Identifiers

- Identifiers must start with a letter, a currency character ($), or a connecting character such as the underscore (_). Identifiers cannot start with a number!
- After the first character, identifiers can contain any combination of letters, currency characters, connecting characters, or numbers.
- In practice, there is no limit to the number of characters an identifier can contain.
- You can't use a Java keyword as an identifier. Table 1-1 lists all of the Java keywords including one new one for 5.0, enum.
- Identifiers in Java are case-sensitive. foo and FOO are two different identifiers.

Class Access Modifiers

- ***Public***
  - Can be seen by any class in any package.
- ***Default (no modifier)***
  - Can only be seen by classes in the same package.

Class Non-Access Modifiers

- ***Abstract***
  - Any class with even one abstract method must be marked abstract itself
  - All abstract methods in an abstract class must be marked public and abstract.
  - All methods in an interface are implicitly public and abstract
  - An abstract class is allowed to have no abstract methods.
  - The first concrete class to extend an abstract class MUST implement all abstract methods.
- ***Interface***
  - All interfaces are implicitly abstract, but not implicitly public.
  - All variables defined in an interface are implicitly public, static, and final. In other words, interfaces can declare only constants, not instance variables.
  - Interface methods must not be static.
  - Because interface methods are abstract, they cannot be marked final, strictfp, or native.
  - An interface can extend one or more other interfaces.
  - An interface cannot extend anything but another interface
  - An interface cannot implement another interface or class.
  - An interface must be declared with the keyword interface.
  - Interface types can be used polymorphically.
- ***Strictfp***
  - Any method in this class will conform to IEEE 754 standards regarding floating point values.
- ***Final***
  - These classes cannot be extended.

Member Access Modifiers

- ***Public***

- Can be accessed by any class in any package.

- ***Private***
  - Can only be accessed by code inside the same class. Cannot be inherited. Despite the fact that a subclass can define a method by the same name, it's not inherited; only coincidentally named the same.

- ***Default (no modifier)***
  - Can only be accessed by classes and subclasses in the same package.
- ***Protected***
  - Can be accessed by classes and subclasses in the same package. Can additionally be accessed (through inheritance only, not the dot operator) by a subclass **even if it's in a different package**. Also, once inherited, becomes private to all other classes except subclasses in the same package.
  - Access modifiers can never be applied to local variables

Member Non-Access Modifiers

- ***Final***
  - Prevents a method from being overridden.
  - Final arguments are the same as final local variables. Basically it makes the variable(or argument) a constant.
  - Final reference variables can never be assigned to a different object, however the object it references can continue to be modified.
- ***Abstract***
  - Abstract methods contain no body and end with a semi-colon instead of curly braces. They are to be implemented by a subclass that inherits them. Abstract methods can never be private, final, or static.
- ***Synchronized***
  - Synchronized methods can only be accessed by one thread at a time. Cannot be applied to classes or variables… only methods. Can be combined with any of the 3 access modifiers or be default.
- ***Native***
  - Native methods are implemented in platform dependant code (usually C). Can only be applied to methods. They end in a semi-colon just like an abstract method.
- ***Strictfp***
  - These methods must conform to the IEEE 754 standard regarding floating point values.
- ***Var-args***
  - Indicates an argument of varying length.
  - Must indicate the argument type followed by an ellipsis, then a space and the name of the array to hold the parameters received. Example : void doStuff(int… x){}.
  - Other parameters can be included in the method, but the var-arg must be the last parameter.
  - You may only use one var-arg per method.
- ***Transient***

    - Will be skipped when attempting to serialize the object that contains it.
    - Can only apply to instance variables.
- ***Volatile***
    - Can only be applied to instance variables.
- ***Static***
    - Exist independently of any instantiated objects in the class. In other words, all instances of a given class share the same value for any static variable.
    - Can be assigned to methods, variables, class nested within another class, & initialization blocks.

Constructor Declarations

- Every class has a constructor. If you don't declare one, the compiler will build one for you.
- Never, **EVER** returns a type.
- Can use all three access modifiers.
- Can take arguments, including var-args.
- Must have the same name as the class.
- Can't be static, final, or abstract.

Variable Declaration

- ***Primitives***
    - byte, char, short, int, long, float, double, boolean
- ***Reference***
    - refer to objects of a type or subtype.
- ***Instance***
    - can be any of the four access levels
    - can be marked final or transient.
    - cannot be abstract, synchronized, strictfp, native, or static
- ***Local***
    - Local variables are variables declared within a method.
    - Is destroyed when the method exits.
    - Can only be modified with final, which makes them constants.
    - Must be initialized before use.
- ***Enums***
    - Restricted to having one of only a few pre-defined values.
    - Can never invoke an Enum constructor directly.
    - Example: enum CoffeeSize{BIG, HUGE, OVERWHELMING}; CoffeeSize cs = CoffeeSize.BIG;
    - The values in the list are all constants.
    - Can have constructors, instance variables, methods, and a constant specific class body.
        - Constructor
            - semicolon required after last element in enum declaration | example: enum CoffeeSize{BIG(5), MEDIUM(10), LARGE(15); ....}
            - each element can have more than one initializing value with multiple overloaded constructors just like a reg instantiation
                - example: enum CoffeeSize{BIG(5), MEDIUM("Med Sized", 10);

        ....}
        - Method
            - just like any other method in a class
        - Constant Specific Body Class
            - each constant can have it's own specific class
            - these classes can define methods that override the methods defined in the actual enum
    - Can be declared both as a class or inside a class as a member, but never inside a method.
        - When declared inside another class, must be accessed through it's enclosing class's class name
        - example: myCoffee = Coffee.CoffeeSize.BIG;
    - semi colon is always optional for the main braces
    - semi colon is only required for the ending brace of a a constant specific class body if there's code following the closing brace.
    - every enum has a static method values() that returns an Array of the values in the enum. | example: Order.Combos[] = Order.Combos.values();

Inheritance

- Used to promote code reuse
- ***Used for polymorphism***
    - Treat any object from a subclass as if it's an object from a superclass of that subclass
    - A reference variable can be of only one type than can never change. But the object it references can change.
    - Unless marked final, a reference variable can be reassigned to other objects.
    - **The reference variable's type determines the methods than can be invoked on the object while that variable is referencing.**
    - **A reference variable can refer to any object of the same type as the declared variable or any subtype of the declared type.**
    - A reference variable can be declared as a class type or an interface type. If it's declared as an interface type, it can reference any object of any class that implements that interface.
    - If the reference points to an object that is a subtype of the reference type, the JVM (java virtual machine) will look at run-time to determine if the subclass of the reference has overridden an instance method of the reference class. If it has, it will invoke the subclass' method instead of the reference type.
- ***Overriding methods***
    - Argument lists must be identical, otherwise it's overloading
    - Return type must be the same or a subtype of the original method
    - Access level can't be more restrictive
    - Instance methods can only be overridden if they are inherited by the subclass
    - Overriding method can throw any unchecked (runtime) exception, regardless of the original method's exceptions.
    - Overriding method can NOT throw any new or broader exceptions
    - You cannot override a method marked final

    - You cannot override a method marked static
- ***Overloading methods***
    - Must change the argument list
    - Can change return type
    - Can change access modifier
    - Can declare new or broader exceptions
    - Methods can be overloaded in the same class or in different classes through inheritance.
    - When passing a reference to an object to an overloaded method, the method is chosen by the compiler based on the reference. **UNLESS** the method is overloaded in the object's subclass of the superclass reference variable… in which case polymorphism works.
    - Primitive widening uses the "smallest" method argument possible.
    - ***Boxing***
        - The compiler will always choose to widen a parameter before choosing to box it.
        - You CANNOT widen from one wrapper type to another. (IS-A fails.)
        - You CANNOT widen and then box. (An int can't become a Long.)*instanceof*
        - You can box and then widen. (An int can become an Object, via Integer.)
        - You can combine var-args with either widening or boxing.

