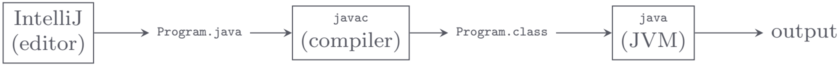
Programming Model

Outline

Programming in Java

Programming in Java

The Java workflow



Programming in Java

The Java workflow

IntelliJ (editor) → `Program.java` → `javac` (compiler) → `Program.class` → `java` (JVM) → output

Program.java

```
[package dsa;]

// Import statements.
...

// Class definition.
public class Program [implements <name>] {
    // Field declarations.
    ...

    // Constructor definitions.
    ...

    // Method definitions.
    ...

    // Function definitions.
    ...

    // Inner class definitions.
    ...
}
```

Programming in Java

Programming in Java

Program: `HelloWorld.java`

Programming in Java

Program: `HelloWorld.java`

$\rightsquigarrow$ Standard output: the message "Hello, World"

Programming in Java

Program: `HelloWorld.java`

⇝ Standard output: the message "Hello, World"

```
>_ ~/workspace/dsa/programs
$ _
```

Program: `HelloWorld.java`

⇝ Standard output: the message "Hello, World"

```
>_ ~/workspace/dsa/programs

$ javac -d out src/HelloWorld.java
```

Programming in Java

Program: `HelloWorld.java`

⇝ Standard output: the message “Hello, World”

```
>_ ~/workspace/dsa/programs

$ javac -d out src/HelloWorld.java
$ _
```

Programming in Java

Program: `HelloWorld.java`

⇝ Standard output: the message “Hello, World”

```
>_ ~/workspace/dsa/programs

$ javac -d out src/HelloWorld.java
$ java HelloWorld
```

Programming in Java

Program: `HelloWorld.java`

⇝ Standard output: the message "Hello, World"

```
>_ ~/workspace/dsa/programs

$ javac -d out src/HelloWorld.java
$ java HelloWorld
Hello, World
$ _
```

Programming in Java

Programming in Java

HelloWorld.java

```java
// Writes the message "Hello, World" to standard output.

import stdlib.StdOut;

public class HelloWorld {
    // Entry point.
    public static void main(String[] args) {
        StdOut.println("Hello, World");
    }
}
```

Programming in Java

The application programming interface (API) for a library provides a summary of the functions in the library

The application programming interface (API) for a library provides a summary of the functions in the library

Example

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

Errors in a Program

Errors in a Program

Syntax errors are identified and reported by `javac` when it compiles a program

Errors in a Program

Syntax errors are identified and reported by `javac` when it compiles a program

Example

HelloWorld.java

```java
// Writes the message "Hello, World" to standard output.

import stdlib.StdOut;

public class HelloWorld {
    // Entry point.
    public static void main(String[] args) {
        StdOut.println("Hello, World")
    }
}
```

Errors in a Program

Syntax errors are identified and reported by `javac` when it compiles a program

Example

HelloWorld.java

```java
// Writes the message "Hello, World" to standard output.

import stdlib.StdOut;

public class HelloWorld {
    // Entry point.
    public static void main(String[] args) {
        StdOut.println("Hello, World")
    }
}
```

~/workspace/dsa/programs

```
$ _
```

Errors in a Program

Syntax errors are identified and reported by `javac` when it compiles a program

Example

HelloWorld.java

```java
// Writes the message "Hello, World" to standard output.

import stdlib.StdOut;

public class HelloWorld {
    // Entry point.
    public static void main(String[] args) {
        StdOut.println("Hello, World")
    }
}
```

~/workspace/dsa/programs

```
$ javac -d out src/HelloWorld.java
```

Errors in a Program

Syntax errors are identified and reported by `javac` when it compiles a program

Example

HelloWorld.java

```java
// Writes the message "Hello, World" to standard output.

import stdlib.StdOut;

public class HelloWorld {
    // Entry point.
    public static void main(String[] args) {
        StdOut.println("Hello, World")
    }
}
```

~/workspace/dsa/programs

```
$ javac -d out src/HelloWorld.java
HelloWorld.java:8: error: ';' expected
        StdOut.println("Hello, World")
                                      ^
1 error
$ _
```

Errors in a Program

Errors in a Program

Semantic errors are also identified and reported by `javac` when it compiles a program

Errors in a Program

Semantic errors are also identified and reported by `javac` when it compiles a program

Example

HelloWorld.java

```java
// Writes the message "Hello, World" to standard output.

public class HelloWorld {
    // Entry point.
    public static void main(String[] args) {
        StdOut.println("Hello, World");
    }
}
```

Errors in a Program

Semantic errors are also identified and reported by `javac` when it compiles a program

Example

HelloWorld.java

```java
// Writes the message "Hello, World" to standard output.

public class HelloWorld {
    // Entry point.
    public static void main(String[] args) {
        StdOut.println("Hello, World");
    }
}
```

~/workspace/dsa/programs

```
$ _
```

Errors in a Program

Semantic errors are also identified and reported by `javac` when it compiles a program

Example

HelloWorld.java

```java
// Writes the message "Hello, World" to standard output.

public class HelloWorld {
    // Entry point.
    public static void main(String[] args) {
        StdOut.println("Hello, World");
    }
}
```

~/workspace/dsa/programs

```
$ javac -d out src/HelloWorld.java
```

Errors in a Program

Semantic errors are also identified and reported by `javac` when it compiles a program

Example

HelloWorld.java

```java
// Writes the message "Hello, World" to standard output.

public class HelloWorld {
    // Entry point.
    public static void main(String[] args) {
        StdOut.println("Hello, World");
    }
}
```

~/workspace/dsa/programs

```
$ javac -d out src/HelloWorld.java
HelloWorld.java:6: error: cannot find symbol
        StdOut.println("Hello, World");
        ^
  symbol:   variable StdOut
  location: class HelloWorld
1 error
$ _
```

Errors in a Program

Errors in a Program

Logic errors are not identified or reported by `javac` or `java`, but produce unintended output

Errors in a Program

Logic errors are not identified or reported by `javac` or `java`, but produce unintended output

Example

HelloWorld.java

```java
// Writes the message "Hello, World" to standard output.

import stdlib.StdOut;

public class HelloWorld {
    // Entry point.
    public static void main(String[] args) {
        StdOut.print("Hello, World");
    }
}
```

Errors in a Program

Logic errors are not identified or reported by `javac` or `java`, but produce unintended output

Example

HelloWorld.java

```java
// Writes the message "Hello, World" to standard output.

import stdlib.StdOut;

public class HelloWorld {
    // Entry point.
    public static void main(String[] args) {
        StdOut.print("Hello, World");
    }
}
```

~/workspace/dsa/programs

```
$ _
```

Errors in a Program

Logic errors are not identified or reported by `javac` or `java`, but produce unintended output

Example

HelloWorld.java

```java
// Writes the message "Hello, World" to standard output.

import stdlib.StdOut;

public class HelloWorld {
    // Entry point.
    public static void main(String[] args) {
        StdOut.print("Hello, World");
    }
}
```

~/workspace/dsa/programs

```
$ javac -d out src/HelloWorld.java
```

Errors in a Program

Logic errors are not identified or reported by `javac` or `java`, but produce unintended output

Example

HelloWorld.java

```java
// Writes the message "Hello, World" to standard output.

import stdlib.StdOut;

public class HelloWorld {
    // Entry point.
    public static void main(String[] args) {
        StdOut.print("Hello, World");
    }
}
```

~/workspace/dsa/programs

```
$ javac -d out src/HelloWorld.java
$ _
```

Errors in a Program

Logic errors are not identified or reported by `javac` or `java`, but produce unintended output

Example

HelloWorld.java

```java
// Writes the message "Hello, World" to standard output.

import stdlib.StdOut;

public class HelloWorld {
    // Entry point.
    public static void main(String[] args) {
        StdOut.print("Hello, World");
    }
}
```

~/workspace/dsa/programs

```
$ javac -d out src/HelloWorld.java
$ java HelloWorld
```

Errors in a Program

Logic errors are not identified or reported by `javac` or `java`, but produce unintended output

Example

HelloWorld.java

```java
// Writes the message "Hello, World" to standard output.

import stdlib.StdOut;

public class HelloWorld {
    // Entry point.
    public static void main(String[] args) {
        StdOut.print("Hello, World");
    }
}
```

~/workspace/dsa/programs

```
$ javac -d out src/HelloWorld.java
$ java HelloWorld
Hello, World$ _
```

Input and Output

Input and Output

input $\longrightarrow$ `Program.java` $\longrightarrow$ output

Input and Output

input $\longrightarrow$ `Program.java` $\longrightarrow$ output

Input types:

Input and Output

$$\text{input} \longrightarrow \boxed{\texttt{Program.java}} \longrightarrow \text{output}$$

