Data Structures and Algorithms in Java

Procedural Programming: Basic Data Types

Outline

1. Data Types
2. Expressions
3. Statements
4. Strings
5. Integers
6. Doubles
7. Booleans
8. Operator Precedence

Data Types

Data Types

A data type specifies a range of values along with a set of operations defined on those values

Data Types

A data type specifies a range of values along with a set of operations defined on those values

Java supports basic and reference data types

Data Types

A data type specifies a range of values along with a set of operations defined on those values

Java supports basic and reference data types

Eight basic data types

1. `boolean` for true and false values with logical operations
2. `byte` for 8-bit integers with arithmetic operations
3. `char` for 16-bit characters with arithmetic operations
4. `short` for 16-bit integers with arithmetic operations
5. `int` for 32-bit integers with arithmetic operations
6. `float` for 32-bit single-precision real numbers with arithmetic operations
7. `long` for 64-bit integers with arithmetic operations
8. `double` for 64-bit double-precision real numbers with arithmetic operations

A literal represents a basic data-type value

A literal represents a basic data-type value

Example

- `true` and `false` are `boolean` literals
- `'*'` is a `char` literal
- `42` is an `int` literal
- `1729L` is a `long` literal
- `3.14159D` is a `double` literal

An identifier represents a name

An identifier represents a name

Each identifier is a sequence of letters, digits, underscore symbols, and dollar symbols, not starting with a digit

An identifier represents a name

Each identifier is a sequence of letters, digits, underscore symbols, and dollar symbols, not starting with a digit

Example: `abc`, `abc_`, `aBC123`, and `$abc` are valid identifiers whereas `abc*`, `1abc`, and `abc+` are not

A variable associates a name with a data-type value

A variable associates a name with a data-type value

Example: `age`

A variable associates a name with a data-type value

Example: `age`

A constant variable is one whose value does not change during the execution of a program

A variable associates a name with a data-type value

Example: `age`

A constant variable is one whose value does not change during the execution of a program

Example: `SPEED_OF_LIGHT`

A variable associates a name with a data-type value

Example: `age`

A constant variable is one whose value does not change during the execution of a program

Example: `SPEED_OF_LIGHT`

A variable's value is accessed as `<name>` or `<target>.<name>`

A variable associates a name with a data-type value

Example: `age`

A constant variable is one whose value does not change during the execution of a program

Example: `SPEED_OF_LIGHT`

A variable's value is accessed as `<name>` or `<target>.<name>`

Example: `age`, `SPEED_OF_LIGHT`, and `Math.PI`

An operator represents a data-type operation

An operator represents a data-type operation

Example

- `+`, `-`, `*`, `/`, and `%` represent arithmetic operations
- `!`, `||`, and `&&` represent logical operations

Many programming tasks involve not only operators, but also functions

Many programming tasks involve not only operators, but also functions

We will use functions

1. From implicitly imported system libraries (`java.lang` package)
2. From explicitly imported third-party libraries (`stdlib` and `dsa` packages)
3. That we define ourselves

Many programming tasks involve not only operators, but also functions

We will use functions

1. From implicitly imported system libraries (`java.lang` package)
2. From explicitly imported third-party libraries (`stdlib` and `dsa` packages)
3. That we define ourselves

A function is called as `<name>(<arg1>, <arg2>, ...)` or `<library>.<name>(<arg1>, <arg2>, ...)`

Many programming tasks involve not only operators, but also functions

We will use functions

1. From implicitly imported system libraries (`java.lang` package)
2. From explicitly imported third-party libraries (`stdlib` and `dsa` packages)
3. That we define ourselves

A function is called as `<name>(<arg1>, <arg2>, ...)` or `<library>.<name>(<arg1>, <arg2>, ...)`

Example: `StdOut.println("Hello, World")` and `Math.sqrt(2)`

Many programming tasks involve not only operators, but also functions

We will use functions

1. From implicitly imported system libraries (`java.lang` package)
2. From explicitly imported third-party libraries (`stdlib` and `dsa` packages)
3. That we define ourselves