Reference Variable Casting

- ***Down cast***

    When you cast from a higher super class reference DOWN to a more specific class. Example: Animal animal = new Dog(); Dog d = (Dog) animal;
    Compiler cannot catch improper downcasts, and they will result in runtime errors.
    Example: Animal animal = new Animal(); Dog d = (Dog) animal.
- ***Up cast***

    When you cast from a lower subclass reference UP to a less specific superclass.
    Example: Dog dog = new Dog(); Animal a = dog;
    Upcasting is done implicitly and therefore doesn't require and explicit cast.

Constructors

- Can use any access modifier
- Must match the name of the class
- Must have no return type
- It's legal (but stupid) to have a method with the same name as the constructor
- If you don't have a constructor, the compiler will create a default, no-arg constructor for you
- First call to every constructor is this() or super()
- If you create a constructor but omit super() or this(), the compiler will create a no-arg call to super () for you
- You cannot make a call to an instance method or access any instance variables until the super() constructor runs
- Only static variables and methods can be accessed as part of the super() or this() call
- Abstract classes have constructors that are called when a concrete class instantiates them

- Interfaces have no constructor, b/c they are not part of the Object inheritance tree.
- Constructors can only be invoked by other constructors.
- If the super constructor has arguments, you must supply a proper super() in your sub constructor or compile will fail.

Statics

- Static methods and variables can be accessed directly through the class without instantiating an object.
- They can also be accessed through any object.
- Static methods can be redefined in a subclass **BUT NOT OVERRIDDEN!**

Coupling and Cohesion

- ***Coupling***
  - The degree to which one class knows about another. One class knowing about another class only through it's public API is loose coupling.
  - One class knowing about another class' variables and methods that should be private is tight coupling.
  - Loose coupling is good, and tight coupling is bad.
- ***Cohesion***
  - The degree to which a class has a single, well focused purpose.
  - High cohesion is good.

Stack and Heap

- Instance variables and objects live on the heap.
- Local variables live on the stack.

Primitive Casting

- The result of an expression involving anything int-sized or smaller always equals an int.
- Smaller to bigger casting (byte to int) is implicit.
- Larger to smaller casting (int to byte) is explicit. Example: Float f = 3.4; int i = (int) f;
- Casting a float as an int loses all digits after the decimal. Example: int i = (int)3.45; In this example, i will = simply 3
- Casting to smaller means you truncate all the bits from the larger value out of the smaller bits that don't fit
  - if you are left with a "sign" bit, you must perform 2's compliment conversion by flipping all bits, adding 1 to the result, and then attach negative sign.
- Compound assignment operators (+=, -=, *=, /=) contain an implicit cast.

Variable Scope

- Static
  - Longest scope.
  - Created when the class is loaded in the JVM
- Instance
  - 2nd Longest scope
  - Created when a new instance is created.
  - Live until instance is removed.
- Local

    - 3rd Longest scope
    - Live as long as their method remains on the stack
- Block
    - 4th Longest scope
    - Live as long as the code block is executing.
- Local method variables still live, but are out of scope in a nested method. Example: int go(){int x = 0 go2(){x++;}}

Passing Variables Into Methods

- Variables pass by copy
- Primitives will not preserve changes b/c the copied data is the literal.
- References will preserve changes b/c the copied data is the address of the object.

Arrays

- Arrays are objects.
- State the type, followed by [] on either side of a following identifier. Example: int []key or int key[].
- Multi-dimensional arrays follow the same syntax. Example: int [][]key or int key[][].
- Illegal to include a size in the declaration. Must be put in the instantiation.
- Multi-dimensional arrays only require the first dimension to be sized when instantiated. Example: int[][] myIntArray = new int[3][];
- Primitive array elements can be any primitive that implicitly converts. Example: myInt[0] can be a short, char, or byte
- Reference array elements can be any reference that is a subtype. Example: myAnimals[0] can be any subtype of Animal.
- The actual variable to the array has different set of rules than the elements. You cannot substitute an implicitly converted primitive here. (a short array can't be referenced by an integer array reference)
- You CAN reference a subtype of an object array with a super type though. (a dog array can be referenced by an animal reference)
- You can declare, initialize, and construct an array all on one line. Example: int[] myInts = {1,2,3};
- Syntax for declaring, initializing, and constructing a multi-dimensional array. Example: int[][] myIntArrays = { {1,2,3}, {4,5}, {6,7,8}};
- Remember when initializing and constructing on the same line, do not include the size.

Order of Execution

- ***Methods***
    - When passing Object parameters into an overloaded method:
        - Exact signature runs first.
        - Next up the inheritance is next, and so on.
        - Var Args are always picked last.
    - When passing primitive parameters into an overloaded method:
        - Exact signature runs frist
        - Widening happens next if possible.
        - Boxing happens next if possible

        - Var-Args run last.
- ***Classes***
    - When a class loads, the order is:
        - Static initialization blocks
        - Main method ()
        - Super constructor
        - Instance initialization blocks
        - Constructor

Wrapper Classes

- ***Boolean***
- ***Byte***
- ***Character***
- ***Double***
- ***Float***
- ***Integer***
- ***Short***
- ***Long***
- Wrapper objects are immutable
- All wrappers except for Character have 2 constructors. One that takes a primitive of the wrapper, and one that takes a string representation of the primitive. They also throw a NumberFormatException.
- ***valueOf()***
    - Static Method.
    - One takes a String as the only arg. Example: Integer i = Integer.valueOf("10");
    - One takes a String and a radix (base) arg. Integer i = Integer.valueOf("10", 2);
    - One takes a primitive as the only arg. Example: Integer I = Integer.valueOf(10);
    - Returns a new Wrapper of the type that invoked the method.
- ***parseXxx()***
    - Static Method.
    - One takes a String as the only arg. Example: int i = Integer.parseInt("24");
    - One takes a String and a radix (base) arg. int i = Integer.parseInt("24", 2);
    - Returns a primitive of the passed value.
- ***xxxValue()***
    - Instance method.
    - Takes no args. Example: byte b = myByteWrapper.byteValue();
    - Converts the wrapped value of a numeric to its primitive.
    - When you convert from a wider type to a narrower type (float to short), the result is truncated instead of rounded.
- ***toString()***
    - Three versions in the wrapper classes
        - All wrapper classes
            - No-arg, non-static that returns a string with the value of the primitive that's wrapped.
            - Example: String str = wrappedInt.toString();