Input types:

- ⇝ Command-line input

Input and Output

$$\text{input} \longrightarrow \boxed{\texttt{Program.java}} \longrightarrow \text{output}$$

Input types:

- ⇝ Command-line input
- ⇝ Standard input

Input and Output

input $\longrightarrow$ $\boxed{\texttt{Program.java}}$ $\longrightarrow$ output

Input types:

- ⇝ Command-line input
- ⇝ Standard input
- ⇝ File input

Input and Output

input $\longrightarrow$ `Program.java` $\longrightarrow$ output

Input types:

- ⇝ Command-line input
- ⇝ Standard input
- ⇝ File input

Output types:

Input and Output

input $\longrightarrow$ `Program.java` $\longrightarrow$ output

Input types:

- Command-line input
- Standard input
- File input

Output types:

- Standard output

Input and Output

input $\longrightarrow$ `Program.java` $\longrightarrow$ output

Input types:

- ⇝ Command-line input
- ⇝ Standard input
- ⇝ File input

Output types:

- ⇝ Standard output
- ⇝ File output

Command-line inputs are strings listed right next to the program name during execution

```
>_ ~/workspace/ipp/programs

$ java Program input1 input2 input3 ...
```

Command-line inputs are strings listed right next to the program name during execution

```
>_ ~/workspace/ipp/programs

$ java Program input1 input2 input3 ...
```

The inputs are accessed within the entry point function in the program as `args[0]`, `args[1]`, `args[2]`, and so on

Input and Output · Command-line Input

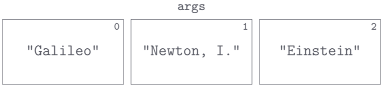
Command-line inputs are strings listed right next to the program name during execution

```
>_ ~/workspace/ipp/programs
$ java Program input1 input2 input3 ...
```

The inputs are accessed within the entry point function in the program as `args[0]`, `args[1]`, `args[2]`, and so on

Example

```
>_ ~/workspace/ipp/programs
$ java Program Galileo "Newton, I." Einstein
```



Program: `UseArgument.java`

Program: `UseArgument.java`

$\rightsquigarrow$ Command-line input: a name

Program: `UseArgument.java`

- ⇝ Command-line input: a name
- ⇝ Standard output: a message containing the name

Program: `UseArgument.java`

- ⇝ Command-line input: a name
- ⇝ Standard output: a message containing the name

```
>_ ~/workspace/dsa/programs

$ _
```

Program: `UseArgument.java`

- ⇝ Command-line input: a name
- ⇝ Standard output: a message containing the name

```
>_ ~/workspace/dsa/programs

$ java UseArgument Alice
```

Program: `UseArgument.java`

- ⇝ Command-line input: a name
- ⇝ Standard output: a message containing the name

```
>_ ~/workspace/dsa/programs

$ java UseArgument Alice
Hi, Alice. How are you?
$ _
```

Program: `UseArgument.java`

- ⇝ Command-line input: a name
- ⇝ Standard output: a message containing the name

```
>_ ~/workspace/dsa/programs

$ java UseArgument Alice
Hi, Alice. How are you?
$ java UseArgument Bob
```

Program: `UseArgument.java`

- ⇝ Command-line input: a name
- ⇝ Standard output: a message containing the name

```
>_ ~/workspace/dsa/programs

$ java UseArgument Alice
Hi, Alice. How are you?
$ java UseArgument Bob
Hi, Bob. How are you?
$ _
```

Program: `UseArgument.java`

- ⇝ Command-line input: a name
- ⇝ Standard output: a message containing the name

```
>_ ~/workspace/dsa/programs

$ java UseArgument Alice
Hi, Alice. How are you?
$ java UseArgument Bob
Hi, Bob. How are you?
$ java UseArgument Carol
```

Program: `UseArgument.java`

- ⇝ Command-line input: a name
- ⇝ Standard output: a message containing the name

```
>_ ~/workspace/dsa/programs

$ java UseArgument Alice
Hi, Alice. How are you?
$ java UseArgument Bob
Hi, Bob. How are you?
$ java UseArgument Carol
Hi, Carol. How are you?
$ _
```

UseArgument.java

```java
// Accepts a name as command-line argument; and writes a message containing that name to standard
// output.

import stdlib.StdOut;

public class UseArgument {
    // Entry point.
    public static void main(String[] args) {
        StdOut.print("Hi, ");
        StdOut.print(args[0]);
        StdOut.println(". How are you?");
    }
}
```

Primitive Types

Primitive Types

A data type (primitive or reference) is a set of values along with a set of operations defined on those values

Primitive Types

A data type (primitive or reference) is a set of values along with a set of operations defined on those values

Primitive types:

Primitive Types

A data type (primitive or reference) is a set of values along with a set of operations defined on those values

Primitive types:

- `boolean` - true and false values with logical operations

Primitive Types

A data type (primitive or reference) is a set of values along with a set of operations defined on those values

Primitive types:

- `boolean` - true and false values with logical operations
- `byte` - 8-bit integers with arithmetic operations

Primitive Types

A data type (primitive or reference) is a set of values along with a set of operations defined on those values

Primitive types:

- `boolean` - true and false values with logical operations
- `byte` - 8-bit integers with arithmetic operations
- `char` - 16-bit characters with arithmetic operations

Primitive Types

A data type (primitive or reference) is a set of values along with a set of operations defined on those values

Primitive types:

- `boolean` - true and false values with logical operations
- `byte` - 8-bit integers with arithmetic operations
- `char` - 16-bit characters with arithmetic operations
- `short` - 16-bit integers with arithmetic operations

Primitive Types

A data type (primitive or reference) is a set of values along with a set of operations defined on those values

Primitive types:

- `boolean` - true and false values with logical operations
- `byte` - 8-bit integers with arithmetic operations
- `char` - 16-bit characters with arithmetic operations
- `short` - 16-bit integers with arithmetic operations
- `int` - 32-bit integers with arithmetic operations

Primitive Types

A data type (primitive or reference) is a set of values along with a set of operations defined on those values

Primitive types:

- `boolean` - true and false values with logical operations
- `byte` - 8-bit integers with arithmetic operations
- `char` - 16-bit characters with arithmetic operations
- `short` - 16-bit integers with arithmetic operations
- `int` - 32-bit integers with arithmetic operations
- `float` - 32-bit single-precision real numbers with arithmetic operations

Primitive Types

A data type (primitive or reference) is a set of values along with a set of operations defined on those values

Primitive types:

- `boolean` - true and false values with logical operations
- `byte` - 8-bit integers with arithmetic operations
- `char` - 16-bit characters with arithmetic operations
- `short` - 16-bit integers with arithmetic operations
- `int` - 32-bit integers with arithmetic operations
- `float` - 32-bit single-precision real numbers with arithmetic operations
- `long` - 64-bit integers with arithmetic operations

Primitive Types

A data type (primitive or reference) is a set of values along with a set of operations defined on those values

Primitive types:

- $\rightsquigarrow$ `boolean` - true and false values with logical operations
- $\rightsquigarrow$ `byte` - 8-bit integers with arithmetic operations
- $\rightsquigarrow$ `char` - 16-bit characters with arithmetic operations
- $\rightsquigarrow$ `short` - 16-bit integers with arithmetic operations
- $\rightsquigarrow$ `int` - 32-bit integers with arithmetic operations
- $\rightsquigarrow$ `float` - 32-bit single-precision real numbers with arithmetic operations
- $\rightsquigarrow$ `long` - 64-bit integers with arithmetic operations
- $\rightsquigarrow$ `double` - 64-bit double-precision real numbers with arithmetic operations

Expressions · Literals

A literal is a representation of a data-type value

A literal is a representation of a data-type value

Examples:

A literal is a representation of a data-type value

Examples:

- ⇝ `true` and `false` are `boolean` literals

A literal is a representation of a data-type value

Examples:

- ⇝ `true` and `false` are `boolean` literals
- ⇝ `'*'` is a `char` literal

A literal is a representation of a data-type value

Examples:

- `true` and `false` are `boolean` literals
- `'*'` is a `char` literal
- `42` is an `int` literal

A literal is a representation of a data-type value

Examples:

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

A literal is a representation of a data-type value

Examples:

