1. **Why objects?**
   1. **Modularity**: source code for an object can be written and maintained independently of source code for other objects.
   2. **Information-hiding**: by interacting only with an object’s methods, the details of its internal implementation remain hidden from the outside world.
      1. **Encapsulation**: classes and the objects created from them encase their attributes and methods (state and behavior).
   3. **Code re-use**: if an object already exists, you can use that object in your program.
   4. **Pluggability and debugging ease**: if a particular object turns out to be problematic, remove it from your application and plug in a different object in its replacement instead of starting all over.
2. **Modifiers**
   1. Access-Level: restrict which classes can access; use the most restrictive that makes sense.
      1. public: accessible by all classes.
         1. Classes
         2. Interfaces
         3. Constructors: if used outside of class.
         4. Constants
      2. private: accessible only by its own class.
         1. Fields: *indirectly* accessible using **getters** and **setters**
         2. Methods
      3. package-private: visible only within own package.
         1. Default
         2. No modifier required
      4. protected: package-private, but accessible by subclass of its class in another package.
   2. final: constant.
      1. Constants
      2. **Compile-time error**: if application tries to modify.
   3. static: only 1 copy of variable in memory regardless of number of instances, which can be manipulated by any single instance and/or without even instantiating.
      1. Constants
3. **Return Types**
   1. void: nothing.
   2. Primitive data types
   3. Reference data types: arrays and ArrayLists.
      1. **Array**: non-dynamic storage of multiple elements.
         1. **Java**: type[] typeArray;
      2. **ArrayList**: class that supports dynamic arrays.
         1. **Java**: ArrayList<type> arrayList = newArrayList();
         2. add()
         3. remove()
   4. Instances
   5. Classes
   6. Interfaces
   7. **null**: can be returned in place of primitive/reference data, instances, classes, or interfaces.
4. **Naming Conventions**
   1. Class: Class
   2. Constructor: Class
   3. Instance: instanceName
   4. Interface: IClass
   5. Constants: CONSTANT\_NAME
   6. Variables: variableName
      1. Legal
         1. Case-sensitive
         2. Sequence of Unicode digits, beginning with a letter, $, or \_
         3. No white-space or reserved keywords
      2. Convention
         1. Use full words, not cryptic abbreviations.
         2. Always begin with a lowercase letter.
            1. Multiple words: first letter of each word after the first in uppercase.
         3. Do not use $ or \_.
   7. Methods: doIt
      1. Legal: any legal identifier.
      2. Convention: same as variable, but verb followed by adjectives/nouns.
   8. Parameters
      1. Legal
         1. Must be unique and cannot be same as method, local variable, or constructor name.
         2. Can **shadow** class name or constructor name.
            1. this: resolves shadowing by referencing *current object*.
            2. **Explicit constructor invocation**: shadowing constructor name.

Invocation of another constructor MUST be the first line in the constructor.

1. **OOP Terminology**
   1. **Object**: software bundle of related state and behavior.
      1. **State**: characteristics of the object.
         1. **Fields**
      2. **Behavior**: actions of the object.
         1. **Methods**: task performance specific to a task.
            1. **Java**: optionalModifier returnType method(type optionalArguments) {\*/.../\*}
            2. **Method call**: calling a method to perform tasks, returning to code when it…

*completes all the statements in the method*.

*reaches a*return*statement*.

Required for non-void methods

Data type must match return type

**Compiler error**: value is returned for void method.

*throws an exception*.

* + - * 1. **Method signature**: name and parameters of method.

**Method overloading**: variables with the same name are allowed, provided method signatures differ in number and/or types, but not return types.

* 1. **Class**: blueprint from which individual objects are created
     1. **Java**: public class Class {\*/.../\*}
        1. **Class body**: area between {}s for field, constructor, and methods.
     2. **Declaration**: associate a variable name with an object type, but NOT create an object.
        1. **Java**: Class instance;
     3. **Instantiation**: allocating memory for creating an object of the defined class.
        1. **Initialization**: object is initialized by a constructor.
           1. **Java**: Class instance = new Constructor();
           2. **Null pointer error**: object is modified without being initialized.
        2. **Instances**: created object of a class, possibly with differing instance variables.
        3. **Constructors**: create and initialize the object by allocating memory.
           1. **Java**: optionalModifier Constructor(type optionalArguments) {\*/.../\*}
           2. Can be overloaded

