Content

• (Our) Basic Program
• 3 Key Basic Types (& String)
• Static vs. Final
• Working With Variables
• Mathematical & Relational Operators
• Logical Operators & Assignment Operators
• Input & Output
• Escape Sequences
• Control Structures
• Errors & Exceptions
• Classes
(Our) Basic Program

/** Program Comments */
import java.io.*;
import java.util.*;

public class BasicProgram {
    public static void main( String[] args ) {
        boolean myBool;  // 3 - Basic Identifiers
        int myInt;
        double myDouble = 3.14;

        String myString;  // String Class
        Scanner jin = new Scanner(System.in);
        // Scanner Class
    }
}
3 Primitive (built-in) Types

- Variables (3 - parts)
  - Type -> Kind of variable we're working with
  - Identifier -> Unique name. Use camel case
  - Value - > Limited by type.

- The Big 3
  - boolean: true or false [defaults to false]
  - int: [defaults to 0]
  - double: [defaults to 0]
    - Double should never be used for precise values, such as currency. (java.math.BigDecimal class instead) [defaults to 0]
Variables (creating/using/naming)

- **Declaration**: Declares existence & type of a variable.
  - `int x;`

- **Initialization**: To give a variable a starting value:
  - `x = 33;`

- **Naming Conventions**:
  - Variable names are case-sensitive (Age != age)
  - Can begin with any letter, $ or _ (but don’t use $ or _)
  - Can contain any letter, number, $ or _ (don’t use $ or _)
  - Make the variable name meaningful (age, height, width)
  - For longer variable use "camel-case" (myAge)
  - Use all capitals for final values (MAXHEIGHT);
Literals

- A literal is the source code representation of a fixed value; literals are represented directly in your code without requiring computation.

- As shown below, it's possible to assign a literal to a variable of a primitive type:
  - int decVal = 26;  // The number 26, in decimal
  - int hexVal = 0x1a;  // The number 26, in hexadecimal
  - int binVal = 0b11010;  // The number 26, in binary
  - double d1 = 123.4;
  - double d2 = 1.234e2;  // in scientific notation
Strings

- A "string literal" is just a list of alpha-numeric characters.
  - "Bring me CODE!!!"

- The String class allows us to create String objects that contain strings literals (wrapper class).
  - String battlecry = "Bring me CODE!!!"

- You can't actually change a "string literal" in Java, however you can have a variable that points to a string object change and point to a new string object.
  - battlecry = new String("Code or Die!!");

- In JAVA parlance String objects are "immutable" meaning they can't be changed (they lack mutators).
What about the old string

```java
String s1 = new String("Give me code, ");
String s1 = new String("or give me death ");
```

- On the 1st line we created an object reference (s1) to a string ("Give me code, "); and then set aside space on the heap for that string.
- In the second line, we reassigned where the object reference (s1) points, orphaning the first string literal.
- This is NOT a problem in Java, as the JVM will detect that string is orphaned and delete it.
- Java has automatic GARBAGE COLLECTION
String Concatenation

- Strings can be linked together using the '+' character:
  
  ```java
  String s1 = "Give me code, ";
  String s2 = "or give me death ";
  battlecry = s1 + s2;
  ```

- Strings can be concatenated with non-strings (automatic conversion will take place) but at least one of the objects to be joined MUST be a string.
  
  ```java
  int x = 1000;
  battlecry = battlecry + x + " times!";
  ```

- Escape sequences can be added to strings:
  ```java
  battlecry += "\n";
  ```

- String can be printed:
  ```java
  System.out.println(battlecry);
  ```
Static & Final

- The keyword static is used to indicate a variable or method that DOES NOT belong or refer to any individual object!
- A static variable belongs to the CLASS in which it is declared and can be accessed by EVERY object derived from that class.
- A static method belongs to the CLASS in which it is declared and does not operate on any individual object of that class.
- The main() method of your program class must always be declared to be static.
- Final variables, can be used by anyone but will NEVER CHANGE their value.
Casting (converting types)

- We can safely assign:
  - byte -> short -> int -> long -> float -> double

- If you try and go the other way you get an error.