- `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

A variable is a name associated with a data-type value

A variable is a name associated with a data-type value

Example: `total` representing the running total of a sequence of numbers

A variable is a name associated with a data-type value

Example: `total` representing the running total of a sequence of numbers

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

A variable is a name associated with a data-type value

Example: `total` representing the running total of a sequence of numbers

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

Example: `SPEED_OF_LIGHT` representing the known speed of light

A variable is a name associated with a data-type value

Example: `total` representing the running total of a sequence of numbers

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

Example: `SPEED_OF_LIGHT` representing the known speed of light

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

Expressions · Variables

A variable is a name associated with a data-type value

Example: `total` representing the running total of a sequence of numbers

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

Example: `SPEED_OF_LIGHT` representing the known speed of light

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

Examples: `total`, `SPEED_OF_LIGHT`, `args`, and `Math.PI`

Expressions · Operators

An operator is a representation of a data-type operation

An operator is a representation of a data-type operation

`+`, `-`, `*`, `/`, and `%` represent arithmetic operations

Expressions · Operators

An operator is a representation of a data-type operation

`+`, `-`, `*`, `/`, and `%` represent arithmetic operations

`!`, `||`, and `&&` represent logical operations

Expressions · Operators

An operator is a representation of a data-type operation

`+`, `-`, `*`, `/`, and `%` represent arithmetic operations

`!`, `||`, and `&&` represent logical operations

The comparison operators `==`, `!=`, `<`, `<=`, `>`, and `>=` operate on numeric values and produce a boolean result

Expressions · Operators

Operator precedence (highest to lowest)

`-`	negation
`*`, `/`, `%`	multiplication, division, remainder
`+`, `-`	addition, subtraction
`<`, `<=`, `>`, `>=`	less than, less than or equal, greater than, greater than or equal
`==`, `!=`	equal, not equal
`=`	assignment
`!`, `||`, `&&`	logical not, logical or, logical and

Expressions · Operators

Operator precedence (highest to lowest)

`-`	negation
`*, /, %`	multiplication, division, remainder
`+, -`	addition, subtraction
`<, <=, >, >=`	less than, less than or equal, greater than, greater than or equal
`==, !=`	equal, not equal
`=`	assignment
`!, ||, &&`	logical not, logical or, logical and

Parentheses can be used to override precedence rules

Expressions · Functions

Many programming tasks involve not only built-in operators, but also functions

Many programming tasks involve not only built-in operators, but also functions

We will use functions:

Many programming tasks involve not only built-in operators, but also functions

We will use functions:

- ⇝ From automatic system libraries (`java.lang` package)

Many programming tasks involve not only built-in operators, but also functions

We will use functions:

- ⇝ From automatic system libraries (`java.lang` package)
- ⇝ From imported system libraries (`java.util` package)

Many programming tasks involve not only built-in operators, but also functions

We will use functions:

- ⇝ From automatic system libraries (`java.lang` package)
- ⇝ From imported system libraries (`java.util` package)
- ⇝ From imported third-party libraries (`stdlib` and `dsa` packages)

Many programming tasks involve not only built-in operators, but also functions

We will use functions:

- ⇝ From automatic system libraries (`java.lang` package)
- ⇝ From imported system libraries (`java.util` package)
- ⇝ From imported third-party libraries (`stdlib` and `dsa` packages)
- ⇝ We define ourselves

Expressions · Functions

Many programming tasks involve not only built-in operators, but also functions

We will use functions:

- ⇝ From automatic system libraries (`java.lang` package)
- ⇝ From imported system libraries (`java.util` package)
- ⇝ From imported third-party libraries (`stdlib` and `dsa` packages)
- ⇝ We define ourselves

A function is called as `[<library>].<name>(<argument1>, <argument2>, ...)`

Many programming tasks involve not only built-in operators, but also functions

We will use functions:

- ⇝ From automatic system libraries (`java.lang` package)
- ⇝ From imported system libraries (`java.util` package)
- ⇝ From imported third-party libraries (`stdlib` and `dsa` packages)
- ⇝ We define ourselves

A function is called as `[<library>].<name>(<argument1>, <argument2>, ...)`

Some functions (called non-void functions) return a value while others (called void functions) do not return any value

Examples:

Examples:

java.lang.Math

`static double sqrt(double x)` returns $\sqrt{\mathtt{x}}$

Examples:

java.lang.Math

`static double sqrt(double x)` returns $\sqrt{x}$

java.lang.Integer

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

Examples:

java.lang.Math

`static double sqrt(double x)` returns $\sqrt{\texttt{x}}$

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`

Examples:

java.lang.Math

`static double sqrt(double x)` returns $\sqrt{x}$

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`

java.util.Arrays

`static void sort(Comparable[] a)` sorts the array `a`

Examples (contd.):

Examples (contd.):

stdlib.StdOut

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

Examples (contd.):

stdlib.StdOut

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

Examples (contd.):

stdlib.StdOut

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

Expressions

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

Examples:

Expressions

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

Examples:

- ⇝ `2, 4`

Expressions

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

Examples:

- $\rightsquigarrow$ `2, 4`
- $\rightsquigarrow$ `a, b, c`

Expressions

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

Examples:

- `2, 4`
- `a, b, c`
- `b * b - 4 * a * c`

Expressions

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

Examples:

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

Expressions

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

Examples:

- `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)`

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"` and `"Cogito, ergo sum"`

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"` and `"Cogito, ergo sum"`

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"` and `"Cogito, ergo sum"`

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"` and `"Cogito, ergo sum"`

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

Example: `"Hello, world\n"`

Two strings can be concatenated using the `+` operator

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"` and `"Cogito, ergo sum"`

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

Example: `"Hello, world\n"`

Two strings can be concatenated using the `+` operator

Example: `"Hello, World" + "!"` evaluates to `"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"` and `"Cogito, ergo sum"`

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

Example: `"Hello, world\n"`

Two strings can be concatenated using the `+` operator

Example: `"Hello, World" + "!"` evaluates to `"Hello, World!"`

The `+` operator can also be used to convert primitives to strings

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"` and `"Cogito, ergo sum"`

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

Example: `"Hello, world\n"`

Two strings can be concatenated using the `+` operator

Example: `"Hello, World" + "!"` evaluates to `"Hello, World!"`

The `+` operator can also be used to convert primitives to strings

Example: `"PI = " + 3.14159` evaluates to `"PI = 3.14159"`

Statements

Statements

A statement is a syntactic unit that expresses some action to be carried out

Statements

A statement is a syntactic unit that expresses some action to be carried out

Import statement

```
import <library>;
```

Statements

A statement is a syntactic unit that expresses some action to be carried out

Import statement

```
import <library>;
```

Examples

```
import java.util.Arrays;
import stdlib.StdOut;
```

Statements

Statements

Function call statement

```
[<library>].<name>(<argument1>, <argument2>, ...);
```

Statements

Function call statement

```
[<library>].<name>(<argument1>, <argument2>, ...);
```

Examples

```
StdOut.print("Cogito, ");
StdOut.print("ergo sum");
StdOut.println();
```

Statements

Statements

Declaration statement

```
<type> <name>;
```

Statements

Declaration statement

```
<type> <name>;
```

The initial value for the variable is `false` for `boolean`, `0` for other primitive types, and `null` for any reference type

Statements

Declaration statement

```
<type> <name>;
```

The initial value for the variable is `false` for `boolean`, `0` for other primitive types, and `null` for any reference type

Assignment statement

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

Statements

Declaration statement

```
<type> <name>;
```

The initial value for the variable is `false` for `boolean`, `0` for other primitive types, and `null` for any reference type

Assignment statement

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

Declaration and assignment statements combined

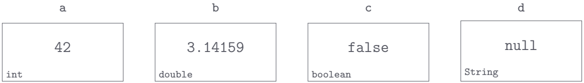
```
<type> <name> = <expression>;
```

Statements

Statements

Examples

```java
int a = 42;
double b = 3.14159D;
boolean c;
String d;
```



Statements

Statements

Equivalent assignment statement forms:

Statements

Equivalent assignment statement forms:

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

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

Statements

Equivalent assignment statement forms:

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

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

```
<name>++;
++<name>;
<name> = <name> + 1;
```

Statements

Equivalent assignment statement forms:

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

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

```
<name>++;
++<name>;
<name> = <name> + 1;
```

```
<name>--;
--<name>;
<name> = <name> - 1;
```

Statements

Equivalent assignment statement forms:

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

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

```
<name>++;
++<name>;
<name> = <name> + 1;
```

```
<name>--;
--<name>;
<name> = <name> - 1;
```

Example

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

Statements

Statements

Program: `Quadratic.java`

Statements

Program: `Quadratic.java`

$\rightsquigarrow$ Command-line input: a (double), b (double), and c (double)

Statements

Program: `Quadratic.java`

- ⇝ Command-line input: a (double), b (double), and c (double)
- ⇝ Standard output: roots of the quadratic equation $ax^2 + bx + c = 0$

Statements

Program: `Quadratic.java`

- Command-line input: a (double), b (double), and c (double)
- Standard output: roots of the quadratic equation $ax^2 + bx + c = 0$

```
>_ ~/workspace/dsa/programs

$ _
```

Statements

Program: `Quadratic.java`

- Command-line input: a (double), b (double), and c (double)
- Standard output: roots of the quadratic equation $ax^2 + bx + c = 0$

```
>_ ~/workspace/dsa/programs

$ java Quadratic 1 -5 6
```

