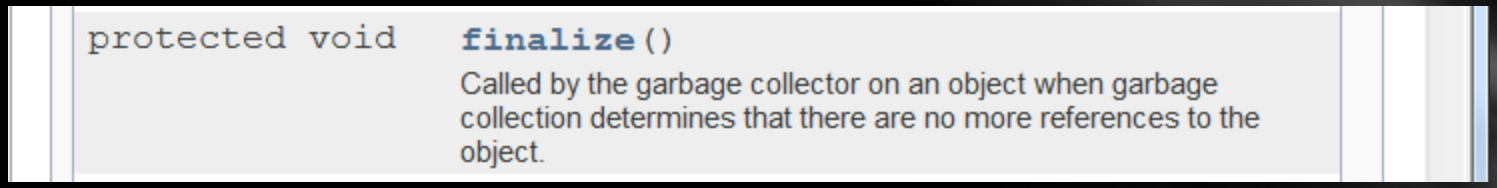


Methods

Modifier and Type	Method and Description
`protected Object`	**`clone`**`()` Creates and returns a copy of this object.
`boolean`	**`equals`**`(Object obj)` Indicates whether some other object is "equal to" this one.

- **Object clone()** …Creates and returns a copy of the current object.
- boolean **equals** (Object obj) …Indicates whether some other object is "equal to" the current one.

Object methods

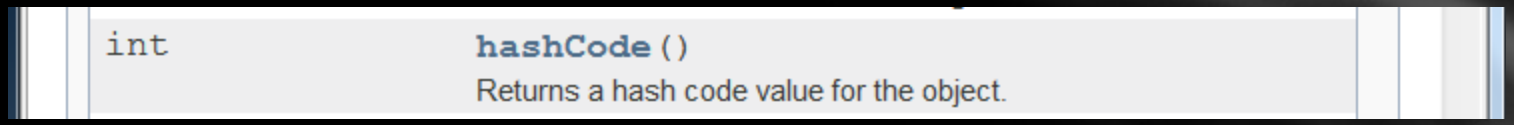


protected void **finalize()**

Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.

- void **finalize**()…Called by the ***garbage collector*** *(an internal Java process)* on an object when the garbage collector *(part of the computer process)* senses that there are no more references to this object, in other words, it is free floating in RAM, and the allocated memory needs to be recycled for reuse.
- *You will most likely never actually use this method in your programming process…it happens internally.*

Object methods



- int **hashCode**()…A method that generates a unique value that represents this object in memory, sort of like a Social Security Number.
- Every different object constructed or instantiated during the execution of the program will have its own unique hash code value.

Object methods

- String **toString**()…Returns a string representation of this object.
- There are several other Object methods, but we'll just look at these five for now.

Back to the String class

- Now that we know the origin of objects, let's return to the String class, which is a descendent, or sub-class, of the Object class
- The Object class is a super-class, or ancestor of the String class

Inheritance

- The String class "inherits" all the characteristics of the Object class, namely the methods.
- This means the String class owns all of the same methods as the Object class, including clone(), equals(), finalize(), hashCode(), and toString().

Inheritance

- It also has several methods of its own, in addition to the Object ones, like length(), substring(), charAt(), and indexOf(), just to name a few.

Sample Program

Let's look at a program that will explore the relationship between Objects and Strings…

Sample Program

go.java *

```
1  //Program to demonstrate Objects
2  public class go
3  {
4      public static void main(String [] args)
5      {
6          //This is an Object object variable, which
7          //is simply a pointer that can point to
8          //an instance of an Object, but it doesn't
9          //point to anything yet, nor does it
10         //contain anything.
11         Object o;
12         System.out.println(o);
13     }
14 }
```

Task List

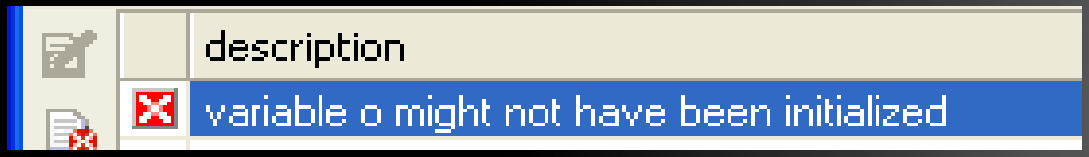
description	source	location
variable o might not have been initialized	java	line 12

It didn't compile?? What the heck?!

Sample Program

This program does not compile!!!

If you try to output an object variable that doesn't point to anything, the compiler will "yell" at you with the following error message, "*variable o might not have been initialized*", which means, "*you can't output an object variable that isn't pointing to anything yet!*"



Sample Program

go.java *

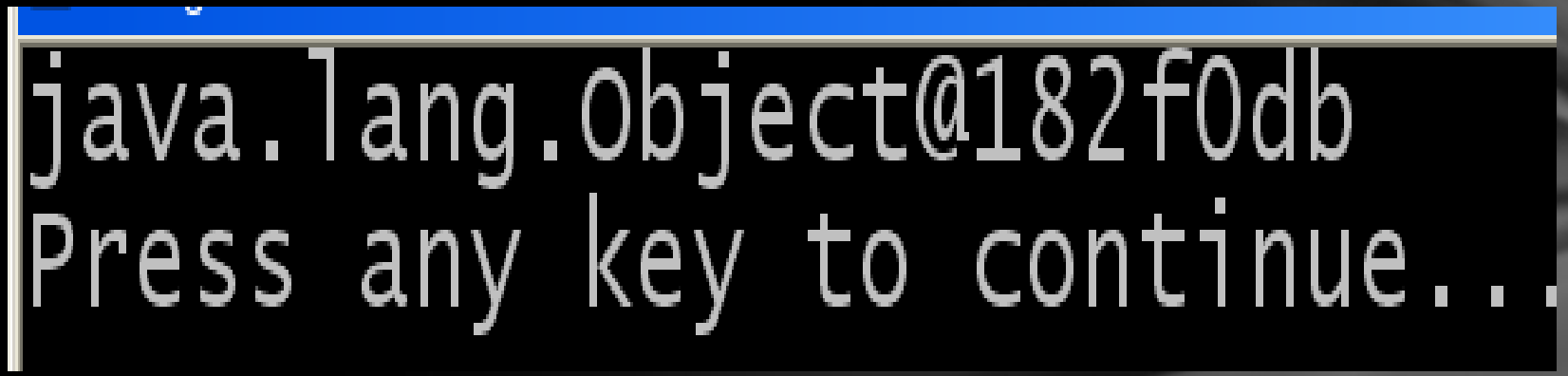
```
1  //Program to demonstrate Objects
2  public class go
3  {
4      public static void main(String [] args)
5      {
6          //This is an Object object variable, which
7          //is simply a pointer that can point to
8          //an instance of an Object, but it doesn't
9          //point to anything yet, nor does it
10         //contain anything.
11         Object o;
12         //This constructs a new Object and
13         //assigns it to the Object variable o
14         o = new Object();
15         //This outputs the Object
16         System.out.println(o);
17     }
18 }
```

Sample Program

```
java.lang.Object@182f0db
Press any key to continue...
```

- This program does run, and this is the output, which looks kind of strange.

Sample Program



- The output simply indicates the class it belongs to and the hexadecimal representation of the hash code of the object (more on hash codes later)

Sample Program

Now we'll mix things up a bit.

You saw earlier how a String object behaves

Now you will see that an ***Object*** object variable can "reference" or "point to" a String

Sample Program

go.java

```
1  //Program to demonstrate Objects
2  public class go
3  {
4      public static void main(String [] args)
5      {
6          Object o;
7          o = new Object();
8          //This outputs the Object
9          System.out.println(o);
10         //The superclass Object object variable can point
11         //to a String subclass object
12         o = new String("hello");
13         System.out.println(o);
14     }
15 }
```

```
java.lang.Object@182f0db
hello
Press any key to continue...
```

- At first the Object variable o references an Object, which when output shows the hashcode, as we saw previously.