A function is called as `<name>(<arg1>, <arg2>, ...)` or `<library>.<name>(<arg1>, <arg2>, ...)`

Example: `StdOut.println("Hello, World")` and `Math.sqrt(2)`

A function that does not return a value is called a void function (eg, `StdOut.println()`)

Many programming tasks involve not only operators, but also functions

We will use functions

1. From implicitly imported system libraries (java.lang package)
2. From explicitly imported third-party libraries (stdlib and dsa packages)
3. That we define ourselves

A function is called as <name>(<arg1>, <arg2>, ...) or <library>.<name>(<arg1>, <arg2>, ...)

Example: StdOut.println("Hello, World") and Math.sqrt(2)

A function that does not return a value is called a void function (eg, StdOut.println())

A function that returns a value is called a non-void function (eg, Math.sqrt())

Example

java.lang.Math

`static double pow(double x, double y)`	returns x^y
`static double sqrt(double x)`	returns $\sqrt{\mathtt{x}}$

java.lang.System

`static void exit(int x)`	shuts down the JVM with exit code `x`

java.lang.Integer

`static int parseInt(String s)`	returns `int` value of `s`

java.lang.Double

`static double parseDouble(String s)`	returns `double` value of `s`

Example

stdlib.StdOut

`static void println(Object x)`	prints an object and a newline to standard output
`static void print(Object x)`	prints an object to standard output

stdlib.StdRandom

`static double uniform(double a, double b)`	returns a double chosen uniformly at random from the interval `[a, b)`
`static boolean bernoulli(double p)`	returns `true` with probability `p` and `false` with probability `1 - p`

stdlib.StdStats

`static double mean(double[] a)`	returns the average value in the array `a`
`static double stddev(double[] a)`	returns the sample standard deviation in the array `a`

Expressions

Expressions

An expression is a combination of literals, variables, operators, and non-void function calls

Expressions

An expression is a combination of literals, variables, operators, and non-void function calls

Every expression has a type and a value

Expressions

An expression is a combination of literals, variables, operators, and non-void function calls

Every expression has a type and a value

Example

- `2, 4`
- `a, b, c`
- `b * b - 4 * a * c`
- `Math.sqrt(b * b - 4 * a * c)`
- `(-b + Math.sqrt(b * b - 4 * a * c)) / (2 * a)`

Statements

Statements

A syntactic unit that expresses some action to be carried out

Statements

A syntactic unit that expresses some action to be carried out

Example

```
import stdlib.StdOut;

String message = "Hello, World";
StdOut.println(message);
```

```
import <library>;
```

```
import <library>;
```

Example

```
import stdlib.StdOut;
import dsa.BinarySearch;
```

Statements · Function Call Statements

```
<name>(<arg1>, <arg2>, ...);
<library>.<name>(<arg1>, <arg2>, ...);
```

```
<name>(<arg1>, <arg2>, ...);
<library>.<name>(<arg1>, <arg2>, ...);
```

Example

```
StdOut.println("To be, or not to be, that is the question.");
System.exit(0);
```

Declaration statement

```
<type> <name>;
<type> <name1>, <name2>, <name3>, ...;
```

Initial value is `false` for `boolean` type, `0` for other basic types, and `null` for reference types

Declaration statement

```
<type> <name>;
<type> <name1>, <name2>, <name3>, ...;
```

Initial value is `false` for `boolean` type, `0` for other basic types, and `null` for reference types

Assignment statement

```
<name> = <expression>;
```

Declaration statement

```
<type> <name>;
<type> <name1>, <name2>, <name3>, ...;
```

Initial value is `false` for `boolean` type, `0` for other basic types, and `null` for reference types

Assignment statement

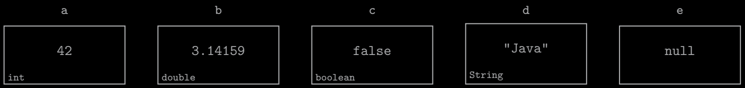
```
<name> = <expression>;
```

Initialization statement

```
<type> <name> = <expression>;
<type> <name1> = <expression1>, <name2> = <expression2>, <name3> = <expression3>, ...;
```

