Class 5

- **Inheritance & polymorphism**
  - 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.
  - But there is another advantage that we haven’t seen yet: It promotes code reuse and avoids code duplication.
  - Example

![Class Hierarchy Diagram]

- Syntax: class Developer extends Employee

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

- Somehow the Die class has a toString() method. Where did it come from?
- Answer: It inherited it from its parent or superclass. Every class has a parent class except for the Object class which is the grandaddy of them all. If you don’t specify a parent class, then Object is used as the parent.
- 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]
- 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](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 grand parent.

• 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.

```java
//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:
  ```java
  Object o = new Die();
  ```
  o What do you think will happen when we try to execute it? Error? [run it]
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?**
  - Use the "extends" keyword:
    ```java
    public class Die extends Object
    {
        ...
    }
    ```
  - 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)
    - In Java Variables can be declared with the "static" keyword. Example: static int y = 0;
    - 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 an instance.
    - It is a variable which belongs to the class and not to object(instance)
    - 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
    - A single copy to be shared by all instances of the class
    - A static variable can be accessed directly by the class name and doesn't need any object
    - Syntax: <class-name>.<variable-name>

    - Example
      ```java
      public class Book {
          public final static String NAME = "I'm a static variable";
      }
      ```
  - public class Application {
    ```java
    public static void main(String[] args) {
        System.out.println(Book.NAME);
    }
    ```
  }

- **Static Methods**
  - 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.

- To avoid having to create any objects of the class in that situation, Java has the keyword "static".
- It is a method which belongs to the class and not to the object(instance)
- A static method can access only static data. It cannot access non-static data (instance variables)
- A static method can be accessed directly by the class name and doesn't need any object
- Syntax: <class-name>.<method-name>
- Where else have we seen static methods? [Math.random]
- Example

```java
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**
  - How do you determine equality?
    1. For arithmetic or boolean operators, the answer is simple: you test with the equals operator (==).
    2. 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.
    3. 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).
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
  - Comparing two Die objects for greater or less than regarding the value side up.
  - What if you want to know which is greater? `Equals()` won’t do it.
  - Suggestions? [isGreater() method]
  - Actually, we’ll call it `compareTo()`

```java
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;
}