Programming Model
Programming in Java

The Java workflow

IntelliJ (editor)

Program.java

javac (compiler)

Program.class

java (JVM)

Output

/edit

Program.java

```java
// Import statements.
...

// Class definition.
public class Program {
  // Field declarations.
  ...
  // Constructor definitions.
  ...
  // Method definitions.
  ...
  // Function definitions.
  ...
  // Inner class definitions.
  ...
}
```
Programming in Java

The Java workflow

IntelliJ (editor) -> Program.java -> javac (compiler) -> Program.class -> java (JVM) -> output
Programming in Java

The Java workflow

```
[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

Program:

HelloWorld.java

• Standard output: the message "Hello, World"
Program: HelloWorld.java
Program: HelloWorld.java

- Standard output: the message “Hello, World”
Program: HelloWorld.java

- Standard output: the message “Hello, World”
Program: **HelloWorld.java**

- Standard output: the message “Hello, World”

> `~/workspace/dsaj/programs`

$ javac -d out src/HelloWorld.java
Program: HelloWorld.java

- Standard output: the message “Hello, World”

```bash
> ~/workspace/dsaj/programs

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

- Standard output: the message “Hello, World”

```
> ~/workspace/dsaj/programs

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

- Standard output: the message “Hello, World”
// 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

HelloWorld.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");
    }
}
The application programming interface (API) for a library provides a summary of the functions in the library.

Example:

`/list` (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.
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

<table>
<thead>
<tr>
<th>stdlib.StdOut</th>
</tr>
</thead>
<tbody>
<tr>
<td>static void println(Object x)</td>
</tr>
<tr>
<td>static void print(Object x)</td>
</tr>
</tbody>
</table>
Errors in a Program

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

Example

```
// 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");
    }
}
```
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

```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

```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/dsaj/programs
$ _
```
Errors in a Program

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

Example

```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")
    }
}
```

```
$ javac -d out src/HelloWorld.java
```
Syntax errors are 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:

```java
// Writes the message "Hello, World" to standard output.
public class HelloWorld {
    // Entry point.
    public static void main (String[] args) {
        StdOut.println("Hello, World");
    }
}
```
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

```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

```
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

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

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

```
$ 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

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

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

```
$ 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

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

Example:

```
HelloWorld.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.
Errors in a Program

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

Example

```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

```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");
    }
}
```

```bash
$ ~/workspace/dsaj/programs
```
Errors in a Program

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

Example

```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");
    }
}
```

$ 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

```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");
    }
}
```

```bash
~/workspace/dsaj/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

```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/dmaj/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

```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");
    }
}
```

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

- Command-line input
- Standard input
- File input

- Standard output
- File output
Input and Output

Input types:
• Command-line input
• Standard input
• File input

Output types:
• Standard output
• File output

input → Program → output
Input and Output

Input types:
- Command-line input
- Standard input
- File input

Output types:
- Standard output
- File output
Input and Output

Input types:
- Command-line input
- Standard input
- File input

Output types:
- Standard output
- File output
Input and Output

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

```
~/workspace/dsaj/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/dsaj/programs
$ java Program Galileo " Isaac Newton " Einstein
```

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

```bash
> ~/workspace/dsaj/programs
$ java Program input1 input2 input3 ...
```
Command-line inputs are strings listed right next to the program name during execution

```
> ~/workspace/dmaj/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.
Command-line inputs are strings listed right next to the program name during execution

```
$ 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

```
$ java Program Galileo "Isaac Newton" Einstein

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Galileo&quot;</td>
<td>&quot;Isaac Newton&quot;</td>
<td>&quot;Einstein&quot;</td>
</tr>
</tbody>
</table>
```
Input and Output

Program: UseArgument.java

• Command-line input: a name
• Standard output: a message containing the name
Input and Output

Program: UseArgument.java
Program: UseArgument.java

- Command-line input: a name
Program: UseArgument.java

- Command-line input: a name
- Standard output: a message containing the name
Input and Output

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/dsaj/programs
$ java UseArgument Alice
```
**Input and Output**

**Program:** UseArgument.java

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

```
$ ~/workspace/dsaj/programs
$ java UseArgument Alice
Hi, Alice. How are you?
$ 
```
Program: UseArgument.java

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

```bash
$ ~workspace/dsaj/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

```bash
$ 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

