Definition

A *construct* of a language is a syntactically meaningful subpart of a language formed (constructed) in accordance with the syntactic rules of a language and has a significant and coherent purpose.
Java Constructs

• Expression stands for a value (during execution)
• Declaration introduces a name (in the program)
• Statement performs an action (during execution)
• Type describes a set of values (in the program)
Arithmetic

\[ 2+7, \ 5\times4 \]

\[ 2.9+7.4, \ 5.2\times4.3 \]

Avoid the use of `+` for conversion to strings, can be written using formatter strings which are more flexible and clearer.
Division and Mod

The binary operators / and % are very important. Use them for non-negative integers.

In other cases think carefully and use the Math class:

```java
int ceilDiv (int x, int y)
int floorDiv (int x, int y)
int ceilMod (int x, int y)
int floorMod (int x, int y)
```

To fail instead of wrapping around:

```java
int ceilDivExact (int x, int y)
int floorDivExact (int x, int y)
int decrementExact (int x)
int incrementExact (int x)
int negateExact (int x)
int absExact (int x)
int addExact (int x, int y)
int subtractExact (int x, int y)
```
Other Boolean Operators

relational: <, >, <=, >=

equality: ==, !=

conditional: &&, ||
if Expression

String url = args.length>0?args[0]:"http://www.cs.fit.edu/~ryan/";
String s = (Math.cos(0.0) > 0.0) ? "positive" : "non-positive";
flip(n-1, base, isZ?c:c-1, prefix+'0');
score[player?0:1] += turn;
out.setRGB(row, col, (row+col)%2==0?WHITE:BLACK);
return (c==0)?t:Integer.MAX_VALUE;
Bit Operations

Boolean operations extend naturally over sequences of bits of the same length.

\[
a = 0b0011_1100; \quad -- \quad 60
\]

\[
b = 0b0000_1101; \quad -- \quad 13
\]

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp; (bitwise and)</td>
<td>Binary AND Operator copies a bit to the result if it exists in both operands.</td>
<td>((A &amp; B)) will give 12 which is 0000 1100</td>
</tr>
<tr>
<td></td>
<td>(bitwise or)</td>
<td>Binary OR Operator copies a bit if it exists in either operand.</td>
</tr>
<tr>
<td>^ (bitwise XOR)</td>
<td>Binary XOR Operator copies the bit if it is set in one operand but not both.</td>
<td>((A ^ B)) will give 49 which is 0011 0001</td>
</tr>
<tr>
<td>~ (bitwise compliment)</td>
<td>Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.</td>
<td>((~A)) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.</td>
</tr>
<tr>
<td>(&lt;) (left shift)</td>
<td>Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.</td>
<td>(A \ll 2) will give 240 which is 1111 0000</td>
</tr>
<tr>
<td>(&gt;&gt;) (right shift)</td>
<td>Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.</td>
<td>(A \gg 2) will give 15 which is 1111</td>
</tr>
<tr>
<td>(&gt;&gt;&gt;) (zero fill right shift)</td>
<td>Shift right zero fill operator. The left operands value is moved right by the number of bits specified by the right operand and shifted values are filled up with zeros.</td>
<td>(A \gg\gg 2) will give 15 which is 0000 1111</td>
</tr>
</tbody>
</table>
int x;
int y;
x = 365 / 12;
print (x);
y = 366 / 12;
print (y);

final int x = 365 / 12;
print (x);
final int y = 366 / 12;
print (y);

Java allows the programmer to place declarations anywhere that statements go. But declarations are not actions performed by the running program. They are acts of communication to the translator (before execution begins.)
Simple Statements

• Empty
• Expression statements
  • Assignment, e.g., \( x = 2.0 \times \text{Math.pi} \);
  • Subprocedure invocation, e.g., \( \text{StdDraw.line}(0.2, 0.2, 0.8, 0.2) \);
• Break (see loops)
• Continue (see loops)
• Yield (see switch **expressions**)
• Return (in methods)
• Throw (exception handling)
• Assert
Statements

- Simple statements (including assert)
- Block – sequence of statements in curly braces
- Labeled – usually used with `break`
- Conditional: `Grade.java`
- Switch: `Month.java`, `Days.java`
  ```java
  switch expression
  ```
- While: `WhileDemo.java`
- For: `WordCount.java`; for each (coming up)
- Try (later)
- Synchronize (not covered)
Loops

When to use `for`, when to use `while`?
Loops

When to use *for*, when to use *while*?

Use *for* when you know the number of times the body of the loop will be executed.
For each

double[] a = {1.2, 3.0, 0.8};
double sum = 0;
for (double d: a) {
    sum += d;
}
System.out.format("sum = %.2f\n", sum);

- only access, elements cannot be assigned
- only single structure at a time
- only single element, can’t compare successive elements
- only forward, can’t iterate backwards, by twos, etc.
One-and-one-half Loops

Test at the beginning (while), test at the end (do-while), test in the middle.

Main.java

DRY = Don’t Repeat Yourself
Try Resource

Simple, isolated programs like those that do calculations are less common these days. More and more common are programs that interact with the environment: network, other programs, etc. Often these programs claim resources (provided by the operating system) and it is tidy to return them when unneeded.

```java
try {
    final Scanner stdin = new Scanner (System.in, "US-ASCII")
    // Use 'stdin'
}
```

try {
    final DirectoryStream<Path> listing = Files.newDirectoryStream(dir);
    // Use 'listing'
} catch (final IOException exc) {
    exc.printStackTrace(System.err);
}
See sermon on correctness and the `assert` statement
correct.pdf
Anatomy of a static method, figure page 188.
How to you call a static method: Class.name or just “name” if you are calling from the same class.
Expressions
For each Statement

double[] a = {1.2, 3.0, 0.8};
double sum = 0;
for (double d : a) {
    sum += d;
}
System.out.format("sum = %.2f\n", sum);

• only access, elements cannot be assigned
• only single structure at a time
• only single element, can’t compare successive elements
• only forward, can’t iterate backwards, by twos, etc.
### Static Methods

Anatomy of a static method, S&W, 2.1, figure page 188.

