Data Structures and Algorithms

- Data Structures: how we organize information on a computer
- Algorithms: efficient methods of doing standard tasks
- Complexity: how do we measure efficiency/resource usage?

Introduction to Java

- There are several pages on comparing Java and Python available on the course Moodle web site. I'm giving them to you mainly for reference in case you forget the Java way. [http://www.cs.colby.edu/courses/F13/cs231/JavavsPython.html]
- The best way to understand Java is to jump right in, so we'll do so.
- Here are 4 of the most important things you need to understand about Java that distinguishes it from Python:
  - Java has different syntax. No big deal. You'll be surprised how easily you'll be able to adapt to the new syntax.
  - Java is compiled not interpreted and so running a program is a 2-step process (Java is a compiled language. The code is read by the java compiler (javac) and converted into Java bytecode, which is stored in a .class file. To execute the code, you run the java virtual machine (java) and tell it the name of the class whose main method you want to execute. Java bytecode is not code that is directly executable by a CPU (though some native Java computers have been built). Instead, the bytecode runs on a Java Virtual Machine [JVM], which is a program that runs on your computer and translates the Java bytecode into actual machine instructions).
  - Java is statically typed, which means that variables have types, not just data. Variables can only store data of the declared type of the variable.
  - Java is an OO language, meaning virtually everything in it consists of classes, objects and methods. What is a class, object, and method?
- A typical program creates some objects and has them execute their methods. Note there are a few non-objects (called primitives). These are integer, character, boolean, and floating point values. We'll see them a lot but you should mostly focus on the objects and classes.
- To create objects to do things for you, you first have to declare a class. What is a class? [a template indicating the data and methods that can be executed by that object].
- Java has thousands of built-in classes that you can use (API) - Java application programming interface is a list of all classes that are part of the Java development kit (JDK). It includes all Java packages, classes, and interfaces, along with their methods, fields, and constructors. These prewritten classes provide a tremendous amount of functionality to a programmer. [https://docs.oracle.com/javase/7/docs/api/]
- But you can also create new classes of your own. In fact, creating your own new classes is almost all the code that you'll ever write in Java.
- Overall format: Java source code is stored in a simple text file with suffix ".java". You give the file the same name as the class declared in the file. If you declare a class named Hello, then the file should have the name Hello.java.
• First program (The code will be posted online) - Let us take a simple program—it is probably the one we all create when learning a new programming language! Almost all you’ll ever do in a Java program is declare new classes and when you the run the program, you are just running a starting "main" method, which might call other methods.

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

- Single-line comments begin with // characters. Multi-line comments are enclosed within /* ..... */
- Java is case-sensitive; all keywords belonging to the language itself are spelled as in the examples using lower-case characters.
- You declare a class by specifying the class keyword followed by a non-reserved identifier that names it. By convention, the first letter of a class's name is uppercased and subsequent characters are lowercased (for example, Employee). If a name consists of multiple words, the first letter of each word is uppercased (such as SavingsAccount). This naming convention is called camelCasing. They can be any combo of letters, digits, underscores, and dollar signs that doesn’t begin with a digit and that isn’t a keyword. It is recommended that you don’t use the dollar sign.
- The start of a class is the public/private designation followed by the name of the class.
“public” is optional but you should include it by default. It means the class is usable by every other part of the program. The other accessibility options for classes are useful only in large projects.

The {...} braces enclose a block of code—the outer ones being the class definition, and the inner ones the block of the main function.

Whitespace doesn’t matter to Java (but it will to your readers!)

Java programs must have one named main()

The main has no ‘return’ value (void), and it is public (accessible).

The keyword static applied to a function or variable, states it must remain in memory until the program ends

The term in parentheses represents arguments that may be passed to main when it runs; normally they may be omitted. Strings args[]: communication with the command line

System.out.println( ) outputs data to the output stream. Here, it sends the character string in quotes to the screen, followed by a ‘newline’ character (System.out.println( )) without ln does not send an end-of-line character.

