

# Object-Oriented Programming II

Michael C. Hackett

Assistant Professor, Computer Science

Community  
College  
*of* Philadelphia

# Lecture Topics

- Constructors
- Arithmetic
  - Python vs Java Basic Math Operations
  - Math object
  - Mixed Type Arithmetic
  - Typecasting
- Utility Methods
- Formatted Printing

# Constructors

- Constructors are a special type of method that prepares the instance of the object.
- Constructors always have the same name as the class.
- Classes may have multiple constructors.
  - This is referred to as ***overloading*** or having ***overloaded constructors***.

# The No Argument (No-Arg) Constructor

```
class Circle {  
  
    double radius;  
    String color;  
  
    Circle() {  
        radius = 1;  
        color = "Red";  
    }  
  
}
```

- Replaces the default constructor added by the compiler.
  - A class only has a default constructor when the class has no constructors defined.
- The code in the no-arg constructor's body will be executed when the no-arg constructor is called.
- In this example, the radius field will be set to 1 and the color field is set to Red when the no-arg constructor is called.

# The No Argument (No-Arg) Constructor

```
class TestProgram {  
  
    public static void main(String[] args) {  
        Circle demo = new Circle();  
  
        System.out.println("The circle's radius is " + demo.radius);  
        System.out.println("The circle's color is " + demo.color);  
    }  
}
```

## Output

```
The circle's radius is 1  
The circle's color is Red
```

# Constructors

- In many cases, you will want to pass data into a constructor to set the values of the object's fields.
- ***Parameters*** are variables that represent data that is given (or *passed*) to a constructor.
  - The data given to the parameters are called ***arguments***.
  - Though, *parameter* and *argument* are often used interchangeably
- If a constructor declares a parameter list, a value for each parameter must be present.
  - The parameter list must include the data type for each parameter.

# Constructors

```
class Circle {
```

```
    double radius;  
    String color;
```

```
    Circle() {  
        radius = 1;  
        color = "Red";  
    }
```

```
    Circle(double r) {  
        radius = r;  
        color = "Red";  
    }
```

```
}
```

- There are now two ways to instantiate a Circle object
  - Without an argument (radius is set to 1; color is set to Red)
  - With a double-type argument (radius is set to the value passed to the r parameter; color is set to Red)

# Constructors

```
class TestProgram {  
  
    public static void main(String[] args) {  
        Circle demo = new Circle();  
        Circle demo2 = new Circle(76.5);  
  
        System.out.println("The first circle's radius is " + demo.radius);  
        System.out.println("The second circle's radius is " + demo2.radius);  
    }  
}
```

## Output

```
The first circle's radius is 1  
The second circle's radius is 76.5
```



# Constructors

```
class Circle {  
  
    double radius;  
    String color;  
  
    Circle() {  
        radius = 1;  
        color = "Red";  
    }  
  
    Circle(double r) {  
        radius = r;  
        color = "Red";  
    }  

```

```
    Circle(double r, String c) {  
        radius = r;  
        color = c;  
    }  

```

```
}
```

- There is now a third way to instantiate a Circle object
  - With a double-type argument (radius is set to the value passed to the r parameter) followed by a String-type argument (color is set to the value passed to the c parameter)

# Constructors

## Output

The first circle's radius is 1  
The first circle's color is Red

The second circle's radius is 76.5  
The second circle's color is Red

The third circle's radius is 100.4  
The third circle's color is Blue

```
class TestProgram {  
  
    public static void main(String[] args) {  
        Circle demo = new Circle();  
        Circle demo2 = new Circle(76.5);  
        Circle demo3 = new Circle(100.4, "Blue");  
  
        System.out.println("The first circle's radius is " + demo.radius);  
        System.out.println("The first circle's color is " + demo.color);  
        System.out.println();  
        System.out.println("The second circle's radius is " + demo2.radius);  
        System.out.println("The second circle's color is " + demo2.color);  
        System.out.println();  
        System.out.println("The third circle's radius is " + demo3.radius);  
        System.out.println("The third circle's color is " + demo3.color);  
    }  
}
```

# Constructors

- There is no limit to the number of constructors in a class.
- However, each constructor must have a unique ***signature***.
  - A constructor signature consists of its name and parameter list data types.

