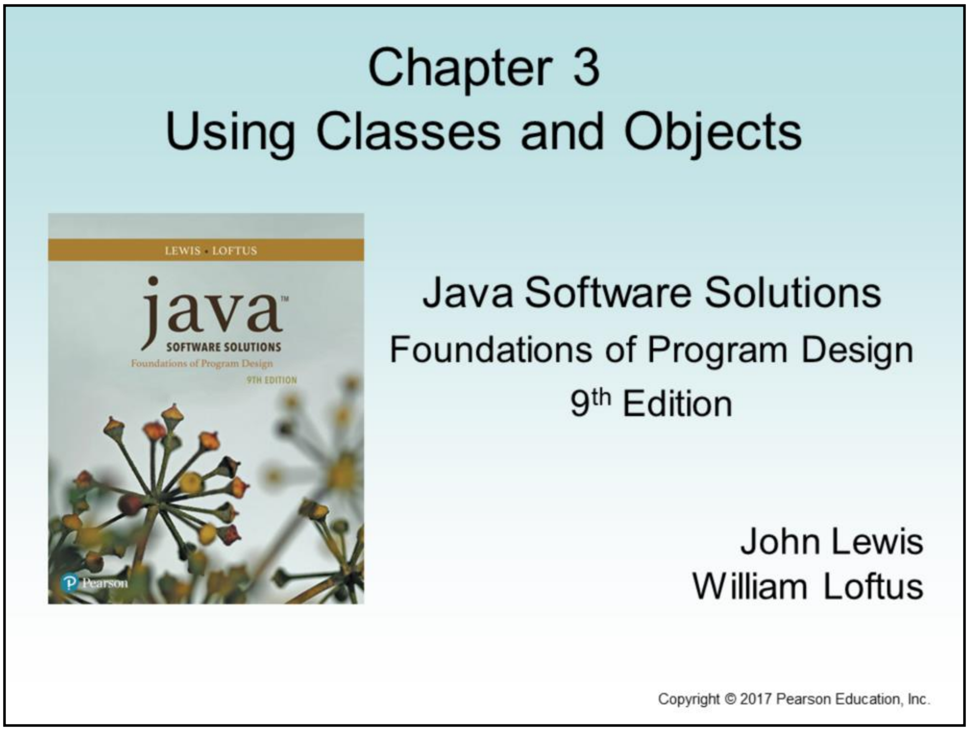


Chapter 3
Using Classes and Objects

LEWIS · LOFTUS

java™

SOFTWARE SOLUTIONS

Foundations of Program Design

9TH EDITION

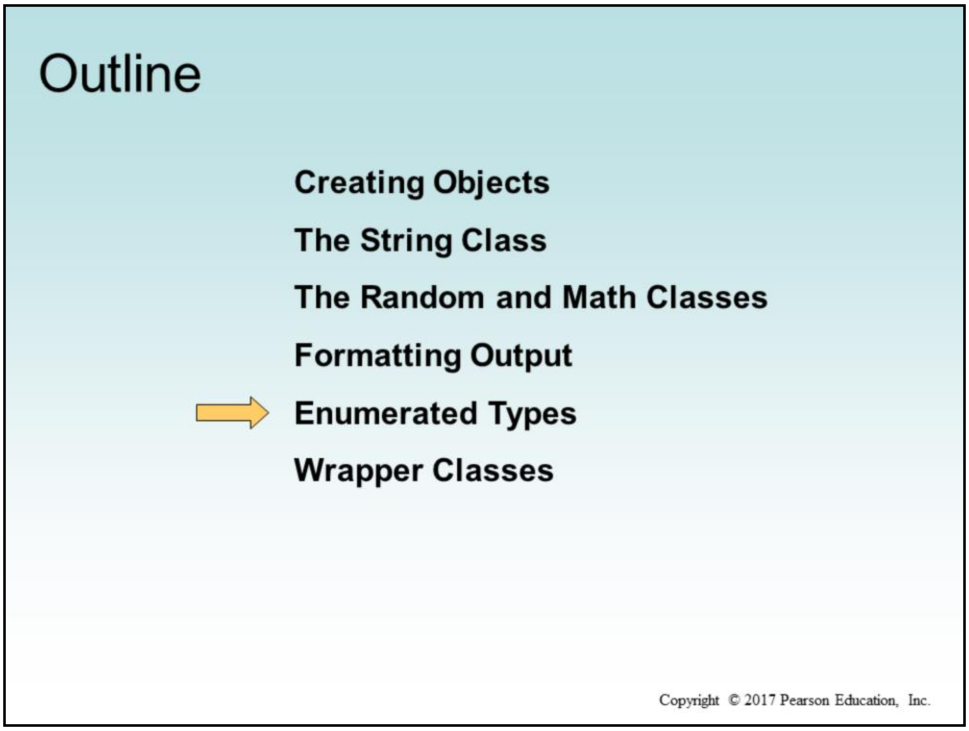
Pearson

Java Software Solutions

Foundations of Program Design

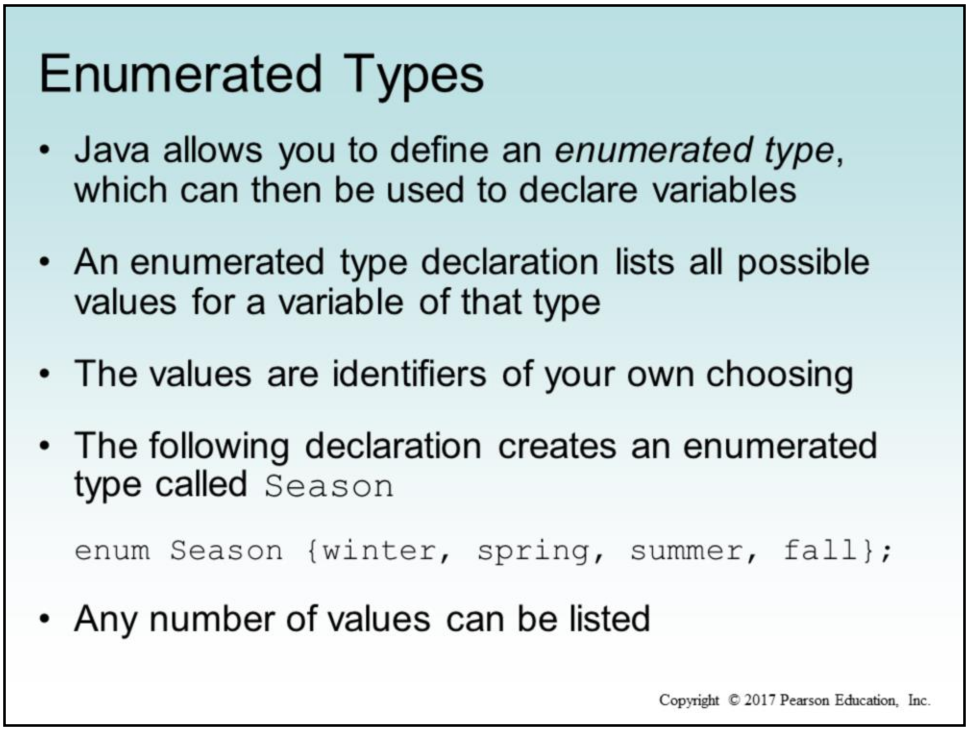
9th Edition

John Lewis
William Loftus



Outline

- Creating Objects
- The String Class
- The Random and Math Classes
- Formatting Output
- ⇨ Enumerated Types
- Wrapper Classes



Enumerated Types

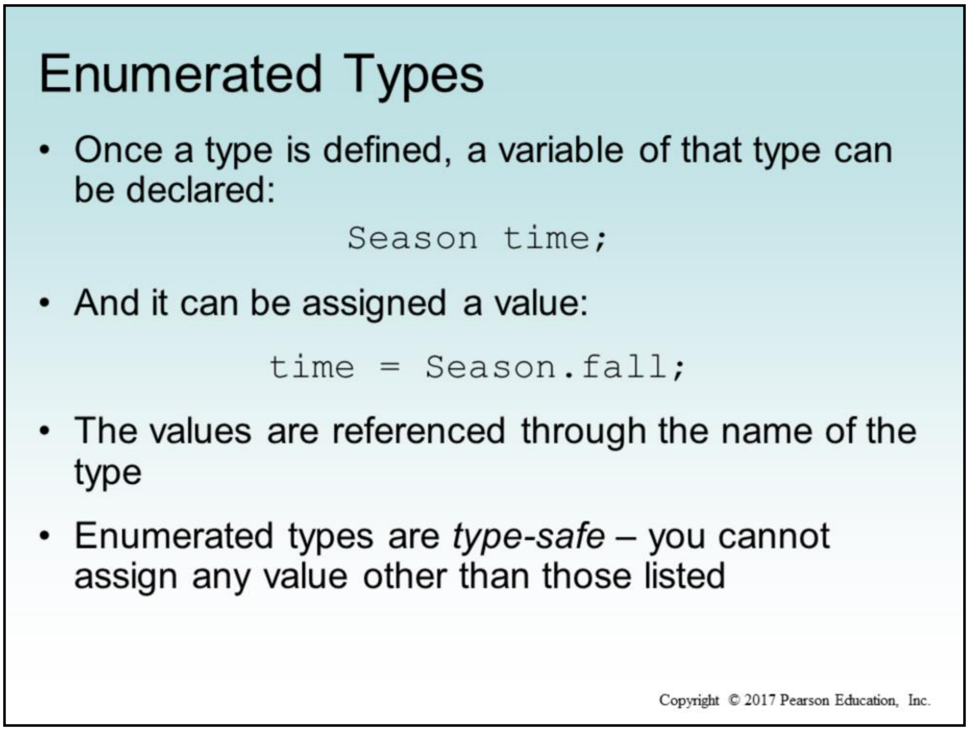
- Java allows you to define an *enumerated type*, which can then be used to declare variables
- An enumerated type declaration lists all possible values for a variable of that type
- The values are identifiers of your own choosing
- The following declaration creates an enumerated type called `Season`

```
enum Season {winter, spring, summer, fall};
```

- Any number of values can be listed

-An enumerated type is a data type that contains only a specific number of values

-Note that Season above is the name of our enumerated data type, **not** a variable



Enumerated Types

- Once a type is defined, a variable of that type can be declared:

```
Season time;
```

- And it can be assigned a value:

```
time = Season.fall;
```

- The values are referenced through the name of the type
- Enumerated types are *type-safe* – you cannot assign any value other than those listed