Each executable statement is terminated with a semicolon. (an error if you don’t have it)

• Compile and run
  o Make sure you have JDK installed on your computer to be able to compile Java
  o In the Terminal, we navigate to the appropriate directory.
  o We compile: javac Hello.java
  o We run: java Hello

The following example shows a declaration of a public class Die with a single private int field numFaces and two constructor methods. numFaces indicates which face of the Die is on top.

```java
public class Die {
    private int numFaces; // instance variables

    //Constructor for objects of class Die
    public Die() {
        numFaces = 6;
    }

    public Die(int faces) {
        numFaces = faces;
    }
}
```

• Compile and run
  o Compile but doesn’t do anything
• 3 parts: field declaration, constructor declaration, method declaration.
  o Fields are for the data held by an object
  o Constructors are called when new objects are being created
Methods are the behavior of the objects, that is, what they can do for you if you ask them nicely.

- Each of them needs an accessibility specification. "Private" means that it can only be accessed within the class. "Public" means everything can access it. Your default should be to use private for fields and public for everything else.
- Every field needs one type. Python variables can have values of any type (dynamically typed), but Java variables can only have values of one type and you specify what that type is when you declare the variable (statically typed). One way to think about the difference is that in Python, data has a type but in Java, data and variables both have types and if you want to assign data to a variable, the types better match. In Java, a variable can only have one type of data and you have to declare that type before you use the variable. Why the restriction? [HW] [It allows Java to detect errors where you accidentally assign the wrong type of data to a variable.]
- Primitive data types: byte, short, int, long, float, double, boolean, char [https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html]
- Constructors have names that are the same as the class name. They are mostly used to initialize values of the fields.
- Methods must include a return type in their declaration. In our example here, the first method returns an integer and the second returns nothing and so we declare it to be of type "void".
- Variable numFaces
  - The field is not declared static. This means that it can have a different value for each instance of Die. A field like this is called an instance field or instance variable. Since this is the most common kind of field, it is often simply called a field.
  - The field is declared private. Instance fields are normally declared private. This means that they cannot be accessed by methods in other classes. When other classes do things with Die instances, code in those classes can’t access private fields directly. This is an example of information hiding.
  - Therefore, any class that wants to access the variable numFaces should access them through these getters and setters. They are called accessor functions.
- Let's add a main and create a couple of objects and print the top face value. We add the roll() method. The implementation involves the static random() method from the built-in Math class. This method returns a random double which is at least 0 and less than 1. If we multiply the result by 6, throw away any fractional part by casting it to an int, and add 1, we get a random int between 1 and 6 inclusive.

```java
public class Die {
    private int numFaces; // instance variables
    
    //Constructor for objects of class Die
    public Die() {
        this.numFaces = 6;
    }

    public Die(int numFaces) {
        this.numFaces = numFaces;
    }
}
```
// accessor
public int getNumFaces() {
    return this.numFaces;
}

// Roll the die and return the number
// that ends up on top.
public int roll() {
    double r; // declare
    r = Math.random();
    int topFace;
    topFace = (int) (r*this.numFaces) + 1;
    return topFace;
}

public static void main(String Args[]) {
    Die die1 = new Die(); // create a Die instance called die1 (we are using
class Die to create objects)

    System.out.println("Die 1 has " + die1.numFaces + " faces");
    System.out.println( "Rolling die 1" );
    for (int i = 0; i < 20; i++) {
        System.out.println("Roll output: " + die1.roll() );
    }

    Die die2 = new Die(4); //another instance called die2
    System.out.println("Die 2 has " + die2.numFaces + " faces");
    System.out.println( "Rolling die 2" );
    for (int i = 0; i < 20; i++) {
        System.out.println("Roll output: " + die2.roll() );
    }
}

} // end Die class

• this. Keyword
  o this is a reference variable that refers to the current object.

  o Problem (without this keyword)

    class Student{
        int id;
        String name;

        Student(int id,String name){
            id = id;
            name = name;
        }

        void display(){
            System.out.println(id+" "+name); }

        public static void main(String args[]){

}
```java
Student s1 = new Student(111, "Jane");
Student s2 = new Student(321, "David");
s1.display();
s2.display();
}
}
```

- The parameter/local variable and instance variables are the same - that is why we are using this keyword to distinguish between local variable and instance variable.