**Circle()**

**Circle(double r)**

**Circle(double r, String c)**

Signatures:

Circle()

Circle(double)

Circle(double, String)

# Python and Java Basic Math Comparisons

Purpose	Python	Java
Addition	<code>x = y + z</code>	<code>x = y + z;</code>
Subtraction	<code>x = y - z</code>	<code>x = y - z;</code>
Multiplication	<code>x = y * z</code>	<code>x = y * z;</code>
Division	<code>x = y / z</code> <code>x = y // z</code>	<code>x = y / z;</code>
Mod Division	<code>x = y % z</code>	<code>x = y % z;</code>
Exponentiation	<code>x = y ** z</code>	<code>x = Math.pow(y, z);</code>
Square Root	<code>import math</code> <code>x = math.sqrt(y)</code>	<code>x = Math.sqrt(y);</code>
Round	<code>x = round(y)</code>	<code>x = Math.round(y);</code>
Round (Ceiling)	<code>import math</code> <code>x = math.ceil(y)</code>	<code>x = Math.ceil(y);</code>
Round (Floor)	<code>import math</code> <code>x = math.floor(y)</code>	<code>x = Math.floor(y);</code>

# Python and Java Basic Math Comparisons

Purpose	Python	Java
Augmented Assignment (Addition)	<code>x += y</code>	<code>x += y;</code>
Augmented Assignment (Subtraction)	<code>x -= y</code>	<code>x -= y;</code>
Augmented Assignment (Multiplication)	<code>x *= y</code>	<code>x *= y;</code>
Augmented Assignment (Division)	<code>x /= y</code> <code>x //= y</code>	<code>x /= y;</code>
Augmented Assignment (Mod Division)	<code>x %= y</code>	<code>x %= y;</code>
Augmented Assignment (Exponentiation)	<code>x **= y</code>	N/A

# Math object

- Provides access to mathematical functions like rounding, exponents, and square roots.
- Unlike Python's math module, Java's Math object does not need to be imported.
  - It's always available to call on.

# Math object – Rounding

- Round method takes a **double or float**.
  - Returns a **long** if double.
  - Returns an **int** if float.

```
double originalNumber;  
long roundedNumber;  
originalNumber = 25.6;  
roundedNumber = Math.round(originalNumber);  
System.out.println("The rounded number is: " + roundedNumber);
```

The rounded number is: 26.0

# Math object – Rounding Up

- Ceiling (ceil) method takes a **double or float**.
  - Always returns a **double**.

```
double originalNumber;  
double roundedNumber;  
originalNumber = 15.1;  
roundedNumber = Math.ceil(originalNumber);  
System.out.println("The rounded number is: " + roundedNumber);
```

The rounded number is: 16.0



# Math object – Rounding Down

- Floor method takes a **double or float**.
  - Always returns a **double**.

```
double originalNumber;  
double roundedNumber;  
originalNumber = 18.8;  
roundedNumber = Math.floor(originalNumber);  
System.out.println("The rounded number is: " + roundedNumber);
```

The rounded number is: 18.0

# Math object – Square Root

- Square Root function takes any numeric value;
  - Always returns a **double**.

```
int myNumber;  
double squareRoot;  
myNumber = 16;  
squareRoot = Math.sqrt(myNumber);  
System.out.println("The square root is: " + squareRoot);
```

The square root is: 4.0

# Math object – Exponents

- Power (pow) function takes two parameters: base and exponent
  - Always returns a **double**.

```
double base = 2;  
int exponent = 3;  
double result;  
result = Math.pow(base, exponent);  
System.out.println("The result is: " + result);
```

```
The result is: 8.0
```