Example

```java
int a = 42;
double b = 3.14159;
boolean c;
String d = "Java", e;
```



Example (swapping idiom)

```
int a = 42;
int b = 1729;

int t = a;
a = b;
b = t;

StdOut.println(a);
StdOut.println(b);
```

Variable Trace

line #	a	b	t

>_

Example (swapping idiom)

```
int a = 42;
int b = 1729;

int t = a;
a = b;
b = t;

StdOut.println(a);
StdOut.println(b);
```

Variable Trace

line #	a	b	t
1	42		

```
>_
```

Example (swapping idiom)

```
1 int a = 42;
2 int b = 1729;
3
4 int t = a;
5 a = b;
6 b = t;
7
8 StdOut.println(a);
9 StdOut.println(b);
```

Variable Trace

line #	a	b	t
2	42	1729	

>_

Example (swapping idiom)

```java
1 int a = 42;
2 int b = 1729;
3
4 int t = a;
5 a = b;
6 b = t;
7
8 StdOut.println(a);
9 StdOut.println(b);
```

Variable Trace

line #	a	b	t
4	42	1729	42

Example (swapping idiom)

```
1 int a = 42;
2 int b = 1729;
3
4 int t = a;
5 a = b;
6 b = t;
7
8 StdOut.println(a);
9 StdOut.println(b);
```

Variable Trace

line #	a	b	t
5	1729	1729	42

>_

Example (swapping idiom)

```
1 int a = 42;
2 int b = 1729;
3
4 int t = a;
5 a = b;
6 b = t;
7
8 StdOut.println(a);
9 StdOut.println(b);
```

Variable Trace

line #	a	b	t
6	1729	42	42

>_

Example (swapping idiom)

```
int a = 42;
int b = 1729;

int t = a;
a = b;
b = t;

StdOut.println(a);
StdOut.println(b);
```

Variable Trace

line #	a	b	t
8	1729	42	42

>_

```
1729
```

Example (swapping idiom)

```java
int a = 42;
int b = 1729;

int t = a;
a = b;
b = t;

StdOut.println(a);
StdOut.println(b);
```

Variable Trace

line #	a	b	t
9	1729	42	42

```
>_
1729
42
```

Example (swapping idiom)

```java
int a = 42;
int b = 1729;

int t = a;
a = b;
b = t;

StdOut.println(a);
StdOut.println(b);
```

Variable Trace

line #	a	b	t

```
>_
1729
42
```

Example (variable update)

```
int x = 1;
x = x * 10;
x = x / 2;
x = x % 3;
x = x + 2;
x = x - 1;

StdOut.println(x);
```

Variable Trace

line #	x

>_

Example (variable update)

```
int x = 1;
x = x * 10;
x = x / 2;
x = x % 3;
x = x + 2;
x = x - 1;

StdOut.println(x);
```

Variable Trace

line #	x
1	1

```
>_
```

Example (variable update)

```
1 int x = 1;
2 x = x * 10;
3 x = x / 2;
4 x = x % 3;
5 x = x + 2;
6 x = x - 1;
7
8 StdOut.println(x);
```

Variable Trace

line #	x
2	10

>_

Example (variable update)

```java
int x = 1;
x = x * 10;
x = x / 2;
x = x % 3;
x = x + 2;
x = x - 1;

StdOut.println(x);
```

Variable Trace

line #	x
3	5

>_

Example (variable update)

```
int x = 1;
x = x * 10;
x = x / 2;
x = x % 3;
x = x + 2;
x = x - 1;

StdOut.println(x);
```

Variable Trace

line #	x
4	2

>_

Example (variable update)

```
1 int x = 1;
2 x = x * 10;
3 x = x / 2;
4 x = x % 3;
5 x = x + 2;
6 x = x - 1;
7
8 StdOut.println(x);
```

Variable Trace

line #	x
5	4

>_

Example (variable update)

```
int x = 1;
x = x * 10;
x = x / 2;
x = x % 3;
x = x + 2;
x = x - 1;

StdOut.println(x);
```