- Solution (with this keyword)

```java
class Student {
    int id;
    String name;

    Student(int id, String name) {
        this.id = id;
        this.name = name;
    }

    void display() {
        System.out.println(id + " 
```

- Note: If local variables (formal arguments) and instance variables are different, there is no need to use this keyword

- Compiling several files. If your program is broken into several Java files because you created several different classes A.java, B.java, C.java, you need to compile all of them that you change since last compilation before running. So do

  javac A.java B.java C.java
  javac *.java
  java A //if A is the class with the main method you are running

- Arithmetic Operators
  +  Additive operator (also used for String concatenation)
  -  Subtraction operator
  *  Multiplication operator
  /  Division operator
  %  Remainder operator

- Equality and Relational Operators
== Equal to
!= Not equal to
> Greater than
>= Greater than or equal to
< Less than
<= Less than or equal to

- **For Loops**
  - Most languages have for loops as well as while loops. While loops can do everything a for loop can, so why add for loops to a language? [convenience, if you know how many times you will be looping or exact set of data you'll be looping through].

  - Example:
    ```java
    for(int i = 0; i < 1000; i++) {
        System.out.println(i);
    }
    ```

  - for loop format:
    ```java
    for(<init>; <condition>; <update>) {
        <body>
    }
    ```

  - Here's what happens:
    - <init> is executed
    - while <condition> is true
    - <body> is executed
    - <update> is executed

  - Vars declared in init are local to the for loop
  - Some of the 4 parts are optional (i.e., could be left blank). **Guess which ones.** [All 4 parts are optional.]
    - You could declare i outside of loop. The only difference is then i is not local to the for loop and instead is local to the main method
    - You can leave out the condition, which means that the condition is always true (as if true were used as the condition).

  - Example

    ```java
    public class TestForLoop {
        public static void main(String args[]) {
            for(int x = 10; x < 20; x = x + 1) {
                System.out.print("value of x: "+ x);
            }
            System.out.print("\n");
        }
    }
    ```
• While loops.
  o A while loop statement in Java programming language repeatedly executes the code in the body of the loop as long as a given condition is true. When the condition becomes false, program control passes to the line immediately following the loop.
  o Format:
    while(Boolean_expression) {
      // Statements
    }
  o Example
    public class TestWhileLoop {
      public static void main(String args[]) {
        int y = 10;
        while( y < 20 ) {
          System.out.print("value of y : "+ y);
          y++;
          System.out.print("\n");
    }
    }

• Import statements
  o Java has a huge library of classes (more than 3 thousand of them) that it has already created that are ready for your use. This is wonderful.
  o For example, there are the Math, String, and System classes you used in Die.java. They are part of a built-in package (like the __built-in__ package in Python) called java.lang.
  o There are other libraries as well that, like in Python, you need to import.
  o An example is the Random class. You use it to choose random values [https://docs.oracle.com/javase/8/docs/api/java/util/Random.html]. Note that it does much more than the Math.random() method does and so is the preferred way of dealing with randomness except if you want a random double between 0 and 1.
  o Problem: [Create a Java class that creates a Random object and calls one of its methods. You get an error message]

    public class TestRandom {
      public static void main( String args[] ) {
        // create random object
        Random generator = new Random();

        // check next int value
        System.out.println("Next int value: " + generator.nextInt(1000));
      }
    }
Solution: Look in API for info on Random class. Note that it is in the java.util package, so you can fix the problem by adding an "import java.util.Random;" at the beginning of the file. (You are telling javac to go into the java.util package and find the Random class and import it.)

Add `import java.util.*;`.

In summary, all the Java library classes are in packages. The classes in the java.lang package are automatically imported. The java.lang package is a core part of the language and so it is implicitly imported for you automatically in all Java files. In contrast, all classes in all other packages need to be imported. Use the documentation to find out which package your class is in. Most classes we will use are in the default java.lang or java.util.

You can also import a whole package instead of just one class from that package:

```java
import java.util.*;
```

Then you can use all the classes in that package without worrying about the package prefix.

**Generics**

Before generics, we can store any type of objects in collection i.e. non-generic. Now generics, forces the java programmer to store specific type of objects.

A class like `ArrayList<E>` is a generic type. It has a type parameter E that represents the type of the elements stored in the list. Instead of just using a `ArrayList`, not saying anything about the type of elements the list contains, we can use a `ArrayList<String>` or a `ArrayList<Integer>`, thereby specifying that we mean a list of strings or integral values respectively.

The other class in the java.util package that you'll use is the `ArrayList` class. An `ArrayList` is like a Python list in that it stores a list of data values.