Sample Program

go.java

```
1  //Program to demonstrate Objects
2  public class go
3  {
4      public static void main(String [] args)
5      {
6          Object o;
7          o = new Object();
8          //This outputs the Object
9          System.out.println(o);
10         //The superclass Object object variable can
11         //to a String subclass object
12         o = new String("hello");
13         System.out.println(o);
14     }
15 }
```

```
java.lang.Object@182f0db
hello
Press any key to continue...
```

- When o is re-referenced, or pointed at a new String, the output is different!

Sample Program

The words **superclass** and **subclass** can thought of as a "mother" and "child" relationship

go.java

```
1  //Progra                    Objects
2  public c
3  {
4      publ                   in(String [] a
5      {
6          O
7          o      ew Object();
8          //     ls outputs the Object
9          Sy     stem.out.println(o);
10         //The superclass Object object variable can
11         //to a String subclass object
12         o = new String("hello");
13         System.out.println(o);
```

- W   s re-
  d, or pointed at a new String, the output is different!

```
java.lang.Object@182f0db
hello
Press any key to continue...
```

Object recycling

- Now let's stop and think about something for a moment (another detour).
- The object variable o was pointing to an Object object at first, then it was "re-pointed", or referenced to a String object.

Object recycling

- What happens when an object variable like o "lets go" of its current object and "points to", or is referenced to another object, like the String object in this case?

Object recycling

- Answer: the object that is released by the object variable and is not "pointed to" or referenced anymore is automatically "recycled", and the memory it occupied can be used again by another object.

Object recycling

- The method that does this is the finalize() method you saw earlier, which EVERY object owns because it is a method that belongs to the Object class, which is the "superclass" or *mother class* for all Java classes.

Object recycling

- void **finalize**()…Called (automatically) by the garbage collector on an object when garbage collection determines that there are no more references to the object.

Sample Program 2 – pointing "up"

- Now that you understand object recycling, let's continue our lesson.
- You just saw that on Object variable can point to a String object with no problem.
- Can a String object variable point "up" to, or reference, an Object object?

Sample Program 2 – pointing "up"

go.java

Compile File

```
1  //Program to demonstrate Objects
2  public class go
3  {
4      public static void main(String [] args)
5      {
6          Object o;
7          o = new Object();
8          //This outputs the Object
9          System.out.println(o);
10         //The superclass Object object variable can
11         //to a String subclass object
12         o = new String("hello");
13         System.out.println(o);
14         //Attempt to use a String object variable
15         //to reference an Object object
16         String s = o;
17     }
18 }
```

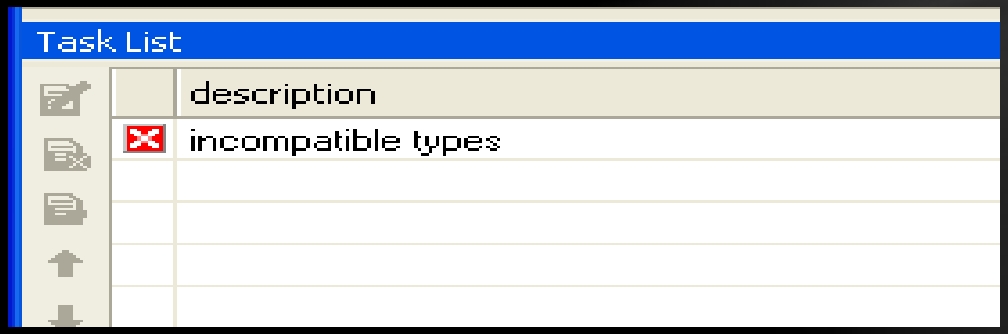
Task List

description	resource	location
incompatible types	.java	line 16

Not again!
What the hey?

Sample Program 2 – pointing "up"

- As you can see, this did not work…"incompatible types" is the error message!



Sample Program 2 – pointing "up"

- Even though o is pointing to a String object, the String object variable s cannot point to the same String object that o is pointing to!
- Why not!

Sample Program 2 – pointing "up"

- Here's why…
- Since the Object variable can point to ANY object, the compiler cannot be guaranteed that it points to a String in this case, therefore it will not allow this assignment to occur, even though it would work in this case.

Sample Program 2 – pointing "up"

- However, there is a way get around this problem by casting, an important technique you learned in an earlier lesson.
- String s = (String)o;
- This should now work.

Sample Program 2 – pointing "up"

go.java

```
1  //Program to demonstrate Objects
2  public class go
3  {
4      public static void main(String [] args)
5      {
6          Object o;
7          o = new Object();
8          //This outputs the Object
9          System.out.println(o);
10         //The superclass Object object variable can
11         //to a String subclass object
12         o = new String("hello");
13         System.out.println(o);
14         //Attempt to use a String object variable
15         //to reference an Object object.
16         String s = (String)o;
17         System.out.println(s);
18     }
19 }
```

Build Output

```
--------------------Configuration: <Default>--------------------

Process completed.
```

Sample Program 2 – pointing "up"

- And it did!
- Here's the output…

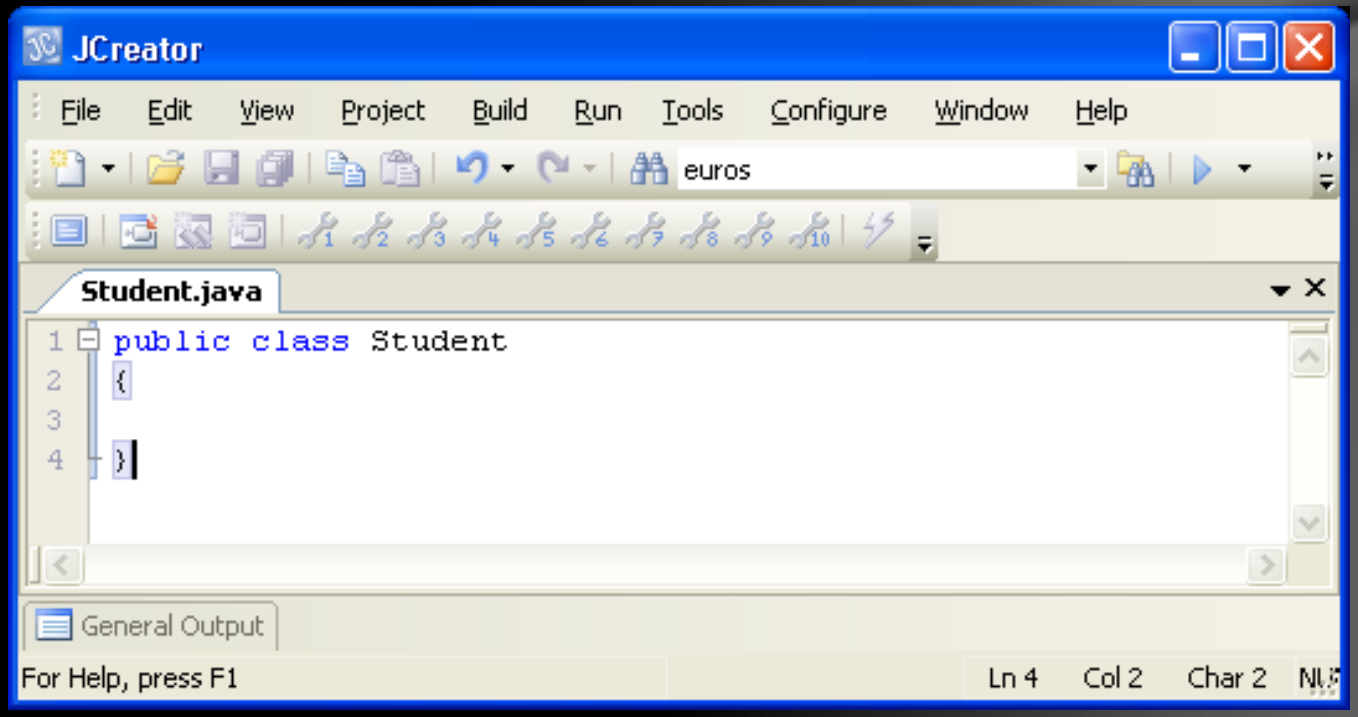
```
C:\Program Files\Xinox Software\JCreatorV3 LE\GE2001.exe
java.lang.Object@182f0db
hello
hello
Press any key to continue...
```

A Student class

- Now let's attempt to design our own class.
- First we must decide on a type of object to design.
- Let's do something familiar, like a Student!
- We're going to design a class definition for YOU!!!