**Compile-time error**: 2 constructors with same number and type of arguments for the same class.

* + - * 1. No constructor: default 🡪 parent class 🡪 … 🡪 Object class.
      1. **Default no-argument constructor**: only allocates memory.
    1. **Inheritance**: creating a new class (**subclass**) using an existing class (**superclass**).
       1. **Java**: public Subclass extends Superclass {\*/.../\*}
          1. 1 subclass 🡨 ONLY 1 *direct* superclass
          2. 1 superclass 🡪 many subclasses
       2. Purpose
          1. *Create a subclass from superclass*
          2. *Add to superclass.*
          3. *Code for subclasses is easy to read.*
          4. *Document the state and behavior that each superclass defines, since that code will not appear in the source code of each subclass*
    2. **Garbage collection**: environment automatically deletes objects when it determines that there are no more references to that object.
  1. **Variables**
     1. Fields
        1. **Java**: optionalModifier datatype name;
           1. .: used to access fields.
        2. **Class variable**: static field; common to all instances.
        3. **Instance variable**: non-static field; unique to each instance.
     2. **Local variable**: temporary state variables that exist (and are only visible to) a method.
     3. Parameters and arguments: pass **primitive** or **reference** data types by *value* (only within method scope).
        1. **Parameters**: variables that go between ()s in method signature.
           1. varargs: passed for arbitrary number of parameters.

**Java**: method(type… parameterName) {\*/…/\*}

* + - 1. **Arguments**: variables that go between ()s in method call.
      2. Format
         1. **Format specifiers**: embedded in static text of a format string.

d: integer.

f: floating point.

n: line terminator (platform-specific).

x: integer as HEX.

s: any value as a string.

tB: integer as locale-specific month name.

Many conversions

* + - * 1. **Precision**

**Width**: minimum width of formatted value; left-padded with blanks.

**Flags**: additional formatting options.

+: always have + sign.

0: 0 is padding character.

-: pad on right.

,: format number with locale-specific thousands separators.

* + - * 1. **Argument Index**: specify argument; < to match same as previous.
  1. **Interface**: collection of related methods with empty bodies that tell objects what to do, but NOT how.
     1. **Java**: optionalModifiers interface Interface {\*/.../\*}
     2. Abilities
        1. Can be…
           1. implemented by many classes, along with other interfaces.

**Java**: public class Class implements Interface1, Interface2 {\*/.../\*}

* + - * 1. extended by other interfaces.

**Java**: public interface Interface2 extends Interface1, Interface2 {\*/.../\*}

* + - 1. CANNOT be instantiated.
    1. Purpose
       1. *Separate programmers or groups of programmers agree upon how their software will interact.*
       2. *Each programmer or group develops their specific model independent of the other programmer or group.*
       3. *No concern if the developed models will interact correctly since the programmers or groups defined ahead of time the interface.*
    2. Contains ONLY…
       1. *constants*
       2. *method signatures*
       3. *default methods*
       4. *static methods*
       5. *nested types*
    3. **Interface body**
       1. **Abstract methods**: followed by a ;, but no () and does not contain an implementation.
       2. **Default methods**: defined with the default modifier.
          1. *Adding new functionality to the existing interfaces*
          2. *Allows for binary compatibility with code written for older versions of those interfaces*
          3. *Specify the method definition in an interface is a default method with*default*at the beginning of the method signature*
          4. Extending

Do not reference the default method at all

Extended interface inherits default method

Re-declare the default method, making it abstract

Redefine the default method to override it

* + - 1. **Static methods**: use static.
      2. **Constant declarations**: implicitly public, static, and final.
    1. Notes
       1. When defining a reference variable with an interface as its type, any object assigned to it MUST be an instanceof a class that implements the interface.
       2. When implementing an interface, a class can be a type of their own class, superclass, and the interface.
  1. **Package**: namespace for organizing classes and interfaces in a logical manner, making them easier to manage.
     1. java.util
        1. ArrayList
        2. Random
  2. **Operators**: special symbols that perform operations on 2 or 3 **operands** and return a result.
     1. **Prefix and Postfix** (++x, x++, etc.)
     2. **Unary** (++x, –x, +x, –x, ~!): only have 1 operator.
     3. **Arithmetic**
        1. **Multiplicative** (\*, /, %)
        2. **Additive** (+, -)
     4. **Bitwise shift** (<<, >>, >>>)
     5. **Equality** (==, !=) and **Relational** (<, >, <=, >=, instanceof)
     6. **Arithmetic**
        1. **Multiplicative** (\*, /, %)
        2. **Additive** (+, -)
     7. **Bitwise operators** [L 🡪 R]
     8. **Bitwise operators** (&, ^, |) [L 🡪 R]
     9. **Logical**
        1. **Conditional** (&&, ||, !)
        2. **Ternary** (?:): shorthand for if-then-else statement, with 3 operands.
           1. if the result of the condition is true

result coded between ? and : is executed.