API [https://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html](https://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html)

Example code:

```java
import java.util.*;
ArrayList list = new ArrayList();
list.add(3);
int x = (int) list.get(0);
```

But I don't want you to write code like this. When you use an `ArrayList`, you are almost always storing data of one particular kind, like integers or strings or Dice. In those cases, I want you to declare that your `ArrayList` will be storing that kind of data.

Example code:

```java
import java.util.*;
ArrayList<Integer> list1 = new ArrayList<Integer>();
list1.add(3);
```
list1.add(4);
int x = list.get(0); // no typecast necessary
System.out.println(list2.get(0) + list2.get(1));

ArrayList<String> list2 = new ArrayList<String>();
list2.add("Hello");
String s = list2.get(0); // no typecast necessary

Note: Casting is an operation that allows us to change the type of a value. We can take a value of one type and cast it into an equivalent value of another type.

double d1 = 3.2;
double d2 = 3.9999;
int i1 = (int) d1; // i1 gets value 3
int i2 = (int) d2; // i2 gets value 3
double d3 = (double) i2; // d3 gets value 3.0

- The advantage of doing it this way is that (a) it is easier to understand and (b) the Java compiler will catch errors where you accidentally try to add a value of the wrong type to the ArrayList.
  list1.add("hi"); // ← ERROR!
- Always use generic types. Don't ever create an ArrayList without the brackets.
- If you want a list that can store any kind of data, use <Object> as the generic type.
- Example:

  ArrayList<Object> list = new ArrayList<Object>();
  list.add(3); // no error
  list.add("Hi"); // no error

- **Generic Types Summary**
  - Generic types are used to restrict lists and other data structures to storing particular kinds of data.
  - If you want a list that can store any kind of data, use <Object> as the generic type.

- **Primitive wrappers**
  - ArrayList<Object> can hold any kind of data in the list. So you can have statements such as
    
    list.add("a string");
    list.add(3);
  - This works fine, except for one problem: ArrayLists actually store only objects and not primitives. But then how can we add an int?
  - Answer: Behind the scenes, Java "wraps" the primitive in an object. It actually converts the primitive 3 to an Integer object that stores the value 3 and this object is what is stored in the list.
  - **Autoboxing** is the automatic conversion that the Java compiler makes between the primitive types and their corresponding object wrapper classes. For example,
converting an int to an Integer, a double to a Double, and so on. If the conversion goes the other way, this is called **unboxing**.

- When autoboxing, it wraps ints in Integers, booleans in Booleans, chars in Characters, and doubles in Doubles.
- Java automatically unbox as well:
  
  ```java
  Integer ob = new Integer(3);
  int x = ob;  //auto unboxes the integer and assigns 3 to x
  ```
- Occasionally, the wrapping rears its ugly head:
  
  ```java
  int x = list.get(0);
  ```
  
  This will crash since the ArrayList's `get` method has return type of Object, and so the method could return any type of object, not just an int. Therefore, the assignment is illegal.
- The solution is to typecast the value returned by `get`:
  
  ```java
  int x = (Integer) list.get(0);
  ```
  
  or
  
  ```java
  int x = (int) list.get(0);
  ```

  These statements are equivalent and tell the Java compiler 4 things:
  
  1. Call the get method of the list
  2. The 0-th element is actually an Integer object
  3. Unwrap the Integer to just the int it contains
  4. Assign that int to a new variable `x`.

- Recall Primitive data types: byte, short, int, long, float, double, Boolean, char. If your variable is not one of the 8 types we discussed before, then it is non primitive type (a variable whose type is an object). We call such variable **object reference variable**.

- **Java's memory model**
  
  - All variables are allocated space in memory to store something.
  - Java divides the computer's memory into two parts: a stack and a heap.
  - Stack is the section of memory where we will store temporary information such as method invocations, local variables;
  - Objects are stored in Heap memory.

![Diagram of Stack and Heap](image-url)

- The Stack
  
  Where method invocations and local variables live

- The Heap
  
  Where ALL objects live

  ```java
  public static void main(String[] args) {
    go();
    doStuff();
  }
  ```
**Primitives vs objects**

i. All primitive variables store their value in that space.

ii. All reference variables (object reference variable) store a pointer in that space to the object they refer to.

