Basics of Java

Components of a Java Program

- **statements** - a statement is some action or sequence of actions, given as a command in code. A statement ends with a semi-colon (;)
- **blocks** - A block is a set of statements enclosed in set braces {}. Blocks can be nested
- **classes** - A class is a blueprint for building objects in Java
  - Every Java program has at least one class
  - Programmers can define new classes
  - There are many pre-built classes in the Java SDK
- **methods** - A method is a function that belongs to a class
  - In Java, *all* functions are methods, meaning they are always contained in some class
- A Java program can be made up of multiple classes, spread across multiple code files, and it will typically make use of some SDK libraries as well
- **the `main` method** - Every Java application must have a `main` method, which defines where the program begins. In Java, the `main` method belongs to a class. Any class can have a `main` method. The `main` method looks like this:

```
public static void main(String[] args) {
    // statements
}
```

Java source code files

- The Java compiler imposes some specific rules on the naming of source code files.
- A Java source code file has a base name, along with the extenion `.java`
- A source file can contain one or more classes
- **if** there are multiple classes in a code file, one and only one of them should be declared to be `public`
- The base name of the filename *must* match the name of the class that is declared to be `public` in the file.
  - If there is only one class in the file, the filename must match that class name
  - class names in Java *are* case sensitive
- Example: This class belongs in the file `Yadda.java`

```
class Yadda {
    public static void main(String[] args) {
        System.out.println("Yadda Yadda Yadda");
    }
}
```

- Example 2: This file must be named `Daffy.java`

```java
class Bugs {
    public static void main(String[] args) {
        System.out.println("What's up doc?");
    }
}

public class Daffy {
    public static void main(String[] args) {
        System.out.println("You're dethpicable");
    }
}
```

Statements

- Statements in Java are made up of the following
- **reserved words** - words that have pre-defined meanings in the Java language
- **identifiers** - words that are created by programmers for names of variables, functions, classes, etc.
- **literals** - literal values written in code, like strings or numbers
- integer literal - an actual integer number written in code (4, -10, 18)
- float literal - an actual decimal number written in code (4.5, -12.9, 5.0)
- character literal - a character in single quotes ('F', 'a', '\n')
- string literal - a string in double quotes ("Hello", "Bye", "Wow!\n")
- **operators** - special symbols that perform certain actions on their operands
  - A **uniary** operator has one operand
  - A **binary** operator has two operand
  - A **ternary** operator has three operand (there is only one of these)
- Calls to methods (functions)

Escape Sequences

- String and character literals can contain special *escape sequences* that represent single characters that cannot be represented with a single character in code

Escape Sequence	Meaning
\n	new line
\t	tab
\b	backspace
\r	carriage return
\"	double quote
\'	single quote
\\	backslash

Comments

Comments are used to improve the readability of code. Comments are ignored by the compiler. There are two styles of comments in Java: * block style - comments enclosed in a block that starts with `/*` and ends with `*/`

```
/* This is a comment */
```

- Line style - comment follows the double slash marker `//`. Everything after this mark, to the end of the line, is a comment

```
int x;       // This is a comment
x = 3;       // This is a comment
```

Variables

Variables are used to store data. Every Java variable has a * **Name** – chosen by the programmer (aka *the identifier*) * **Type** – specified in the declaration of the variable * **Size** – determined by the type * **Value** – the data stored in the variable's memory location

Identifiers

Identifiers are the names for things (variables, functions, etc.) in the language. Some identifiers are built-in, and others can be created by the programmer. * User-defined identifiers can consist of letters, digits, underscores, and the dollar-sign `$` * Must start with a non-digit * Identifiers are case sensitive (`count` and `Count` are different variables) * *Reserved words* (keywords) cannot be used as identifiers * an identifier can be any length

Style conventions (for identifiers)