        - All numeric classes
            - Additional, static that takes a primitive arg of the appropriate type and returns it as a string.
            - Example: String str = Double.toString(3.441);
        - Integer & Long
            - Additional, static that takes a primitive and radix args and returns the computed value as a string.
            - Example: String str = Long.toString(10001, 2);

Boxing, ==, and equals()

- == checks to see if 2 objects have the same memory address (ie. Are the same object)
- equals() checks to see if 2 objects are "meaningfully equal" (ie Are the ***same type*** and contain the same primitive.
- There are a few exceptions where == will consider 2 different objects the same even when they are 2 separate objects.
    - Boolean
    - Byte
    - Character (from /u0000 to /u00f7)
    - Short and Integer (from -128 to 127)
- When == is used to compare a primitive and a wrapper, the wrapper will be unwrapped and then compared primitive to primitive.

Garbage Collection

- Garbage Collection is only concerned with the heap.
- When the garbage collector runs, its purpose is to find and delete objects that cannot be reached.
- The JVM decides when to run the GC.
- You can ask the JVM to run the GC, but there are no guarantees.
- An object is eligible for GC when no live thread can access it.
- When counting the number of object created, don't forget to include any addition object that get created as member variables in a class, for example a member array, or member variable that's a wrapper, as wrappers are objects.

Switch Statements

- break statements are optional
- expression being switched must be either an enum or implicitly castable to an int (byte,char,short)
- if the expression is to be implicitly cast to an int (byte,char,short,) you can't use a case that's larger than the actual expression's bit type. Example: byte b = 20; switch(b){ case 128:} is illegal b/c a byte can never contain a value of 128. This will cause a possible loss of precision error.
- the case must be a COMPILE-TIME constant. Example: final int a = 0; You must initialize and declare at the same time for the constant to be COMPILE TIME.
- you can't have 2 cases with the same value or you will get a compile time error.
- boxing is legal. you can switch a Wrapped int.
- you can use curly's for code blocks for each case section.

For Loops

- the third expression in the loop can be one or more expressions that aren't necessarily iterations.
- none of the expression in the loop have to be filled in.
- all blank expressions results in an infinite loop.
- unlabeled break exits the entire loop; labeled break exits the labeled loop.
- unlabeled continue exits only the current iteration; labeled continue skips the rest of the labeled loop's current iteration, but continues with the next one.
- the enhanced for loop must initialize it's first variable inside the loop.
- infinite loops do NOT throw stack overflow exceptions if nothing is added to the stack!

Labeled Statements

- most commonly labeled are loop statements in conjunction with break and continue.
- used in situations where you have a nested loop and want to specify from which loop to break or to which loop you want to continue.
- must be placed just before the statement being labeled.
- syntax: [valid identifier][:] Example: myLabel:

Exceptions

- 2 Types
  - Run-time - do not have to be declared.
    - Null Pointer Exception
    - ArrayIndexOutOfBoundsException
    - ClassCastException
    - IllegalArgumentException
    - IllegalStateException
    - NumberFormatException
  - Checked- must be declared.
- your catch clauses must be in order of increasingly superclass, or your code will not compile.
- if you rethrow an exception from a catch clause, you must also declare the exception you throw.
- StackOverflowError is an ERROR; not an exception.
- you can both handle and declare an exception in the same method.
- ***Finally***
  - Optional
  - Runs if exception thrown, not thrown, caught, or not caught.

Assertions

- 2 types
  - Very Simple
    - makes the assertion and then throws the AssertionError. Example: assert(y<x);
  - Simple
    - makes the assertion and then throws the Assertion
    - also passes a value to the stack trace to provide a string that prints further info about the error. Example: assert(y<x) : "x is " + y + " y is " + y;

- use -ea when running to turn on assertions
- use -da when running to disable assertions
- -ea and -da can use 3 diff types of args
  - ☠□ args - works on all classes
  - Package name - works on package specified and all other sub packages.
  - Class name - works on a specified class.
- Don't use assertions to
  - to validate arguments to a public method. (unless you know something is never ever supposed to happen, like reaching the default of a switch statement)
  - to validate command-line args
  - that can cause side effects.

Strings, StringBuffers, & StringBuilders

- ***StringBuilder***
  - Important Methods
    - append() - updates the value of the object by adding the value passed into it.
    - delete() - a substring is removed based off the arg(s) passed to it. (arg1 is zero-based, arg2 if passed is 1 based)
    - insert() - a string (passed into 2nd arg) is inserted at a zero based index (passed into first arg).
    - reverse() - the characters in the calling string are reversed.
    - toString() - returns the value of the string
  - Same as StringBuffer, except it's faster, b/c it's not synchronized.
  - New to Java 6
  - not immutable
  - equals() is NOT overridden. to do meaningfully equal must convert toString(). Example: sb1.toString().equals(sb2.toString());
- ***StringBuffer***
  - Same as StringBuilder, except it's slower, b/c it IS synchronized.
  - (see StringBuilder for the additional notes, as synchronization is the only difference between the two)
- ***String***
  - Important Methods
    - charAt() - returns the character located at the specific index
    - concat() - appends one string to the end of another AND RETURNS THE NEW STRING.
    - equalsIgnoreCase() - determines the equality of two strings, ignoring case
    - length() - returns the number of characters in a string
    - replace() - replace occurrences of a character with a new character AND RETURNS THE NEW STRING
    - substring() - returns a part of a string
    - toString() - returns the value of a string
    - toLowerCase() - returns a string with uppercase characters converted
    - toUpperCase() - returns a string with lowercase characters converted
    - trim() - removes whitespace from the ends of a string AND RETURNS THE

NEW STRING

- Immutable.
- Each character in a String is a 16bit character.
- Most of the utility methods return a new String, so you must assign it to a reference, or you simply create the new string, but it's lost.
- the shortcut method (String s = "jason";) only makes 1 object, the new string object "jason".
- the standard method String s = new String("jason"); makes 2 objects, "jason" and the new string object.