- Primitive variable represents the actual value of the variable whereas object reference variable represents a way to get to the object.

**Example**

```java
{  
  int x;  
  Date d1, d2;  
  d1 = new Date(18,11,2002);  
  d2 = d1;  
}
```

**Local variables vs fields**

- There are two kinds of variables: instance variables/fields and local variables/parameters
- Every method has a stack frame with space for all its local variables (while it is being executed)
- Fields reside in the objects on the heap.

**Local/Instance/Class Variable:** [http://www.fredosaurus.com/notes-java/data/variables/45local-inst-class.html](http://www.fredosaurus.com/notes-java/data/variables/45local-inst-class.html) (Check Table)
- Instance variables – declared inside a class. Must be initialized before use (else compiler error)
- Local variables – declared within a method
Parameters are just local variables in Java except that they are initialized by the caller instead of in the method body.
Local variables' lifetimes consist of the time the method that contains them starts until the method returns.
Field's lifetimes consist of the time from the creation of the object until the garbage collection of the object.
  - garbage collection is the process of looking at heap memory, identifying which objects are in use and which are not, and deleting the unused objects (http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html)
Local variable's scope is from the point of declaration to the end of the enclosing scope.
Field's scope depends on whether they are public or private. The scope of private fields is the whole class body in which the fields are declared. The scope of public fields is everywhere.

<table>
<thead>
<tr>
<th>local var/params</th>
<th>instance var/field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Location</td>
<td>stack frame</td>
</tr>
<tr>
<td>Lifetime</td>
<td>during method execution</td>
</tr>
<tr>
<td>Scope</td>
<td>remaining body of method</td>
</tr>
<tr>
<td></td>
<td>heap</td>
</tr>
<tr>
<td></td>
<td>object creation to garbage</td>
</tr>
<tr>
<td></td>
<td>class if private, all if public</td>
</tr>
</tbody>
</table>

Recall that the `new` command allocates memory for a new object, including space for all the fields of the object, clears out that memory, calls the constructor to initialize the fields, and then returns a pointer to the memory.
You can have more than one constructor, as long as they differ on the number or types of the parameters.
Method calls' parameters are **passed by value**
  - Primitive values are copied into the parameters
  - Objects are not copied. Instead the pointers to the objects are copied into the parameters.

Example
```java
public class Main {
    public static void main(String[] args) {
        Foo f = new Foo("f");
        changeReference(f); // It won't change the reference!
        modifyReference(f); // It will modify the object that the reference variable "f" refers to!
    }
    public static void changeReference(Foo a) {
```
1. Declaring a reference named f of type Foo and assign it to a new object of type Foo with an attribute "f".
   ```java
   Foo f = new Foo("f");
   ```

2. As you call the method changeReference, the reference a will be assigned to the object which is passed as an argument.
   ```java
   changeReference(f);
   ```

3. Declaring a reference named b of type Foo and assign it to a new object of type Foo with an attribute "b".
   ```java
   Foo b = new Foo("b");
   ```
5. \( a = b \) is re-assigning the reference \( a \) NOT \( f \) to the object whose its attribute is "b".

6. As you call `modifyReference(Foo c)` method, a reference \( c \) is created and assigned to the object with attribute "f".

7. `c.setAttribute("c");` will change the attribute of the object that reference \( c \) points to it, and it's same object that reference \( f \) points to it.

(http://stackoverflow.com/questions/40480/is-java-pass-by-reference-or-pass-by-value)

- Methods that return a value return the actual value if primitive and return a pointer to the object if it is not primitive.
• Recall that fields are initialized to default values when an object is constructed. What if one field was of type String? Can you guess the default value?