Statements

Program: `Quadratic.java`

- Command-line input: a (double), b (double), and c (double)
- Standard output: roots of the quadratic equation $ax^2 + bx + c = 0$

```
>_ ~/workspace/dsa/programs

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

Statements

Program: `Quadratic.java`

- ⇝ Command-line input: a (double), b (double), and c (double)
- ⇝ Standard output: roots of the quadratic equation $ax^2 + bx + c = 0$

```
>_ ~/workspace/dsa/programs

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

Statements

Program: `Quadratic.java`

- ⇝ Command-line input: a (double), b (double), and c (double)
- ⇝ Standard output: roots of the quadratic equation $ax^2 + bx + c = 0$

```
>_ ~/workspace/dsa/programs

$ 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
$ _
```

Statements

Statements

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);
    }
}
```

Statements

Conditional (if) statement

```
if (<expression>) {
    <statement>
    ...
} else if (<expression>) {
    <statement>
    ...
} else if (<expression>) {
    <statement>
    ...
} else {
    <statement>
    ...
}
<statement>
...
```

Statements

Statements

Program: `Grade.java`

Statements

Program: `Grade.java`

- ⇝ Command-line input: a percentage *score* (double)

Statements

Program: `Grade.java`

- ⇝ Command-line input: a percentage *score* (double)
- ⇝ Standard output: the corresponding letter grade

Statements

Program: `Grade.java`

- ⇝ Command-line input: a percentage *score* (double)
- ⇝ Standard output: the corresponding letter grade

```
>_ ~/workspace/dsa/programs

$ _
```

Statements

Program: `Grade.java`

- ⇝ Command-line input: a percentage *score* (double)
- ⇝ Standard output: the corresponding letter grade

```
>_ ~/workspace/dsa/programs

$ java Grade 97
```

Statements

Program: `Grade.java`

- ⇝ Command-line input: a percentage *score* (double)
- ⇝ Standard output: the corresponding letter grade

```
>_ ~/workspace/dsa/programs

$ java Grade 97
A
$ _
```

Statements

Program: `Grade.java`

- ⇝ Command-line input: a percentage *score* (double)
- ⇝ Standard output: the corresponding letter grade

```
>_ ~/workspace/dsa/programs

$ java Grade 97
A
$ java Grade 56
```

Statements

Program: `Grade.java`

- ⇝ Command-line input: a percentage *score* (double)
- ⇝ Standard output: the corresponding letter grade

```
>_ ~/workspace/dsa/programs

$ java Grade 97
A
$ java Grade 56
F
$ _
```

Statements

Statements

Grade.java

```java
import stdlib.StdOut;

public class Grade {
    public static void main(String[] args) {
        double score = Double.parseDouble(args[0]);
        if (score >= 93) {
            StdOut.println("A");
        } else if (score >= 90) {
            StdOut.println("A-");
        } else if (score >= 87) {
            StdOut.println("B+");
        } else if (score >= 83) {
            StdOut.println("B");
        } else if (score >= 80) {
            StdOut.println("B-");
        } else if (score >= 77) {
            StdOut.println("C+");
        } else if (score >= 73) {
            StdOut.println("C");
        } else if (score >= 70) {
            StdOut.println("C-");
        } else if (score >= 67) {
            StdOut.println("D+");
        } else if (score >= 63) {
            StdOut.println("D");
        } else if (score >= 60) {
            StdOut.println("D-");
        } else {
            StdOut.println("F");
        }
    }
}
```

Statements

Conditional expression

```
... <expression> ? <expression1> : <expression2> ...
```

Statements

Statements

Program: `Flip.java`

Statements

Program: `Flip.java`

$\rightsquigarrow$ Standard output: “Heads” or “Tails”

Statements

Program: `Flip.java`

⇝ Standard output: “Heads” or “Tails”

```
>_ ~/workspace/dsa/programs

$ _
```

Statements

Program: `Flip.java`

⇝ Standard output: “Heads” or “Tails”

```
>_ ~/workspace/dsa/programs

$ java Flip
```

Statements

Program: `Flip.java`

⇝ Standard output: "Heads" or "Tails"

```
>_ ~/workspace/dsa/programs

$ java Flip
Heads
$ _
```

Statements

Program: `Flip.java`

⇝ Standard output: “Heads” or “Tails”

```
>_ ~/workspace/dsa/programs

$ java Flip
Heads
$ java Flip
```

Statements

Program: `Flip.java`

⇝ Standard output: "Heads" or "Tails"

```
>_ ~/workspace/dsa/programs

$ java Flip
Heads
$ java Flip
Heads
$ _
```

Statements

Program: `Flip.java`

⇝ Standard output: "Heads" or "Tails"

```
>_ ~/workspace/dsa/programs

$ java Flip
Heads
$ java Flip
Heads
$ java Flip
```

Statements

Program: `Flip.java`

⇝ Standard output: “Heads” or “Tails”

```
>_ ~/workspace/dsa/programs

$ java Flip
Heads
$ java Flip
Heads
$ java Flip
Tails
$ _
```

Statements

Statements

Flip.java

```java
import stdlib.StdOut;
import stdlib.StdRandom;

public class Flip {
    public static void main(String[] args) {
        String result = StdRandom.bernoulli(0.5) ? "Heads" : "Tails";
        StdOut.println(result);
    }
}
```

Statements

Loop (while) statement

```
while (<expression>) {
    <statement>
    ...
}
<statement>
...
```

Statements

Statements

Program: `NHellos.java`

Statements

Program: `NHellos.java`

$\rightsquigarrow$ Command-line input: n (int)

Statements

Program: `NHellos.java`

- $\rightsquigarrow$ Command-line input: n (int)
- $\rightsquigarrow$ Standard output: n Hellos

Statements

Program: `NHellos.java`

- ⇝ Command-line input: n (int)
- ⇝ Standard output: n Hellos

```
>_ ~/workspace/dsa/programs

$ _
```

Statements

Program: `NHellos.java`

- ⇝ Command-line input: n (int)
- ⇝ Standard output: n Hellos

```
>_ ~/workspace/dsa/programs

$ java NHellos 10
```

Statements

Program: `NHellos.java`

- ⇝ Command-line input: n (int)
- ⇝ Standard output: n Hellos

```
>_ ~/workspace/dsa/programs

$ java NHellos 10
Hello # 1
Hello # 2
Hello # 3
Hello # 4
Hello # 5
Hello # 6
Hello # 7
Hello # 8
Hello # 9
Hello # 10
$ _
```

Statements

Statements

NHellos.java

```java
import stdlib.StdOut;

public class NHellos {
    public static void main(String[] args) {
        int n = Integer.parseInt(args[0]);
        int i = 1;
        while (i <= n) {
            StdOut.println("Hello # " + i);
            i++;
        }
    }
}
```

Statements

Statements

Loop (for) statement

```
for (<initialization>; <expression>; <update>) {
    <statement>
    ...
}
<statement>
...
```

Statements

Statements

Program: `Harmonic.java`

Statements

Program: `Harmonic.java`

$\rightsquigarrow$ Command-line input: n (int)

Statements

Program: `Harmonic.java`

- $\rightsquigarrow$ Command-line input: n (int)
- $\rightsquigarrow$ Standard output: the nth harmonic number $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$

Statements

Program: `Harmonic.java`

- ⇝ Command-line input: n (int)
- ⇝ Standard output: the nth harmonic number $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$

```
>_ ~/workspace/dsa/programs
$ _
```

Statements

Program: `Harmonic.java`

- ⇝ Command-line input: n (int)
- ⇝ Standard output: the nth harmonic number $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$

```
>_ ~/workspace/dsa/programs

$ java Harmonic 10
```

Statements

Program: `Harmonic.java`

- ⇝ Command-line input: n (int)
- ⇝ Standard output: the nth harmonic number $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$

```
>_ ~/workspace/dsa/programs

$ java Harmonic 10
2.9289682539682538
$ _
```

Statements

Program: `Harmonic.java`

- ⇝ Command-line input: n (int)
- ⇝ Standard output: the nth harmonic number $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$

```
>_ ~/workspace/dsa/programs

$ java Harmonic 10
2.9289682539682538
$ java Harmonic 1000
```

Statements

Program: `Harmonic.java`

- ⇝ Command-line input: n (int)
- ⇝ Standard output: the nth harmonic number $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$

```
>_ ~/workspace/dsa/programs

$ java Harmonic 10
2.9289682539682538
$ java Harmonic 1000
7.485470860550343
$ _
```

Statements

Program: `Harmonic.java`

- ⇝ Command-line input: n (int)
- ⇝ Standard output: the nth harmonic number $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$