# Math object methods

Method	Return Type	Description	Possible Exceptions
<code>sqrt(double/float/long/int)</code>	double	Returns the square root of the supplied number.	None
<code>pow(double, double)</code>	double	Returns the first parameter raised to the power of the second parameter.	None
<code>round(double/float)</code>	long if double; int if float	Rounds the supplied number to the nearest whole number and returns it.	None
<code>ceil(double/float)</code>	double	Rounds the supplied number up to the nearest whole number and returns it.	None
<code>floor(double/float)</code>	double	Rounds the supplied number down to the nearest whole number and returns it.	None
<code>abs(double/float/long/int)</code>	double	Returns the absolute value of the supplied numbers.	None
<code>log(double)</code>	double	Returns the natural logarithm of the supplied number.	None
<code>toDegrees(double)</code>	double	Returns the supplied number (radians) in degrees.	None
<code>toRadians(double)</code>	double	Returns the supplied number (degrees) in radians.	None

There are more than these. For a complete list go to: <https://docs.oracle.com/javase/8/docs/api/java/lang/Math.html>

# Mixed Type Arithmetic

- Special rules apply when performing arithmetic operations on numbers of different types. For example, adding a byte and a double together or dividing an int by a short. *What data type is the result of that arithmetic?*
- Arithmetic operations performed only on a combination of *bytes, shorts, and ints* **ALWAYS return an int!**
- Arithmetic operations performed with *longs, floats, or doubles* **return with the highest ranked data type.**

# Mixed Type Arithmetic

- Numeric Primitive Data Type Ranking:
  1. double (Highest Ranked)
  2. float
  3. long
  4. int
  5. short
  6. byte (Lowest Ranked)

# Mixed Type Arithmetic

First Operand	Second Operand	Resulting Type
byte, short, int	byte, short, int	int
byte, short, int, long	long	long
byte, short, int, long, float	float	float
byte, short, int, long, float, double	double	double

# Mixed Type Arithmetic

- Examples:
  - *int + byte = int*
  - *byte \* short = int*
  - *int - int = int*
  - *byte / Long = Long*
  - *Long - Long = Long*
  - *short \* float = float*
  - *float \* float = float*
  - *double - byte = double*
  - *double + float = double*



# Mixed Type Arithmetic

- Examples:

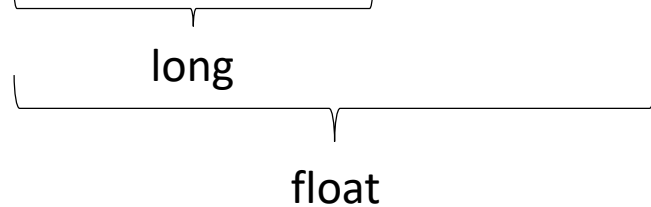
- $\underbrace{int + byte}_{int} - byte = int$   
 $\underbrace{\hspace{1.5cm}}_{int}$

- $byte + \underbrace{Long * int}_{long} = Long$   
 $\underbrace{\hspace{1.5cm}}_{long}$

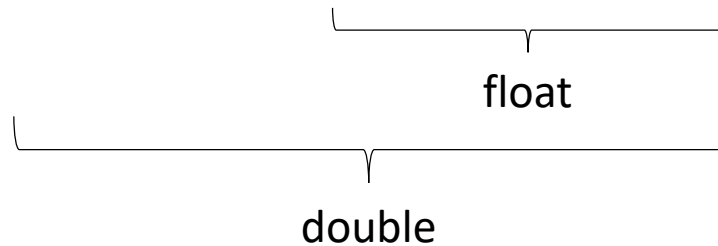
# Mixed Type Arithmetic

- Examples:

- $\text{long} - \text{int} + \text{float} = \text{float}$



- $\text{double} + \text{int} / \text{float} = \text{double}$





# Mixed Type Arithmetic

- Examples:

- $\text{float} + \text{int} / \text{byte} = \text{float}$

- $54.6 + 6 / 12 = \underline{54.6}$

  
int (6 / 12 = 0.5 -> 0)

  
float (54.6 + 0 = 54.6)

Be careful with operator precedence and the truncation of fractional amounts with integer division

# Typecasting

- The process of typecasting the data of a higher ranked type into a lower ranked type is called ***narrowing***.

```
double myDouble = 453.87;  
int myInt;  
myInt = (int)myDouble;
```




Put the desired data type, in parenthesis, before the variable name to ***typecast*** the value as that type.