- Otherwise we have to explicitly change the data type:
  - int i=0;
  - float f;
  - f=(float)i; // Cast int as float

- Remember the adding or subtracting 0.5 in order to round trick.
  - int i = (int) (f + 0.5);
Mathematical Operators

+ Addition
- Subtraction
* Multiplication
/ Division
% Modulus (also called remainder)

• Notice: no exponent! While there is no exponentiation symbol in Java, there is a Math.pow( ) method which will be discussed later.

• Modulus:

  \[ 22 \% 6 = 4 \] (because \( 22 / 6 = 3 \) with a remainder of 4)

• PEMDAS

  \[ 2 + 7 \* 3 \text{ is } 23 \] (not 27).
Relational & Logical Operators

- The majority of these operators will probably look familiar to you.
- Keep in mind that you must use "==", not "=" when testing if two primitives are equal.
- Watch for short-circuited operations!!!
Assignment Operators

- All are right-associative ie a=b=c groups as a=(b=c) vs (a=b)=c
- All are used with primitive data types except = and += which can be used with Strings
- Operators of the form op = cast their result to the type of the left-operand

<table>
<thead>
<tr>
<th>Op</th>
<th>Example</th>
<th>Equivalent Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>+=</td>
<td>x += y;</td>
<td>x = (x + y);</td>
</tr>
<tr>
<td>-=</td>
<td>x -= y;</td>
<td>x = (x - y);</td>
</tr>
<tr>
<td>*=</td>
<td>x *= y;</td>
<td>x = (x * y);</td>
</tr>
<tr>
<td>/=</td>
<td>x /= y;</td>
<td>x = (x / y);</td>
</tr>
<tr>
<td>%=</td>
<td>x %= y;</td>
<td>x = (x % y);</td>
</tr>
<tr>
<td>&amp;=</td>
<td>x &amp;= y;</td>
<td>x = (x &amp; y);</td>
</tr>
<tr>
<td></td>
<td>=</td>
<td>x</td>
</tr>
<tr>
<td>^=</td>
<td>x ^= y;</td>
<td>x = (x ^ y);</td>
</tr>
<tr>
<td>&lt;&lt;=</td>
<td>x &lt;&lt;= y;</td>
<td>x = (x &lt;&lt; y);</td>
</tr>
<tr>
<td>&gt;&gt;=</td>
<td>x &gt;&gt;= y;</td>
<td>x = (x &gt;&gt; y);</td>
</tr>
<tr>
<td>&gt;&gt;&gt;=</td>
<td>x &gt;&gt;&gt;= y;</td>
<td>x = (x &gt;&gt;&gt; y);</td>
</tr>
</tbody>
</table>
The Unary Operators

- The unary operators require only one operand; they perform various operations such as incrementing/decrementing a value by one, negating an expression, or inverting the value of a boolean.
- + Unary plus operator
- - Unary minus operator
- ++ Increment operator; increments a value by 1
- -- Decrement operator; decrements a value by 1
- ! Logical complement; inverts value of a boolean

**NOTE:** There is a difference between: x++ and ++ x

```java
x = 3;
System.out.println(++x);  // System.out.println(x++);
```
Input

- A scanner object is one way in which we can get information from the command line.
  
  ```java
  Scanner inS = new Scanner(System.in);
  // System.in is InputStream
  Scanner inF = new Scanner(new FileReader("file"));
  ```

- Trying to read input that is not there will throw an exception, so use the "has" functions to check if a type of token exists.