Variable Trace

line #	x
6	3

>_

Example (variable update)

```
int x = 1;
x = x * 10;
x = x / 2;
x = x % 3;
x = x + 2;
x = x - 1;

StdOut.println(x);
```

Variable Trace

line #	x
8	3

```
>_
3
```

Example (variable update)

```
int x = 1;
x = x * 10;
x = x / 2;
x = x % 3;
x = x + 2;
x = x - 1;

StdOut.println(x);
```

Variable Trace

line #	x

```
>_
3
```

The assignment statement

```
<name> = <name> <operator> <expression>
```

is equivalent to

```
<name> <operator>= <expression>
```

where `<operator>` is `*`, `/`, `%`, `+`, or `-`

The assignment statement

```
<name> = <name> <operator> <expression>
```

is equivalent to

```
<name> <operator>= <expression>
```

where `<operator>` is `*`, `/`, `%`, `+`, or `-`

Example

```
x = x * 10;
x = x / 2;
x = x % 3;
x = x + 2;
x = x - 1;
```

are equivalent to

```
x *= 10;
x /= 2;
x %= 3;
x += 2;
x -= 1;
```

More equivalent assignment statements

```
x = x + 1;
x++;
++x;
```

```
x = x - 1;
x--;
--x;
```

Strings

Strings

The `String` data type, which is a reference type, represents strings (sequences of characters)

Strings

The `String` data type, which is a reference type, represents strings (sequences of characters)

A `String` literal is specified by enclosing a sequence of characters in matching double quotes

Strings

The `String` data type, which is a reference type, represents strings (sequences of characters)

A `String` literal is specified by enclosing a sequence of characters in matching double quotes

Example: `"Hello, World"`

Strings

The `String` data type, which is a reference type, represents strings (sequences of characters)

A `String` literal is specified by enclosing a sequence of characters in matching double quotes

Example: `"Hello, World"`

Tab, newline, backslash, and double quote characters are specified using escape sequences `"\t"`, `"\n"`, `"\\"`, and `"\""`

Strings

The `String` data type, which is a reference type, represents strings (sequences of characters)

A `String` literal is specified by enclosing a sequence of characters in matching double quotes

Example: `"Hello, World"`

Tab, newline, backslash, and double quote characters are specified using escape sequences `"\t"`, `"\n"`, `"\\"`, and `"\""`

Example: `"Hello, world\n"`

Strings

The `String` data type, which is a reference type, represents strings (sequences of characters)

A `String` literal is specified by enclosing a sequence of characters in matching double quotes

Example: `"Hello, World"`

Tab, newline, backslash, and double quote characters are specified using escape sequences `"\t"`, `"\n"`, `"\\"`, and `"\""`

Example: `"Hello, world\n"`

Operations

- Concatenation (`+`) — eg, `"123" + "456"` and `"123" + 456` evaluate to `"123456"`

`DateFormats.java`

Command-line input	*d* (String), *m* (String), and *y* (String) representing a date
Standard output	the date in different formats

DateFormats.java

Command-line input	*d* (String), *m* (String), and *y* (String) representing a date
Standard output	the date in different formats

~/workspace/dsaj

```
$ _
```

DateFormats.java

Command-line input	*d* (String), *m* (String), and *y* (String) representing a date
Standard output	the date in different formats

~/workspace/dsaj

```
$ java DateFormats 14 03 1879
```

DateFormats.java

Command-line input	*d* (String), *m* (String), and *y* (String) representing a date
Standard output	the date in different formats

~/workspace/dsaj

```
$ java DateFormats 14 03 1879
14/03/1879
03/14/1879
1879/03/14
$ _
```

Strings · Example (Date Formats)

DateFormats.java

```java
import stdlib.StdOut;

public class DateFormats {
    public static void main(String[] args) {
        String d = args[0];
        String m = args[1];
        String y = args[2];
        String dmy = d + "/" + m + "/" + y;
        String mdy = m + "/" + d + "/" + y;
        String ymd = y + "/" + m + "/" + d;
        StdOut.println(dmy);
        StdOut.println(mdy);
        StdOut.println(ymd);
    }
}
```