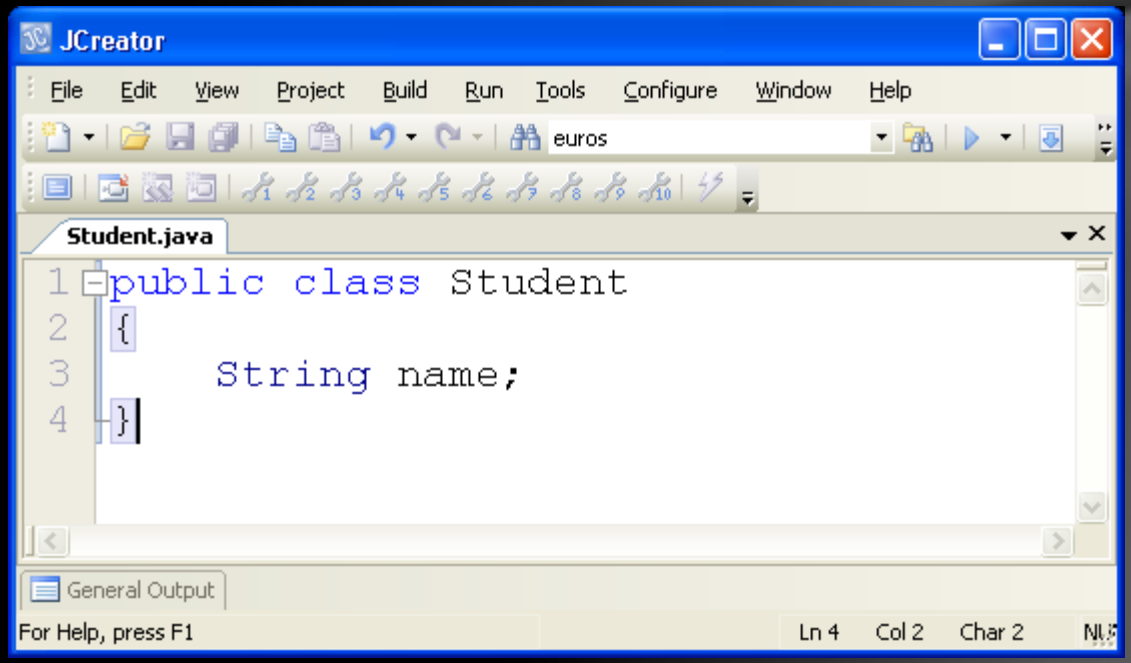
A Student class

- First, we set out the heading, which by convention is capitalized and set as public.



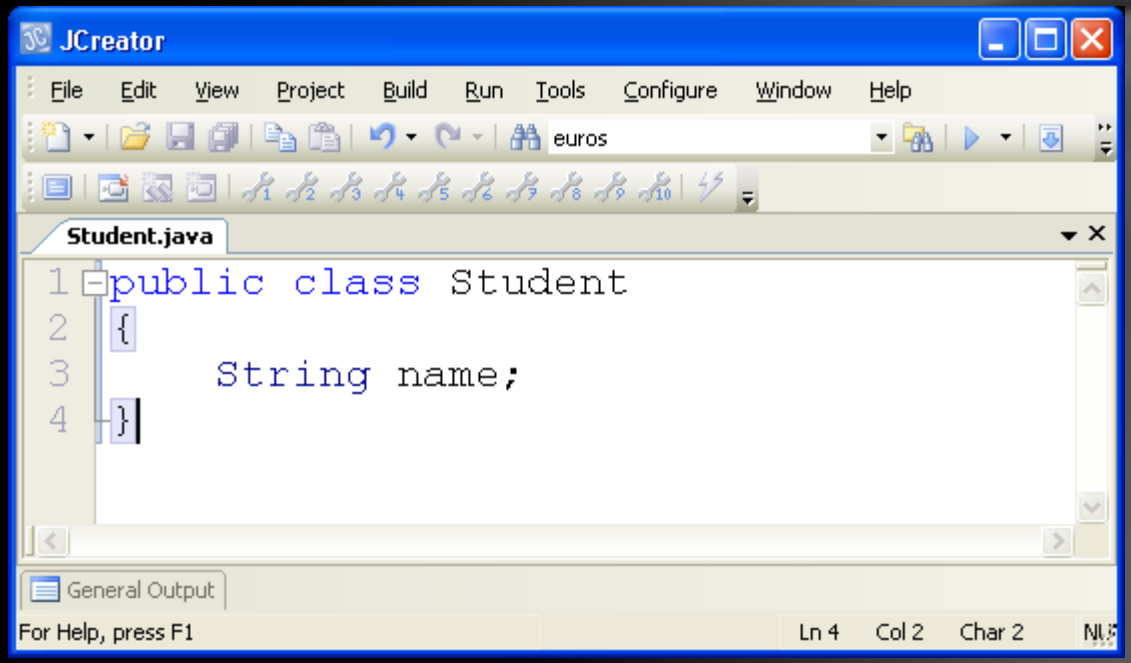
A Student class

- Next we need to decide what characteristics go with a typical Student…like your name!



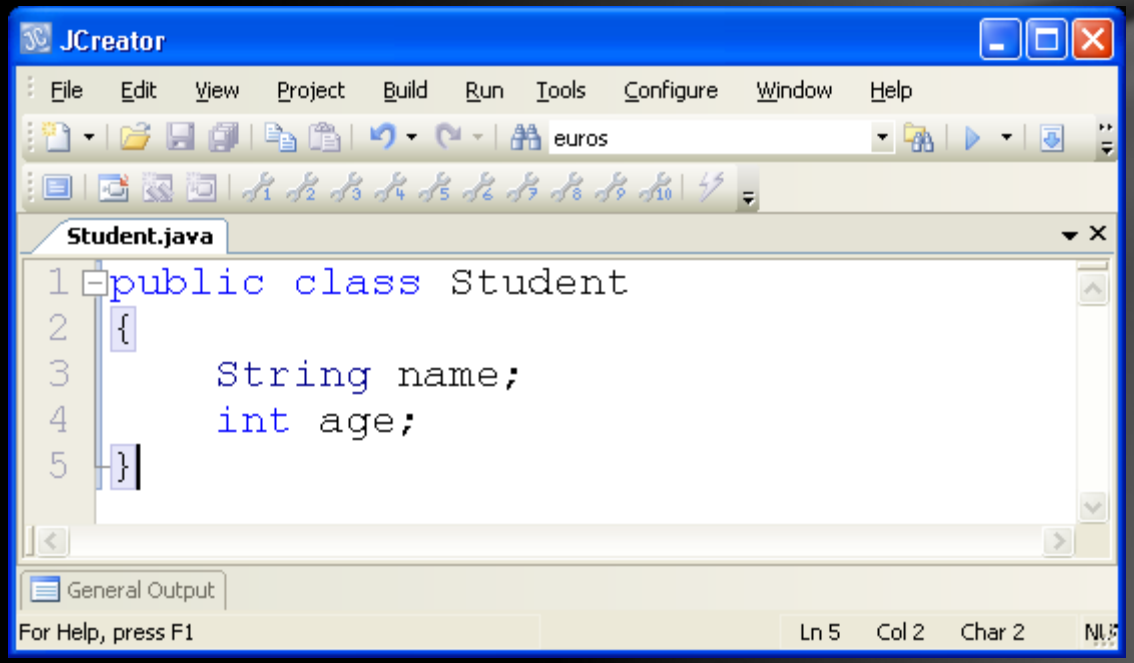
A Student class

- Data variables that belong to an object are called instance variables or fields.