```bash
> ~/workspace/dsaj/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/dsaj/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?
$ _
```
// 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?);
    }
}
// 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

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
- double - 64-bit double-precision real numbers with arithmetic operations
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:

- 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.
- double: 64-bit double-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
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
- **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
- **double** - 64-bit double-precision real numbers with arithmetic operations
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
- `double` - 64-bit double-precision real numbers with arithmetic operations
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
- `double` - 64-bit double-precision real numbers with arithmetic operations
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
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
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
- double - 64-bit double-precision real numbers with arithmetic operations
Expressions

A literal is a representation of a 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
Expressions

A literal is a representation of a data-type value
Expressions

A literal is a representation of a data-type value

Example:
Expressions

A literal is a representation of a data-type value

Example:

- true and false are boolean literals
A literal is a representation of a data-type value

Example:

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

A literal is a representation of a data-type value

Example:

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

A literal is a representation of a data-type value

Example:

- `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

Example:
- true and false are boolean literals
- 'ast' is a char literal
- 42 is an int literal
- 1729L is a long literal
- 3.14159D is a double literal
Expressions

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>
```

Example:

- `total`, `SPEED_OF_LIGHT`, `args`, and `Math.PI`
Expressions

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

Example:
- `total` representing the running total of a sequence of numbers
- `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`
A variable is a name associated with a data-type value

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

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

Example: \texttt{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
Expressions

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
Expressions

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

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]`

Example: `total`, `SPEED_OF_LIGHT`, `args`, and `Math.PI`
Expressions

An operator is a representation of a data-type operation. The arithmetic operations are:

- +
- -
- *
- /
- %

The logical operations are:

- !
- ||
- &&

The comparison operators are:

- ==
- !=
- <
- <=
- >
- >=

These operators operate on numeric values and produce a boolean result.
Expressions

An operator is a representation of a data-type operation
Expressions

An operator is a representation of a data-type operation

+, -, *, /, and % represent arithmetic operations
An operator is a representation of a data-type operation

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

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

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

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
Operator precedence (highest to lowest)

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>negation</td>
</tr>
<tr>
<td>*, /, %</td>
<td>multiplication, division, remainder</td>
</tr>
<tr>
<td>+, -</td>
<td>addition, subtraction</td>
</tr>
<tr>
<td>&lt;, &lt;=, &gt;, &gt;=</td>
<td>less than, less than or equal, greater than, greater than or equal</td>
</tr>
<tr>
<td>==, !=</td>
<td>equal, not equal</td>
</tr>
<tr>
<td>=</td>
<td>assignment</td>
</tr>
<tr>
<td>!,</td>
<td></td>
</tr>
</tbody>
</table>
Operator precedence (highest to lowest)

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>negation</td>
</tr>
<tr>
<td>*, /, %</td>
<td>multiplication, division, remainder</td>
</tr>
<tr>
<td>+, -</td>
<td>addition, subtraction</td>
</tr>
<tr>
<td>&lt;, &lt;=, &gt;, &gt;=</td>
<td>less than, less than or equal, greater than, greater than or equal</td>
</tr>
<tr>
<td>==, !=</td>
<td>equal, not equal</td>
</tr>
<tr>
<td>=</td>
<td>assignment</td>
</tr>
<tr>
<td>!,</td>
<td></td>
</tr>
</tbody>
</table>

Parentheses can be used to override precedence rules
Expressions

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)
- That we define ourselves

A function is called as 

\[
\text{[<library>.]<name>(<argument1>, <argument2>, ...)
\]

Some functions (called non-void functions) return a value while others (called void functions) do not return any value
Many programming tasks involve not only built-in operators, but also functions
Expressions

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

We will use functions:
Expressions

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)
- That we define ourselves

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

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)

A function is called as

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

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)
- That we define ourselves
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)
- That we define ourselves

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

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)
- That we define ourselves

A function is called as [library.]name(arg1, arg2, ...)

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

Example

/list

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 according to the natural order of its objects

static void sort(Object[] a, Comparator c) sorts the array a according to the order induced by the comparator c
Expressions

Example

java.lang.Math

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

java.lang.Integer

static int parseInt(String s) returns \(\text{int value of } s\)

java.lang.Double

static double parseDouble(String s) returns \(\text{double value of } s\)

java.util.Arrays

static void sort(Comparable[] a) sorts the array \(a\) according to the natural order of its objects

static void sort(Object[] a, Comparator c) sorts the array \(a\) according to the order induced by the comparator \(c\)
Expressions

Example

```java
// java.lang.Math

double sqrt(double x) returns \sqrt{x}
```
### Expressions

#### Example

<table>
<thead>
<tr>
<th>Class</th>
<th>Method Description</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.lang.Math</td>
<td>static double sqrt(double x)</td>
<td>returns $\sqrt{x}$</td>
</tr>
<tr>
<td>java.lang.Integer</td>
<td>static int parseInt(String s)</td>
<td>returns int value of s</td>
</tr>
</tbody>
</table>
Expressions