While you can legally pick any name for a variable that follows the *rules*, it is also a good idea to follow common programming conventions, for easy-to-read code. Here are some conventions used in the Java SDK * class and interface names start with an uppercase letter * variable names and method names start with a lowercase letter * *constants* are usually in ALL CAPS * When using names that are made up of multiple words, capitalize the first letter of each word after the first - `numberOfMathStudents` * In addition, it is good to pick mostly meaningful identifiers, so that it's easy to remember what each is for

```
int numStudents;    // good name
String firstName;   // good name
int a, ns;          // not so good name
String fn;          // not so good name
```

Primitive Data Types

Java has a small set of what are known as *primitives.* These are basic data types that are predefined for the language * **char** - used for storing single characters (letters, digits, special symbols, etc.) * **boolean** - has two possible values, `true` or `false`. * Interger types - for storage of integer values - byte - short - int - long * floating point types - for storage of decimal numbers - float - double

Declaring variables

- Inside a block, variables must be declared before they can be used in later statements in the block.
- Declaration format `typeName variableName1, variableName2, ...;`

```
int numStudents;     // variable of type integer
double weight;       // variable of type double
char letter;         // variable of type char
boolean flag;        // variable of type boolean
int test1, test2, finalExam;
double average, gpa;
```

Initializing Variables

- To **declare** a variable is to tell the compiler it exists, and to reserve memory for it
- To **initialize** a variable is to load a value into it for the first time
- One common way to initialize variables is with an *assignment statement*

```
int numStudents;
double weight;
char letter;

numStudents = 26;
weight = 165.35;
letter = 'A';
```

- Variables of built-in types can be declared and initialized in the same line

```
int numStudents = 26;
double weight = 165.35;
boolean flag = false;
```

Constant Variables

- A variable can be decalred constant by using the keyword `final`
- A constant variable (while an oxymoron term) is a variable that cannot change value

```
final double PI = 3.14159;
```

- After this, PI cannot be changed. So the following would not work

```
PI = 3
```

- By convention, we use all upper case letters to denote a constant variable

Operators

- Special built-in symbols that have functionality, and work on *operands*
- **Operand** – an input to an operator
- **Arity** – how many operands an operator takes
  - *uniary operator* – has one operand
  - *biary operator* – has two operands
  - *triary operator* – has three operands
- Examples

```
int x, y = 5, z;
z = 10;     // assignment operator (binary)
x = y + z;  // addition (binary)
x = -y;     // -y is a unary operation (negation)
x++;        // unary (increment)
```

- **Cascading** - linking of multiple operators, espically of related categories, together in a single statement

```
x = a + b + c - d + e;  // cascading arithmetic operators
x = y = z = 3;          // cascading assignment operators
```

- this works because the result of one operation send back the answer (i.e. a *return value*) in its place, to be used in the next piece of the statement.
- In the above, `(a + b)` happens first, then the answer becomes the first operand in the next `+` operation.
- **Precedence** – rules specifying which operators come first in a statement containing multiple operators

```
x = a + b * c;      // b * c happens first, since * has a
                    // higher precedence than +
```

- **Associativity** – rules specifying which operators are evaluated first when they have the same level of precedence.
  - *Most* (but not all) operators associate from left to right

Assignment Operator

- Value on the right side (R-value) is assigned to (i.e stored in) the location (variable) on the left side (L-value)
  - **R-value** – any expression that evaluates to a single value (name comes from "right" side of assignment operator)

  - **L-value** – A storage location! (**not** any old expression). A variable or reference to a location. (name comes from the "left" side of assignment operator)
  - Typical usage:

`variable_name = expression`

- The assignment operator returns the L-value (which now stores the new value)
- Examples

```
x = 5;
y = 10.3;
z = x + y;  // right side can be an expression
a + 3 = b   // ILLEGAL! Left side must be a storage location
```

- Associates right-to-left

```
x = y = z = 5;      // z = 5 evaluated first, returns z
```