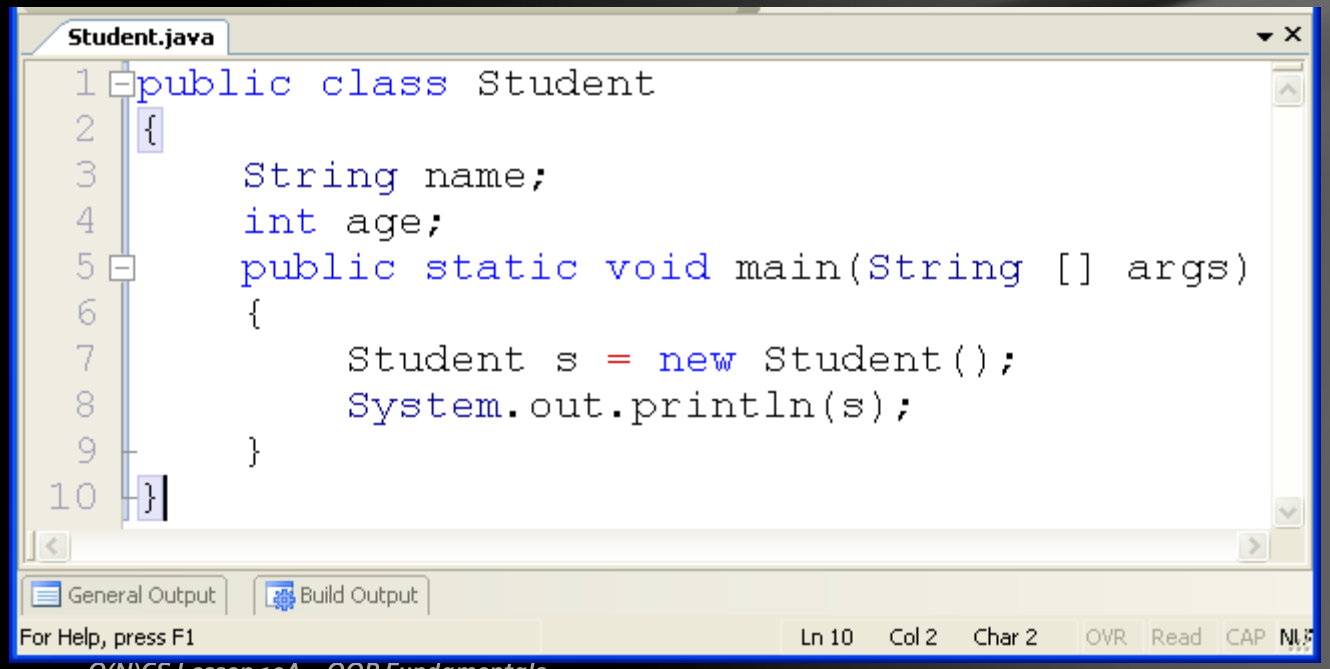
A Student class

- An object can own as many instance variables as needed.
- Let's add another…your age.



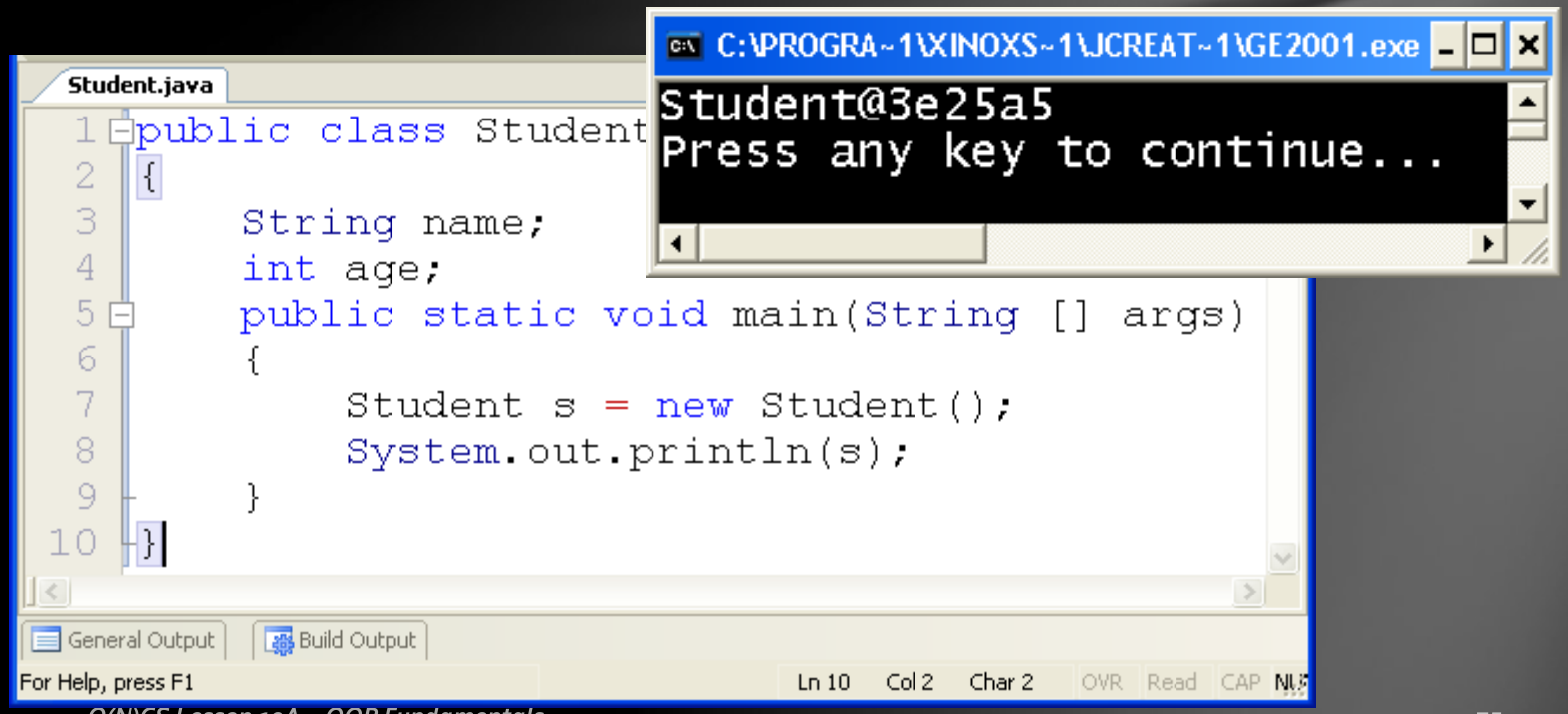
A Student class

- Now let's just see what happens if we create a Student object and output it.