Integers

Integers

The `int` data type represents integers

Integers

The `int` data type represents integers

An `int` literal is specified as a sequence of digits `0` through `9`

Integers

The `int` data type represents integers

An `int` literal is specified as a sequence of digits `0` through `9`

Example: `42`

Integers

The `int` data type represents integers

An `int` literal is specified as a sequence of digits `0` through `9`

Example: `42`

Operations

- Addition (`+`) — eg, `5 + 2` evaluates to `7`
- Subtraction/negation (`-`) — eg, `5 - 2` evaluates to `3` and `-(-3)` evaluates to `3`
- Multiplication (`*`) — eg, `5 * 2` evaluates to `10`
- Division (`/`) — eg, `5 / 2` evaluates to `2`
- Remainder (`%`) — eg, `5 % 2` evaluates to `1`

Integers · Example (Sum of Squares)

SumOfSquares.java

Command-line input	x (int) and y (int)
Standard output	$x^2 + y^2$

Integers · Example (Sum of Squares)

SumOfSquares.java

Command-line input	x (int) and y (int)
Standard output	$x^2 + y^2$

~/workspace/dsaj

```
$ _
```

SumOfSquares.java

Command-line input	x (int) and y (int)
Standard output	$x^2 + y^2$

~/workspace/dsaj

```
$ java SumOfSquares 3 4
```

SumOfSquares.java

Command-line input	x (int) and y (int)
Standard output	$x^2 + y^2$

~/workspace/dsaj

```
$ java SumOfSquares 3 4
25
$ _
```

SumOfSquares.java

Command-line input	x (int) and y (int)
Standard output	$x^2 + y^2$

~/workspace/dsaj

```
$ java SumOfSquares 3 4
25
$ java SumOfSquares 6 8
```

SumOfSquares.java

Command-line input	x (int) and y (int)
Standard output	$x^2 + y^2$

~/workspace/dsaj

```
$ java SumOfSquares 3 4
25
$ java SumOfSquares 6 8
100
$ _
```

Integers · Example (Sum of Squares)

SumOfSquares.java

```
import stdlib.StdOut;

public class SumOfSquares {
    public static void main(String[] args) {
        int x = Integer.parseInt(args[0]);
        int y = Integer.parseInt(args[1]);
        int result = x * x + y * y;
        StdOut.println(result);
    }
}
```

Doubles

Doubles

The `double` data type represents floating-point numbers

Doubles

The `double` data type represents floating-point numbers

A floating-point literal is specified as a sequence of digits with a decimal point

Doubles

The `double` data type represents floating-point numbers

A floating-point literal is specified as a sequence of digits with a decimal point

Example: `3.14159`

Doubles

The `double` data type represents floating-point numbers

A floating-point literal is specified as a sequence of digits with a decimal point

Example: `3.14159`

Scientific notation can be used to represent very large and very small numbers

Doubles

The `double` data type represents floating-point numbers

A floating-point literal is specified as a sequence of digits with a decimal point

Example: `3.14159`

Scientific notation can be used to represent very large and very small numbers

Example: `6.022e23` represents 6.022×10^{23} and `6.674e-11` represents 6.674×10^{-11}

Doubles

The `double` data type represents floating-point numbers

A floating-point literal is specified as a sequence of digits with a decimal point

Example: `3.14159`

Scientific notation can be used to represent very large and very small numbers

Example: `6.022e23` represents 6.022×10^{23} and `6.674e-11` represents 6.674×10^{-11}

Operations

- Addition (`+`) — eg, `16.0 + 0.5` evaluates to `16.5`
- Subtraction/negation (`-`) — eg, `16.0 - 0.5` evaluates to `15.5` and `-(-3.0)` evaluates to `3.0`
- Multiplication (`*`) — eg, `16.0 * 0.5` evaluates to `8.0`
- Division (`/`) — eg, `16.0 / 0.5` evaluates to `32.0`