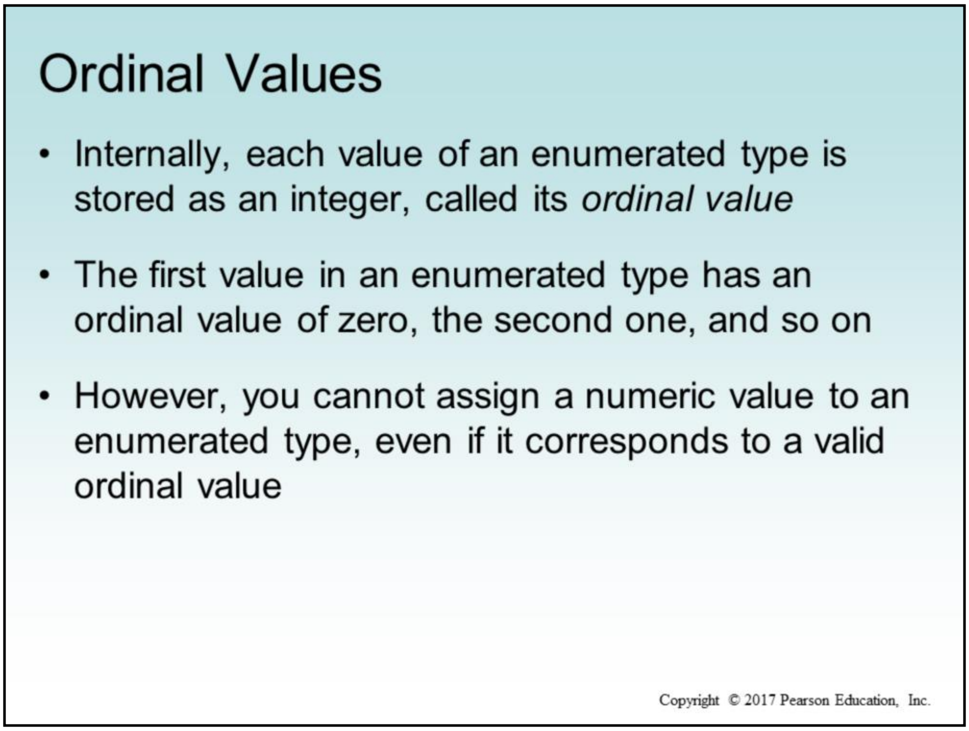
-As with other data types, we declare a variable of this enumerated type

-In the example above, **time** is a variable declared of the **Season** enumerated type

-Trying to use an identifier not within the enumerated type results in a compile-time error

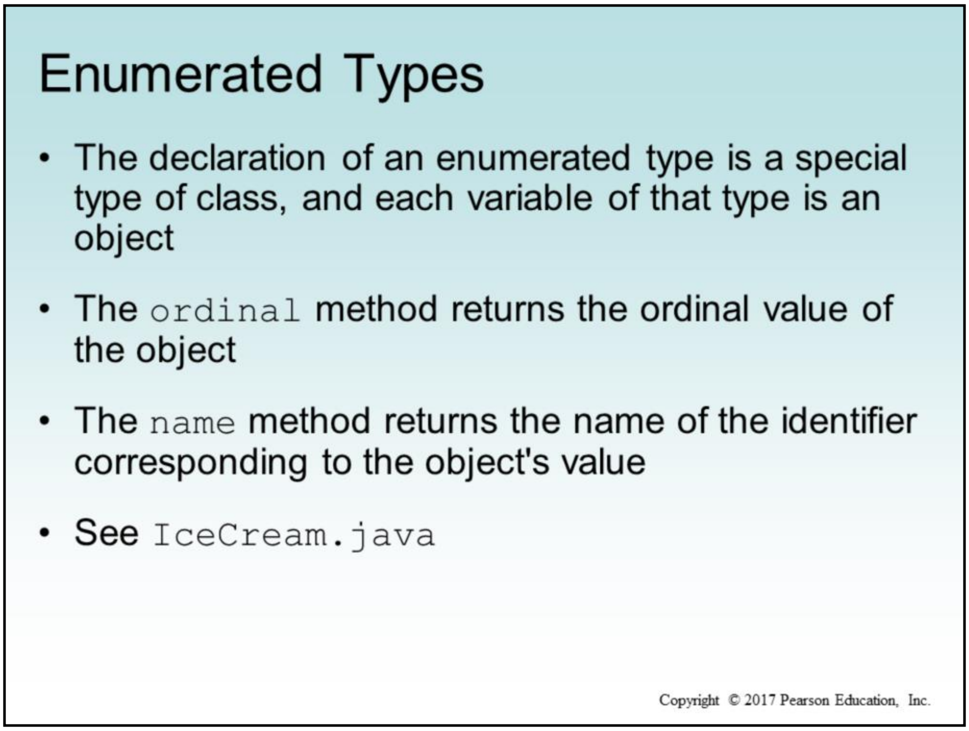
-For example, the following results in a compile-time error

time = Season.autumn;