File Navigation & I/O

- Must import java.io.*;
- ***Important classes***
    - File - an abstract representation of file and directory pathnames
        - Not used to read or write data
        - make new empty files
        - search for files
        - delete files
        - make directories
        - work with paths
        - constructor 1 takes a String to look for a file name and create a file object that either refers to nothing or to the file that exists with the String as the name, BUT NOT THE ACTUAL FILE!
        - constructor 2 takes a directory(file) and a string to create a new file name in the subdirectory provided, BUT AGAIN, NOT THE ACTUAL FILE!
    - FileReader - low level utility methods for reading character files
        - usually wrapped in a higher object such as BufferedReader
        - read single characters
        - read whole streams of characters
        - read fixed number of characters
        - constructor takes a File object.
    - BufferedReader - make low level classes like FileReader more efficient and easier to use
        - reads large chunks of characters at a time into a buffer
        - provides some convenient utility methods for easier character reading
    - FileWriter - write to character files
        - usually wrapped in a higher object such as BufferedWriter or PrintWriter
        - white characters or strings
        - constructor takes a File object & creates the actual file.
    - BufferedWriter - make low level classes like FileWriter more efficien and easier to use
        - writes large chunks to a file at once
        - provides some convenient utility methods for easier character writing
    - PrintWriter - wrapper class with new constructors
        - can now use in place of a lower level writer wrapped in a wrapper such as a

        - BufferedWriter
    - Console - new to Java 6
        - read input from the console
        - write formatted output to the console
- ***Important methods***
    - File
        - createNewFile()- **throws IOException**
            - creates new File object with the file name provided to the constructor of the calling File object, unless a file of that name already exists, in which case it just assigns the reference to the existing file.
            - if a directory is provided, the above happens in the directory provided.
            - returns a boolean value indicating a successful or unsuccessful file creation.
        - exists() - returns boolean value representing if the ***file object or directory object*** exists;
        - delete() - deletes files and directories. You can't delete a directory if it's not empty.
        - isDirectory() - returns boolean value representing if the File object is an existing directory.
        - isFile() - returns boolean value representing if the File object is an existing file.
        - list() - returns a String[] of the names of the files and directories in the string passed into it.
        - mkdir() - creates a directory from a File object.
        - renameTo() renames files and directories. You can rename a non-empty directory. You must pass a new File object with the name you want to the current object to rename it.
    - FileWriter
        - write() - writes a string of characters to the file passed to the FileWriter object.
        - flush() - flushes the string fully
        - close() - closes the file
    - BufferedWriter
        - write() - same as FileWriter
        - flush() - same as FileWriter
        - close() - same as FileWriter
        - newLine() - writes a new line character to the line (effectively adding a carriage return)
    - PrintWriter
        - write() - same as FileWriter
        - flush() - same as FileWriter
        - close() - same as FileWriter
        - print() - writes to the file.
        - println() - writes a new line (including the newline character) to the file.
        - format()
        - printf()

    - FileReader
        - read() - reads the contents of the File object passed to it at creation into a char[] passed to the actual read() method
        - close() - closes the file
    - BufferedReader
        - read() - same as FileReader
        - readLine() - reads the contents of the File object passed into it at creation into a String. returns a null when there is no more data to return.
        - close() - closes the file
    - Console
        - construction is Console myConsole = System.console(); NOT = new Console();
        - readLine() - returns a String containing whatever the user keyed in. takes an optional String arg to print to console as prompt. Example: String str = readLine("Input: ");
        - readPassword() - returns a char[] NOT a String. Takes 2 optional args. Arg 1 is format String. Arg 2 is String to print to console as prompt.

Serialization

- Saves an object's state to a file so that it can be rebuilt later.
- Not for static variables.
- The class of the object being serialized MUST implement the Serializable interface
- Any states inherited from a superclass that isn't Serializable will be reset to their initial values b/c the constructor for the superclass will run.
- Every element in an array or collection must be Seralizable or the serialization will fail.
- Classes throw checked exceptions.
- ***Classes & Methods***
    - FileOutputStream
        - takes a String file name as a param for storing the object to be serialized
        - is wrapped by ObjectOutputStream
    - ObjectOutputStream
        - wraps FileOutputStream
        - writeObject()
            - takes an object as a param
            - writes the state of the object to the file passed into it's wrapped FileOutputStream object.
    - FileInputStream
        - takes a String file name as a param for reading the stored object to be rebuilt
        - is wrapped by ObjectInputStream
    - ObjectInputStream
        - wraps FileInputStream
        - readObject()
            - takes an object as a param and reads it
            - returns it as an Object which MUST be explicitly cast to the proper type. Example: Dog d = (Dog) oos.readObject();

- ***Transient***
    - All the objects that are part of the object must implement Serializable
    - Any that do not implement Serializable, must be marked as transient & will not be serialized.
    - upon rebuild, those marked transient must be manually rebuilt using 2 methods (must call both in same order to work)
        - **private void writeObject(ObjectOutputStream os){} (throws IOException)**
            - *os.defaultWriteObject()* must be called to do initial write
            - *os.writeXxx(reference.getterMethod())* must be called to do manual write
        - **private void readObject(ObjectInputStream is){} (throws IOException)**
            - *is.defaultReadObject()* must be called to do initial read
            - *reference = new Object(is.readXxx())* must be called to do manual read

Dates, Numbers, and Currency

- Java.util package
    - Date
        - mostly deprecated
        - instance represents a mutable date and time to the number of miliseconds past from 1/1/1970 to present (stored in long)
        - toString() returns a String containing current date and time
    - Calendar
        - large variety of methods to convert and manipulate dates and times
        - instantiate like so: Calendar myCalendar = Calendar.getInstance(); NOT with new().
        - getTime() returns the Date stored within
        - setTime(Date) stores a Date within
        - add() adds or subtracts units of time for whatever field you specify. Example: myCal.add(Calendar.MONTH, 2); adds 2 months to the date.
        - roll() same as add, but won't increment or decrement the larger parts. Example: if the date is 12/31/10, myCal.roll(Calendar.DAY, 1); yields 12/1/10 instead of 1/1/11.
    - Locale
        - used with DateFormat & NumberFormat
        - format dates, numbers, and currency for specific locales
        - constructor takes either a language, or a language and country
        - getDisplayCountry()
            - no arg returns the way to say the name of the country in the local locale (english for me obviously)
            - pass it a Locale and it returns how to say the name of the country in the local you pass it. (if there is no country arg in the Local passed, there will be empty but not null)
        - getDisplayLanguage()
            - same as getDisplayCountry() but with the language spoken in the locale

- Java.text package
  - DateFormat
    - used to format dates
    - instantiate like so: DateFormat df = DateFormat.getInstance(); or DateFormat df = DateFormat.getDateInstance([style],[Locale]);
    - format() takes a date to format, and returns a String of the date in the format set into the DateFormat object.
    - parse() reverses the process by taking a String in the form of a DateFormat, and parses it into a Date object.
      - throws ParseException (must handle)
  - NumberFormat
    - format numbers and currencies
    - instantiate like so: NumberFormat nf = NumberFormat.getInstance([Locale]) or NumberFormat nf = NumberFormat.getNumberInstance([Locale]) or NumberFormat nf = NumberFormat.getCurrencyFormat([Locale]);
    - format() just like DateFormat,
      - if there are more than the default number of fractional digits in a float or double, format() will round, not truncate.
    - getMaximumFractionDigits() returns the number of fraction digits used by default by format() **DOESNT WORK FOR PARSE()**
    - setMaximumFractionDigitis() sets the number **DOESNT WORK FOR PARSE()**
    - parse() returns a String composed of the numbers passed to it in a String.
      - throws ParseException (must handle)
      - setParseIntegerOnly() takes a boolean & sets or doesn't set the NumberFormat object to parse just the integers in a float or double

Parsing, Tokenizing, & Formatting.