Quadratic.java

Command-line input	a (double), b (double), and c (double)
Standard output	the two roots of the quadratic equation $ax^2 + bx + c = 0$, computed as $\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

Quadratic.java

Command-line input	a (double), b (double), and c (double)
Standard output	the two roots of the quadratic equation $ax^2 + bx + c = 0$, computed as $\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

~/workspace/dsaj

```
$ _
```

Quadratic.java

Command-line input	a (double), b (double), and c (double)
Standard output	the two roots of the quadratic equation $ax^2 + bx + c = 0$, computed as $\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

~/workspace/dsaj

```
$ java Quadratic 1 -5 6
```

Quadratic.java

Command-line input	a (double), b (double), and c (double)
Standard output	the two roots of the quadratic equation $ax^2 + bx + c = 0$, computed as $\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

~/workspace/dsaj

```
$ java Quadratic 1 -5 6
Root 1 = 3.0
Root 2 = 2.0
$ _
```

Quadratic.java

Command-line input	a (double), b (double), and c (double)
Standard output	the two roots of the quadratic equation $ax^2 + bx + c = 0$, computed as $\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

~/workspace/dsaj

```
$ java Quadratic 1 -5 6
Root 1 = 3.0
Root 2 = 2.0
$ java Quadratic 1 -1 -1
```

Quadratic.java

Command-line input	a (double), b (double), and c (double)
Standard output	the two roots of the quadratic equation $ax^2 + bx + c = 0$, computed as $\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

~/workspace/dsaj

```
$ java Quadratic 1 -5 6
Root 1 = 3.0
Root 2 = 2.0
$ java Quadratic 1 -1 -1
Root 1 = 1.618033988749895
Root 2 = -0.6180339887498949
$ _
```

Doubles · Example (Quadratic Formula)

Doubles · Example (Quadratic Formula)

</> Quadratic.java

```java
import stdlib.StdOut;

public class Quadratic {
    public static void main(String[] args) {
        double a = Double.parseDouble(args[0]);
        double b = Double.parseDouble(args[1]);
        double c = Double.parseDouble(args[2]);
        double discriminant = b * b - 4 * a * c;
        double root1 = (-b + Math.sqrt(discriminant)) / (2 * a);
        double root2 = (-b - Math.sqrt(discriminant)) / (2 * a);
        StdOut.println("Root # 1 = " + root1);
        StdOut.println("Root # 2 = " + root2);
    }
}
```

Booleans

Booleans

The `boolean` data type represents truth values (true or false) from logic

Booleans

The `boolean` data type represents truth values (true or false) from logic

The two `boolean` literals are `true` and `false`

Booleans

The `boolean` data type represents truth values (true or false) from logic

The two `boolean` literals are `true` and `false`

Operations

- Logical not (`!`)
- Logical or (`||`)
- Logical and (`&&`)

Booleans

The `boolean` data type represents truth values (true or false) from logic

The two `boolean` literals are `true` and `false`

Operations

- Logical not (`!`)
- Logical or (`||`)
- Logical and (`&&`)

Truth tables for the logical operations

x	!x
false	true
true	false

x	y	x || y
false	false	false
false	true	true
true	false	true
true	true	true

x	y	x && y
false	false	false
false	true	false
true	false	false
true	true	true

Two values of the same basic type can be compared using comparison operators, the result of which is a boolean value

Booleans · Comparison Operators

Two values of the same basic type can be compared using comparison operators, the result of which is a boolean value

Comparison operators

- Equal (`==`) — eg, `5 == 2` evaluates to `false`
- Not equal (`!=`) — eg, `5 != 2` evaluates to `true`
- Less than (`<`) — eg, `5 < 2` evaluates to `false`
- Less than or equal (`<=`) — eg, `5 <= 2` evaluates to `false`
- Greater than (`>`) — eg, `5 > 2` evaluates to `true`
- Greater than or equal (`>=`) — eg, `5 >= 2` evaluates to `true`

`LeapYear.java`

Command-line input	y (int)
Standard output	`true` if y is a leap year and `false` otherwise