```
>_ ~/workspace/dsa/programs

$ java Harmonic 10
2.9289682539682538
$ java Harmonic 1000
7.485470860550343
$ java Harmonic 10000
```

Statements

Program: `Harmonic.java`

- ⇝ Command-line input: n (int)
- ⇝ Standard output: the nth harmonic number $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$

```
>_ ~/workspace/dsa/programs

$ java Harmonic 10
2.9289682539682538
$ java Harmonic 1000
7.485470860550343
$ java Harmonic 10000
9.787606036044348
$ _
```

Statements

Statements

Harmonic.java

```java
import stdlib.StdOut;

public class Harmonic {
    public static void main(String[] args) {
        int n = Integer.parseInt(args[0]);
        double total = 0.0;
        for (int i = 1; i <= n; i++) {
            total += 1.0 / i;
        }
        StdOut.println(total);
    }
}
```

Statements

Statements

The if, while, and for statements can be nested within one another

Statements

Statements

Program: `DivisorPattern.java`

Statements

Program: `DivisorPattern.java`

$\rightsquigarrow$ Command-line input: n (int)

Statements

Program: `DivisorPattern.java`

- ⇝ Command-line input: n (int)
- ⇝ Standard output: a table where entry (i, j) is a star ("*") if j divides i or i divides j and a space (" ") otherwise

Statements

Program: `DivisorPattern.java`

- ⇝ Command-line input: n (int)
- ⇝ Standard output: a table where entry (i, j) is a star ("*") if j divides i or i divides j and a space (" ") otherwise

```
>_ ~/workspace/dsa/programs

$ _
```

Statements

Program: `DivisorPattern.java`

- Command-line input: n (int)
- Standard output: a table where entry (i, j) is a star ("*") if j divides i or i divides j and a space (" ") otherwise

```
>_ ~/workspace/dsa/programs

$ java DivisorPattern 10
```

Program: `DivisorPattern.java`

⇝ Command-line input: n (int)

⇝ Standard output: a table where entry (i, j) is a star ("*") if j divides i or i divides j and a space (" ") otherwise

```
>_ ~/workspace/dsa/programs

$ java DivisorPattern 10
* * * * * * * * * * 1
* *   *   *   *   * 2
*   *     *     *   3
* *   *       *     4
*       *         * 5
* * *     *         6
*           *       7
* *   *       *     8
*   *           *   9
* *     *         * 10
$ _
```

Statements

Statements

DivisorPattern.java

```java
import stdlib.StdOut;

public class DivisorPattern {
    public static void main(String[] args) {
        int n = Integer.parseInt(args[0]);
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= n; j++) {
                if (i % j == 0 || j % i == 0) {
                    StdOut.print("* ");
                } else {
                    StdOut.print("  ");
                }
            }
            StdOut.println(i);
        }
    }
}
```

Statements

Break statement

```
break;
```

Break statement

```
break;
```

Example

```
for (int n = 10, i = 0; true; i += 2) {
    if (i == n) {
        break;
    }
    StdOut.println(i + " ");
}
StdOut.println();
```

Break statement

```
break;
```

Example

```
for (int n = 10, i = 0; true; i += 2) {
    if (i == n) {
        break;
    }
    StdOut.println(i + " ");
}
StdOut.println();
```

```
0 2 4 6 8
```

Statements

Continue statement

```
continue;
```

Continue statement

```
continue;
```

Example

```
for (int n = 10, i = 0; i <= n; i++) {
    if (i % 2 == 0) {
        continue;
    }
    StdOut.print(i + " ");
}
StdOut.println();
```

Continue statement

```
continue;
```

Example

```
for (int n = 10, i = 0; i <= n; i++) {
    if (i % 2 == 0) {
        continue;
    }
    StdOut.print(i + " ");
}
StdOut.println();
```

```
1 3 5 7 9
```

Declaration

```
<type>[] <name>;
```

Declaration

```
<type>[] <name>;
```

Creation

```
<name> = new <type>[<capacity>];
```

Declaration

```
<type>[] <name>;
```

Creation

```
<name> = new <type>[<capacity>];
```

Explicit initialization

```
int n = <name>.length;
for (int i = 0; i < n; i++) {
    <name>[i] = <expression>;
}
```

Arrays · One-dimensional (1D) Arrays

Declaration

```
<type>[] <name>;
```

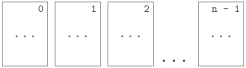
Creation

```
<name> = new <type>[<capacity>];
```

Explicit initialization

```
int n = <name>.length;
for (int i = 0; i < n; i++) {
    <name>[i] = <expression>;
}
```

Memory model for `<name>[]`



Arrays · One-dimensional (1D) Arrays

Program: `Sample.java`

Program: `Sample.java`

$\rightsquigarrow$ Command-line input: m (int) and n (int)

Program: `Sample.java`

- ⇝ Command-line input: m (int) and n (int)
- ⇝ Standard output: a random sample (without replacement) of m integers from the interval $[0, n)$

Program: `Sample.java`

- ⇝ Command-line input: m (int) and n (int)
- ⇝ Standard output: a random sample (without replacement) of m integers from the interval $[0, n)$

```
>_ ~/workspace/dsa/programs

$ _
```

Program: `Sample.java`

- ⇝ Command-line input: m (int) and n (int)
- ⇝ Standard output: a random sample (without replacement) of m integers from the interval $[0, n)$

```
>_ ~/workspace/dsa/programs

$ java Sample 6 16
```

Program: `Sample.java`

- Command-line input: m (int) and n (int)
- Standard output: a random sample (without replacement) of m integers from the interval $[0, n)$

```
>_ ~/workspace/dsa/programs

$ java Sample 6 16
10 7 11 1 8 5
$ _
```

Program: `Sample.java`

- ⇝ Command-line input: m (int) and n (int)
- ⇝ Standard output: a random sample (without replacement) of m integers from the interval $[0, n)$

```
>_ ~/workspace/dsa/programs

$ java Sample 6 16
10 7 11 1 8 5
$ java Sample 10 1000
```

Program: `Sample.java`

- ⇝ Command-line input: m (int) and n (int)
- ⇝ Standard output: a random sample (without replacement) of m integers from the interval $[0, n)$

```
>_ ~/workspace/dsa/programs

$ java Sample 6 16
10 7 11 1 8 5
$ java Sample 10 1000
258 802 440 28 244 256 564 11 515 24
$ _
```

Program: `Sample.java`

- ⇝ Command-line input: m (int) and n (int)
- ⇝ Standard output: a random sample (without replacement) of m integers from the interval $[0, n)$

```
>_ ~/workspace/dsa/programs

$ java Sample 6 16
10 7 11 1 8 5
$ java Sample 10 1000
258 802 440 28 244 256 564 11 515 24
$ java Sample 20 20
```

Program: `Sample.java`

⇝ Command-line input: m (int) and n (int)

⇝ Standard output: a random sample (without replacement) of m integers from the interval $[0, n)$

```
>_ ~/workspace/dsa/programs

$ java Sample 6 16
10 7 11 1 8 5
$ java Sample 10 1000
258 802 440 28 244 256 564 11 515 24
$ java Sample 20 20
15 11 13 1 5 8 16 7 0 4 10 18 19 14 3 12 2 6 9 17
$ _
```

Sample.java

```java
import stdlib.StdOut;
import stdlib.StdRandom;

public class Sample {
    public static void main(String[] args) {
        int m = Integer.parseInt(args[0]);
        int n = Integer.parseInt(args[1]);
        int[] perm = new int[n];
        for (int i = 0; i < n; i++) {
            perm[i] = i;
        }
        for (int i = 0; i < m; i++) {
            int r = StdRandom.uniform(i, n);
            int temp = perm[r];
            perm[r] = perm[i];
            perm[i] = temp;
        }
        for (int i = 0; i < m; i++) {
            StdOut.print(perm[i] + " ");
        }
        StdOut.println();
    }
}
```

Arrays · Two-dimensional (2D) Arrays

Declaration

```
<type>[][] <name>;
```

Declaration

```
<type>[][] <name>;
```

Creation

```
<name> = new <type>[<capacity>][<capacity>];
```

Declaration

```
<type>[][] <name>;
```

Creation

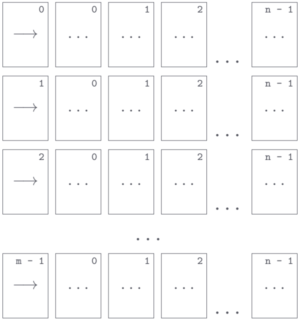
```
<name> = new <type>[<capacity>][<capacity>];
```

Explicit initialization