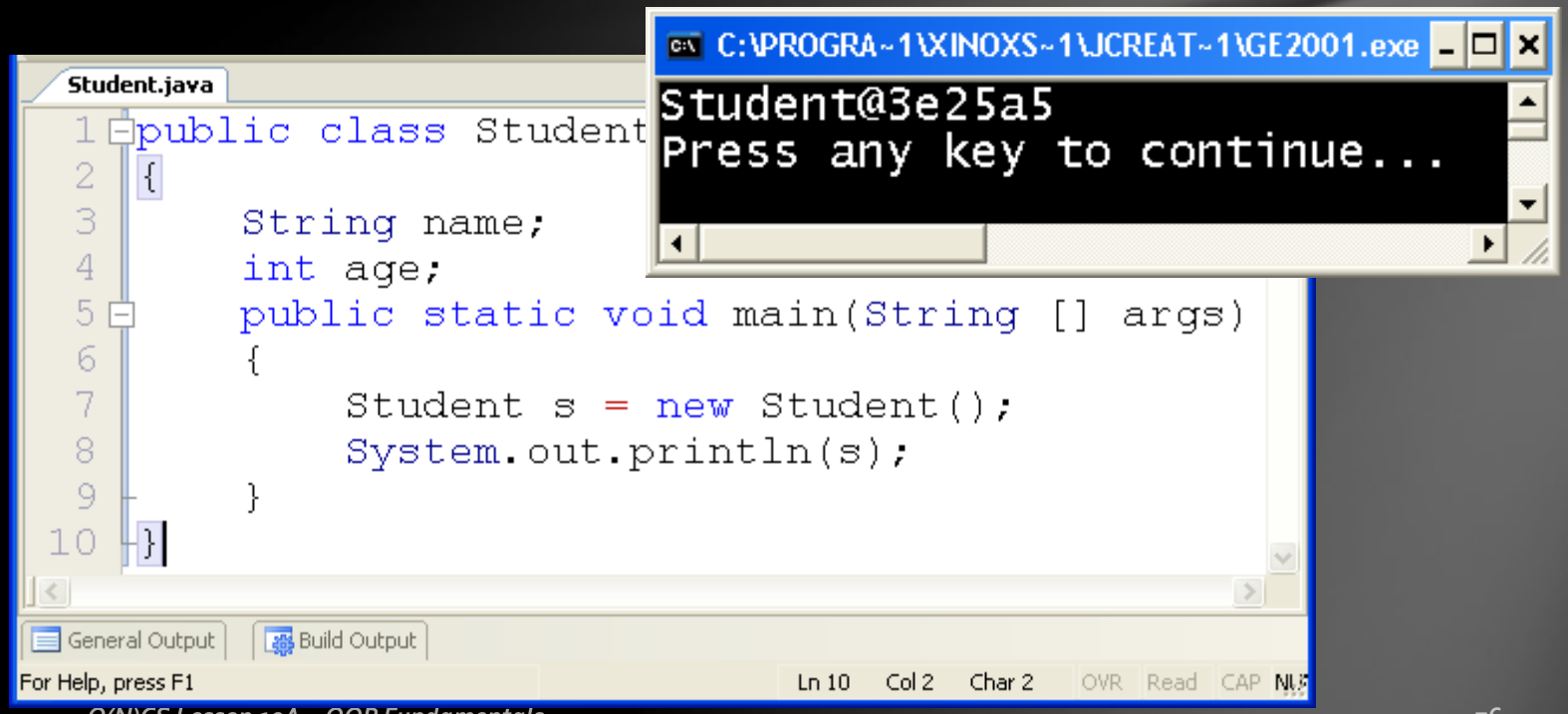
A Student class

- To do this we simply add the main method to the Student class, create a Student object, and output it.



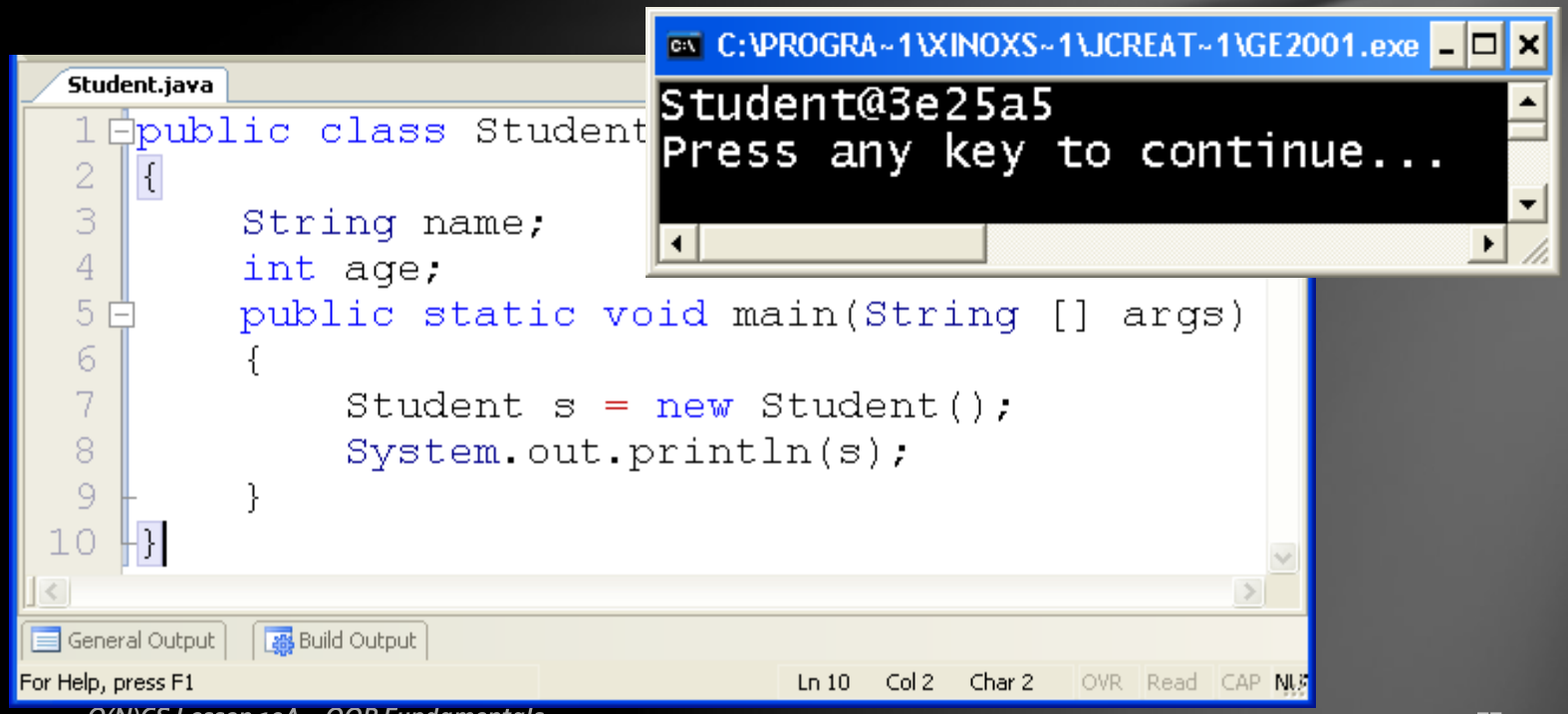
A Student class

- As you can see, the program worked just fine, but the output is not exactly what we want.



A Student class

- As it did for the Object output we saw earlier, the name of the class and the hex value hash code are shown.



Overriding the toString() method

- We need to further develop our class to adjust (or override) the output method called toString(), inherited from the Object class, which will allow us to customize the output for our Student object.
- Let's make it include the name and age in the output string.

Overriding the toString() method

- Here is the result…

Student.java

```
1  public class Student
2  {
3      String name;
4      int age;
5      public String toString()
6      {
7          return "Name="+name+" Age="+age;
8      }
9      public static void main(String [] args)
10     {
11         Student s = new Student();
12         System.out.println(s);
13     }
14 }
```

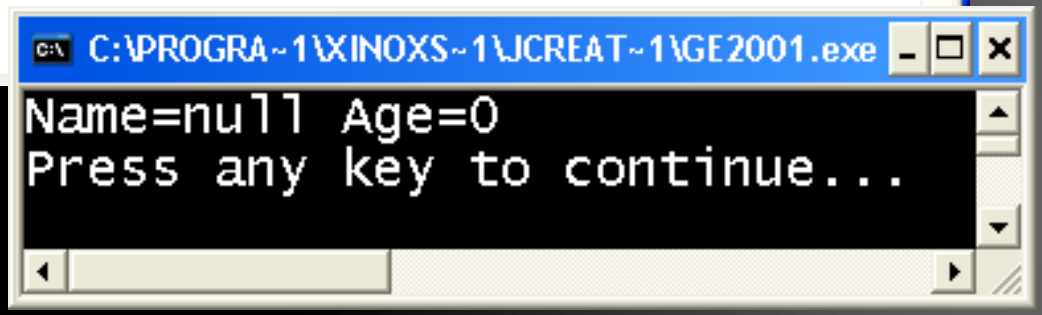
```
C:\PROGRA~1\XINOXS~1\JCREAT~1\GE2001.exe
Name=null Age=0
Press any key to continue...
```

Overriding the toString() method

- The structure of the toString() method, inherited from the Object class, is shown below. A class can customize this method, or **override** it to fit it's own special needs.

