

## 2: More types, Methods, Conditionals

# Outline

---

- Lecture I Review
- More types
- Methods
- Conditionals

Kinds of values that can be stored and manipulated.

**boolean**: Truth value (**true** or **false**).

**int**: Integer (0, 1, -47).

**double**: Real number (3.14, 1.0, -2.1).

**String**: Text ("hello", "example").

# Variables

**Named location that stores a value**

**Example:**

```
String a = "a";
```

```
String b = "letter b";
```

```
a = "letter a";
```

```
String c = a + " and " + b;
```

# Operators

---

## Symbols that perform simple computations

- Assignment: =
- Addition: +
- Subtraction: -
- Multiplication: \*
- Division: /

# Exercise 1

```
class GravityCalculator {  
    public static void main(String[] args) {  
        double gravity = -9.81;  
        double initialVelocity = 0.0;  
        double fallingTime = 10.0;  
        double initialPosition = 0.0;  
        double finalPosition = .5 * gravity * fallingTime *  
                               fallingTime;  
        finalPosition = finalPosition +  
                       initialVelocity * fallingTime;  
        finalPosition = finalPosition + initialPosition;  
        System.out.println("An object's position after " +  
                           fallingTime + " seconds is " +  
                           finalPosition + " m.");  
    }  
}
```

# Exercise I

```
double finalPosition = .5 * gravity * fallingTime *  
                      fallingTime;  
finalPosition = finalPosition + initialVelocity  
                      * fallingTime;  
finalPosition = finalPosition + initialPosition;
```

**OR**

```
double finalPosition = .5 * gravity * fallingTime *  
                      fallingTime;  
finalPosition = finalPosition + initialVelocity  
                      * fallingTime;  
finalPosition += initialPosition;
```



# Questions from last lecture?

---

# Outline

---

- Lecture I Review
- **More types**
- Methods
- Conditionals

Division (“/”) operates differently on integers and on doubles!

Example:

```
double a = 5.0/2.0; // a =2.5
```

```
int b = 4/2; // b = 2
```

```
int c = 5/2; // c = 2
```

```
double d = 5/2; // d= 2.0
```

Precedence like math, left to right

Right hand side of = evaluated first

Parenthesis increase precedence

```
double x = 3 / 2 + 1; // x = 2.0  
double y = 3 / (2 + 1); // y = 1.0
```

Java verifies that types always match

```
String five = 5; // ERROR!
```

```
./Root/Main.java:8: error: incompatible types: int cannot be converted to String
    String five = 5;
               ^
1 error
```

## What is a casting?

- Taking an Object of one particular type and “turning it into” another Object type.

```
int a = 2;                      // a = 2
double a = 2;                    // a = 2.0 Implicit

int a = 18.7;                    // ERROR
int a = (int)18,7:              // a = 18

double a = 2/3;                  // a = 0.0
double a = (double)2/3;          // a = 0.666 ...

double d = 5.25;
int i = (int) d;                // d = 5 (Explicit) DOWNCAST

int d = 5;
double i = d;                   // i = 5.0 (Implicit) UPCAST
```

- Lecture | Review
- More types
- Methods
- Conditionals

# Methods

## Java Methods

- A collection of statements that are grouped together to perform an operation.
  - `System.out.println()`→
    - The system actually executes several statements in order to display a message on the console.
- The only required elements of a method declaration are the method's return type, name, a pair of parentheses, (), and a body between braces, {}.