Booleans · Example (Leap Year)

LeapYear.java

Command-line input	y (int)
Standard output	`true` if y is a leap year and `false` otherwise

~/workspace/dsaj

```
$ _
```

Booleans · Example (Leap Year)

LeapYear.java

Command-line input	y (int)
Standard output	`true` if y is a leap year and `false` otherwise

~/workspace/dsaj

```
$ java LeapYear 2020
```

LeapYear.java

Command-line input	y (int)
Standard output	`true` if y is a leap year and `false` otherwise

~/workspace/dsaj

```
$ java LeapYear 2020
true
$ _
```

LeapYear.java

Command-line input	y (int)
Standard output	`true` if y is a leap year and `false` otherwise

~/workspace/dsaj

```
$ java LeapYear 2020
true
$ java LeapYear 1900
```

Booleans · Example (Leap Year)

LeapYear.java

Command-line input	y (int)
Standard output	`true` if y is a leap year and `false` otherwise

~/workspace/dsaj

```
$ java LeapYear 2020
true
$ java LeapYear 1900
false
$ _
```

LeapYear.java

Command-line input	y (int)
Standard output	`true` if y is a leap year and `false` otherwise

~/workspace/dsaj

```
$ java LeapYear 2020
true
$ java LeapYear 1900
false
$ java LeapYear 2000
```

LeapYear.java

Command-line input	y (int)
Standard output	`true` if y is a leap year and `false` otherwise

~/workspace/dsaj

```
$ java LeapYear 2020
true
$ java LeapYear 1900
false
$ java LeapYear 2000
true
$ _
```

LeapYear.java

Command-line input	y (int)
Standard output	`true` if y is a leap year and `false` otherwise

~/workspace/dsaj

```
$ java LeapYear 2020
true
$ java LeapYear 1900
false
$ java LeapYear 2000
true
$ _
```

A leap year is one that is divisible by 4 *and* is not divisible by 100 *or* is divisible by 400

Booleans · Example (Leap Year)

Booleans · Example (Leap Year)

</> LeapYear.java

```java
import stdlib.StdOut;

public class LeapYear {
    public static void main(String[] args) {
        int y = Integer.parseInt(args[0]);
        boolean result = y % 4 == 0 && y % 100 != 0 || y % 400 == 0;
        StdOut.println(result);
    }
}
```

Operator Precedence

Operator Precedence

From highest to lowest

<table>
<tr><td>+, -</td><td>unary</td></tr>
<tr><td>*, /, %</td><td>multiplicative</td></tr>
<tr><td>+, -</td><td>additive</td></tr>
<tr><td><, <=, >, >=</td><td>comparison</td></tr>
<tr><td>==, !=</td><td>comparison</td></tr>
<tr><td>=, *=, /=, %=, +=, -=</td><td>assignment</td></tr>
<tr><td>!, ||, &&</td><td>logical</td></tr>
</table>

Operator Precedence

From highest to lowest

`+, -`	unary
`*, /, %`	multiplicative
`+, -`	additive
`<, <=, >, >=`	comparison
`==, !=`	comparison
`=, *=, /=, %=, +=, -=`	assignment
`!, ||, &&`	logical

Example: `2 + 3 * 4` evaluates to `14`

Operator Precedence

From highest to lowest

`+, -`	unary
`*, /, %`	multiplicative
`+, -`	additive
`<, <=, >, >=`	comparison
`==, !=`	comparison
`=, *=, /=, %=, +=, -=`	assignment
`!, ||, &&`	logical

Example: `2 + 3 * 4` evaluates to `14`

Parentheses can be used to override precedence rules

Operator Precedence

From highest to lowest

`+, -`	unary
`*, /, %`	multiplicative
`+, -`	additive
`<, <=, >, >=`	comparison
`==, !=`	comparison
`=, *=, /=, %=, +=, -=`	assignment
`!, ||, &&`	logical

Example: `2 + 3 * 4` evaluates to `14`

Parentheses can be used to override precedence rules

Example: `(2 + 3) * 4` evaluates to `20`