Example

- `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
<table>
<thead>
<tr>
<th>Method</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>java.lang.Math</code></td>
<td><code>static double sqrt(double x)</code> returns $\sqrt{x}$</td>
</tr>
<tr>
<td><code>java.lang.Integer</code></td>
<td><code>static int parseInt(String s)</code> returns <code>int</code> value of <code>s</code></td>
</tr>
<tr>
<td><code>java.lang.Double</code></td>
<td><code>static double parseDouble(String s)</code> returns <code>double</code> value of <code>s</code></td>
</tr>
<tr>
<td><code>java.util.Arrays</code></td>
<td><code>static void sort(Comparable[] a)</code> sorts the array <code>a</code> according to the natural order of its objects</td>
</tr>
<tr>
<td><code>java.util.Arrays</code></td>
<td><code>static void sort(Object[] a, Comparator c)</code> sorts the array <code>a</code> according to the order induced by the comparator <code>c</code></td>
</tr>
</tbody>
</table>
Expressions

Example (contd.)

```java
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
```

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

```java
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

Example (contd.)
Expressions

Example (contd.)

<table>
<thead>
<tr>
<th>stdlib.StdOut</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>static void println(Object x)</td>
<td>prints an object and a newline to standard output</td>
</tr>
<tr>
<td>static void print(Object x)</td>
<td>prints an object to standard output</td>
</tr>
</tbody>
</table>
### stdlib.StdOut

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>static void println(Object x)</td>
<td>prints an object and a newline to standard output</td>
</tr>
<tr>
<td>static void print(Object x)</td>
<td>prints an object to standard output</td>
</tr>
</tbody>
</table>

### stdlib.StdRandom

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>static double uniform(double a, double b)</td>
<td>returns a double chosen uniformly at random from the interval ([a, b))</td>
</tr>
<tr>
<td>static boolean bernoulli(double p)</td>
<td>returns true with probability (p) and false with probability (1 - p)</td>
</tr>
</tbody>
</table>
### stdlib.StdOut

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>static void println(Object x)</code></td>
<td>prints an object and a newline to standard output</td>
</tr>
<tr>
<td><code>static void print(Object x)</code></td>
<td>prints an object to standard output</td>
</tr>
</tbody>
</table>

### stdlib.StdRandom

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>static double uniform(double a, double b)</code></td>
<td>returns a double chosen uniformly at random from the interval $[a, b)$</td>
</tr>
<tr>
<td><code>static boolean bernoulli(double p)</code></td>
<td>returns <code>true</code> with probability $p$ and <code>false</code> with probability $1 - p$</td>
</tr>
</tbody>
</table>

### stdlib.StdStats

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>static double mean(double[] a)</code></td>
<td>returns the average value in the array <code>a</code></td>
</tr>
<tr>
<td><code>static double stddev(double[] a)</code></td>
<td>returns the sample standard deviation in the array <code>a</code></td>
</tr>
</tbody>
</table>
An expression is a combination of literals, variables, operators, and non-void function calls that evaluates to a value.

Example:

- \(2, 4\)
- \(a, b, c\)
- \(b \times b - 4 \times a \times c\)
- \(\text{Math.sqrt}(b \times b - 4 \times a \times c)\)
- \((-b + \text{Math.sqrt}(b \times b - 4 \times a \times c)) / (2 \times a)\)
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

Example:
Expressions

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

Example:

- \( 2, 4 \)
An expression is a combination of literals, variables, operators, and non-void function calls that evaluates to a value.

Example:

- 2, 4
- a, b, c
Expressions

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

Example:
- 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

Example:

- 2, 4
- a, b, c
- b * b - 4 * a * c
- Math.sqrt(b * b - 4 * a * c)
An expression is a combination of literals, variables, operators, and non-void function calls that evaluates to 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)
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".
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.

Example:

"Hello, World"

"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"
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"
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"
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"
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.
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

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

Import statement
import <library>;

Example
import java.util.Arrays;
import stdlib.StdOut;
A statement is a syntactic unit that expresses some action to be carried out.
A statement is a syntactic unit that expresses some action to be carried out

Import statement

```java
import <library>;
```
A statement is a syntactic unit that expresses some action to be carried out

Import statement

```java
import <library>;
```

Example

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

Function call statement

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

Example

```java
StdOut . print ("Cogito , ");
StdOut . print ("ergo sum ");
StdOut . println ();
```
Function call statement

```
[{library}.]name((<argument1>, <argument2>, ...));
```
Function call statement

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

Example

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

**Declaration statement**

```plaintext
<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**

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

**Declaration and assignment statements combined**

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

Declation statement

\[\text{<type> <name>;}\]

The initial value for the variable is \textit{false} for \textit{boolean}, \textit{0} for other primitive types, and \textit{null} for any reference type.