Be careful! Narrowing can lead to loss of precision!  
In the example above, the value stored at the memory location referenced by myInt is 453 NOT 453.87

# Typecasting

- If you forget to typecast when narrowing, your code will not compile.

```
double myDouble = 453.87;  
int myInt;  
myInt = myDouble;
```



Compile-time error

```
incompatible types: possible lossy conversion from double to int  
----  
(Alt-Enter shows hints)
```


# Typecasting

- Consider the code below. An int value, 4500, is narrowed to a byte. However, a byte can only hold up to 127. What happens?

```
int someInt = 4500;  
byte someByte;  
someByte = (byte)someInt;
```

# Typecasting

```
int someInt = 4500;  
byte someByte;  
someByte = (byte)someInt;
```

- 4500, in binary (and in memory,) looks like this: 1000110010100
- A byte can only hold eight bits: 1000110010100  


These bits will be ignored.  
The technical term is **overflow**
- The binary value of the data in memory referenced by someByte is now 10010100
- This binary value, when converted to a signed decimal integer is -108 (Way off from 4500)

# Typecasting

- The process of typecasting the data of a lower ranked type into a higher ranked type is called ***widening***.
- No real issues with this. You can just assign the lower ranked value into a variable of a higher rank.

```
double myDouble;  
int myInt = 34653;  
myDouble = myInt;
```

- The higher-ranked data type will have enough room to accommodate the value of the lower-ranked type.
- Remember: you can't change a variable's type after it has been declared.



# Typecasting

Source Data Type (x)	Target Data Type	Statement
Any numeric primitive	<b>byte</b>	<b>(byte)x</b>
Any numeric primitive	<b>short</b>	<b>(short)x</b>
Any numeric primitive	<b>int</b>	<b>(int)x</b>
Any numeric primitive	<b>long</b>	<b>(long)x</b>
Any numeric primitive	<b>float</b>	<b>(float)x</b>
Any numeric primitive	<b>double</b>	<b>(double)x</b>
Any numeric primitive	<b>String</b>	<b>String.valueOf(x)</b>

# Typecasting

Source Data Type (x)	Target Data Type	Statement
String	byte	<code>Byte.parseByte(x)</code>
String	short	<code>Short.parseShort(x)</code>
String	int	<code>Integer.parseInt(x)</code>
String	long	<code>Long.parseLong(x)</code>
String	float	<code>Float.parseFloat(x)</code>
String	double	<code>Double.parseDouble(x)</code>

# Methods

- Recall from the previous lecture that, at a minimum, methods must specify:
  - **The type of data they return**
  - **The name of the method**
  - **A parameter list**
- This was demonstrated with the Circle class's toString method.

# Methods

- Methods are generally categorized as an accessor method or a mutator method.
  - An **accessor method** retrieves (or “gets”) data from an object.
  - A **mutator method** changes (or “sets”) the object’s state.
  - Some methods may be a hybrid of both, where the method changes the object’s data *and* returns information from the object.
- Some methods neither alter the state of the object nor return information from the object.
  - These are often referred to as **utility methods**.

# Utility Methods

- The Circle class's toString method is mostly a utility method.

```
String toString() {  
    String text = "Circle radius: " +  
                  radius +  
                  " and color: " +  
                  color;  
    return text;  
}
```

- Though, it is somewhat an accessor since it returns information about the object's state (its current radius and color)
  - However, those data aren't usable unless it is properly parsed out of the String

# Utility Methods

- Suppose we want to add a method that calculates and return the Circle object's area, based on its current radius.
  - Formula for the area of a circle:  $\pi \times r^2$

```
double area() {  
    double result = Math.PI * Math.pow(radius, 2);  
    return result;  
}
```

- This is primarily a utility method but is somewhat an accessor since it returns information about the object's state.
  - No explicit information about the radius or color, though.