<table>
<thead>
<tr>
<th>Method</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>int nextInt()</td>
<td>Returns the next token as an int. If the next token is not an integer, InputMismatchException is thrown.</td>
</tr>
<tr>
<td>long nextLong()</td>
<td>Returns the next token as a long. If the next token is not an integer, InputMismatchException is thrown.</td>
</tr>
<tr>
<td>float nextFloat()</td>
<td>Returns the next token as a float. If the next token is not a float or is out of range, InputMismatchException is thrown.</td>
</tr>
<tr>
<td>double nextDouble()</td>
<td>Returns the next token as a long. If the next token is not a float or is out of range, InputMismatchException is thrown.</td>
</tr>
<tr>
<td>String next()</td>
<td>Finds and returns the next complete token from this scanner and returns it as a string; a token is usually ended by whitespace such as a blank or line break. If not token exists, NoSuchElementException is thrown.</td>
</tr>
<tr>
<td>String nextLine()</td>
<td>Returns the rest of the current line, excluding any line separator at the end.</td>
</tr>
<tr>
<td>void close()</td>
<td>Closes the scanner.</td>
</tr>
<tr>
<td>boolean hasNextLine()</td>
<td>Returns true if the scanner has another line in its input; false otherwise.</td>
</tr>
<tr>
<td>boolean hasNextInt()</td>
<td>Returns true if the next token in the scanner can be interpreted as an int value.</td>
</tr>
<tr>
<td>boolean hasNextFloat()</td>
<td>Returns true if the next token in the scanner can be interpreted as a float value.</td>
</tr>
</tbody>
</table>
There are two options:

a. System.out.print();

b. System.out.println();

We can output both text, variables, and equation results:

a. System.out.println("Hi");

b. System.out.print("Hi " + yourName + " \n ");

c. System.out.println( 2 + 36 / 6 );

Writing to a file is useful and requires the creation of a file variable and a FileWriter stream.
Escape Sequences

- In Java any character that is preceded by a backslash (\) is known as escape sequence, which has special meaning for the compiler.
- Escape sequences allow us to change the structure of output and to print characters that we would otherwise be unable to print.

<table>
<thead>
<tr>
<th>Escape Sequence</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>\n</td>
<td>newline</td>
</tr>
<tr>
<td>\t</td>
<td>tab</td>
</tr>
<tr>
<td>\b</td>
<td>backspace</td>
</tr>
<tr>
<td>\f</td>
<td>form feed</td>
</tr>
<tr>
<td>\r</td>
<td>return</td>
</tr>
<tr>
<td>&quot;</td>
<td>&quot; (double quote)</td>
</tr>
<tr>
<td>'</td>
<td>' (single quote)</td>
</tr>
<tr>
<td>\</td>
<td>\ (back slash)</td>
</tr>
<tr>
<td>\uDDDD</td>
<td>character from the Unicode character set (DDDD is four hex digits)</td>
</tr>
</tbody>
</table>
Control Structures (1) - SELECTION

- BEWARE OF THE DANGLING ELSE PROBLEM!
  
  ```
  if ( num > 0 )
  
  if ( num < 10 )
    mystring = "aaa";
  else
    mystring = "bbbb";
  ```

- In your programs ALWAYS used '{' and '}' to delineate your statements.

```java
if (testscore >= 90) {
    grade = 'A';
} else if (testscore >= 80) {
    grade = 'B';
} else if (testscore >= 70) {
    grade = 'C';
} else if (testscore >= 60) {
    grade = 'D';
} else {
    grade = 'F';
}
```
(2) – REPETITION (ITERATION)

**WHILE**

while(pass == false) {
   System.out.println("No!");
}

**FOR**

for( i=1; i<5; i++) {
   System.out.print( i + " ");
}

**FOR-EACH**

Shortened version of a for loop used for arrays and other collections

```java
for (int x : arr1) {
   System.out.println(x);
}
```

FOR-EACH loops CANNOT be used to MODIFY an array or ArrayList.
Errors and Exceptions

• When your program is running it may encounter an error and crash.

• Specific types of errors can be identified by the JVM and given names when they are discovered.

• These errors are called "exceptions" and you can write your code to deal with them (keep your program from crashing).

• Common Exceptions:
  ◦ ArithmeticException
  ◦ NullPointerException
  ◦ ClassCastException
  ◦ ArrIndexOutOfBoundsException
  ◦ IndexOutOfBoundsException
  ◦ IllegalArgumentException
Try, Throw and Catch

- Creating and throwing exceptions give our programs the chance to recover from errors that would otherwise crash the system.