Student.java

```
1  public class Student
2  {
3      String name;
4      int age;
5      public String toString()
6      {
7          return "Name="+name+" Age="+age;
8      }
9      public static void main(String [] args)
10     {
11         Student s = new Student();
12         System.out.println(s);
13     }
14 }
```

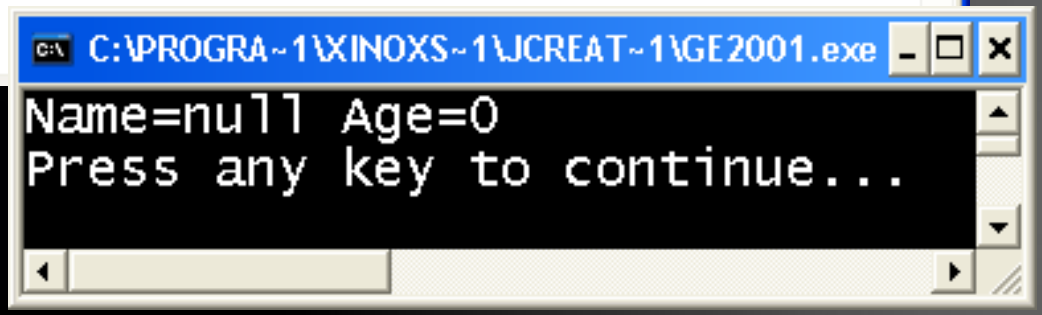


Overriding the toString() method

- It has a header, public String toString(), a pair of brackets, and a return statement which can return any **String** you want, but usually includes the instance fields of the object in some way, like this one.

Student.java

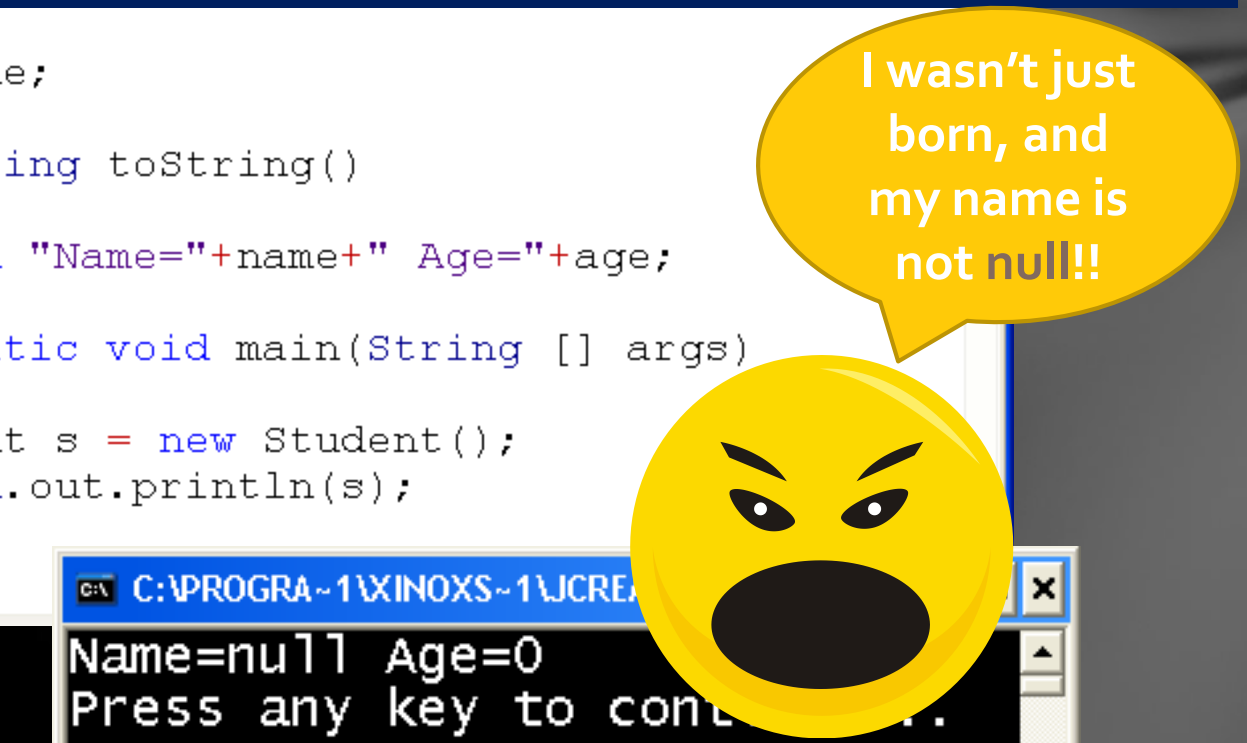
```
1  pub
2  {
3      String name;
4      int age;
5      public String toString()
6      {
7          return "Name="+name+" Age="+age;
8      }
9      public static void main(String [] args)
10     {
11         Student s = new Student();
12         System.out.println(s);
13     }
14 }
```



Overriding the toString() method

- Unfortunately, the values in name and age are not really appropriate, primarily since we haven't given them starting values. Instead, since they are members of this class and belong to this object, the compiler gives them the default values for Strings and integers, null and zero.

```
2  {
3      String name;
4      int age;
5      public String toString()
6      {
7          return "Name="+name+" Age="+age;
8      }
9      public static void main(String [] args)
10     {
11         Student s = new Student();
12         System.out.println(s);
13     }
14 }
```

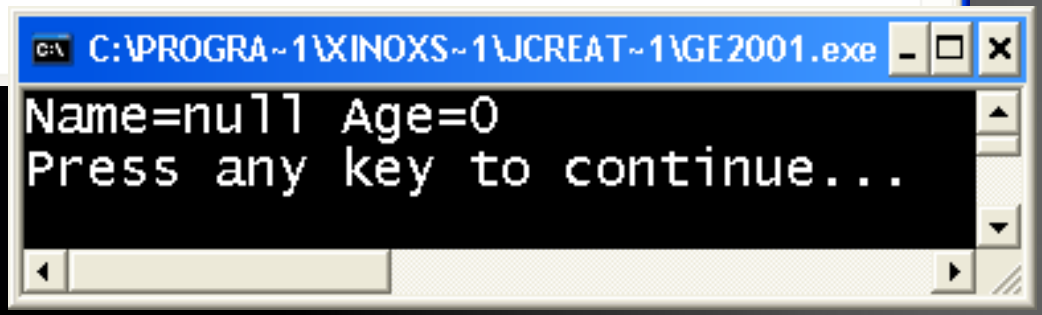


Student class constructor

- To remedy this problem, we must create another method for the Student class called a **constructor**.
- The job of the constructor is simple: to create an object of the class and assign beginning values to the instance fields.