```
int m = <name>.length;
for (int i = 0; i < m; i++) {
    int n = <name>[i].length;
    for (int j = 0; j < n; j++) {
        <name>[i][j] = <expression>;
    }
}
```

Arrays · Two-dimensional (2D) Arrays

Memory model for <name>[][]



Program: `SelfAvoid.java`

Program: `SelfAvoid.java`

$\rightsquigarrow$ Command-line input: n (int) and $trials$ (int)

Program: `SelfAvoid.java`

- ⇝ Command-line input: n (int) and $trials$ (int)
- ⇝ Standard output: percentage of dead ends encountered in $trials$ self-avoiding random walks on an $n \times n$ lattice

Program: `SelfAvoid.java`

- ⇝ Command-line input: n (int) and *trials* (int)
- ⇝ Standard output: percentage of dead ends encountered in *trials* self-avoiding random walks on an $n \times n$ lattice

```
>_ ~/workspace/dsa/programs
$ _
```

Program: `SelfAvoid.java`

- ⇝ Command-line input: n (int) and *trials* (int)
- ⇝ Standard output: percentage of dead ends encountered in *trials* self-avoiding random walks on an $n \times n$ lattice

```
>_ ~/workspace/dsa/programs

$ java SelfAvoid 20 1000
```

Program: `SelfAvoid.java`

- ⇝ Command-line input: n (int) and *trials* (int)
- ⇝ Standard output: percentage of dead ends encountered in *trials* self-avoiding random walks on an $n \times n$ lattice

```
>_ ~/workspace/dsa/programs

$ java SelfAvoid 20 1000
33% dead ends
$ _
```

Program: `SelfAvoid.java`

- ⇝ Command-line input: n (int) and *trials* (int)
- ⇝ Standard output: percentage of dead ends encountered in *trials* self-avoiding random walks on an $n \times n$ lattice

```
>_ ~/workspace/dsa/programs

$ java SelfAvoid 20 1000
33% dead ends
$ java SelfAvoid 40 1000
```

Program: `SelfAvoid.java`

- ⇝ Command-line input: n (int) and *trials* (int)
- ⇝ Standard output: percentage of dead ends encountered in *trials* self-avoiding random walks on an $n \times n$ lattice

```
>_ ~/workspace/dsa/programs

$ java SelfAvoid 20 1000
33% dead ends
$ java SelfAvoid 40 1000
78% dead ends
$ _
```

Program: `SelfAvoid.java`

- ⇝ Command-line input: n (int) and *trials* (int)
- ⇝ Standard output: percentage of dead ends encountered in *trials* self-avoiding random walks on an $n \times n$ lattice

```
>_ ~/workspace/dsa/programs

$ java SelfAvoid 20 1000
33% dead ends
$ java SelfAvoid 40 1000
78% dead ends
$ java SelfAvoid 80 1000
```

Program: `SelfAvoid.java`

- ⇝ Command-line input: n (int) and *trials* (int)
- ⇝ Standard output: percentage of dead ends encountered in *trials* self-avoiding random walks on an $n \times n$ lattice

```
>_ ~/workspace/dsa/programs

$ java SelfAvoid 20 1000
33% dead ends
$ java SelfAvoid 40 1000
78% dead ends
$ java SelfAvoid 80 1000
98% dead ends
$ _
```

Arrays · Two-dimensional (2D) Arrays

SelfAvoid.java

```java
import stdlib.StdOut;
import stdlib.StdRandom;

public class SelfAvoid {
    public static void main(String[] args) {
        int n = Integer.parseInt(args[0]);
        int trials = Integer.parseInt(args[1]);
        int deadEnds = 0;
        for (int t = 0; t < trials; t++) {
            boolean[][] a = new boolean[n][n];
            int x = n / 2;
            int y = n / 2;
            while (x > 0 && x < n - 1 && y > 0 && y < n - 1) {
                a[x][y] = true;
                if (a[x - 1][y] && a[x + 1][y] && a[x][y - 1] && a[x][y + 1]) {
                    deadEnds++;
                    break;
                }
                int r = StdRandom.uniform(1, 5);
                if (r == 1 && !a[x + 1][y]) {
                    x++;
                } else if (r == 2 && !a[x - 1][y]) {
                    x--;
                } else if (r == 3 && !a[x][y + 1]) {
                    y++;
                } else if (r == 4 && !a[x][y - 1]) {
                    y--;
                }
            }
        }
        StdOut.println(100 * deadEnds / trials + "% dead ends");
    }
}
```

Defining Functions

Defining Functions

Function definition

```
public|private static void|<type> <name>(<parameter1>, <parameter2>, ...) {
    <statement>
    ...
}
```

Defining Functions

Function definition

```
public|private static void|<type> <name>(<parameter1>, <parameter2>, ...) {
    <statement>
    ...
}
```

Return statement

```
return [<expression>];
```

Defining Functions

Function definition

```
public|private static void|<type> <name>(<parameter1>, <parameter2>, ...) {
    <statement>
    ...
}
```

Return statement

```
return [<expression>];
```

Example

```
private static boolean isPrime(int x) {
    if (x < 2) {
        return false;
    }
    for (int i = 2; i <= x / i; i++) {
        if (x % i == 0) {
            return false;
        }
    }
    return true;
}
```

Defining Functions

Defining Functions

Properties of functions:

Defining Functions

Properties of functions:

⇝ Arguments are passed by value

Defining Functions

Properties of functions:

- ⇝ Arguments are passed by value
- ⇝ Function names can be overloaded

Defining Functions

Properties of functions:

- ⇝ Arguments are passed by value
- ⇝ Function names can be overloaded
- ⇝ A function has a single return value but may have multiple return statements

Defining Functions

Properties of functions:

- ⇝ Arguments are passed by value
- ⇝ Function names can be overloaded
- ⇝ A function has a single return value but may have multiple return statements
- ⇝ A function can have side effects

Defining Functions

Defining Functions

Program: `HarmonicRedux.java`

Defining Functions

Program: `HarmonicRedux.java`

$\rightsquigarrow$ Command-line input: n (int)

Defining Functions

Program: `HarmonicRedux.java`

- ⇝ Command-line input: n (int)
- ⇝ Standard output: the nth harmonic number $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$

Defining Functions

Program: `HarmonicRedux.java`

- ⇝ Command-line input: n (int)
- ⇝ Standard output: the nth harmonic number $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$

```
>_ ~/workspace/dsa/programs

$ _
```

Defining Functions

Program: `HarmonicRedux.java`

- Command-line input: n (int)
- Standard output: the nth harmonic number $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$

```
>_ ~/workspace/dsa/programs

$ java HarmonicRedux 10
```

Defining Functions

Program: `HarmonicRedux.java`

- Command-line input: n (int)
- Standard output: the nth harmonic number $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$

```
>_ ~/workspace/dsa/programs

$ java HarmonicRedux 10
2.9289682539682538
$ _
```

Defining Functions

Program: `HarmonicRedux.java`

- Command-line input: n (int)
- Standard output: the nth harmonic number $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$

```
>_ ~/workspace/dsa/programs

$ java HarmonicRedux 10
2.9289682539682538
$ java HarmonicRedux 1000
```

Defining Functions

Program: `HarmonicRedux.java`

- Command-line input: n (int)
- Standard output: the nth harmonic number $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$

```
>_ ~/workspace/dsa/programs

$ java HarmonicRedux 10
2.9289682539682538
$ java HarmonicRedux 1000
7.485470860550343
$ _
```

Defining Functions

Program: `HarmonicRedux.java`

- Command-line input: n (int)
- Standard output: the nth harmonic number $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$

```
>_ ~/workspace/dsa/programs

$ java HarmonicRedux 10
2.9289682539682538
$ java HarmonicRedux 1000
7.485470860550343
$ java HarmonicRedux 10000
```

Defining Functions

Program: `HarmonicRedux.java`

- Command-line input: n (int)
- Standard output: the nth harmonic number $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$

```
>_ ~/workspace/dsa/programs

$ java HarmonicRedux 10
2.9289682539682538
$ java HarmonicRedux 1000
7.485470860550343
$ java HarmonicRedux 10000
9.787606036044348
$ _
```

Defining Functions

Defining Functions

HarmonicRedux.java

```java
import stdlib.StdOut;

public class HarmonicRedux {
    public static void main(String[] args) {
        int n = Integer.parseInt(args[0]);
        StdOut.println(harmonic(n));
    }

    private static double harmonic(int n) {
        double total = 0.0;
        for (int i = 1; i <= n; i++) {
            total += 1.0 / i;
        }
        return total;
    }
}
```

A recursive function is one that calls itself, has a base case, addresses subproblems that are smaller in some sense, and does not address subproblems that overlap

A recursive function is one that calls itself, has a base case, addresses subproblems that are smaller in some sense, and does not address subproblems that overlap

Example (computing $n!$)

$$n! = \begin{cases} n(n-1)! & \text{if } n > 0\text{, and} \\ 1 & \text{if } n = 0 \end{cases}$$