- Example:

```java
String filename = Scanner.nextLine();
try {
    // Create the file
    new File(filename).createNewFile();
} catch (IOException e) {
    // Print exception that occurred
    System.out.println( filename + " : " + e.getMessage() );
} // Program continues after IOException handler is executed
```
Classes
Content

• Classes & "the heap"
• Objects & Classes
• Public, Private and Static
• Methods (aka Functions, aka Procedures)
  ◦ General methods
  ◦ Class Constructors
  ◦ Class Accessors & Mutators
  ◦ Overloading
• Scope (more than just a mouthwash)
• References
  ◦ NULL
  ◦ Passing by reference (Parameters)
Objects & Classes

- A class is a pattern (template, description) of a type of object including the facts (state, data fields, member fields) that describe a type of object and the functions (methods, procedures, behaviors) that can operate on an object of that type.
  - Blueprints for a house.
  - Dress Pattern

- An object is an instantiation (an instance, a construct) created from a class. You can make as many objects as you want from a class.
  - 1 house blueprint -> multiple different houses
  - 1 dress pattern -> multiple dresses.
class Pet{
    double weight;
    Person owner;

    // .... Other functions ...

    // @ symbol used to indicate input parameters / return values
    /**
     * isHappyToSee() -> is happy to see a particular person.
     * @param p The person who the pet sees.
     * @return true if the Person is their owner, false otherwise.
     */
    boolean isHappyToSee(Person p) {
        return p == owner;
    }
}
Objects

public class driverProgram{
    public static void main(String[] args){
        Pet spot = new Pet();
        Pet rover = spot;
        // rover is an alias for spot.
        Pet cujo = new Pet();
        // How many pet object exist?
    }
}
Classes & "the heap"

- A class is a pattern or template for:
  1. Creating objects (which are just collections of related variables) on the heap.
  2. Defining methods (functions/procedures) allowed to work on the object member variables for that type.

- Remember the keyword "new" means we are "going to the heap".
- If we don't go to the heap (don't initialize our variable) then we have a variables whose value is NULL as it points to nothing.
Public, Private

- Some of the member variables inside of an object we may want to protect and/or limit access too.
- We can do this through "access specifiers".
- Public means anyone can access!
- Private means only methods that are inside the class can access.

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Class</th>
<th>Package</th>
<th>Subclass</th>
<th>World</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>protected</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>no modifier</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>private</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>
Don't forget that a CLASS is a thing (it's a pattern)!

Your main program is a CLASS and all of it's methods and variables should be declared static.

Static -&gt; Stays put!
- Memory allocation only happens once.
- It belongs to the class itself, not any objects of the class.
- Static variables have only one instance but can be seen by every object in the class.
- Static methods DO NOT work on objects, but instead work on the class itself (usually static variables in the class).
Methods (aka Functions, aka Procedures)

- A method (function, procedure) is a block of code with a name. When I want that code, I call it by its name.
- A method declaration (the thing that creates a method) has four different parts (listed in order):
  - Access specifier.
  - Return type
  - Name
  - Parameter list
- We often include methods in our classes to control how the member variables of a classes objects are created/manipulated.
public double hoursSlept(double lbsOfFood) {
    return 24.0*lbsOfFood/weight;
}

// Note use of @ to indicate input parameters and return values
/**
* isHappyToSee() -> is happy to see a particular person.
* @param p The person who the pet sees.
* @return true if the Person is their owner, false otherwise.
*/
public boolean isHappyToSee(Person p) {
    return p == owner;
}
Class Constructors

- When you create a new instance (a new object) of a class using the new keyword, a constructor for that class is called.

- Constructors are used to initialize the instance variables (fields) of an object. Constructors are like methods, but with some important differences.
  - Constructor name is same as class name.
  - There is no return type given (or return statement) in a constructor. The value is this object itself so there is no need to indicate a return value.
  - Constructors can call other constructors (using this and using super). In general compiler automatically inserts a call to the parameterless superclass constructor.
Class Constructors

- **Default constructor.** If you don't define a constructor for a class, a default parameterless constructor is automatically created by the compiler.

- The default constructor calls the default parent constructor (super()) and initializes all instance variables to default value (zero for numbers, null for objects, and false for booleans).