- ***Regular Expressions***
  - Pattern - holds a representation of the regex expression to be used by instances of Matcher
    - instantiate like so: Pattern p = Pattern.compile(regex String here);
  - Matcher - invokes regex engine to preforming matching operations
    - instantiate like so: Matcher m = p.matcher(String to search through here);
    - **find()** - searches for expression and returns boolean result.
    - **start()** - returns the index for the found expression
    - **group()** - returns the found expression as a String
  - Metacharacters
    - \d - digits
    - \s - whitespace characters
    - \w - a word character (letters, digits, or _)
    - . - any character
    - [] - specific characters
    - [a-f] - specific range of characters
    - [a-fA-F] specific ranges of characters

        - + - one or more (greedy)
        - * - zero or more (greedy)
        - ? - zero or one (greedy)
    - Scanner - can do more advanced token extraction and search than the Pattern/Match classes
        - Constructors
            - new Scanner(String source)
            - new Scanner(File source)
            - new Scanner(Stream source)
        - **findInLine(String pattern)** - returns a String with the found pattern.
        - **hasNext()** - checks to see if the String passed into the Scanner has a String in the next spot and returns a boolean
        - **hasNextXxx()** - checks to see if the String passed into the Scanner has an integer in the next spot and returns a boolean
        - **next()** - returns the String in the next spot
        - **nextXxx()** - returns the Xxx in the next spot
        - **useDelimiter(regex)** - set the delimiter with a regex
- ***Tokenizing***
    - Separating tokens from the delimiters that separate them.
    - String.split() - takes regex as it's arg and returns a String[] with the tokens produced by the split
- ***Formatting***
    - ***printf(***format string, arg(s)***)*** - same as format
    - ***format(***format string, args(s)***)*** - same as printf
    - ***format string -*** %[arg_index$][flags][[width][.precision]conversion char
        - arg_index$ - integer indicating which arg gets printed in this position
        - flags
            - "-" left justify
            - "+" include a sign (+or-)
            - "0" pad this arg with zeros
            - "," use locale specific grouping separators (ie comma 304,204)
            - "(" enclose negative numbers in parentheses
        - width - indicates max number of characters to print
        - .precision - num digits to print after the decimal point
        - conversion - the type of arg you're formatting
            - b boolean
            - c char
            - d integer
            - f floating point
            - s string

Overriding hashCode() and equals()

- ***toString() -*** all objects have it
    - override it - public String toString(){}
    - allows you to define what is printed when you send the object to the output stream.

- ***equals()* -** defaults to same evaluation as "==" and compares bit code (aka references for objects)
    - override it - public boolean equals(){}
    - if 2 objects are equal with equals(), they must have equal hashcodes.
- ***hashCode()* -** sort of an ID number, but not necessarily unique.
    - override it - public int hashCode(){}
    - if 2 objects are equal() they must return same hashcode() int result
    - if 2 objects are NOT equal() they can return non int results.

Collections

- Ordered **-** means the collection maintains some order with regard to when the object was added to it.
- Sorted **-** means the collection puts the objects added to it in some natural sorted order. (Spaces, UpperCase, Lowercase, Numbers)
    - Comparable() - defines how instances can be compared to one another for sorting.
    - Comparator() - defines how instances can be compared to one another for sorting.
    - in general, Collections can hold objects but not primitives. (primitives are autoboxed)
    - Collections.sort() - used to sort objects in a Collection. Does not work for any Object that doesn't implement the Comparable or Comparator interface, i.e. Object.
- ***List*** **-** *Extends Collection (List of things)*
    - *Ordered by index*
    - *Unsorted*
    - cares about the added object's index
    - objects can be added at a specific index or (by default) to the end of the List
    - *Key methods*
        - get(int index)
        - indexOf(Object o)
        - add(int index, Object o)
        - size()
        - hasNext() - returns true if the collection has at least one more element
        - next() - returns the next object in the collection AND moves forward.
        - iterator() - returns the address of the first element in the list.
        - toArray() returns an object array populated with the elements in the List.
        - toArray(Array a) returns an array of the type passed into it, populated with the items in the List.
            - Example: Integer[] ia = new Integer[3];

ia = myList.toArray(ia);

-
    - **ArrayList**
        - fastest random access for fast iteration through the elements
    - **Vector -** same as ArrayList, but synchronized.
    - **LinkedList**
        - objects are linked to one another allowing for extra methods for fast insertion and deletion

        - iterates slower than ArrayList
        - now implements the Queue interface
    - **Iterator**
        - Pre-Java 5 code (before enhanced loop) had to use iterators to move through a List.
            - hasNext() - returns if there's another item in the next slot (doesn't move the index)
            - next() - returns the item in the next slot (moves the index)
            - Example: Iterator<Dog> i3 = myDogList.iterator();

```
While(i3.hasNext()){

        Dog aDog = i3.next();

        System.out.println(aDog.getName());

}
```

- ***Set***- *Extends Collection (Unique things)*
    - doesn't allow duplicates
    - **HashSet -** unsorted & unordered
        - uses hashCode for retrieval
        - use when need no duplicates and don't need to iterate.
    - **LinkedHashSet** - ordered version of HashSet
        - all the elements are linked
        - use when you need no duplicates but want to iterate over the elements. (in the order they were inserted)
    - ***SortedSet -*** *Extends Set*
        - ***NavigableSet -*** *Extends SortedSet*
            - **TreeSet -** ordered and sorted
                - objects added MUST BE MUTUALLY COMPARABLE!
                - sorts elements in ascending order (natural)
                - can be altered for custom order using comparable() or comparator()
                - implements NavigableSet()
                    - Java5 or earlier required you to create a new TreeSet, and set it equal to a returned collection via methods.
                    - Java6 allows you to simply return elements using methods from the NavigableSet() Interface
                        - lower() - returns element less than the given element
                        - floor() - returns the element less than OR EQUAL TO the given element
                        - higher() - returns element greater than the given element
                        - ceiling() - returns element greater than OR EQUAL TO the given element

                - pollFirst() - returns (and removes from the Set) the first element in the Set
                - pollLast() - returns (and removes from the Set) the last element in the Set
                - descendingSet() - returns the Set in reversed order.