Assignment statement

\[\text{<name> = <expression>;}\]
Statements

Declaration statement

\[ \text{<type> <name>;} \]

The initial value for the variable is \text{false} for \text{boolean}, \text{0} for other primitive types, and \text{null} for any reference type.

Assignment statement

\[ \text{<name> = <expression>;} \]

Declaration and assignment statements combined

\[ \text{<type> <name> = <expression>;} \]
Statements

Example

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

```
a b c d
42 3.14159 false null
```
Statements

Example

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

Equivalent assignment statement forms

\begin{align*}
\text{<name>} & \quad \text{<operator> = <expression>;} \\
\text{<name>} & \quad = \text{<name>} \quad \text{<operator>} \quad \text{<expression>;} \\
\text{where} & \quad \text{<operator>} \quad \text{is} \quad +, - *, /, \text{or} \%.
\end{align*}

\begin{align*}
\text{<name>} & \quad \text{<name>}++; \\
& \quad ++ \text{<name>}; \\
\text{<name>} & \quad = \text{<name>} + 1; \\
\text{<name>} & \quad = \text{<name>} - 1; \\
\text{<name>} & \quad --\text{<name>}; \\
\text{Example} & \quad x += 1; \\
& \quad x = x + 1; \\
& \quad ++x; \\
& \quad x ++;
\end{align*}
Statements

Equivalent assignment statement forms

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

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

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

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

Equivalent assignment statement forms

\[
\text{<name> <operator> = <expression>};
\]
\[
\text{<name> = <name> <operator> <expression>};
\]

where \text{<operator>} is +, -, *, /, or %

\[
\text{<name>++;}
\]
\[
\text{++<name>};
\]
\[
\text{<name> = <name> + 1;}
\]

\[
\text{<name>--;}
\]
\[
\text{--<name>};
\]
\[
\text{<name> = <name> - 1;}
\]

Example

\[
x += 1;
\]
\[
x = x + 1;
\]
\[
++x;
\]
\[
+++x;
\]
\[
x++;\]
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
Program: Quadratic.java

- Command-line input: \( a \) (double), \( b \) (double), and \( c \) (double)
Program: Quadratic.java

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

$ java Quadratic 1 -5 6 $
Program: Quadratic.java

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

```bash
$ ~/workspace/dsaj/programs
$ java Quadratic 1 -5 6
Root # 1 = 3.0
Root # 2 = 2.0
$ _
```
Program: Quadratic.java

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

```bash
> ~/workspace/dsaj/programs
$ java Quadratic 1 -5 6
Root # 1 = 3.0
Root # 2 = 2.0
$ java Quadratic 1 -1 -1
```
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/dsaj/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
$ 
```
import stdlib.StdOut;

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);
    }
}
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

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

```plaintext
if (<expression>) {
    <statement>
    ...
} else if (<expression>) {
    <statement>
    ...
} else if (<expression>) {
    <statement>
    ...
} else {
    <statement>
    ...
} ...
```
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
Program: Grade.java

• Command-line input: a percentage score (double)
Program: Grade.java

- Command-line input: a percentage score (double)
- Standard output: the corresponding letter grade
Program: Grade.java

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

```
> ~/workspace/dsaj/programs

$ 
```
Program: Grade.java

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

```bash
~/workspace/dsaj/programs

$ java Grade 97
```
Program: Grade.java

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

```
$ ~workspace/dsaj/programs
$ java Grade 97
A
$ 
```
Program: Grade.java

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

```
> ~/workspace/dsaj/programs
$ java Grade 97
A
$ java Grade 56
```
Program: Grade.java

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

```bash
> ~/workspace/dsaj/programs
$ java Grade 97
A
$ java Grade 56
F
$ _
```
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");
        }
    }
}
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> ...
Conditional expression

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

Program: Flip.java

- Standard output: "Heads" or "Tails"
Program: Flip.java

Standard output: "Heads" or "Tails"
Program: Flip.java

- Standard output: “Heads” or “Tails”
Program: Flip.java

- Standard output: “Heads” or “Tails”
Program: Flip.java

- Standard output: “Heads” or “Tails”

```
$ java Flip
```
Program: Flip.java

- Standard output: “Heads” or “Tails”

```bash
> ~/workspace/dsaj/programs

$ java Flip
Heads
$ _
```
Program: Flip.java

- Standard output: “Heads” or “Tails”

```bash
> ~/workspace/dsaj/programs

$ java Flip
Heads
$ java Flip
```
Program: Flip.java

- Standard output: “Heads” or “Tails”