Student.java

```
1  pub
2  {
3      String name;
4      int age;
5      public String toString()
6      {
7          return "Name="+name+" Age="+age;
8      }
9      public static void main(String [] args)
10     {
11         Student s = new Student();
12         System.out.println(s);
13     }
14 }
```

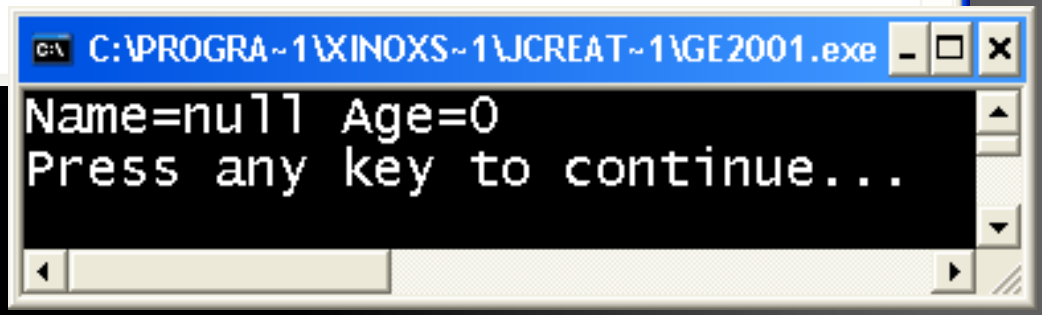


Student class constructor

- However, before we do that, you might be wondering how we were able to construct the Student object shown below ***without defining a constructor***...

Student.java

```
1  public class Student
2  {
3      String name;
4      int age;
5      public String toString()
6      {
7          return "Name="+name+" Age="+age;
8      }
9      public static void main(String [] args)
10     {
11         Student s = new Student();
12         System.out.println(s);
13     }
14 }
```

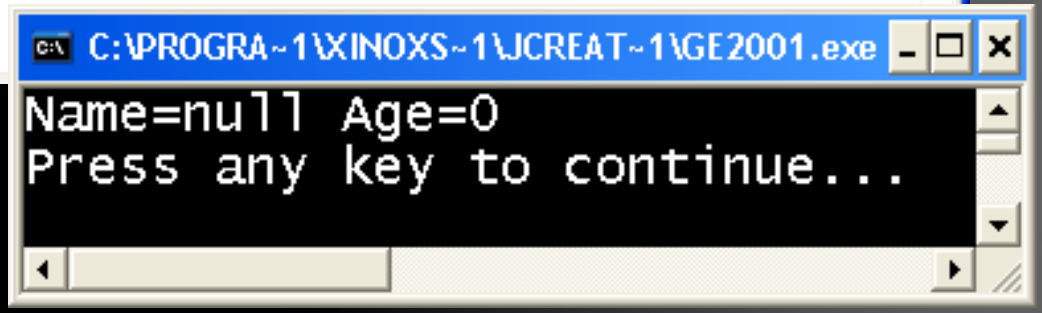


Student class constructor

- The answer to that is simple…the compiler uses a default constructor when none is defined, which automatically gives default values to any instance fields, as we see here.

Student.java

```
1  public class Student
2  {
3      String name;
4      int age;
5      public String toString()
6      {
7          return "Name="+name+" Age="+age;
8      }
9      public static void main(String [] args)
10     {
11         Student s = new Student();
12         System.out.println(s);
13     }
14 }
```



Student class constructor

- Here is an example of a default constructor that can be explicitly defined which gives an actual name and age to the object instead of just null and zero.

Student.java

```
1  public class Student
2  {
3      String name;
4      int age;
5      public Student()
6      {
7          name = "Joe";
8          age = 16;
9      }
10     public String toString()
11     {
12         return "Name="+name+" Age="+age;
13     }
14     public static void main(String [] args)
15     {
16         Student s = new Student();
17         System.out.println(s);
18     }
19 }
```

```
C:\PROGRA~1\XINOXS~1\JCREAT~1\GE2001.exe
Name=Joe Age=16
Press any key to continue...
```

Student class constructor

- The basic structure of a default constructor is the **header**, public Student(), with the word **public**, always followed by the **name of the class**, ***no*** "void" or return type, and no formal parameters (empty parentheses).

Student.java

```
1  public class Student
2  {
3      String name;
4      int age;
5      public Student()
6      {
7          name = "Joe";
8          age = 16;
9      }
10     public String toString()
11     {
12         return "Name="+name+" Age="+age;
13     }
14     public static void main(String [] args)
15     {
16         Student s = new Student();
17         System.out.println(s);
18     }
19 }
```

```
C:\PROGRA~1\XINOXS~1\JCREAT~1\GE2001.exe
Name=Joe Age=16
Press any key to continue...
```

Student class constructor

- As you can see by the output, the default constructor we defined now takes over from the default compiler version.

Student.java

```
1  public class Student
2  {
3      String name;
4      int age;
5      public Student()
6      {
7          name = "Joe";
8          age = 16;
9      }
10     public String toString()
11     {
12         return "Name="+name+" Age="+age;
13     }
14     public static void main(String [] args)
15     {
16         Student s = new Student();
17         System.out.println(s);
18     }
19 }
```

```
C:\PROGRA~1\XINOXS~1\JCREAT~1\GE2001.exe
Name=Joe Age=16
Press any key to continue...
```

Student class constructor

- Now we need to define another constructor, that's right, another one.
- Recall earlier in the lesson that there were TWELVE ways to construct a String, so there is no limit to the number of ways you can define a constructor for your class!

```
Stud
 1
 2
 3      String name;
 4      int age;
 5      public Student()
 6      {
 7          name = "Joe";
 8          age = 16;
 9      }
10      public String toString()
11      {
12          return "Name="+name+" Age="+age;
13      }
14      public static void main(String [] args)
15      {
16          Student s = new Student();
17          System.out.println(s);
18      }
19  }
```

```
Name=Joe Age=16
Press any key to continue...
```

Student class constructor

- All constructors start the same way…public ClassName, but then each one must be different after this part.
- Our next constructor will have a different **parameter signature**.
- This is how the compiler will differentiate between the two constructors…by the **parameter signature**, which must be unique!

```
3       String name;
4       int age;
5       public Student()
6       {
7           name = "Joe";
8           age = 16;
9       }
10      public String toString()
11      {
12          return "Name="+name+" Age="+age;
13      }
14      public static void main(String [] args)
15      {
16          Student s = new Student();
17          System.out.println(s);
18      }
19  }
```

```
Name=Joe Age=16
Press any key to continue...
```

Student class constructor

- A **parameter signature** is simply what is included, if anything, inside the parentheses.
- The default constructor always has an empty parameter list, or empty parentheses… (), like the one we just defined.

```
1
2
3       String name;
4       int age;
5       public Student()
6       {
7           name = "Joe";
8           age = 16;
9       }
10      public String toString()
11      {
12          return "Name="+name+" Age="+age;
13      }
14      public static void main(String [] args)
15      {
16          Student s = new Student();
17          System.out.println(s);
18      }
19  }
```

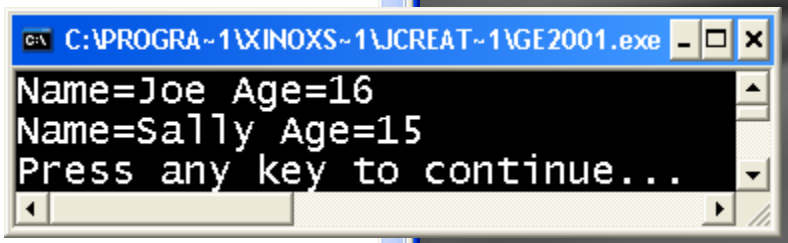
```
Name=Joe Age=16
Press any key to continue...
```

Student class constructor

- Below is the definition for a new constructor that receives a String and an integer, so that we can construct an object with a name other than "Joe" and an age other than 16.