- ***<u>Map</u>*** **-** *Doesn't actually extend Collection (Things with a unique ID)*
    - Any class used as part of a key for a map must override hashcode() and equals()
    - allows you to search for a value based on it's ID key
    - allows you to ask for a collection of just the values or just the keys
    - Map<Object, Object> myMap = new HashMap<Object, Object>();
    - myMap.put([key], [value]);
    - myMap.get([key]);
    - **HashMap** - unsorted & unordered
        - uses hashCode for retrieval
        - use if you don't care about the order
        - allows mutiple null values and one null key
        - **LinkedHashMap -** *extends HashMap*
            - all the elements are linked
            - use when you want to iterate over the elements in the order they where inserted.
    - **Hashtable** - synchronized version of HashMap
        - cannot have ANYTHING null
    - ***SortedMap*** **- Extends Map**
        - ***NavigableMap*** **- Extends SortedMap**
            - **TreeMap** - ordered and sorted
                - sorts elements in ascending order (natural)
                - can be altered for custom order using comparable() or comparator()
                - implements NavigableMap()
                    - Java5 or earlier required you to create a new SortedMap and set it equal to a returned Sorted map via methods
                    - Java6 allows you to simply return elements using methods from the NavigableMap() Interface
                        - lowerKey() - returns element less than the given element
                        - floorKey() - returns the element less than OR EQUAL TO the given element
                        - higherKey() - returns element higher than the given element
                        - ceilingKey() - returns element greater than OR EQUAL TO the given element
                        - pollFirstEntry() - returns (and removes from the map) the first key/value pair in the Map
                        - pollLastEntry() - returns (and removes from the map) the last key/value pair in the Map

            - descendingMap() - returns the map in reverse order
    -
- ***<u>Queue</u>* - Extends Collection** *(Things arranged by the order in which they are to be processed)*
    - usually FIFO
    - support all the Collection methods
    - additional methods for adding and subtracting elements as well as reviewing queue elements
        - offer() - adds an element to the PriorityQueue
        - poll() - returns highest element and removes it from the PriorityQueue
        - peek() - returns highest element without removing it from the PriorityQueue
    - **PriorityQueue** - for queuing in a non-FIFO manner (priority-in/priority-out) or PIPO
        - priority sorting in the queue accomplished with a comparable or comparator
- ***<u>Utilities</u>***
    - Collections
        - binarySearch()
            - binarySearch(List, key) - search sorted List for key value and return index or insertion point where it would go
            - binarySearch(List, key, Comparator) - search Comparator-sorted list for key value and return index or insertion point where it would go
        - reverse(List) - reverse the order of the elements in a list.
        - reverseOrder()
            - reverseOrder() - returns a Comparator that reverses the sort order of the Collection
            - reverseOrder(Comparator) - returns a Comparator that reverses the sort order of the Collection's Comparator.
        - sort()
            - sort(List) - sorts in natural order
            - sort(List, Comparator) - sorts according to the SortObject's specifications
        - add()
            - add(element) - add element to the Collection: LIST, SET
            - add(element, index) add element at a specific index: LIST
        - contains()
            - contains(Object) - search Collection for Object, return result as boolean: LIST, SET
            - containsKey(Object key) - search Map for key, return result as boolean: MAP
            - containsValue(Object value) - search Map for value, return result as boolean: MAP
        - get()
            - get(index) - get an Object from a List: LIST
            - get(key) - get a key from a Map: MAP
        - indexOf - get index of Object in a list: LIST

        - iterator() - get an Iterator for a List or Set: LIST, SET
        - keySet() - return a Set containing a Map's keys: MAP
        - put(key, value) - add a key/value pair to Map: MAP
        - remove()
            - remove(index) - remove an element via the index: LIST
            - remove(Object) - remove an element via it's value: LIST, SET
            - remove(key) - remove an element via it's key: MAP
        - size() - return the number of elements in a Collection: LIST, MAP, SET
        - toArray()
            - toArray() - return an Object[] containing the elements of a Collection: LIST, SET
            - toArray(T[]) - return an array of type T containing the elements of a Collection: LIST, SET
    - Arrays
        - asList(myArray[]) - returns a List containting the elements of the array passed into it.
        - binarySearch()
            - binarySearch(Object[], key) - search sorted Object array for key value and return index or insertion point where it would go
            - binarySearch(primitive[], key) - search sorted primitive array for key value and return index or insertion point where it would go
            - binarySearch(Object[], key, Comparator) - search Comparator sorted array for key value.
        - equals()
            - equals(Object[], Object[]) - compare two Object arrays to determine if their contents are equal
            - equals(primitive[], primitive[]) - compare two primitive arrays to determine if their contents are equal
        - sort()
            - sort(Object[]) - sort the elements of an Object[] by their natural order
            - sort(primitive[]) - sort the elements of a primitive[] by their natural order
            - sort(Object[], Comparator) - sort the elements of an array using a Comparator
        - toString()
            - toString(Object[]) - create a string containing the contents of an Object[].
            - toString(primitive[]) - create a string containing the contents of a primitive[].
- ***<u>Comparable</u>***
    - used by Collections.sort() and Arrays.sort() to sort Lists and arrays of objects.
    - cannot use primitives.
        - for primitives, use the Wrapper classes (int = Integers, long = Long, etc)
    - class must implement Comparable interface
    - class must also override compareTo()

        - public int compareTo(thisObject o) {
          return
field.compareTo(anotherObject.getField());
          }
            - returns an integer indicating the result of the comparison of the two objects
                - -1 = thisObject < anotherObject
                - 0 = thisObject == anotherObject
                - +1 = thisObject > anotherObject
    - To reverse the order of the sort, swap the order of values in the implementation
        - public int compareTo(thisObject o) {
          return anotherObject.getField().compareTo(field);
          }
- ***Comparator***
    - allows you to sort a collection in multiple ways
    - can be used to sort instances of classes you can't modify
    - class must implement Comparator interface
    - class must also override compare();
        - public int compare(objectOne o1, objectTwo o2){
          return
o1.getField().compareTo(o2.getField());
          }
- ***Searching***
    - uses binarySearch()
    - successful searches return the int index of the element being searched (positive)
    - unsuccessful searches return the int index insertion point of where it would go (negative - 1)
    - collection/array must be sorted before you can search it, otherwise results are unpredictable
    - if sort was natural order, search must be natural order (no comparator)
    - if sort used comparator, search must use same comparator
- ***Converting Lists to Arrays and Arrays to Lists***
    - **Arrays to Lists**
        - Arrays.asList(myArray) - copies an array into a list
        - any changes to the List affects the array and visa-versa.
    - **Lists to Arrays**
        - toArray() - returns a new Object array
            - Object[] oa = myList.toArray()
        - toArray(someArray[]) returns an array into the "someArray[]" array passed into it.
            - Integer[] myIntArray = new Integer[3];
              myIntArray =
myList.toArray(myIntArray);
- ***Backed Collections***

- creates a copy of a portion of a NavigableMap or NavigableSet.
- both the copy and the original may be updated
    - if adding to the copy, the values must be in the original range assigned to the copy.
- **Set**
    - subSet(start, [b], end, [b]) - returns a SortedSet starting at start and ending at end
        - without boolean args, start is inclusive and end is exclusive. Pass boolean args to change them. (if you pass one boolean, you must pass both)
    - headSet(end, [b]) - returns a SortedSet ending at the "end" element
        - default boolean is false, and is exclusive.
    - tailSet(start, [b]) - returns a SortedSet starting with the "start element
        - default boolean is true, and is inclusive.
- **Map**
    - subMap(start, [b], end, [b]) - returns a SortedMap starting at start and ending at end
        - without boolean args, start is inclusive and end is exclusive. Pass boolean args to change them. (if you pass one boolean, you must pass both)
    - headMap(end, [b]) - returns a SortedMap ending at the "end" element
        - default boolean is false, and is exclusive
    - tailMap(start, [b]) - returns a SortedMap starting with the "start" element
        - default boolean is true, and is inclusive