```
> ~/workspace/dsaj/programs
$ java Flip
Heads
$ java Flip
Heads
$ _
```
Program: Flip.java

• Standard output: “Heads” or “Tails”

> `~/workspace/dsaj/programs

$ java Flip
Heads
$ java Flip
Heads
$ java Flip
Program: Flip.java

- Standard output: “Heads” or “Tails”

```bash
$ cd ~/workspace/dsaj/programs
$ java Flip
Heads
$ java Flip
Heads
$ java Flip
Tails
$ 
```
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);
    }
}
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 >
    ...
}
...
Loop (while) statement

```java
while (<expression>) {
    <statement>
    ...
}
...
```
Program: NHellos.java

Command-line input: n

Standard output: n Hellos
Program: NHellos.java
Program:  NHellos.java

- Command-line input:  \( n \) (int)
Statements

Program: NHellos.java

- Command-line input: n (int)
- Standard output: n Hellos
Program: NHellos.java

- Command-line input: \texttt{n (int)}
- Standard output: \texttt{n Hellos}
Program: NHellos.java

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

```bash
$ java NHellos 10
```
Statements

Program: NHellos.java

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

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

$ -
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 ++;
        }
    }
}
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

Loop (for) statement

```plaintext
for ( [ < initialization > ]; [ < expression > ]; [ < update > ] ) {
    <statement >
    ...
}
...
```
Loop (for) statement

```plaintext
for ([<initialization>]; [<expression>]; [<update>]) {
  <statement>
  ...
}
...
```
Statements

Program: Harmonic.java

• Command-line input: \( n \) (int)

• Standard output: the \( n \)th harmonic number

\[ H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} \]
Program: Harmonic.java

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

Standard output: the \( n \)th harmonic number

\[ H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} \]
Program: Harmonic.java

- Command-line input: $n$ (int)
Program: Harmonic.java

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

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

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

> ~/workspace/dsaj/programs

$ java Harmonic 10
Program: Harmonic.java

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

```
$ ~/workspace/dsaj/programs
$ java Harmonic 10
2.9289682539682538
$ _
```
Program: Harmonic.java

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

```
> ~/workspace/dsaj/programs
$ java Harmonic 10
2.9289682539682538
$ java Harmonic 1000
```
Statements

Program: Harmonic.java

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

```bash
~/workspace/dsaj/programs
$ java Harmonic 10
2.9289682539682538
$ java Harmonic 1000
7.485470860550343
$ 
```
**Program:** Harmonic.java

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

```
> ~/workspace/dsaj/programs
$ java Harmonic 10
2.9289682539682538
$ java Harmonic 1000
7.485470860550343
$ java Harmonic 10000
```
Program: Harmonic.java

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

```
> ~/workspace/dsaj/programs
$ java Harmonic 10
2.9289682539682538
$ java Harmonic 1000
7.485470860550343
$ java Harmonic 10000
9.787606036044348
$ _
```
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);
    }
}
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
The if, while, and for statements can be nested within one another.
The if, while, and for statements can be nested within one another
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
Program: DivisorPattern.java

- Command-line input: \texttt{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.
Program: DivisorPattern.java

- Command-line input: \( n \) (int)
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
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
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
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/dsaj/programs
$ java DivisorPattern 10
* * * * * * * * * * 1
* * * * * * * 2
* * * * 3
* * * * 4
* * * 5
* * * 6
* * 7
* * * 8
* * * 9
* * * * 10
$ 
```
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);
        }
    }
}
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

```
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

Break statement

```java
break;
```
**Statements**

**Break statement**

```java
break;
```

**Example**

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

Break statement

```java
break;
```

Example

```java
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;
```

Example

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

```
1 3 5 7 9
```
Continue statement

```java
continue;
```
Statements

Continue statement

```java
continue;
```

Example

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

```java
continue;
```

**Example**

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

```
1 3 5 7 9
```
Arrays

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

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

Explicit initialization:
```
int n = <name>.length; // capacity of <name>
for (int i = 0; i < n; i++) {
    <name>[i] = <expression>;
}
```

Memory model for `<name>[]`:
```
0 ... 1 ... 2 ... ··· n - 1 ...
```
Arrays

Declaration

\[ \text{<type>[]} \text{ <name>}; \]

Creation

\[ \text{<name> = new <type>[]} \text{ <capacity>}; \]

Explicit initialization

\[ \text{n = <name>. length; // capacity of <name>}; \]

\[ \text{for (int i = 0; i < n; i++)} \]

\[ \text{<name>[i] = <expression>}; \]

Memory model for \[ \text{<name>[]} \]

\[ 0 \]

\[ \ldots \]

\[ 1 \]

\[ \ldots \]

\[ 2 \]

\[ \ldots \]

\[ \text{···} \]

\[ n - 1 \]

\[ \ldots \]
Arrays

Declaration

<type>[], <name>;

Creation

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

Memory model for <name>[

0 ...
1 ...
2 ...
···
n - 1 ...

]
Arrays

Declaration

<type>[] <name>;

Creation

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

Explicit initialization

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

Declaration

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

Creation

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

Explicit initialization

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

Memory model for `<name>[