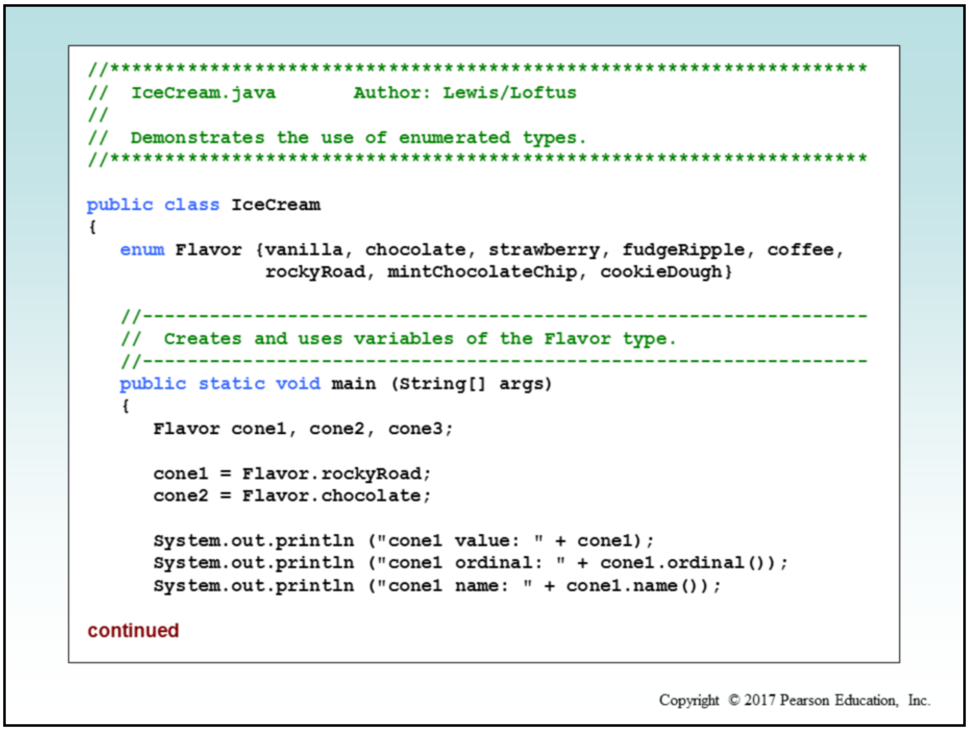
Ordinal Values

- Internally, each value of an enumerated type is stored as an integer, called its *ordinal value*
- The first value in an enumerated type has an ordinal value of zero, the second one, and so on
- However, you cannot assign a numeric value to an enumerated type, even if it corresponds to a valid ordinal value



Enumerated Types

- The declaration of an enumerated type is a special type of class, and each variable of that type is an object
- The `ordinal` method returns the ordinal value of the object
- The `name` method returns the name of the identifier corresponding to the object's value
- See `IceCream.java`



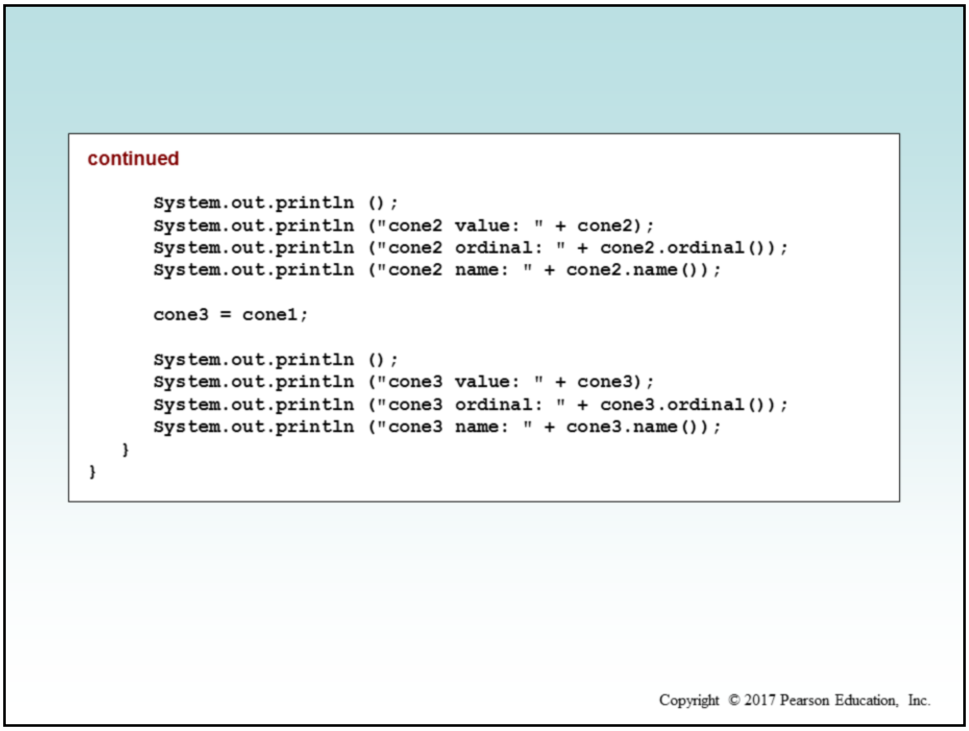
```
//********************************************************************
//  IceCream.java       Author: Lewis/Loftus
//
//  Demonstrates the use of enumerated types.
//********************************************************************

public class IceCream
{
   enum Flavor {vanilla, chocolate, strawberry, fudgeRipple, coffee,
                rockyRoad, mintChocolateChip, cookieDough}

   //-----------------------------------------------------------------
   //  Creates and uses variables of the Flavor type.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      Flavor cone1, cone2, cone3;

      cone1 = Flavor.rockyRoad;
      cone2 = Flavor.chocolate;

      System.out.println ("cone1 value: " + cone1);
      System.out.println ("cone1 ordinal: " + cone1.ordinal());
      System.out.println ("cone1 name: " + cone1.name());
```