```
private static int factorial(int n) {
    if (n == 0) {
        return 1;
    }
    return n * factorial(n - 1);
}
```

A recursive function is one that calls itself, has a base case, addresses subproblems that are smaller in some sense, and does not address subproblems that overlap

Example (computing $n!$)

$$n! = \begin{cases} n(n-1)! & \text{if } n > 0 \text{, and} \\ 1 & \text{if } n = 0 \end{cases}$$

```
private static int factorial(int n) {
    if (n == 0) {
        return 1;
    }
    return n * factorial(n - 1);
}
```

Call trace for `factorial(5)`

```
factorial(5)
```

A recursive function is one that calls itself, has a base case, addresses subproblems that are smaller in some sense, and does not address subproblems that overlap

Example (computing $n!$)

$$n! = \begin{cases} n(n-1)! & \text{if } n > 0, \text{ and} \\ 1 & \text{if } n = 0 \end{cases}$$

```
private static int factorial(int n) {
    if (n == 0) {
        return 1;
    }
    return n * factorial(n - 1);
}
```

Call trace for `factorial(5)`

```
factorial(5)
  5 * factorial(4)
```

A recursive function is one that calls itself, has a base case, addresses subproblems that are smaller in some sense, and does not address subproblems that overlap

Example (computing $n!$)

$$n! = \begin{cases} n(n-1)! & \text{if } n > 0 \text{, and} \\ 1 & \text{if } n = 0 \end{cases}$$

```
private static int factorial(int n) {
    if (n == 0) {
        return 1;
    }
    return n * factorial(n - 1);
}
```

Call trace for `factorial(5)`

```
factorial(5)
  5 * factorial(4)
        4 * factorial(3)
```

A recursive function is one that calls itself, has a base case, addresses subproblems that are smaller in some sense, and does not address subproblems that overlap

Example (computing $n!$)

$$n! = \begin{cases} n(n-1)! & \text{if } n > 0\text{, and} \\ 1 & \text{if } n = 0 \end{cases}$$

```java
private static int factorial(int n) {
    if (n == 0) {
        return 1;
    }
    return n * factorial(n - 1);
}
```

Call trace for `factorial(5)`

```
factorial(5)
  5 * factorial(4)
        4 * factorial(3)
              3 * factorial(2)
```

A recursive function is one that calls itself, has a base case, addresses subproblems that are smaller in some sense, and does not address subproblems that overlap

Example (computing $n!$)

$$n! = \begin{cases} n(n-1)! & \text{if } n > 0\text{, and} \\ 1 & \text{if } n = 0 \end{cases}$$

```
private static int factorial(int n) {
    if (n == 0) {
        return 1;
    }
    return n * factorial(n - 1);
}
```

Call trace for `factorial(5)`

```
factorial(5)
  5 * factorial(4)
        4 * factorial(3)
              3 * factorial(2)
                    2 * factorial(1)
```

A recursive function is one that calls itself, has a base case, addresses subproblems that are smaller in some sense, and does not address subproblems that overlap

Example (computing $n!$)

$$n! = \begin{cases} n(n-1)! & \text{if } n > 0, \text{ and} \\ 1 & \text{if } n = 0 \end{cases}$$

```java
private static int factorial(int n) {
    if (n == 0) {
        return 1;
    }
    return n * factorial(n - 1);
}
```

Call trace for `factorial(5)`

```
factorial(5)
  5 * factorial(4)
        4 * factorial(3)
              3 * factorial(2)
                    2 * factorial(1)
                          1 * factorial(0)
```

A recursive function is one that calls itself, has a base case, addresses subproblems that are smaller in some sense, and does not address subproblems that overlap

Example (computing $n!$)

$$n! = \begin{cases} n(n-1)! & \text{if } n > 0\text{, and} \\ 1 & \text{if } n = 0 \end{cases}$$

```java
private static int factorial(int n) {
    if (n == 0) {
        return 1;
    }
    return n * factorial(n - 1);
}
```

Call trace for `factorial(5)`

```
factorial(5)
  5 * factorial(4)
        4 * factorial(3)
              3 * factorial(2)
                    2 * factorial(1)
                          1 * 1
```

A recursive function is one that calls itself, has a base case, addresses subproblems that are smaller in some sense, and does not address subproblems that overlap

Example (computing $n!$)

$$n! = \begin{cases} n(n-1)! & \text{if } n > 0 \text{, and} \\ 1 & \text{if } n = 0 \end{cases}$$

```
private static int factorial(int n) {
    if (n == 0) {
        return 1;
    }
    return n * factorial(n - 1);
}
```

Call trace for `factorial(5)`

```
factorial(5)
  5 * factorial(4)
        4 * factorial(3)
              3 * factorial(2)
                    2 * 1
```

A recursive function is one that calls itself, has a base case, addresses subproblems that are smaller in some sense, and does not address subproblems that overlap

Example (computing $n!$)

$$n! = \begin{cases} n(n-1)! & \text{if } n > 0 \text{, and} \\ 1 & \text{if } n = 0 \end{cases}$$

```
private static int factorial(int n) {
    if (n == 0) {
        return 1;
    }
    return n * factorial(n - 1);
}
```

Call trace for `factorial(5)`

```
factorial(5)
  5 * factorial(4)
        4 * factorial(3)
              3 * 2
```

A recursive function is one that calls itself, has a base case, addresses subproblems that are smaller in some sense, and does not address subproblems that overlap

Example (computing $n!$)

$$n! = \begin{cases} n(n-1)! & \text{if } n > 0, \text{ and} \\ 1 & \text{if } n = 0 \end{cases}$$

```java
private static int factorial(int n) {
    if (n == 0) {
        return 1;
    }
    return n * factorial(n - 1);
}
```

Call trace for `factorial(5)`

```
factorial(5)
  5 * factorial(4)
        4 * 6
```

A recursive function is one that calls itself, has a base case, addresses subproblems that are smaller in some sense, and does not address subproblems that overlap

Example (computing $n!$)

$$n! = \begin{cases} n(n-1)! & \text{if } n > 0 \text{, and} \\ 1 & \text{if } n = 0 \end{cases}$$

```java
private static int factorial(int n) {
    if (n == 0) {
        return 1;
    }
    return n * factorial(n - 1);
}
```

Call trace for `factorial(5)`

```
factorial(5)
  5 * 24
```

A recursive function is one that calls itself, has a base case, addresses subproblems that are smaller in some sense, and does not address subproblems that overlap

Example (computing $n!$)

$$n! = \begin{cases} n(n-1)! & \text{if } n > 0\text{, and} \\ 1 & \text{if } n = 0 \end{cases}$$

```
private static int factorial(int n) {
    if (n == 0) {
        return 1;
    }
    return n * factorial(n - 1);
}
```

Call trace for `factorial(5)`

```
120
```

Program: `Factorial.java`

Program: `Factorial.java`

$\rightsquigarrow$ Command-line input: n (int)

Program: `Factorial.java`

- ⇝ Command-line input: n (int)
- ⇝ Standard output: $n!$

Program: `Factorial.java`

- ⇝ Command-line input: n (int)
- ⇝ Standard output: $n!$

```
>_ ~/workspace/dsa/programs

$ _
```

Program: `Factorial.java`

- ⇝ Command-line input: n (int)
- ⇝ Standard output: $n!$

```
>_ ~/workspace/dsa/programs

$ java Factorial 0
```

Program: `Factorial.java`

- ⇝ Command-line input: n (int)
- ⇝ Standard output: $n!$

```
>_ ~/workspace/dsa/programs

$ java Factorial 0
1
$ _
```

Program: `Factorial.java`

- ⇝ Command-line input: n (int)
- ⇝ Standard output: $n!$

```
>_ ~/workspace/dsa/programs

$ java Factorial 0
1
$ java Factorial 5
```

Program: `Factorial.java`

- ⇝ Command-line input: n (int)
- ⇝ Standard output: $n!$

```
>_ ~/workspace/dsa/programs

$ java Factorial 0
1
$ java Factorial 5
120
$ _
```

Defining Functions · Recursive Functions

Factorial.java

```java
import stdlib.StdOut;

public class Factorial {
    public static void main(String[] args) {
        int n = Integer.parseInt(args[0]);
        StdOut.println(factorial(n));
    }

    private static int factorial(int n) {
        if (n == 0) {
            return 1;
        }
        return n * factorial(n - 1);
    }
}
```

Scope of Variables

Scope of Variables

The scope of a variable is the part of the program that can refer to that variable by name

Scope of Variables

The scope of a variable is the part of the program that can refer to that variable by name

Example

Harmonic.java