• Objects default to null; ints, longs and shorts to 0; Strings to null; booleans to false (Recall - Default Values: http://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html)

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Default Value (for fields)</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>0</td>
</tr>
<tr>
<td>short</td>
<td>0</td>
</tr>
<tr>
<td>int</td>
<td>0</td>
</tr>
<tr>
<td>long</td>
<td>0L</td>
</tr>
<tr>
<td>float</td>
<td>0.0f</td>
</tr>
<tr>
<td>double</td>
<td>0.0d</td>
</tr>
<tr>
<td>char</td>
<td>'\u0000'</td>
</tr>
<tr>
<td>String (or any object)</td>
<td>null</td>
</tr>
<tr>
<td>boolean</td>
<td>false</td>
</tr>
</tbody>
</table>

• Inheritance & polymorphism
  o One big advantage of OO programming over iterative programming is that it is conceptually easier to organize and understand large programs if they are divided into classes and objects
  o But there is another advantage that we haven’t seen yet: It promotes code reuse and avoids code duplication.
  o Example

![Diagram of class hierarchy with Employee as the superclass and Developer, Architect, ProjectManager, and Tester as subclasses]

  o Syntax: class Developer extends Employee

  o Let’s go back to our Die class and in the main method, add a line.
    o System.out.println(die1.toString());
o What do you think will happen? error?
  o Demo

o Somehow the Die class has a toString() method. Where did it come from?
  o Answer: It inherited it from its parent or superclass. Every class has a parent class except for the Object class which is the granddaddy of them all. If you don’t specify a parent class, then Object is used as the parent.
  o It turns out that the Object class has an toString method and that’s where it came from in our example. [look at online docs for Object class]
  o The Object class, in the java.lang package, sits at the top of the class hierarchy tree. Every class is a descendant, direct or indirect, of the Object class. Every class you use or write inherits the instance methods of Object (https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html)

- A subclass inherits all the methods and fields from its parent class. Since that class inherited methods and fields from its parent, a grandchild inherits from both the parent and the grandparent.

- Exercise: Create a new class called RedDie such that it inherits from the Die class. Add a main method to test it.

- **Method Overriding:** Overriding toString
  o What if we create our own toString method in the Die class and then call toString as we did above? Which toString method gets executed?
  o Answer: The most local one does. So the Die class’ toString method gets executed.
  o This is called **overriding.** The Die toString method overrides the inherited toString method.
  o Why would you want to override the inherited toString method?
  o Answer: The inherited method just returns the string "Die@...", which is not very helpful. A more friendly version of toString() would be nice.
/Override
  public String toString() {
    return Integer.toString( this.numFaces );
  }

• Summary:
  o A class inherits all the methods of its ancestor classes, all the way up the Object class.
  o A subclass can override the inherited methods by defining a method with the same header in the subclass.
  o Fields are also inherited by subclasses. However, if the fields are private in the superclass, then the subclass inherits them but can’t access them.

• Let’s try another line of code to see what happens:
  Object o = new Die();
  o What do you think will happen when we try to execute it? Error? [run it]
  o Since Die is a subclass of Object, every Die "is an" Object and so you can do such assignments. A variable of type A can be assigned any object of type A or of a type that is a subclass of A.

• How do you tell Java that you want your class to be a subclass of another class?
  o Use the "extends" keyword:
    public class Die extends Object
    {
      ...
    }
  o But you don’t have to explicitly extend any class and in that case the Java compiler automatically puts in "extends Object" for you.

• Static fields and methods
  • Static Variables (“static” Keyword = Class Variables)
    o In Java Variables can be declared with the “static” keyword. Example: static int y = 0;
    o When a variable is declared with the keyword static, it’s called a class variable. All instances share the same copy of the variable. A class variable can be accessed directly with the class, without the need to create a instance.
    o It is a variable which belongs to the class and not to object(instance)
    o Static variables are initialized only once, at the start of the execution. These variables will be initialized first, before the initialization of any instance variables
    o A single copy to be shared by all instances of the class
    o A static variable can be accessed directly by the class name and doesn’t need any object
    o Syntax : <class-name>.<variable-name>
    
    o Example
      public class Book {
      
        public final static String NAME = "I’m a static variable";
      }
public class Application {
    public static void main(String[] args) {
        System.out.println(Book.NAME);
    }
}

• Static Methods
  o Methods in a class are usually there to manipulate the data in the fields for you (e.g., get and set them). But what if a class has no fields? Then the objects of the class all do exactly the same thing, so there is never a reason to create more than one object of that class.
  o To avoid having to create any objects of the class in that situation, Java has the keyword "static".
  o It is a method which belongs to the class and not to the object (instance)
  o A static method can access only static data. It cannot access non-static data (instance variables)
  o A static method can be accessed directly by the class name and doesn’t need any object
  o Syntax: <class-name>.<method-name>
  o Where else have we seen static methods? [Math.random]
  o Example

        import java.lang.Math;

        class Another {
            public static void main(String[] args) {
                int result;

                result = Math.min(10, 20); // calling static method min by writing class name

                System.out.println(result);
                System.out.println(Math.max(100, 200));
            }
        }