continued

-Note where we define our enumerated type

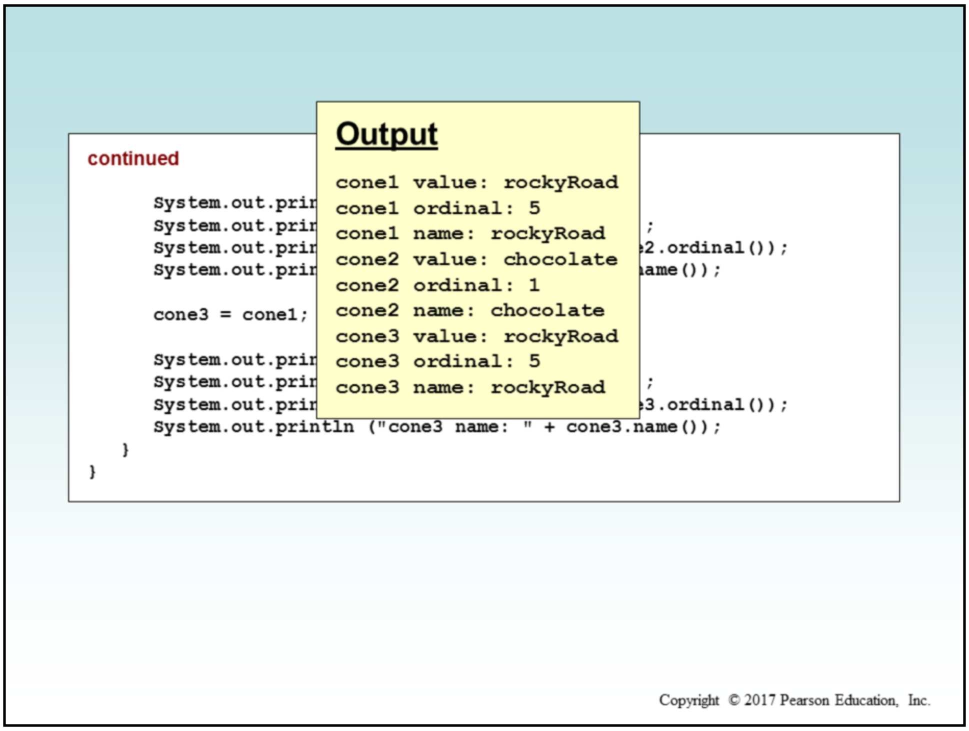
-Since they are a special type of class, we define them outside of a specific class method

-We could have also defined it above (or outside) the class itself



continued

```
      System.out.println ();
      System.out.println ("cone2 value: " + cone2);
      System.out.println ("cone2 ordinal: " + cone2.ordinal());
      System.out.println ("cone2 name: " + cone2.name());

      cone3 = cone1;

      System.out.println ();
      System.out.println ("cone3 value: " + cone3);
      System.out.println ("cone3 ordinal: " + cone3.ordinal());
      System.out.println ("cone3 name: " + cone3.name());
   }
}
```

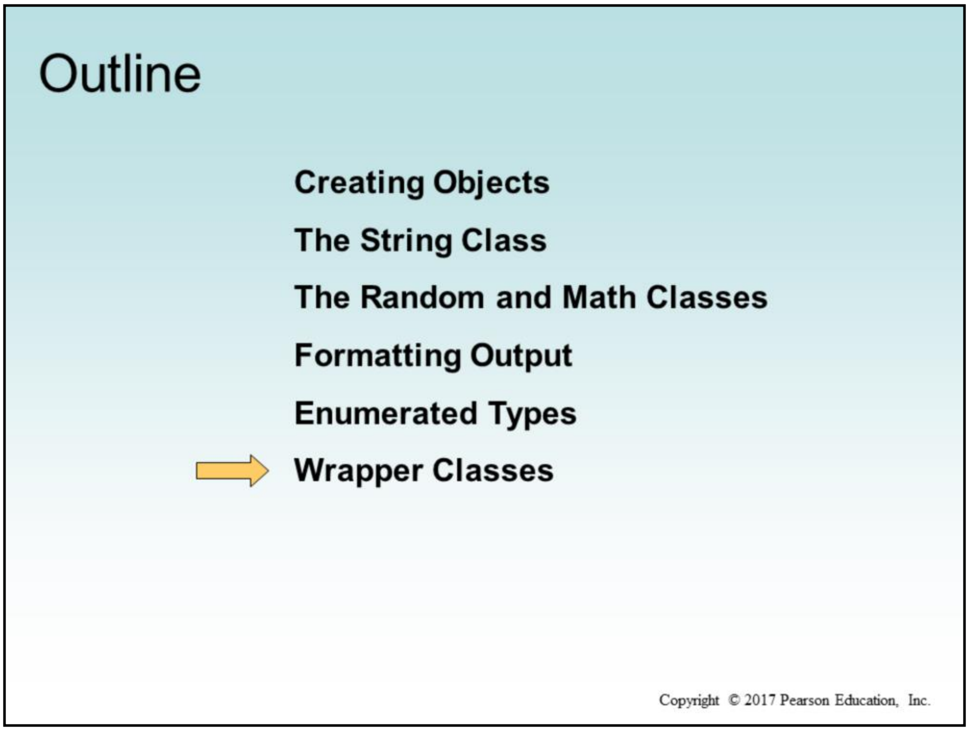


continued

```
      System.out.prin
      System.out.prin                          ;
      System.out.prin                          2.ordinal());
      System.out.prin                          ame());

      cone3 = cone1;

      System.out.prin
      System.out.prin                          ;
      System.out.prin                          3.ordinal());
      System.out.println ("cone3 name:  " + cone3.name());
   }
}
```

Output

```
cone1 value: rockyRoad
cone1 ordinal: 5
cone1 name: rockyRoad
cone2 value: chocolate
cone2 ordinal: 1
cone2 name: chocolate
cone3 value: rockyRoad
cone3 ordinal: 5
cone3 name: rockyRoad
```