```
0 1 2 n-1
...
```
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
Arrays

Program: `Sample.java`

- 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 $[0, n)$
Arrays

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 $[0, n)$

$> ~/workspace/dsaj/programs$

$ $ 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/dsaj/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/dsaj/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)

```bash
> ~/workspace/dsaj/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/dsaj/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 \([0, n]\)

```bash
> ~/workspace/dsaj/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
$ _
```
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();
    }
}
import stdlib.StdOut;
import stdlib.StdRandom;

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

Declaration

[type] [name];

Creation

[name] = new [type][capacity][capacity];

Explicit initialization

int m = [name].length; // # of rows in [name]
for (int i = 0; i < m; i++) {
    int n = [name][i].length; // # of columns in the ith row of [name]
    for (int j = 0; j < n; j++) {
        [name][i][j] = <expression>;
    }
}
Arrays

Declaration

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

Declaration

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

Creation

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

Declaration

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

Creation

```java
{name} = new <type>[<capacity>][<capacity>];
```

Explicit initialization

```java
int m = {name}.length; // # of rows in {name}
for (int i = 0; i < m; i++) {
    int n = {name}[i].length; // # of columns in the ith row of {name}
    for (int j = 0; j < n; j++) {
        {name}[i][j] = <expression>;
    }
}
```
Arrays

Index to row-major order: $k = ni + j$

Row-major order to index: $i = \lfloor k/n \rfloor$ and $j = k \bmod n$
### Memory model for `<name>[][]`

<table>
<thead>
<tr>
<th>n-1</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>n-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>→</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>n-1</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>n-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>→</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>n-1</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>n-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>→</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>...</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>n-1</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>n-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>→</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Arrays

Memory model for `<name>[][]`

Index to row-major order: $k = ni + j$
Arrays

Memory model for $\langle \text{name} \rangle[]$

Index to row-major order: $k = ni + j$

Row-major order to index: $i = \lfloor \frac{k}{n} \rfloor$ and $j = k \mod n$
Arrays

Program: SelfAvoid.java

- Command-line input: \( n \) (int) and \( \text{trials} \) (int)
- Standard output: percentage of dead ends encountered in \( n \times n \) lattice self-avoiding random walks on an
Program: SelfAvoid.java
Program: SelfAvoid.java

- 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
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

```bash
$ ~/workspace/dsaj/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

```shell
> ~/workspace/dsaj/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/dsaj/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

<table>
<thead>
<tr>
<th>Command-line</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>java SelfAvoid 20 1000</code></td>
<td>33% dead ends</td>
</tr>
<tr>
<td><code>java SelfAvoid 40 1000</code></td>
<td>78% dead ends</td>
</tr>
<tr>
<td><code>java SelfAvoid 80 1000</code></td>
<td></td>
</tr>
</tbody>
</table>
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

```bash
> ~/workspace/dsaj/programs
$ java SelfAvoid 20 1000
33% dead ends
$ java SelfAvoid 40 1000
78% dead ends
$ java SelfAvoid 80 1000
98% dead ends
$ _
```
Arrays
Arrays

Escape

Dead End
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");
    }
}
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

Function definition

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

Return statement

return [< expression >];
```

Example

```java
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

Function definition

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

Example

```java
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

Function definition

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

Return statement

```java
return [<expression>];
```
Defining Functions

**Function definition**

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

**Return statement**

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

**Example**

```java
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

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

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

Properties of functions:

- Arguments are passed by value
Properties of functions:

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

Program: HarmonicRedux.java

• Command-line input: \( n \) (int)

• Standard output: the \( n \)th harmonic number
  \[ H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} \]
Defining Functions

Program: HarmonicRedux.java
Defining Functions

Program: HarmonicRedux.java

- Command-line input: $n$ (int)
Program: HarmonicRedux.java

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

```
> ~/workspace/dsaj/programs

$ 
```
Defining Functions

Program: HarmonicRedux.java

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

```
> ~/workspace/dsaj/programs
$ java HarmonicRedux 10
```
Defining Functions