- **Default constructor is created only if there are no constructors.** If you define any constructor for your class, no default constructor is automatically created.
public class Point {
    int m_x;
    int m_y;

    // Constructor
    public Point(int x, int y) {
        m_x = x;
        m_y = y;
    }

    // Parameterless constructor
    public Point() {
        this(0, 0);
        // Calls other constructor.
    }

    // Constructor (same as other example)
    public Point(int x, int y) {
        super();
    // Automatically done if you don't call
    // Some other constructor here
        m_x = x;
        m_y = y;
    }

    // The reason for default call to super() will
    // become clear when we study inheritance.
Class Accessors & Mutators

- To the right we see the person class.
- Notice how all the data fields are **private**.
- This will prevent operations like:

```java
my.firstName = "Bob";
```

- Accessors and Mutators are special functions used to manipulate private variables.

```java
class Person {
    //Private fields
    private String firstName;
    private String middleNames;
    private String lastName;
    private String address;
    private String username;

    //Constructor method
    public Person(String firstName, String middleNames, String lastName, String address) {
        this.firstName = firstName;
        this.middleNames = middleNames;
        this.lastName = lastName;
        this.address = address;
        this.username = "";
    }
}
```
<table>
<thead>
<tr>
<th>Accessors (access)</th>
</tr>
</thead>
<tbody>
<tr>
<td>//Accessor for firstName</td>
</tr>
<tr>
<td>public String getFirstName() {</td>
</tr>
<tr>
<td>return firstName;</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>//Accessor for middleNames</td>
</tr>
<tr>
<td>public String getMiddlesNames() {</td>
</tr>
<tr>
<td>return middleNames;</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>//Accessor for lastName</td>
</tr>
<tr>
<td>public String getLastName() {</td>
</tr>
<tr>
<td>return lastName;</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mutators (set or change)</th>
</tr>
</thead>
<tbody>
<tr>
<td>//Mutator for address</td>
</tr>
<tr>
<td>public void setAddress(String address) {</td>
</tr>
<tr>
<td>this.address = address;</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>//Mutator for username</td>
</tr>
<tr>
<td>public void setUsername(String uname) {</td>
</tr>
<tr>
<td>this.username = username;</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>
Overloading

• Coming up with unique method names can be somewhat tedious.
• Overloading allows methods of a class to share the same name.
• These methods are differentiated by the number and type of arguments passed into the method.
• Common with constructors!

```java
class Person {
    private String firstName;
    private String lastName;

    Person() {
        this.firstName = "";
        this.lastName = "";
    }

    Person(String lname) {
        this.firstName = "";
        this.lastName = lname;
    }

    Person(String fname, String lname) {
        this.firstName = fname;
        this.lastName = lname;
    }
}
```
Scope (more than just a mouthwash)

- The scope of a variable is the part of the program over which the variable name can be referenced.
- You cannot refer to a variable before its declared.
- You can declare variables in several places:
  - In a class body as class fields. Variables declared here are referred to as class-level variables.
  - As parameters of a method or constructor.
  - In a method's body or a constructor's body.
  - Within a statement block, such as inside a while or for block.

**NOTE:** If there is a clash (two variables, same name) the local (or closer) variable declaration is used.
Parameters (and References)

- **REMEMBER!** For **primitive types**, the value of the type is a single (atomic) piece of data!

- **REMEMBER!** For **objects**, the "value" is a reference (a pointer) to a place in memory where the complete collection of the objects data members can be found.

- When a **primitive type** is copied into a method, a copy is made of the value, and the source value is not changed! Parameter has local (or method) scope.

- When a **reference type** is copied into a method, a copy is made of the value (the source value is not changed)! BUT the value is a pointer and what it points too CAN BE CHANGED!!!
Passing by reference (Parameters)

• Be careful when creating objects. If an object is not immediately initialized after creation then it's value is set to null (also sometimes written NULL).

• If you try to reference the member variable of an unitialized object you will get a NullPointerException.

• Example:

Person bill;  // bill has a null value
bill.setLastName("Bixby");
// NullPointerException.
"Programmers are tools for converting caffeine into code."

- Anonymous

You got this....