Generics

- Used primarily for making "type safe" collections that can be checked for accuracy at compile time.
- The compiler will allow you to send a type safe Collection into a legacy, "non-type-safe" method.
    - If the method doesn't alter the Collection, there's no problem, and no warnings.
    - If it does alter the Collection, the compiler has to let the code compile so that legacy and J5+ can play together.
        - The compiler WILL, however issue a warning that you are performing unsafe, unchecked, operations.
- "Typing" info does NOT exist at runtime. It only exists at compile time.
- ***Polymorphism***
    - **Collections**
        - does NOT apply to the generic type
            - ArrayList<Animal> = new ArrayList<Dog>
        - DOES apply to the base type of Collections
            - List<Animal> aList = new ArrayList<Animal>();
        - You CANNOT send a type <Dog> ArrayList to a method expecting a type <Animal>List
            - example: ArrayList<Dog> dList = new ArrayList<Dog>();

```
addAnimal(dList);

public void addAnimal(List<Animal> a){

    a.add(new Dog());

}
```

- You can still add a Dog Object to the ArrayList<Animal> in the a method as long as the actual ArrayList passed in is of <Animal> type and not of a subtype, i.e <Dog>
    - example: `public void addAnimal(List<Animal> a){`
      ```
      a.add(new Dog())
      }
      ```
- **<u>EXCEPTIONS TO THESE RULES</u>**
    - There is a workaround for sending generic subtypes into a method.
        - You must use the wildcard symbol: "?" in the method signature.
            - example: `public void addAnimal(List<? extends Animal> a)`
            - doing so, however, prevents you from adding (using the add() method) anything to the List.
        - If you want to use the wildcard AND still add to the Collection in the method
            - you must use the keyword "super" in the method signature
            - you must also only send only the type in the method declaration or any of it's supertypes.
            - example: `public void addAnimal(List<? super Dog> a)`
                - this allows you to send any list with a generic type of Dog or a super type of Dog into the method and add to it.
        - If you want to declare a method that takes anything that implements a certain interface (Serializable for example)
            - you must say it EXTENDS the interface... not that it IMPLEMENTS the interface like you normally would
            - example: `public doIt(List<? extends Serializable> aList);`

- **Arrays**
    - DOES apply to the base type of arrays
        - `Animal[] aArray = new Dog[3];`
- **Creating a Generic Class**
    - add <T> (stands for type) to the class declaration
        - example: `public class RentalGeneric<T>{}`
    - add the Collection you want to use in the class as a member, making it a collection of the same type as the class, e.g <T>

        - example: private List<T> myList;
    - build your constructor to include the Collection you added
        - example: RentalGeneric(List<T> myList) {this.myList = myList;}
    - build your getter method for T
        - example: public T getRental() { return myList.get(0);}
    - build your custom add method for the Collection of type T
        - example: public void addRentalItem(T thingToAdd) { myList.add(thingToAdd);}
    - You may use more than one type in your generic class definition
        - example: public class RentalGeneric<T, X>{}
    - You may use a form of wildcard notation in your generic class definition, but you use a variable, e.g. T instead of the ?
        - example: public class RentalGeneric<T extends Animal>{}
- **Creating a Generic Method**
    - add <T> after the access modifier in the method
        - example: public <T> void makeArrayList(T t);
    - create your Collection to hold the type of object you passed into the method
        - example: List<T> theList = new ArrayList<T>();
    - now you can add to the list safely
        - example; theList.add(t);
    - You may use a form of wildcard notation in your generic method definition, but you use a variable, e.g. T instead of the ?
        - example: public <T extends Animal> void makeArrayList(T t);

Inner Classes

- 4 Types
    - **Regular**
        - can't have static declarations of any kind
        - from inside the inner class you can instantiate it like any other class: Inner myInner = new Inner();
        - from outside the outer class (or inside the outer class but in a static method) you can only instantiate it through an outer class object: Inner myInner = new Outer().new Inner();
        - to reference the outer class' instance: Outer.this;
        - same access modifiers as a member variable or method.
    - **Method-Local**
        - defined inside a method
        - method must instantiate an object of the inner class to use it.
        - can't use the local variables of the method it's in, unless the local variables are marked as final so that they don't fall off the stack after the method is complete.
        - same access modifiers as a local variable. Can only be abstract or or final.
        - if its declared in a static method it can only access the static members of it's outer class.
    - **Anonymous**

- declared without any name at all
- 3 types
  - *regular type 1* - creates an anonymous subclass of another class
    - example: class Food{

```
Popcorn p = new Popcorn(){
        public void pop(){
        System.out.println("This is the method to an anonymous sublcass of class Popcorn");
        }
    };    <---------------NOTE THE SEMI-COLON
}
```

    - can define non overloaded methods, but can't use them because the ref variable is of the super of the anonymous inner class and can't see the more specialized methods.
  - *regular type 2*- creates an anonymous implementer of an interface
    - example: class Food{

```
        Eatable e = new Eatable(){
            public void eat(){
System.out.println("This is the implemented eat() method to an interface named Eatable");
            }
        };    <---------------NOTE THE SEMI-COLON
}
```

  - *argument defined* - same as type1 or type2 except its created as an argument inside a method call.
    - example: class Food{

```
    Chef c = new Chef();
    c.cook(new Cookable(){
            public void cookIt(){}
    )};    <-------------NOTE THE SEMI-COLON AND
```

THE CLOSE PARENTHESIS

```
                    }


                        interface Cookable{
                            void cookIt();
                        }

                    class Chef{
                            void cook(Cookable c){
                                        System.out.println("Cooked
something");
                            }
                        }
```

- **Static**
    - not really an "inner" class, b/c it's really just a regular static class that's scope is nested within its outer class.
    - doesn't have access to instance variables and non-static methods b/c it's marked static
    - because it's marked static, doesn't have the implicit access relationship the other types of inner classes have.
        - can only be accessed from its outer class or an outer class object.
    - *Access*
        - **From the outer class**
            - Standard syntax: Inner i = new Inner();
        - **From outside the outer class**
            - Must go through enclosing class: Outer.Inner i = new Outer.Inner();

Threads

- Two definitions
    - *Instance of java.lang.Thread*
        - Just an object that lives and dies on the heap.
    - ***A thread of execution***
        - An individual, light-weight process with it's own stack.
        - Very little is guaranteed from one JVM to the next when it comes to threads.