Program: HarmonicRedux.java

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

$>$ ~/workspace/dsaj/programs

$>$ java HarmonicRedux 10
2.9289682539682538
$>$ _
Defining Functions

Program: HarmonicRedux.java

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

```
> ~/workspace/dsaj/programs

$ java HarmonicRedux 10
2.9289682539682538
$ java HarmonicRedux 1000
```
Defining Functions

Program: HarmonicRedux.java

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

```
$ ~/workspace/dsaj/programs
$ java HarmonicRedux 10
2.9289682539682538
$ java HarmonicRedux 1000
7.485470860550343
$ 
```
### Defining Functions

**Program:** HarmonicRedux.java

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

```bash
$ ~/workspace/dsaj/programs
$ java HarmonicRedux 10
2.9289682539682538
$ java HarmonicRedux 1000
7.485470860550343
$ java HarmonicRedux 10000
```
Program: HarmonicRedux.java

• Command-line input: \( n \) (int)

• Standard output: the \( n \)th harmonic number \( H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} \)

$ ~/workspace/dsaj/programs
$ java HarmonicRedux 10
2.9289682539682538
$ java HarmonicRedux 1000
7.485470860550343
$ java HarmonicRedux 10000
9.787606036044348
$ _
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;
    }
}
Defining Functions

```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;
    }
}
```
Defining Functions

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 $n!$)

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

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

Call trace for `factorial(5)`
Defining Functions

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 $n!$)

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

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

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)
```
Defining Functions

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 $n!$)

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

```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)
```
Defining Functions

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 $n!$)

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

```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)
```

Defining Functions

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 $n!$)

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

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

Call trace for `factorial(5)`

```
defactorial(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 $n!$)

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

```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`
Defining Functions

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 $n!$)

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

```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)
```
Defining Functions

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 $n!$)

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

```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
```
Defining Functions

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 $n!$)

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

```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 * 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 * 2
```
Defining Functions

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 $n!$)

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

```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
```
Defining Functions

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`
Defining Functions

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)`

120
Defining Functions

Program: Factorial.java

- Command-line input: \( n \) (int)
- Standard output: \( n! \)
Defining Functions

Program: Factorial.java
Program: Factorial.java

- Command-line input: $n$ (int)
Defining Functions

Program: Factorial.java

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

Program: Factorial.java

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

```
$ ~/workspace/dsa/j/programs
```

```bash
$ _
```
Program: Factorial.java

- Command-line input: \( n \) (int)
- Standard output: \( n! \)
Defining Functions

Program:  Factorial.java

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

```
> ~/workspace/dsaj/programs

$ java Factorial 0
 1
$ _
```
Defining Functions

Program: Factorial.java

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

```
> ~/workspace/dsaj/programs
$ java Factorial 0
1
$ java Factorial 5

```
**Program:** Factorial.java

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

```bash
$ ~/workspace/dsaj/programs
$ java Factorial 0
1
$ java Factorial 5
120
$ _
```
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);
    }
}
Defining Functions

```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

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

Example:

```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);
    }
}
```

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

```
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);
    }
}
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>args</td>
<td>lines 4 — 11</td>
</tr>
<tr>
<td>n</td>
<td>lines 5 — 11</td>
</tr>
<tr>
<td>total</td>
<td>lines 6 — 11</td>
</tr>
<tr>
<td>i</td>
<td>lines 7 — 9</td>
</tr>
</tbody>
</table>
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

static void printf(String fmt, Object... args)
prints args to standard output using the format string fmt
### stdlib.StdOut

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>static void println(Object x)</td>
<td>prints an object and a newline to standard output</td>
</tr>
<tr>
<td>static void print(Object x)</td>
<td>prints an object to standard output</td>
</tr>
<tr>
<td>static void printf(String fmt, Object... args)</td>
<td>prints args to standard output using the format string fmt</td>
</tr>
</tbody>
</table>
Input and Output Revisited

Program: RandomSeq.java

- Command-line input: \( n \) (int), \( \text{lo} \) (double), \( \text{hi} \) (double)
- Standard output: \( n \) random doubles in the range \([\text{lo}, \text{hi}]\), each up to 2 decimal places
Program: RandomSeq.java
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
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
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
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/dsaj/programs
$ java RandomSeq 10 100 200
```
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/dsaj/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
$  
```
import stdlib.StdOut;
import stdlib.StdRandom;

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
", r);
        }
    }
}
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

Standard input is input entered interactively on the terminal. The end of standard input stream is signalled by the end-of-file (EOF) character (\texttt{<ctrl-d>}).