• Equals
  o How do you determine equality?
    i. For arithmetic or boolean operators, the answer is simple: you test with the equals operator (==).
    ii. For object references, though, Java provides both == and the equals() method inherited from java.lang.Object. The equals operator can be confusing, as it simply compares two object references to see if they refer to the same object.
iii. If we want to compare the contents of two objects, then the standard approach is to implement an equals method that does the comparison.

- Consider the code:
  ```java
  int x = 3, y = 3;
  Die d1 = new Die();
  Die d2 = new Die();
  System.out.println(d1.getFaces () + ":" + d2.getFaces ());
  if(x == y) System.out.println("ints are equal");
  if(d1 == d2) System.out.println("Die are equal");
  if(d1.equals(d2)) System.out.println("Die are equal");
  ```

- What happens if we run this? Why doesn't the compiler complain that Dies don't have equals() methods? (inherits equals() method from object superclass).

  ```java
  public boolean equals (Object other)
  ```

- Does the number up on the dice matter when determining equality?

- What if you want to test whether the data held by two objects are equal rather than whether the two objects are really the same object? Create a method that does so!

- Override the inherited equals method

  ```java
  @Override public boolean equals (Object other) {
    Die d = (Die) other;
    return this.numFaces == d.numFaces ;
  }
  ```

- This method public boolean equals (Object other) really shouldn't crash if the other reference is null or if it is an object, but it isn't a Die. Java lets us check the type of each object, so we should take advantage of that and update our code:

  ```java
  @Override
  public boolean equals(Object other) {  
    if (other == null) {
      return false;
    }
    if (other.getClass () != this.getClass()) {
      return false;
    }
    //Die d = (Die) other; // Cast the Object into a Die.
    return this.numFaces == d.numFaces ;
  }
  ```
• **Greater or Less Than**
  o Comparing two Die objects for greater or less than regarding the value side up.
  o What if you want to know which is greater? Equals() won't do it.
  o Suggestions? [isGreater() method]
  o Actually, we’ll call it compareTo()

    public int compareTo(Die that) {
        if(this.getSideUp() > otherDie.getSideUp())
            return 1;
        else if(this.getSideUp() == otherDie.getSideUp())
            return 0;
        else if(this.getSideUp() < otherDie.getSideUp())
            return -1;
    }

• **Interfaces**

An interface is a pure skeleton. It contains the property definition of the object in mind, but not a hint of how to implement it, but the inherited class must implement every bit of it.

```java
interface Bicycle {
    // wheel revolutions per minute
    void changeCadence(int newValue);
    void changeGear(int newValue);
    void speedUp(int increment);
    void applyBrakes(int decrement);
}
```

To implement this interface, the name of your class would change (to a particular brand of bicycle, for example, such as ACMEBicycle), and you’d use the implements keyword in the class declaration:

Example:

```java
class ACMEBicycle implements Bicycle {
    void changeCadence(int newValue) {
    }
    void changeGear(int newValue) {
    }
    void speedUp(int increment) {
    }
    void applyBrakes(int decrement) {
    }
}
```

(https://docs.oracle.com/javase/tutorial/java/concepts/interface.html)
• Abstract Classes

An abstract class is a class that is declared abstract—it may or may not include abstract methods. Abstract classes cannot be instantiated, but they can be subclassed.

An abstract method is a method that is declared without an implementation (without braces, and followed by a semicolon), like this:

```java
abstract void moveTo(double deltaX, double deltaY);
```

An abstract class may contain a certain level of implementation which the inherited class is free to override and implement the empty ones.

```java
abstract class GraphicObject {
    int x, y;
    ...
    void moveTo(int newX, int newY) {
        ...
    }
    abstract void draw();
    abstract void resize();
}

class Circle extends GraphicObject {
    void draw() {
        ...
    }
    void resize() {
        ...
    }
}
```

(http://docs.oracle.com/javase/tutorial/java/IandI/abstract.html)

Disclaimer: Notes adapted from previous CS 231 lecture materials at Colby College.