Outline

Creating Objects

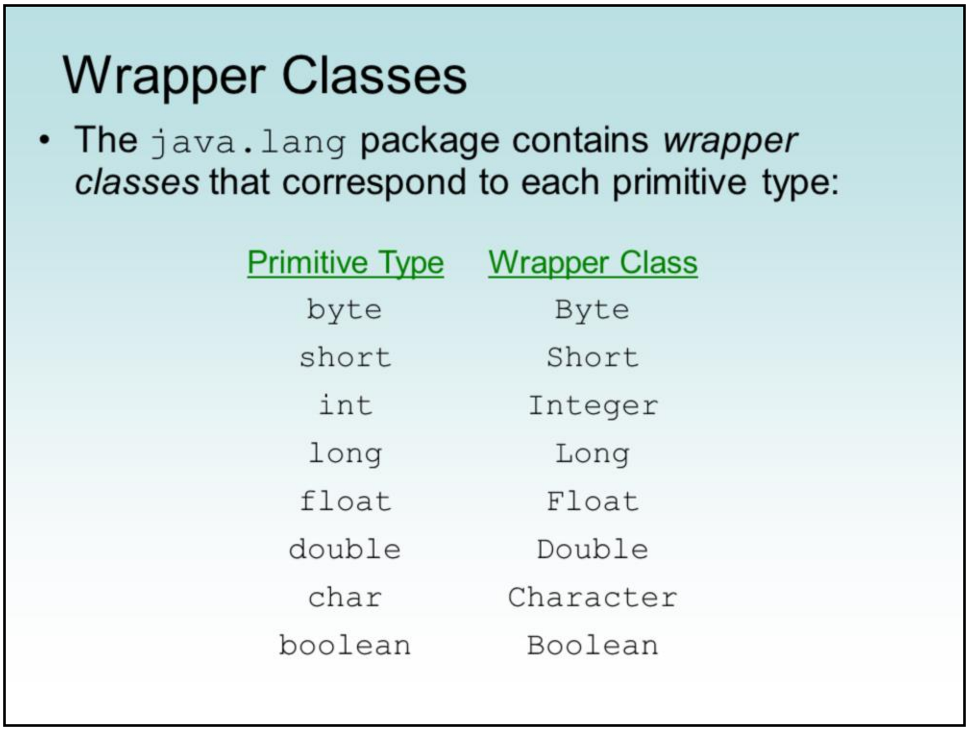
The String Class

The Random and Math Classes

Formatting Output

Enumerated Types

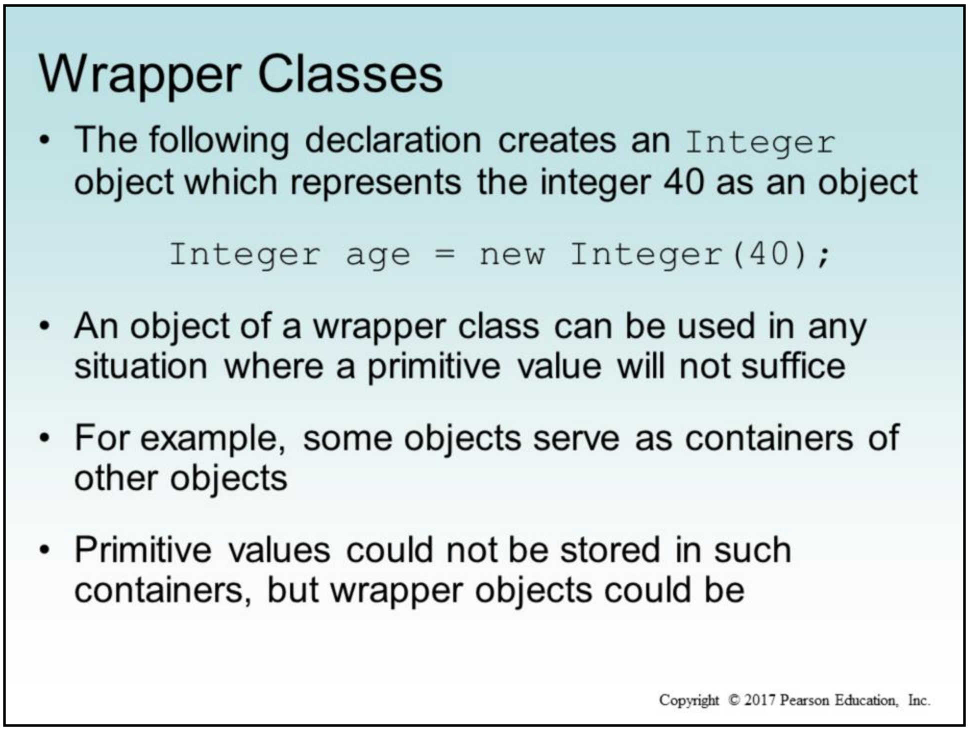
→ Wrapper Classes



Wrapper Classes

- The `java.lang` package contains *wrapper classes* that correspond to each primitive type:

Primitive Type	Wrapper Class
`byte`	`Byte`
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`
`char`	`Character`
`boolean`	`Boolean`



Wrapper Classes

- The following declaration creates an `Integer` object which represents the integer 40 as an object

```
Integer age = new Integer(40);
```

- An object of a wrapper class can be used in any situation where a primitive value will not suffice
- For example, some objects serve as containers of other objects
- Primitive values could not be stored in such containers, but wrapper objects could be