Student.java

```
1  public class Student
2  {
3      String name;
4      int age;
5      public Student(String n, int a)
6      {
7          name = n;
8          age = a;
9      }
10     public Student()
11     {
12         name = "Joe";
13         age = 16;
14     }
15     public String toString()
16     {
17         return "Name="+name+" Age="+age;
18     }
19     public static void main(String [] args)
20     {
21         Student s = new Student();
22         System.out.println(s);
23         s = new Student("Sally",15);
24         System.out.println(s);
25
26     }
```

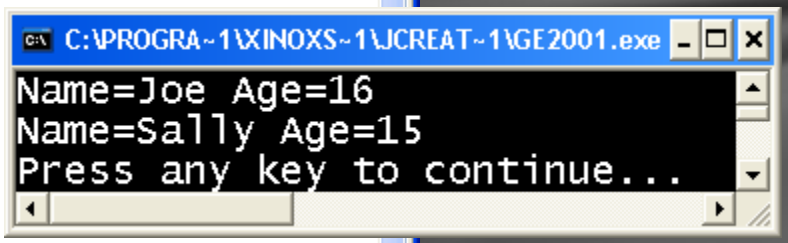


Student class constructor

- You can see in the main program that both constructors are used, the default one with "Joe" and 16, and then the two-parameter constructor, with "Sally" and 15 sent as parameter values.

Student.java

```
1  public class Student
2  {
3      String name;
4      int age;
5      public Student(String n, int a)
6      {
7          name = n;
8          age = a;
9      }
10     public Student()
11     {
12         name = "Joe";
13         age = 16;
14     }
15     public String toString()
16     {
17         return "Name="+name+" Age="+age;
18     }
19     public static void main(String [] args)
20     {
21         Student s = new Student();
22         System.out.println(s);
23         s = new Student("Sally",15);
24         System.out.println(s);
25
26     }
```



Lesson Summary

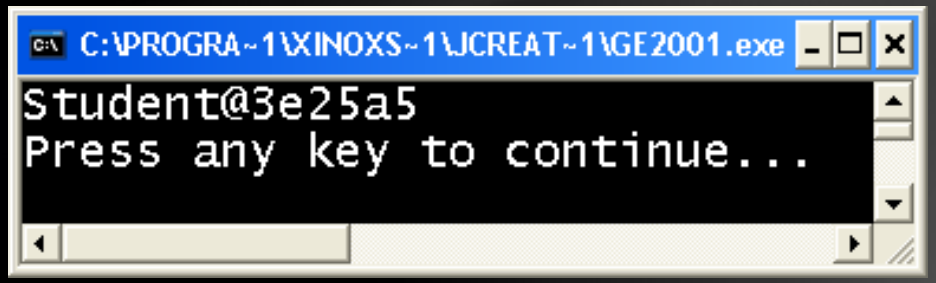
- In this lesson, you learned all about the Object class, the String class, and began to design a customized class of your own, the Student class
- Now it is time to put that program to work.

Lab

- Write a program to implement the Student class as found in the previous slides of this lesson.
- Follow the next few slides to build the class step by step.
- Print out the program source code and a screen shot of the output after completing each method.
- Hint: To get a screen shot of the output window, do the ALT-PrintScreen keyboard combination, paste the result to a Word document, and print from there..

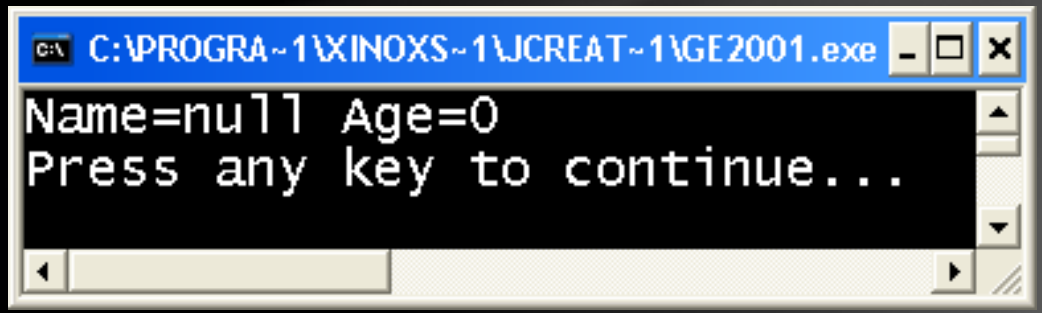
Step 1

- Write the basic Student class that:
  - Constructs a default object using the compiler default constructor
  - Outputs that object, as shown below, using ***method10_1()***



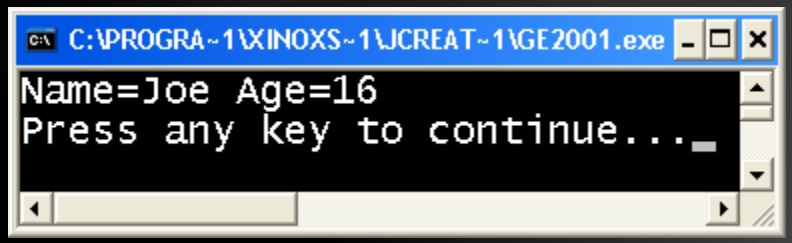
Step 2

- Modify the Student class by
  - Defining its own default constructor as shown earlier in the lesson
  - Override the toString method to create the output shown below



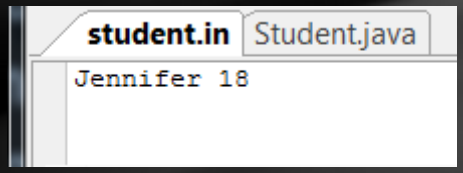
Step 3

- Further modify the Student class
  - Create a default constructor that makes your name and age the default values.
  - Creates a default Student object and output that object, like this:



Step 4

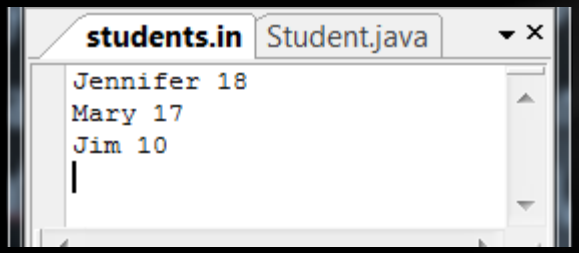
- Add to your Student class a two-parameter constructor, read a name and age from a datafile ("student.in"), to create a Student object, and then output it.

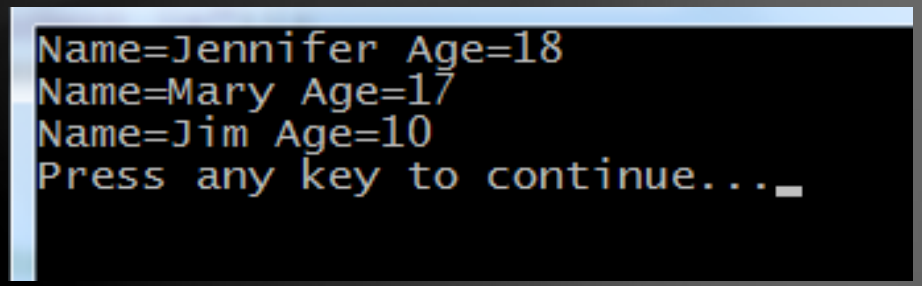



```
Name=Jennifer Age=18
Press any key to continue...
```

Step 5

- Modify Step 4 as follows:
  - Read a series of names and ages from a datafile ("students.in"), creating a Student object with each data set, and output all of the Student objects





CONGRATULATIONS!

- Now you know something about objects and classes
- You know about the Object class, the "mother" (original superclass) of all objects, inherited by all other classes
- You now know how to create your own class and how to override the toString() method,
- Create a default constructor,
- And a two-parameter constructor.
- Now move on to Lesson 10B to learn about the three main aspects of OOP, as well as how to write access methods with private data.

Thanks, and have fun!

To order supplementary materials for all the lessons in this series, including lesson examples, lab solutions, quizzes, tests, and unit reviews, visit the O(N)CS Lessons website, or contact me at

John B. Owen
captainjbo@gmail.com