```java
1  import stdlib.StdOut;
2  
3  public class Harmonic {
4      public static void main(String[] args) {
5          int n = Integer.parseInt(args[0]);
6          double total = 0.0;
7          for (int i = 1; i <= n; i++) {
8              total += 1.0 / i;
9          }
10         StdOut.println(total);
11     }
12 }
```

Variable	Scope
`args`	lines 4 — 11
`n`	lines 5 — 11
`total`	lines 6 — 11
`args`	lines 7 — 9

Input and Output Revisited

Input and Output Revisited

stdlib.StdOut

`static void print(Object x)`	prints an object to standard output
`static void println(Object x)`	prints an object and a newline to standard output
`static void printf(String fmt, Object... args)`	prints `args` to standard output using the format string `fmt`

Input and Output Revisited

Input and Output Revisited

Program: `RandomSeq.java`

Input and Output Revisited

Program: `RandomSeq.java`

$\rightsquigarrow$ Command-line input: n (int), lo (double), hi (double)

Input and Output Revisited

Program: `RandomSeq.java`

- ⇝ Command-line input: n (int), lo (double), hi (double)
- ⇝ Standard output: n random doubles in the range $[lo, hi)$, each up to 2 decimal places

Input and Output Revisited

Program: `RandomSeq.java`

- ⇝ Command-line input: n (int), lo (double), hi (double)
- ⇝ Standard output: n random doubles in the range $[lo, hi)$, each up to 2 decimal places

```
>_ ~/workspace/dsa/programs

$ _
```

Input and Output Revisited

Program: `RandomSeq.java`

- ⇝ Command-line input: n (int), lo (double), hi (double)
- ⇝ Standard output: n random doubles in the range $[lo, hi)$, each up to 2 decimal places

```
>_ ~/workspace/dsa/programs

$ java RandomSeq 10 100 200
```

Input and Output Revisited

Program: `RandomSeq.java`

- ⇝ Command-line input: n (int), lo (double), hi (double)
- ⇝ Standard output: n random doubles in the range $[lo, hi)$, each up to 2 decimal places

```
>_ ~/workspace/dsa/programs

$ java RandomSeq 10 100 200
186.69
102.34
176.05
182.78
161.95
169.34
155.65
154.96
194.41
103.91
$ _
```

Input and Output Revisited

Input and Output Revisited

RandomSeq.java

```java
import stdlib.StdOut;
import stdlib.StdRandom;

public class RandomSeq {
    public static void main(String[] args) {
        int n = Integer.parseInt(args[0]);
        double lo = Double.parseDouble(args[1]);
        double hi = Double.parseDouble(args[2]);
        for (int i = 0; i < n; i++) {
            double r = StdRandom.uniform(lo, hi);
            StdOut.printf("%.2f\n", r);
        }
    }
}
```

Input and Output Revisited

Input and Output Revisited

Standard input is input entered interactively on the terminal

Input and Output Revisited

Standard input is input entered interactively on the terminal

The end of standard input stream is signalled by the end-of-file (EOF) character (`<ctrl-d>`)

Input and Output Revisited

Standard input is input entered interactively on the terminal

The end of standard input stream is signalled by the end-of-file (EOF) character (`<ctrl-d>`)

stdlib.StdIn

`static boolean isEmpty()`	returns `true` if standard input is empty, and `false` otherwise
`static double readDouble()`	reads and returns the next double from standard input

Input and Output Revisited

Input and Output Revisited

Program: `Average.java`

Input and Output Revisited

Program: `Average.java`

- ⇝ Standard input: a sequence of doubles

Input and Output Revisited

Program: `Average.java`

- ⇝ Standard input: a sequence of doubles
- ⇝ Standard output: their average value

Input and Output Revisited

Program: `Average.java`

- ⇝ Standard input: a sequence of doubles
- ⇝ Standard output: their average value

```
>_ ~/workspace/dsa/programs

$ _
```

Input and Output Revisited

Program: `Average.java`

- ⇝ Standard input: a sequence of doubles
- ⇝ Standard output: their average value

```
>_ ~/workspace/dsa/programs

$ java Average
```

Input and Output Revisited

Program: `Average.java`

- ⇝ Standard input: a sequence of doubles
- ⇝ Standard output: their average value

```
>_ ~/workspace/dsa/programs

$ java Average
_
```

Input and Output Revisited

Program: `Average.java`

- ⇝ Standard input: a sequence of doubles
- ⇝ Standard output: their average value

```
>_ ~/workspace/dsa/programs

$ java Average
1.0 5.0 6.0
```

Input and Output Revisited

Program: `Average.java`

- ⇝ Standard input: a sequence of doubles
- ⇝ Standard output: their average value

```
>_ ~/workspace/dsa/programs

$ java Average
1.0 5.0 6.0
-
```

Input and Output Revisited

Program: `Average.java`

- ⇝ Standard input: a sequence of doubles
- ⇝ Standard output: their average value

```
>_ ~/workspace/dsa/programs

$ java Average
1.0 5.0 6.0
3.0 7.0 32.0
```

Input and Output Revisited

Program: `Average.java`

- ⇝ Standard input: a sequence of doubles
- ⇝ Standard output: their average value

```
>_ ~/workspace/dsa/programs

$ java Average
1.0 5.0 6.0
3.0 7.0 32.0
_
```

Input and Output Revisited

Program: `Average.java`

- ⇝ Standard input: a sequence of doubles
- ⇝ Standard output: their average value

```
>_ ~/workspace/dsa/programs

$ java Average
1.0 5.0 6.0
3.0 7.0 32.0
<ctrl-d>
```

Input and Output Revisited

Program: `Average.java`

- ⇝ Standard input: a sequence of doubles
- ⇝ Standard output: their average value

```
>_ ~/workspace/dsa/programs

$ java Average
1.0 5.0 6.0
3.0 7.0 32.0
<ctrl-d>
Average is 10.5
$ _
```

Input and Output Revisited

Input and Output Revisited

Average.java

```java
import stdlib.StdIn;
import stdlib.StdOut;

public class Average {
    public static void main(String[] args) {
        double total = 0.0;
        int count = 0;
        while (!StdIn.isEmpty()) {
            double x = StdIn.readDouble();
            total += x;
            count++;
        }
        double average = total / count;
        StdOut.println("Average is " + average);
    }
}
```

Input and Output Revisited

Input and Output Revisited

Output redirection operator (`>`)

Input and Output Revisited

Output redirection operator (>)

```
>_ ~/workspace/dsa/programs
$ _
```

Output redirection operator (>)

```
>_ ~/workspace/dsa/programs

$ java RandomSeq 1000 100.0 200.0 > data.txt
```

Input and Output Revisited

Output redirection operator (>)

```
>_ ~/workspace/dsa/programs

$ java RandomSeq 1000 100.0 200.0 > data.txt
$ _
```

Input and Output Revisited

Output redirection operator (>)

```
>_ ~/workspace/dsa/programs

$ java RandomSeq 1000 100.0 200.0 > data.txt
$ _
```

Input redirection operator (<)

```
>_ ~/workspace/dsa/programs

$ _
```

Input and Output Revisited

Output redirection operator (>)

```
>_ ~/workspace/dsa/programs

$ java RandomSeq 1000 100.0 200.0 > data.txt
$ _
```

Input redirection operator (<)

```
>_ ~/workspace/dsa/programs

$ java Average < data.txt
```

Input and Output Revisited

Output redirection operator (>)

```
>_ ~/workspace/dsa/programs

$ java RandomSeq 1000 100.0 200.0 > data.txt
$ _
```

Input redirection operator (<)

```
>_ ~/workspace/dsa/programs

$ java Average < data.txt
Average is 149.1812199999999
$ _
```

Input and Output Revisited

Output redirection operator (>)

```
>_ ~/workspace/dsa/programs

$ java RandomSeq 1000 100.0 200.0 > data.txt
$ _
```

Input redirection operator (<)

```
>_ ~/workspace/dsa/programs

$ java Average < data.txt
Average is 149.1812199999999
$ _
```

Piping operator (|)

```
>_ ~/workspace/dsa/programs

$ _
```

Input and Output Revisited

Output redirection operator (>)

```
>_ ~/workspace/dsa/programs

$ java RandomSeq 1000 100.0 200.0 > data.txt
$ _
```

Input redirection operator (<)

```
>_ ~/workspace/dsa/programs

$ java Average < data.txt
Average is 149.1812199999999
$ _
```

Piping operator (|)

```
>_ ~/workspace/dsa/programs

$ java RandomSeq 1000 100.0 200.0 | java Average
```

Input and Output Revisited

Output redirection operator (>)

```
>_ ~/workspace/dsa/programs

$ java RandomSeq 1000 100.0 200.0 > data.txt
$ _
```

Input redirection operator (<)

```
>_ ~/workspace/dsa/programs

$ java Average < data.txt
Average is 149.1812199999999
$ _
```

Piping operator (|)

```
>_ ~/workspace/dsa/programs

$ java RandomSeq 1000 100.0 200.0 | java Average
Average is 150.0588699999999
$ _
```