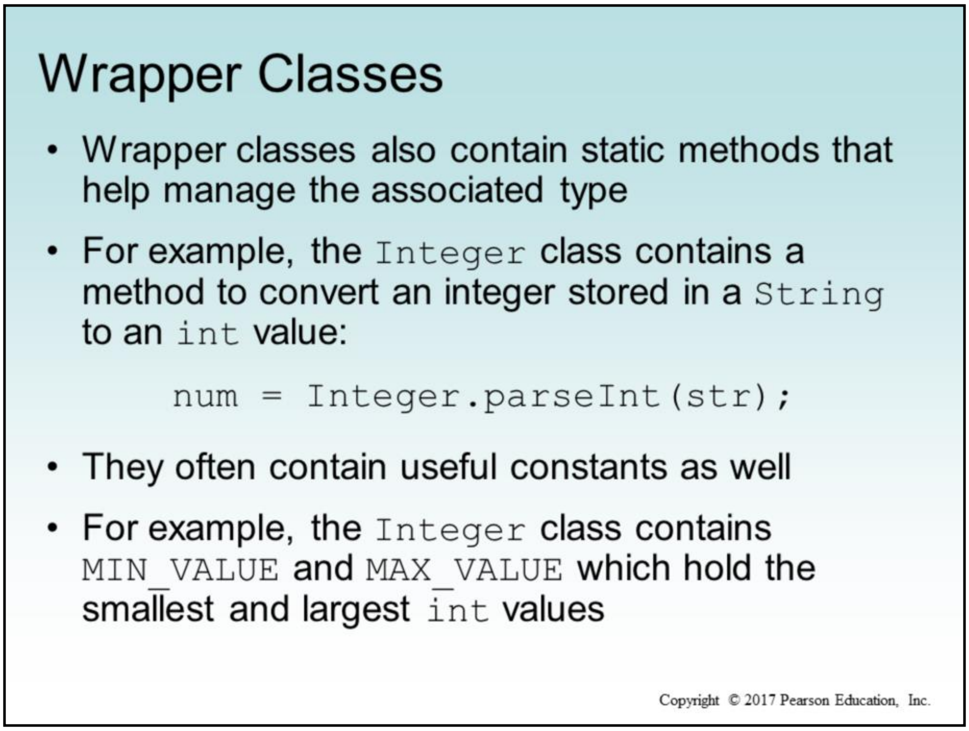
-Wrapper classes each store a field storing a single primitive data type

-Wrapper classes provide functionality for each of the primitive data types

-This functionality includes conversions to other data types or formats

-Since these are classes, we create objects using the new operator

-We then work with the objects as the primitive data type

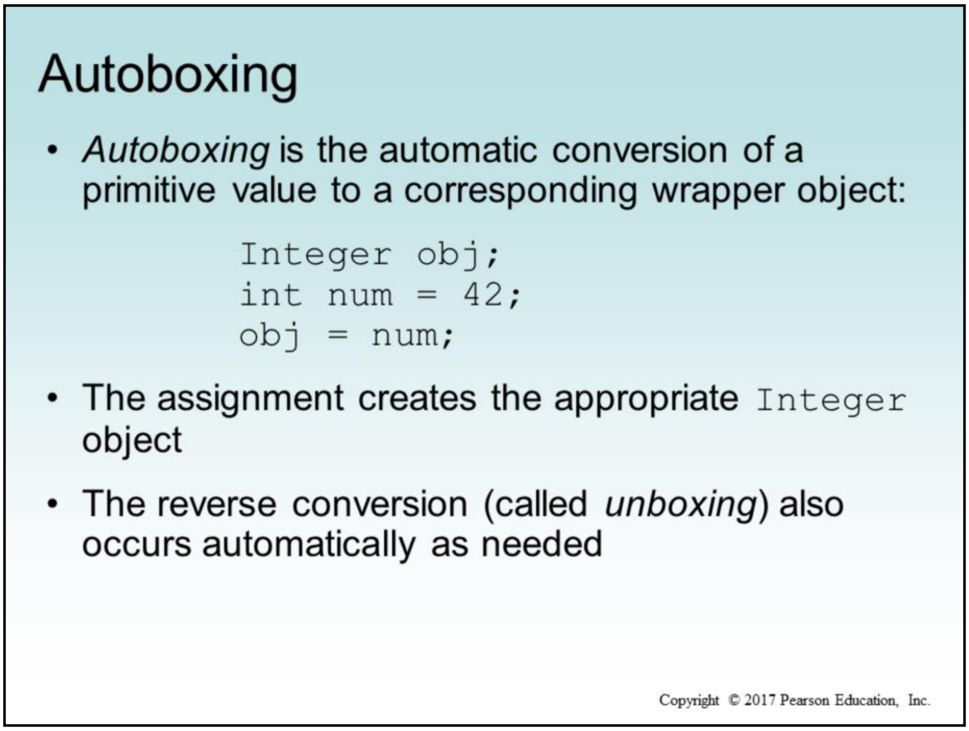


Wrapper Classes

- Wrapper classes also contain static methods that help manage the associated type
- For example, the `Integer` class contains a method to convert an integer stored in a `String` to an `int` value:

```
num = Integer.parseInt(str);
```

- They often contain useful constants as well
- For example, the `Integer` class contains `MIN_VALUE` and `MAX_VALUE` which hold the smallest and largest `int` values



Autoboxing

- *Autoboxing* is the automatic conversion of a primitive value to a corresponding wrapper object:

```
Integer obj;
int num = 42;
obj = num;
```