        - The order of execution for multiple threads is never guaranteed!
- **Five States**
    - *new* - start() has not yet been run
    - *runnable* - start() has been run but run() has not (the thread is now considered alive)
    - *running()* - run() has started but hasn't completed
    - *waiting/blocked/sleeping* - taken out of "runnable" by the JVM, but it's still eligible to return to runnable state
        - waiting - thread's code tells it to wait for some event, when the event happens it sends notification that the thread no longer has to wait, thread returns to runnable state
        - blocked - waiting on input from some input stream for example, when the input arrives, thread returns to runnable state
        - sleeping - thread's run() tells it to sleep for some amount of time, when the time is up, thread returns to runnable state
    - *dead* - run() has been completed
    - The Thread object still exists and can run all the methods associated with it except for start() even after it is dead.
- **Instantiation**
    - Extend java.lang.Thread
        - seldom used unless you actually NEED to extend the class to add functionality.
        - you can overload run(), but the Thread object will only run the "run()" when you call start();
            - you CAN call run() method explicitly from the Thread object, but it just executes on the same thread, e.g. no new thread is created by explicitly calling run().
        - to use, simple create an instance of your class: MyThreadExtender mte = new MyThreadExtender();
    - Implement Runnable interface
        - most common since you're not usually ADDING functionality to the Thread class.
        - also leaves your class open to extend a class it really NEEDS to, should the case arise.
        - to use, create an instance of your class and pass it into the constructor for a new Thread object
            - example: MyRunnable r = new MyRunnable();

Thread t = new Thread(r);

        - you can pass the same "job" (object implementing runnable) to multiple threads; each thread will do the job separately.
- **Important methods**
        - ***start()***
            - thread moves from new to runnable state
            - when the thread gets a chance to execute, it's run() method will run
        - ***yield()*** - sends a thread from running to runnable, but doesn't guarantee it

won't move immediately right back into running state.
        - static, so it only works on the currently executing thread.
    - ***sleep(long milliseconds)*** causes the currently executing thread to sleep for a number of milliseconds
        - throws checked exception
        - static, so it only works on the currently executing thread.
    - ***setPriority(int newPriority)*** - set the priority of the thread for running once runnable
        - thread will take on the priority of whatever thread created it
        - the JVM might squish down your priorities (aka instead of 1 through 10 you assigned, you might get 1 through 5)
        - default priority is 5
        - 3 static constants that define priority range
            - MIN_PRIORITY (1)
            - NORM_PRIORITY (5)
            - MAX_PRIORITY (10)
    - ***join()*** - causes one thread to "join" onto the end of another thread's execution.
        - main(){Thread t = new Thread(r); t.start(); t.join()
        - this tells main "You can't be runnable until t finishes.
        - you can add an timeout arg to give a maximum wait time i.e. t.join(int milliseconds)
    - ***run()*** - starts the new thread of execution for each new thread.
    - ***getName()*** - returns the name of the thread running that Runnable's run() method
    - ***currentThread()*** - returns the currently running thread.
    - ***setName()*** - change the name of the thread
    - ***getID()*** - returns the positive, unique, long, number that represents the thread throughout it's life.
    - **ALL OBJECTS IN JAVA INHERIT THESE "THREAD RELEVANT" METHODS**
        - wait() - causes a thread to give up it's lock and "wait" for a notify() from another object.
        - notify() - notify a thread (which one depends on the JVM) wait()ing on a lock that it's now available
        - notifyAll() - notify a thread (which one depends on the JVM) wait()ing on a lock that it's now available
        - *important notes*
            - a thread can't invoke any of these 3 on an object unless it already owns the object's lock.
            - if a thread wait()s after the notify(s) have been called, the thread deadlocks.
- **Important constructors**
    - Thread()
    - Thread(Runnable target)
    - Thread(Runnable target, String name)

        - Thread(String name)
- **Synchronization**
    - It's dangerous to let more than one thread have access to (and be able to change) the same data at the same time.
    - Synchronization prevents any other threads from interacting with data while another thread is inside the method (even if its sleeping)
    - To synchronize data, make the access private, and synchronize the method changing the data.
        - example: private synchronized void doSomething(int number){}
    - **EVERY OBJECT IN JAVA HAS A BUILT IN LOCK THAT ONLY COMES INTO PLAY WHEN THE OBJECT HAS SYNCHRONIZED CODE**
        - entering a synchronized non-static method automatically acquires the lock for the object executing the method.
    - key points
        - only methods or blocks can be synchronized
        - each object has just one lock
        - a class can have both synchronized and non synchronized methods
        - once a thread acquires the lock, no other synchronized methods may be entered by any other threads until the lock is released.
        - if a thread goes to sleep, it holds onto any locks it has
        - a thread can have more than one lock
        - synchronized block syntax

```
public void doStuff(){
System.out.println("not synchronized");
synchronized(this){
    System.out.println("synchronized");
    }
}
```

        - static methods can be synchronized
            - change the block version to synchronized(MyClass.class) instead of synchronized(this)
        - threads calling non-static synchronized methods in the same class only block each other if they use the same reference, b/c the block is based on *this*
        - threads calling static synchronized methods in the same class will ALWAYS block each other b/c they all lock on the same Class instance.
        - a static synchronized method and a non-static synchronized method will never block each other b/c the Class and *this* don't interfere with one another.
- **Deadlock**
    - When two threads are each waiting on one another to release a lock the other needs.
    - Example:

```
int doIt(){ synchronized(ObjectA){
```

```
synchronized(ObjectB){return doSomething();}
                                                      }
                                        void doItToo(){ synchronized(ObjectB){

synchrnized(ObjectA){doSomethingElse();}
                                                            }
```

- The yellow highlighted sections could possibly lock at the same time, which wouldn't allow the inside synchronizations for either method to continue
- The result is inescapable deadlock

Development

- Compiling and running from the command line
  - Javac [args] filepath
    - -d (arg)
      - sets the destination for the class file created
      - Example: javac -d classes source/Test.java
      - . means to search in the current directory ./myDir is the same as myDir
  - Java [args] classname
    - -cp (or -classpath)
      - sets the classpath
      - Example: java -cp classes Test
      - Example2: (for a file in the com.jason package) java -cp classes com.jason.Test
- Retrieving and setting system properties
  - System.getProperties() - returns Properties object
    - Example: Properties p = System.getProperties();
  - setProperty("myProp", "myValue") - sets a user property to a user value
    - you can also do this at the command line with the -D arg for java command
      - java -DcmdProp=cmdVal TestProps
  - getProperty("myProp") - returns a string that matches the string of the key sent
    - Example: System.out.println(System.getProperties().getProperty("myProp"));
- Using command line args
  - come after java [args] file name
  - delimited by white space outside of quotes
- **Jar**
  - Create manifest from top of packages directory
  - "Jar" from top of packages directory where manifest is
  - jar -cfm MyJar.jar Manifest.txt com/jason/*.class com/jason/Math/*.class
  - jar file can now be run from any directory
  - java -jar MyJar.jar
- **Static Imports**
  - you can import a classes static members
  - import static java.lang.System.out
    - changes System.out.print(); to out.print()

- import static java.lang.Integer.*
  - changes Integer.toHexString(42); to toHexString(42);
- *NOTE*
  - If 2 static classes have the same member i.e. MAX_VALUE   (both Integer and Long)
    - you'll get a compile error b/c the compiler won't know which one to use if you import both Integer.* and Long.*