# Utility Methods

## Output

Circle radius: 100.4 and color: Blue  
The circle's area is 31667.756603

```
class TestProgram {  
  
    public static void main(String[] args) {  
        Circle demo = new Circle(100.4, "Blue");  
  
        System.out.println(demo.toString());  
  
        double a = demo.area();  
        System.out.println("The circle's area is " + a);  
    }  
}
```

# Utility Methods

- Let's also add a method that calculates and returns the Circle object's circumference, based on its current radius.
  - Formula for the circumference of a circle:  $2\pi \times r$

```
double circumference() {  
    double result = 2 * Math.PI * radius;  
    return result;  
}
```

- This is also primarily a utility method but is somewhat an accessor since it returns information about the object's state.
  - No explicit information about the radius or color, though.



# Utility Methods

## Output

Circle radius: 100.4 and color: Blue

The circle's area is 31667.756603

The circle's circumference is 630.831804841

```
class TestProgram {  
  
    public static void main(String[] args) {  
        Circle demo = new Circle(100.4, "Blue");  
  
        System.out.println(demo.toString());  
  
        double a = demo.area();  
        System.out.println("The circle's area is " + a);  
  
        double c = demo.circumference();  
        System.out.println("The circle's circumference is " + c);  
  
    }  
}
```

# Formatted Printing

- The **`System.out.print()`** and **`System.out.println()`** methods print output with no formatting.
- The **`System.out.printf()`** method allows greater control of how numbers are printed.

# Formatted Printing

- Format specifiers begin with % and end with a converter character.
  - The converter character indicates the type of parameter to be formatted.
- Format specifiers are typed directly into a String literal.

```
int age = 75;  
System.out.printf("The value of age is %d", age);
```

```
The value of age is 75
```

# Formatted Printing

- Indicates the corresponding parameter is a decimal (base 10) integer (byte, short, int, long types.)

```
int age = 75;
```

```
int age2 = 65;
```

```
System.out.printf("The ages are %d and %d", age, age2);
```

```
The ages are 75 and 65
```

# Formatted Printing

- Flags are optional and specify how the parameter is to be formatted.
- Inserted between the % and converter character.
- , flag - Inserts commas

```
int lotteryJackpot = 2500000;  
System.out.printf("The jackpot is $%,d", lotteryJackpot);
```

```
The jackpot is $2,500,000
```

# Formatted Printing

- Indicates the corresponding parameter is a floating point decimal (float or double type.)

```
double pi = 3.14159;  
System.out.printf("The value of pi is %f", pi);
```

```
The value of pi is 3.14159
```

# Formatted Printing

- , flag - Inserts commas
- .N flag - Rounds to N decimal places

```
double pi = 3.14159;  
System.out.printf("The value of pi is %.2f", pi);
```

```
The value of pi is 3.14
```

- Note: comma flag must proceed .N flag

# Formatted Printing

- Indicates the corresponding parameter is a character (char type.)
  - %c forces the character, if it's a letter, to lowercase.
  - %C forces the character, if it's a letter, to uppercase.

```
char aChar = 'a';
```

```
char bChar = 'B';
```

```
char cChar = 'c';
```

```
char dChar = 'D';
```

```
System.out.printf("Characters: %C%c%c%C", aChar, bChar, cChar, dChar);
```

```
Characters: AbcD
```



# Formatted Printing

- Forces a new line.
  - Works just like `\n`

```
int height = 44;  
System.out.printf("The\nheight is %d", height);
```

```
The  
height is 44
```

# Formatted Printing

## Output

Circle radius: 100.4 and color: Blue

The circle's circumference is 630.831804841

The circle's circumference is 630.83

The circle's circumference is 630.832

```
class TestProgram {
```

```
    public static void main(String[] args) {  
        Circle demo = new Circle(100.4, "Blue");
```

```
        double c = demo.circumference();
```

```
        System.out.println("The circle's circumference is " + c);
```

```
        System.out.printf("The circle's circumference is %.2f %n", c);
```

```
        System.out.printf("The circle's circumference is %.3f", c);
```

```
    }
```

```
}
```