- Use appropriate types when assigning values to variables

```
int x, y;
x = 5843;
y = -1234;      // assigning integers to int variables

double a, b;
a = 12.98;
b = -345.8;     // can assign decimal numbers to double

float c, d;
c = 12.98f;     //
d = -345.8f;    // put a small 'f' after a literal value
                // to indicate float, instead of double
char letter, symb;
letter = 'Z';
symb = '$';     // can assign character literals to char
                // types
boolean flag;
flag = true;
flag = false;   // can assign true or false to boolean
                //variables
```

- Be careful to not confuse assignment `=` with comparision `==`

Arithmetic Operators

Name	Symbol	Arity	Usage
Add	+	binary	x + y
Subtract	-	binary	x - y

Name	Symbol	Arity	Usage
Multiply	*	binary	x * y
Divide	/	binary	x / y
Modulus	%	binary	x % y
Minus	-	unary	-y

- Division is a special case
  - For types `float` and `double` the `/` operator gives the standard decimal answer

```
double x = 19.0, y = 5.0, z;
z = x / y;          // z is now 3.8
```

  - For integer types, `/` fives the quotient, and `%` gives the remainder (as in long division)

```
int x = 19, y = 5, q, r;
q = x / y;       // q is 3
r = x % y;       // r is 4
```

  - An operation on two operands of the same type returns the same type
  - An operation on mixed primitive types (if compatible) returns the "larger" type (floating point types are "larger" than integers, because no data is lost converting from integer to decimal point)

```
int x = 5;
double y = 3.6;
z = x + y;       // What does z need to be?
                 // x + y returns a double
```

Operator precedence

- Arithmetic has usual precedence
  1. parentheses
  2. Unary minus
  3. `*`, `/`, and `%`
  4. `+` and `-`
  5. Operators on same level associate left to right
- Many different levels of operator precedence
- When in doubt, can always use parentheses
- Example

```
z = a - b * -c + d / (e - f);    // 7 operators in this
                                 // statement. What order
                                 // should they be
                                 // evaluated in?
```

Arithmetic shortcuts

```
v += e;      // same as v = v + e;
v -= e;      // same as v = v - e;
```

```
v *= e;      // same as v = v * e;
v /= e;      // same as v = v / e;
v %= e;      // same as v = v % e;
```

Increment and Decrement Operators

```
++x; // pre-increment (returns reference to new x)
x++; // post-increment (returns value of old x)
    // shortcut for x = x + 1
--x;    // pre-decrement
x--;    // post-decrement
        // shortcur for x = x - 1
```

- Pre-increment: Incrementing is done **before** the value of x is used in the rest of the expression
- Post-increment: Incrementing is done **after** the value of x is used in the rest of the expression
- Note – this only maters if the variable is actually used in another expression. These two statements by themselves have the same effect

```
x++;
++x;
```

- Examples

```
int x = 5, count = 7;
int result = x * ++count;   // result = 40, count = 8

x = 5;
count = 7;
result = x * count++;       // result = 35, count = 8
```

Type Conversions

- When working with mixed primitive types, conversions can take one of two forms
  1. Automatic type conversion
     - When appropriate, the compiler will automatically convert a smaller numeric type into a larger type (where the floating point types are always considered "larger" than the integer types)
  2. Explicit cast operations
     - For all other conversions, the programmer must specify with a cast operation
     - To cast, put the type in parintheses before the expression whose value you are casting

```
int i1 = 5, i2;
short s1 = 3;
double d1 = 23.5, d2;
```

```
float f1 = 12.3f;
byte b1 = 10;

d2 = i1;         // automatically allowed
i1 = b1;         // automatically allowed
s1 = (short)i1; // requires cast operation
                 // (some data may be lost)
i1 = (int)d1;    // requires cast operation
                 // (decimal data may be lost)

d2 = f1 + d1;    // automatically allowed
i2 = b1 + s1;    // automatically allowed
```