# Methods

## Parts

```
class Main {  
    public static void main(String[] arguments)  
    {  
        System.out.println("Hello World");  
    }  
}
```

# Methods

Method declarations have six components:

- Modifiers.
- The return type (or void).
- The method name.
- The parameter list in parenthesis.
- An exception list.
- The method body, enclosed between braces.

```
class Main {  
    public static void main(String[] arguments)  
    {  
        System.out.println("Hello World");  
    }  
}
```

# Adding Methods

```
public static void NAME() {  
    STATEMENTS  
}
```

To call a method:

```
NAME();
```

# Methods

```
class NewLine {  
    public static void newLine() {  
        System.out.println("");  
    }  
  
    public static void threeLines() {  
        newLine();  
        newLine();  
        newLine();  
    }  
  
    public static void main(String[] arguments) {  
        System.out.println("Line 1");  
        threeLines();  
        System.out.println("Line 2");  
    }  
}
```



```
public static void NAME(TYPE NAME) {  
    STATEMENTS  
}
```

To call:

**NAME** (**EXPRESSION**) ;

# Parameters

```
class Square {  
    public static void printSquare(int x) {  
        System.out.println(x*x);  
    }  
  
    public static void main(String[] arguments) {  
        int value = 2;  
        printSquare(value);  
        printSquare(3);  
        printSquare(value*2);  
    }  
}
```

# Parameters

## What's wrong here?

```
class Square {  
    public static void printSquare(int x) {  
        System.out.println(x*x);  
    }  
  
    public static void main(String[] arguments) {  
        printSquare("hello");  
        printSquare(5.5);  
    }  
}
```

# Parameters

## What's wrong here?

```
class Square {  
    public static void printSquare(double x) {  
        System.out.println(x*x);  
    }  
  
    public static void main(String[] arguments) {  
        printSquare(5);  
    }  
}
```

# Multiple Parameters

```
[...] NAME (TYPE NAME, TYPE NAME) {  
    STATEMENTS  
}
```

To call:

```
NAME (arg1, arg2);
```

# Multiple Parameters

```
class Multiply {  
    public static void times (double a, double b) {  
        System.out.println(a * b);  
    }  
  
    public static void main(String[] arguments) {  
        times (2, 2);  
        times (3, 4);  
    }  
}
```

# Return Values

```
public static TYPE NAME () {  
    STATEMENTS  
    return EXPRESSION;  
}
```

**void** means “no returned value”

# Return Values

```
class Square3 {  
    public static void printSquare(double x) {  
        System.out.println(x*x);  
    }  
    public static void main(String[] arguments) {  
        printSquare(5);  
    }  
}
```

```
class Square4 {  
    public static double square(double x) {  
        return x*x;  
    }  
    public static void main(String[] arguments) {  
        System.out.println(square(5));  
        System.out.println(square(2));  
    }  
}
```

# Variable Scope

Variables live in the block ({} ) where they are defined (**scope**)

- Scope starts where the variable is declared
- ... and ends with the block where it was declared
- (the variable lives within the block)

Method parameters are like defining a new variable in the method

# Variable Scope

```
class SquareChange {  
    public static void printSquare(int x) {  
        System.out.println("printSquare x = " + x);  
        x = x * x;  
        System.out.println("printSquare x = " + x);  
    }  
  
    public static void main(String[] arguments) {  
        int x = 5;  
        System.out.println("main x = " + x);  
        printSquare(x);  
        System.out.println("main x = " + x);  
    }  
}
```

```
class Scope {  
    public static void main(String[] arguments) {  
        int x = 5;  
        if (x == 5) {  
            int x = 6;  
            int y = 72;  
            System.out.println("x = " + x + "  
                               y = " + y);  
        }  
    }  
    System.out.println("x = " + x + " y = " + y);  
}
```

# Methods: Building Blocks

## Methods as the way of encapsulating functionality

- Big programs are built out of small methods
- Methods can be individually developed, tested and reused
- User of method does not need to know how it works
  - Black box operations
- In Computer Science, this is called “abstraction”



# Mathematical Functions



Encapsulated  
functionality that we  
can use without  
having to master  
inner details

Overview Package Class Use Tree Deprecated Index Help

Prev Class Next Class Frames No Frames All Classes

Summary: Nested I Field I Constr I Method Detail: Field I Constr I Method

java.lang

**Class Math**

java.lang.Object  
java.lang.Math

---

public final class Math  
extends Object

The class `Math` contains methods for performing basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric functions.

Unlike some of the numeric methods of class `StrictMath`, all implementations of the equivalent functions of class `Math` are not defined to return the bit-for-bit same results. This relaxation permits better-performing implementations where strict reproducibility is not required.