Class.name or just “name”

Scope, S&W, 2.1, figure page 189.

Overloading

```java
public static int abs (final int x) {
    return x<0?-x:x;
}
public static double abs (final double x) {
    return x<0.0?-x:x;
}
```

Overload resolution does take the return type into consideration. Java does promote int values to double, but try to avoid taking advantage of that as that will force the reader to learn the rules of overload resolution.
myBirthDay.plus(2, ChronoUnits.DAYS)

- not capitalized: instance of class, so non-static access
- camel case: method or procedure name
- parentheses: method or procedure invocation
- all uppercase: static final
- capitalized: class name, so static access
A class can contain (static) code which can be used by another class. For example, one might use `Math.hypot`. See page 219, in S&W.
Method Execution/Program Tracing/Runtime Stack

- StackTrace.java
public static int abs(final int x) {
    if (x < 0) return -x else return x;
}
public static double abs(final double x) {
    if (x < 0.0) return -x else return x;
}
public static int abs(final int x) {
    return x < 0 ? -x : x;
}
Scope

The scope of a declaration is the portion of the program in which the identifier declared can be used.
See page 189 of the textbook.
public final class NewtonExample {

    // Sqrt using Newton’s method
    public static double sqrt (final double x) {
        assert x>=0;
        double t = 1.0D;  // a dumb guess
        for (;;) {
            // Improve the current guess, t, with a closer one
            final double t1 = (x/t + t) / 2.0;

            // Stop iterating when there is little improvement
            if (Math.abs(t1-t) < 1e-9) break;
            t = t1;
        }
        return t;
    }

    public static void main (final String[] args) {
        for (final String arg: args) {
            System.out.println (sqrt (Double.parseDouble (arg)));
        }
    }
}
public final class NewtonExample {
    // Sqrt using Newton's method
    public static double sqrt (final double x) {
        assert x>0;
        double t = 1.0D;  // a dumb guess
        for (; ;) {
            // Improve the current guess, t, with a closer one
            final double t1 = (x/t + t) / 2.0;
            // Stop iterating when there is little improvement
            if (Math.abs(t1-t) < 1e-9) break;
            t = t1;
        }
        return t;
    }
    public static void main (final String[] args) {
        for (final String arg: args) {
            System.out.println (sqrt (Double.parseDouble (arg)));
        }
    }
}
call N.main args={"1","2"}  
call D.p x="1"  
return 1.0D  
call N.sqrt x=1.00  
call M.abs x=0.00  
return 0.00  
return 1.00  
call print "1.00"  
return  
call D.p x="2"  
return 2.0D  
call N.sqrt x=2.00  
call M.abs x=0.50  
return 0.50  
call M.abs x=0.42  
return 0.42  
call M.abs x=0.41  
return 0.41  
return 1.41  
call print "1.41"  
return
A class can contain (static) code which can be used by another class. For example, one might use `Math.hypot`. See page 219, in S&W.
Parameter Passing

The names of the parameters to a subprocedure should be used to refer only the arguments, and, so, should be declared to be final in Java. This is hardly ever any exception to the rule. Occasionally one needs an exception in the case of local variables, but hardly ever for the case of formal parameters.
Parameter passing in Java is simple (but involves assignment which is not simple).

```java
public class Main {
    public static void multiply (final int x, final int y) {
        final int p = 5*(x-1);
        System.out.printf ("%d %d%n", y, p);
    }
    public static void main (final String[] args) {
        final int a = Integer.parseInt (args[0]);
        final int ans = multiply (a, a+3);
    }
}

public class Main {
    public static void main (final String[] args) {
        final int a = Integer.parseInt (args[0]);
        multiply: {
            final int x = a; -- assign actual 1 to formal
            final int y = a+3; -- assign actual 2 to formal
            final int p = 5*(x-1); -- body of procedure
            System.out.printf ("%d %d%n", y, p);
        }
    }
}
```
To understand a procedure call conceptually, the body of the subprocedure is inserted in-line with the code of the caller. Of course, it is not practical or desirable to eliminate all subprocedures in the source code. And, in the case of recursion, it is not possible to eliminate the calls since the process would not terminate.
public class Main
    public static int multiply (final int x, final int y) {
        final int p = 5*(x-1);
        return p*y;
    }
    public static void main (final String[] args) {
        final int a = Integer.parseInt (args[0]);
        final int ans = multiply (a, a+3);
    }
}

public class Main {
    public static void main (final String[] args) {
        final int a = Integer.parseInt (args[0]);
        multiply: {
            final int x = a; -- assign actual 1 to formal
            final int y = a+3; -- assign actual 2 to formal
            final int p = 5*(x-1); -- body of procedure
        }
        final int ans = p*y;
    }
}
call-by-value or pass-by-value
This does cause some confusion as the notion of value is ambiguous.
Perhaps parameter passing in Java (and other languages) might best be described
as “call by assignment”: the value of the formal argument is assigned to the formal
parameter as a local variable. The assignment is performed once just before the
subprocedure is executed.
The same mechanism is used primitive types, object types (array, strings, etc.).
Key to understand the execution of any Java program (including subprocedure
 invocation) is sharing and mutable and immutable data types. This is a later topic.
Parameter Passing

- PassByValue.java (information flows in)
- PassString.java (pass an Object)
- PassArray.java (arrays are mutable Objects)
- Wrapper2.java (wrapper classes do not help)
- MultipleReturn.java
- Color.java (is the a better example)