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*4`

`2.9+7.4`, `5.2*4.3`

Avoid the use of + for conversion to strings. Use formatter strings which are more flexible and clearer.

Integer Arithmetic

The binary operators / and % on integers are useful.

```
x =  y * (x / y) + (x % y)
```

Beware of *negative* numbers as `x%y` is negative when `x` is negative contrary to expectations.

Integer Arithmetic

Think carefully about integers of finite size and use the `Math` class (64 bit `long` versions are also in the Java API):

```
int ceilDiv  (int x, int y)
int floorDiv (int x, int y)
int ceilMod  (int x, int y) //ceilMod(-4, -3)==+2; and (-4 %
int floorMod (int x, int y) //floorMod(-4,+3)==+2; and (-4 %
```

To fail instead of wrapping around:

```
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)
int divideExact (int x, int y)
int multiplyExact (int x, int y)
```

Other Boolean Operators

relational: `<`, `>`, `<=`, `>=`
equality: `==`, `!=`
conditional: `&&`, `||`

Definition

Short-circuit evalution is when the second argument of a boolean operation is executed or evaluated only if the first argument does not suffice to determine the value of the expression.

Notice the similarity with the if statement and if the expression. Their “else” part is not evaluated if the test is false.

if Expression

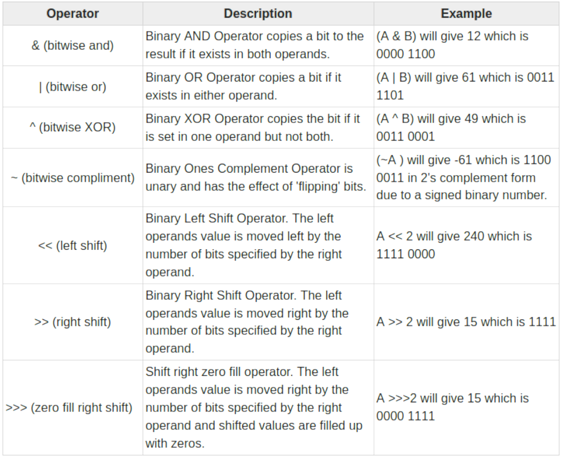
Conditional Operatator

```
String url = args.length>0?args[0]:"http://www.cs.fit.edu/~ry
String s = (Math.cos (0.0) > 0.0) ? "positive" : "non-positiv
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; -- 60
b = 0b0000_1101; -- 13
```



Operator	Description	Example
& (bitwise and)	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12 which is 0000 1100
| (bitwise or)	Binary OR Operator copies a bit if it exists in either operand.	(A | B) will give 61 which is 0011 1101
^ (bitwise XOR)	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49 which is 0011 0001
~ (bitwise compliment)	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.
<< (left shift)	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240 which is 1111 0000
>> (right shift)	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 1111
>>> (zero fill right shift)	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.	A >>>2 will give 15 which is 0000 1111

Bit Operations

Shift Operators

1. The value of `n >> s` is `n` right-shifted `s` bit positions with *sign-extension*. The resulting value is `floor(n / 2s)`. or non-negative values of `n`, this is equivalent to truncating integer division, as computed by the integer division operator `/`, by two to the power `s`.
2. The value of `n >>> s` is `n` right-shifted `s` bit positions *with zero-extension*

```
11111111111111111100111000001011 -12,789
11111111111111111111111111100111    »  9
00000000011111111111111111100111    »> 9
```

Difference Between Statements and Declarations

```
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. The are acts of communication to the translator (before execution begins.)

Simple Statements

- Empty
- Expression statements
  - Assignment, e.g., `x=2.0*Math.pi;`
  - Subprocedure invocation, e.g., `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`
  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

```java
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 resourses (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'
}

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

Try Resource

```
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`

Static Methods

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.

A method (function) may stand for a value, that is, it is an expression (when its return type is not void.) And a method (subprocedure) may peform some actions, that is, it is a statement (when its “return” type is void.)

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

```
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 not 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.

Libraries

A class can contain (static) fields and code which can be used by another class. The Java style capitalization conventions are crucial in detecting how values and methods are accessed.

not capitalized: instance of class, so non-static access

camel case: method or procedure name

parentheses: method or procedure invocation

```
myBirthDay.plus (2, ChronoUnits.DAYS)
```

all uppercase: static final

capitalized: class name, so static access

Overloading

Two functions with the same name. But the argument in one has a different type than the other.

```
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;
}
```

Could use the conditional operator!

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

Overload resolution

By the number and type of the actual arguments.

```
int a = abs (-3);
double d = abs (-3.1D);
```

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.

Scope

```java
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)));
        }
    }
}
```

Declare variableſ
right before you need them.
Localize ſcope!

Sequence of Calls and Returns

```java
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
return
```

Libraries

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). To understand a procedure call conceptually, the body of the subprocedure is

inserted in-line with the code of the caller. The actual arguments are assigned to the formal parameters as local variables.

```
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 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);
    }
  }
}
```

```
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);
  }
}
```

Functions or subprocedures make no different to parameter passing.

```java
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: {
      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
      ans = p*y;
    }
  }
}
```

Parameter Passing

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.

Parameter Passing

call-by-value or pass-by-value

This does 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 mechansim 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 only in
- `PassString.java` – pass an object
- `PassArray.java` (arrays are mutable Objects)
- `Color.java` (is a better example)
- `Wrapper2.java` (wrapper classes do not help)
- `MultipleReturn.java`
- `R.java` – records help

Parameter Passing (Summary)

- Only one paramater massing mechansim
- It is called *call-by-value*
- It is implemented by assigning to the formal parameters as if local variables
- It is designed for information flowing into the subprocedure. (Parameters parameterized subprocedures; abstraction!)
- It it provides some protection from “rogue” or “misbehaving” subprocedures, *but not always*
- `return` is for information flowing back to caler
- Java objects are not passed differently.