By default many of the `Math` methods simply call the equivalent method in `StrictMath` for their implementation. Code generators are encouraged to use platform-specific native libraries or microprocessor instructions, where available, to provide higher-performance implementations of `Math` methods. Such higher-performance implementations still must conform to the specification for `Math`.

The quality of implementation specifications concern two properties, accuracy of the returned result and monotonicity of the method. Accuracy of the floating-point `Math` methods is measured in terms of *ulp*s, units in the last place. For a given floating-point format, an *ulp* of a specific real number value is the distance between the two floating-point values bracketing that numerical value. When discussing the accuracy of a method as a whole rather than at a specific argument, the number of *ulp*s cited is for the worst-case error at any argument. If a method always has an error less than 0.5 *ulp*, the method always returns the floating-point number nearest the exact result; such a method is *correctly rounded*. A correctly rounded method is generally the best a floating-point approximation can be; however, it is impractical for many floating-point methods to be correctly rounded. Instead, for the `Math` class, a larger error bound of 1 or 2 *ulp*s is allowed for certain methods. Informally, with a 1 *ulp* error bound, when the exact result is a representable number, the exact result should be returned as the computed result; otherwise, either of the two floating-point values which bracket the exact result may be returned. For exact results large in magnitude, one of the endpoints of the bracket may be infinite. Besides accuracy at individual arguments, maintaining proper relations between the method at different arguments is also important. Therefore, most methods with more than 0.5 *ulp* errors are required to be *semi-monotonic*: whenever the mathematical function is non-decreasing, so is the floating-point approximation, likewise, whenever the mathematical function is non-increasing, so is the floating-point approximation. Not all approximations that have 1 *ulp* accuracy will automatically meet the monotonicity requirements.

```
public class Main {  
    public static void main(String[] arguments) {  
        int x = 90;  
        Math.sin(x);  
        Math.cos(Math.PI / 2);  
        Math.pow(2, 3);  
        System.out.println(...);  
    }  
}
```

# Outline

---

- Lecture I Review
- More types
- Methods
- Conditionals

```
if (CONDITION) {
```

**STATEMENTS**

/\* statements performed  
when the boolean expression  
results true \*/

```
}
```

# if statement

```
public static void test(int x) {  
    if (x > 5) {  
        System.out.println(x + " is > 5");  
    }  
}  
  
public static void main(String[] arguments) {  
    test(6);  
    test(5);  
    test(4);  
}
```

# Comparison operators

`x > y`: x is greater than y

`x < y`: x is less than y

`x >= y`: x is greater than or equal to y

`x <= y`: x is less than or equal to y

`x == y`: x equals y

( equality: `==`, assignment: `=` )

`&&`: logical AND

`||`: logical OR

```
if (x > 6) {                                if ( x > 6 && x < 9) {  
    if (x < 9) { →...  
        }  
    ...  
}
```

else

```
if (CONDITION) {  
    STATEMENTS  
}  
else {  
    STATEMENTS  
    /* performed when CONDITION is  
       not true */  
}
```

```
public static void test(int x) {  
    if (x > 5) {  
        System.out.println(x + " is > 5");  
    } else {  
        System.out.println(x + " is not > 5");  
    }  
}  
  
public static void main(String[] arguments) {  
    test(6);  
    test(5);  
    test(4);  
}
```

```
if (CONDITION) {  
    STATEMENTS  
}  
else if (CONDITION) {  
    STATEMENTS  
}  
else if (CONDITION) {  
    STATEMENTS  
}  
else {  
    STATEMENTS  
}
```

```
public static void test(int x) {  
    if (x > 5) {  
        System.out.println(x + " is > 5");  
    } else if (x == 5) {  
        System.out.println(x + " equals 5");  
    } else {  
        System.out.println(x + " is < 5");  
    }  
}  
  
public static void main(String[] arguments) {  
    test(6);  
    test(5);  
    test(4);  
}
```



# Questions?

---