* + - * 1. else

result coded between : and ; is executed

* + 1. **Assignments** and **compound assignments** (=, +=, -=, \*=, /=, %=, etc…) [R 🡪 L]
    2. **instanceof**: determines if an object is…
       1. Instance of a class
       2. Instance of subclass
       3. Instance of a class that implements a particular interface

1. **I/O**
   1. Console
      1. System.out.print(): no line break.
      2. System.out.println(): line break.
      3. **Unbuffered Streams**: each read/write request is handled directly by underlying OS.
         1. **Byte streams**: performs I/O of 8-bit bytes.
         2. **Character stream I/O**: translates internal format to local character set (usually 8-bit ASCII or 16-bit chars).
         3. **Line-based I/O**: based on a **line**: string of characters with a **line terminator** (“\r\n”, “\r”, “\n”)
      4. **Buffered Streams**: read data from **buffer** (memory area).
         1. **Java**
            1. inputStream: new BufferedReader(new FileReader(“text.txt”));
            2. outputStream: new BufferedWriter (new FileWriter(“output.txt”));
         2. 4 Classes used to wrap unbuffered streams
            1. Buffered byte streams: BufferedInputStream and BufferedOutputStream
            2. Buffered character streams: BufferedReader and BufferedWriter
         3. **Flushing**: write out a buffer at critical points, without waiting for it to fill.
            1. autoflush: specified by optional constructor argument; supported by some buffered output classes.
            2. flush: manual method of flushing.
         4. **Scanner**: object type useful for breaking down formatted input into tokens and translating tokens according to their data type.
            1. **Java**

Scanner s = null;

try {

s = new Scanner(new BufferedReader(new FileReader(“Scan.txt”)));

while (s.hasNext()){

System.out.println(s.next());

}

} finally {

if (s != null) {

s.close();

}

}

* + - * 1. Default: uses white space to separate tokens.
        2. useDelimiter(): used to specify a different token.
  1. File
     1. **Try-catch block**: allows handling of exceptions in file I/O to prevent program crashing.

try {

in = new FileInputStream (“… .txt”);

out = new FileOutputStream (“… .txt”);

int c;

while ((c = in.read()) != -1) {

out.write(c);

}

} catch (Exception ex) {

System.out.printIn(“Error opening files”);

} finally {

/\*Something\*/

}

* + 1. JFileChooser: displays dialog to allow user to select a file or directory.
    2. **Relative pathing**: ... used within string name to manipulate files in directories, without knowing the absolute directory (from the root).
       1. **Java**: new String(“.../data/BoggleData.txt”);
    3. **Compiler error**: file cannot be opened when no try-catch is included.

1. **Control Flow**
   1. **Decision-Making**
      1. if
      2. if-else
      3. if-else-if-else (*leg* else *not required*)
      4. switch: **switchblock** (primitive data types/classes, enumerated types, String)
         1. String.equals
         2. toLowerCase()
   2. **Looping**
      1. for: executes *n* times.
      2. Enhanced for
         1. **Java**: for (dataType element : array)
      3. while: continually executes a block while a Boolean expression is true; executes 0 to *n* times.
      4. do while: executes 1 to *n* times
   3. **Branching statements**
      1. break
         1. Unlabeled: commonly used in switch or to terminate nested loops.
         2. Labeled: terminate outer statement as well.
      2. continue
         1. Unlabeled: skip current iteration of innermost loop.
         2. Labeled: skip iteration of outer loop.
      3. return: exit from current method and return control flow to where method was invoked.
         1. returns a value
         2. does not return a value
2. **Expressions, Statements, & Blocks**
   1. **Expression**: comprises variables, operators, and method invocations.
   2. **Expression statement**: complete unit of execution and terminating ;.
      1. Assignment: aValue = 8933.234;
      2. Increment: aValue++;
      3. Method invocation: System.out.println("Hello World!");
      4. Object creation: Bicycle myBike = new Bicycle();
      5. Declaration: double aValue = 8933.234;
   3. **Block**: ≥0 statements between {} used wherever a single statement is allowed.
3. **Random Number Generation**
   1. **Math.random()**: function used for random numbers.
   2. **Random**: class used as random number generator.
      1. **Java**
         1. Random randomNumbers = new Random();
         2. void setSeed(long seed): sets the seed of this random number generator using a single long seed.
            1. Repeats same sequence of pseudorandoms for debugging.
      2. Methods: return next pseudorandom, uniformly-distributed element from the Random’s sequence using current time-of-day.
         1. nextInt(int optionalMax)
            1. Range

Default: [–231, 231 – 1]

Adjusted: [shifting value, optionalMax]

optionalMax: scaling factor (how many are in range), as well as last value in range if shifting value = 0.

shifting value: from an expression outside of method to specify first number in range.

multiplier: multiply the output of the method to increase the distances between values.

* + - 1. protected int next(int bits): generates next number.
      2. boolean nextBoolean()
      3. floats and doubles
         1. Range: [0, 1]
         2. double nextDouble()
         3. float nextFloat()
         4. double nextGaussian(): returns next Gaussian-distributed double.

*µ* = 0

*σ* = 1

* + - 1. void nextBytes(byte[] bytes): generates random bytes and places them into a user-supplied bytes array.
      2. long nextLong()