```
/list
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
```
Standard input is input entered interactively on the terminal
Standard input is input entered interactively on the terminal.

The end of standard input stream is signalled by the end-of-file (EOF) character (\texttt{\textasciicircum{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>)

<table>
<thead>
<tr>
<th>stdlib.StdIn</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>static boolean isEmpty()</td>
<td>returns true if standard input is empty, and false otherwise</td>
</tr>
<tr>
<td>static double readDouble()</td>
<td>reads and returns the next double from standard input</td>
</tr>
</tbody>
</table>
Input and Output Revisited

Program: Average.java

• Standard input: a sequence of doubles
• Standard output: their average value
Program: Average.java

Standard input: a sequence of doubles
Standard output: their average value
Program: Average.java

• Standard input: a sequence of doubles
Program: Average.java

- Standard input: a sequence of doubles
- Standard output: their average value
Program: Average.java

- Standard input: a sequence of doubles
- Standard output: their average value
Program: Average.java

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

```bash
> ~/workspace/dsaj/programs
$ java Average
```
Program: Average.java

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

```
~/workspace/dsaj/programs

$ java Average
_
Program: Average.java

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

```
$ java Average
1.0 5.0 6.0
```
Program: Average.java

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

```bash
> ~/workspace/dsaj/programs
$ java Average
1.0 5.0 6.0
-`
```
Program: Average.java

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

$ java Average
1.0 5.0 6.0
3.0 7.0 32.0
Program: Average.java

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

$ java Average
1.0 5.0 6.0
3.0 7.0 32.0
3.0

Program: Average.java

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

```bash
$ java Average
1.0 5.0 6.0
3.0 7.0 32.0
<ctrl-d>
```
Program: Average.java

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

```bash
> ~/workspace/dsaj/programs
$ java Average
1.0 5.0 6.0
3.0 7.0 32.0
<ctrl-d>
Average is 10.5
$ 
```
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);
}
}
```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 (>)
Output redirection operator (>)

```bash
> ~/workspace/dsmaj/programs
```

```bash
$ _
```
Output redirection operator (>)

$ java RandomSeq 1000 100.0 200.0 > data.txt
Output redirection operator (>)

```bash
> ~/workspace/dsaj/programs
$ java RandomSeq 1000 100.0 200.0 > data.txt
$ _
```
### Output redirection operator (>)

<table>
<thead>
<tr>
<th>$&gt; ~/workspace/dsaj/programs</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ java RandomSeq 1000 100.0 200.0 &gt; data.txt</td>
</tr>
<tr>
<td>$ _</td>
</tr>
</tbody>
</table>

### Input redirection operator (<)

<table>
<thead>
<tr>
<th>$&lt; ~/workspace/dsaj/programs</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ _</td>
</tr>
</tbody>
</table>

---

**Input and Output Revisited**
Input and Output Revisited

Output redirection operator (>)

```
$ ~workspace/dsaj/programs
$ java RandomSeq 1000 100.0 200.0 > data.txt
```

Input redirection operator (<)

```
$ ~workspace/dsaj/programs
$ java Average < data.txt
```
Output redirection operator (>)

$ _~/workspace/dsaj/programs
$ java RandomSeq 1000 100.0 200.0 > data.txt
$ _

Input redirection operator (<)

$ _~/workspace/dsaj/programs
$ java Average < data.txt
Average is 149.1812199999999
$ _
## Input and Output Revisited

### Output redirection operator (>)

<table>
<thead>
<tr>
<th>Terminal</th>
<th>Command</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>~/workspace/dsa\j/programs</td>
<td><code>$ java RandomSeq 1000 100.0 200.0 &gt; data.txt</code></td>
<td>$ _</td>
</tr>
</tbody>
</table>

### Input redirection operator (<)

<table>
<thead>
<tr>
<th>Terminal</th>
<th>Command</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>~/workspace/dsa\j/programs</td>
<td><code>$ java Average &lt; data.txt</code></td>
<td>Average is 149.1812199999999</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$ _</td>
</tr>
</tbody>
</table>

### Piping operator (|)

<table>
<thead>
<tr>
<th>Terminal</th>
<th>Command</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>~/workspace/dsa\j/programs</td>
<td><code>$ _</code></td>
<td></td>
</tr>
</tbody>
</table>
Output redirection operator (>)

Output redirection operator (>)

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

Input redirection operator (<)

Input redirection operator (<)

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

Piping operator (|)

Piping operator (|)

```bash
$ java RandomSeq 1000 100.0 200.0 | java Average
```
Input and Output Revisited

Output redirection operator (>)

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

Input redirection operator (<)

```bash
$ java Average < data.txt
Average is 149.18121999999999
$ _
```

Piping operator (|)

```bash
$ java RandomSeq 1000 100.0 200.0 | java Average
Average is 150.05886999999999
$ _
```