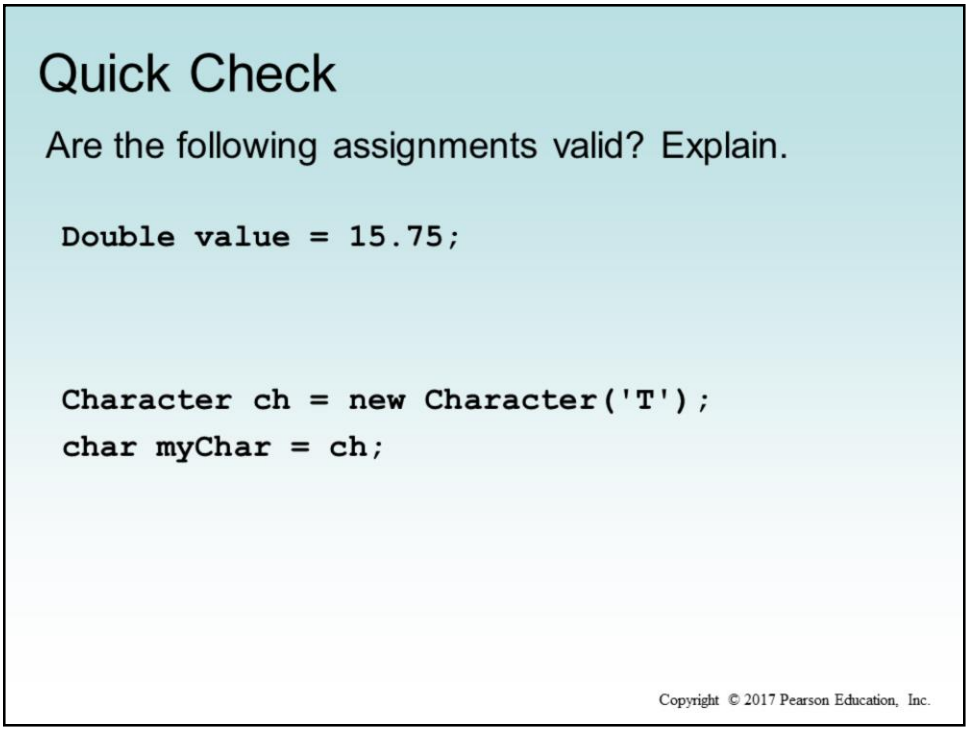
- The assignment creates the appropriate `Integer` object
- The reverse conversion (called *unboxing*) also occurs automatically as needed

-Note how such conversions do not require us to create an object with the **new** operator

-In the example above, an Integer object is automatically created in the assignment obj = num;

-Example of the reverse process called **unboxing**:

```
Integer obj2 = new Integer(77);
int num2;
num2 = obj2;   // automatically extracts the int value (77) from the obj2 object
```

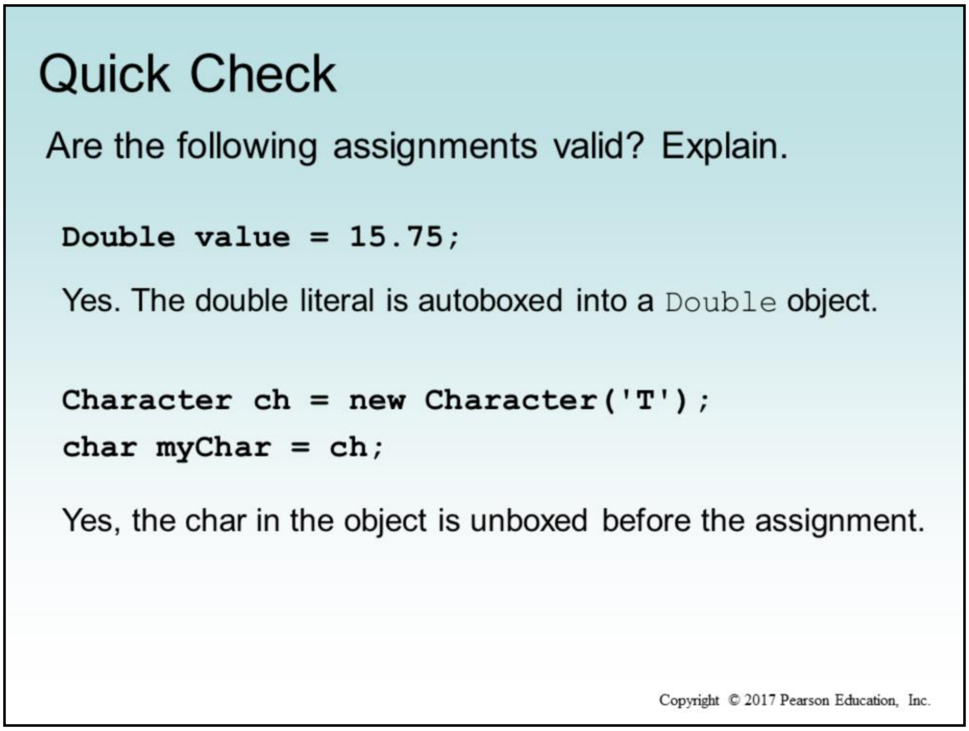


Quick Check

Are the following assignments valid? Explain.

```
Double value = 15.75;
```

```
Character ch = new Character('T');
char myChar = ch;
```



Quick Check

Are the following assignments valid? Explain.

```
Double value = 15.75;
```

Yes. The double literal is autoboxed into a `Double` object.

```
Character ch = new Character('T');
char myChar = ch;
```

Yes, the char in the object is unboxed before the assignment.