# Lecture 6: Hash Code, Nested Classes, Enum

## Learning Outcomes

- Understand the need to override `hashCode` whenever `equals` is overridden and how to use `Arrays`' `hashCode` method to compute a hash code
- Understand static vs. non-static nested class, local class, and anonymous class and when to use / not to use one
- Understand the rules about final / effectively final variable access for local class and anonymous class
- Aware of the limitation when declaring a new anonymous class
- Understand the concept of variable capture
- Be aware that `enum` is a special case of a `class` and share many features as a class -- not just constants can be defined.
- Understand how `enum` is expanded into a subclass of `Enum`
- Know that enum constants can have customized fields and methods.

## 1. Hash Code

We have seen how `HashMap` in Java Collections allows us to store a key-value pair and lookup a value with the key, and how `HashSet` stores item with no duplicates.

Internally, `HashMap` and `HashSet` organize the items into "buckets". Such organization improves efficiency if we need to search through the items, either to find a pair with a given key, or to check if an item already exists in the set. We no longer need to search through the entire collection, but only the "bucket" that the item, if exists, would have been organized into. For this idea to work, each item must be *hashed* deterministically into a bucket.

In Java objects, such hashing operation is determined by the method `hashCode()` of a given key object. `hashCode()` returns an `int`, which `HashMap`, `HashSet`, or other data structures can use to determine which "bucket" to store a (key, value) pair or an item into.

You will learn more about hashing and hash tables in CS2040.

But, what is important here is that two keys (two objects, in general) which are the same ( `equals()` returns `true` ), must have the same `hashCode()`. Otherwise, `HashMap` would fail.

So it is important to ensure that if `o1.equals(o2)`, then `o1.hashCode() == o2.hashCode()`. Note that the reverse does not have to be true -- two objects with the same hash code does not have to be equals.

This property is also useful for implementing `equals()`. For a complex object, comparing every field for equality can be expensive. If we can compare the hash code first, we could filter out objects with different hash code (since they cannot be equal). We only need to compare field by field if the hash code is the same.

Let's see some example:

```java
String s1 = "hello";
String s2 = new String("hello");
String s3 = "goodbye";
s1.hashCode();
s2.hashCode();
s3.hashCode();
```

Lines 4-5 both return 99162326, an integer value that is calculated using a mathematical function, called *hash function*, so that two strings with the same content returns the same value. Line 6 returns 207022353.

We can see the problem when we don't define `hashCode()`. Take our `Point` class from before:

```
jshell> Point p = new Point(0,0);
p ==> (0.0,0.0)

jshell> Point q = new Point(0,0);
q ==> (0.0,0.0)

jshell> p.equals(q);
$5 ==> true

jshell> HashSet<Point> set = new HashSet<>();
set ==> []

jshell> set.add(p);
$7 ==> true

jshell> set.add(q);
$8 ==> true

jshell> set
set ==> [(0.0,0.0), (0.0,0.0)]
```

You can see that we have added two points that are the same into the set, which usually is not what we want. To fix this, we need to write our own `hashCode()`, by overriding the `hashCode()` method in the `Object` class.

Calculating good hash code is an involved topic, and is beyond the scope of CS2030. In Java, the utility class `Arrays` provide us some convenient `hashCode` methods to help us. We can pack the fields of a class into an array, and then call `Arrays.hashCode()` to compute the hash code of these fields. Let's add the following to `Point`:

```java
class Point {
    :
  @Override
  public int hashCode() {
    double[] a = new double[2];
    a[0] = this.x;
    a[1] = this.y;
    return Arrays.hashCode(a);
  }
}
```

Now, `set.add(q)` above returns false -- we can no longer add point (0,0) twice in to the set.

**XOR-ing hash codes**

If you have fields in a class that are of different types (and cannot be packed into a single array), you can pack them into different arrays, call `Arrays.hashCode()` for each of the arrays, and combine the hash code together with the XOR operator `^`. XOR is mathematically a good operator for combining two hash codes, since given two numbers with equal distribution of 0s and 1s, XOR-ing them will give another number with equal distribution of 0s and 1s.

## 2. Nested Class

The second topic for today is nested class. There are four kinds of nested classes in Java. You have seen a static nested class, used inappropriately in Lab 1a [../lab1a/index.html]. Let's see where are some good use cases to use nested classes.

Nested classes are used to group logically relevant classes together. Typically, a nested class is tightly coupled with the container class, and would have no use outside of the container class. Nested classes can be used to encapsulate information within a container class, for instance, when implementation of a class becomes too complex. As such, it is useful for "helper" classes that serve specific purposes.

A nested class is a field of the containing class, and can access fields and methods of the containing class, including those declared as `private`. We can keep the nested class within the abstraction barrier, and declare a nested class as `private`, if there is no need for it to be exposed to the client outside the barrier. On the other hand, because of this, you should really have a nested class only if the nested class belongs to the same encapsulation. Otherwise, the containing class would leak its implementation details to the nested class.

Take the `HashMap<K,V>` class for instance. The implementation of `HashMap<K,V>` [https://github.com/openjdk-mirror/jdk7u-jdk/blob/master/src/share/classes/java/util/HashMap.java] contains several nested classes, including `HashIterator`, which implement an `Iterator<E>` interface for iterating through the key and value pairs in the map, and an `Entry<K,V>` class, which encapsulates a key-value pair in the map. Some of these classes are declared `private`, if they are only used within the `HashMap<K,V>` class.

A nested class can be either static or non-static. Just like static fields and static methods, a *static nested class* is associated with the containing *class*, NOT an *instance*. So, it can only access static fields and static methods of the containing class. A *non-static nested class*, on the other hand, can access all fields and methods of the containing class. A *non-static nested class* is also known as an *inner class*.

```java
class A {
  private int x;
  static int y;

  class B {
    void foo() {
      x = 1; // accessing x from A is OK
    }
  }

  static class C {
    void bar() {
      x = 1; // accessing x from A is not OK since C is static
      y = 1; // accessing y is OK
    }
  }
}
```

### Local Class

We can also declare a class within a function as well, just like a local variable. One example is the `NameComparator` class in Lecture 5 [../lec05/index.html]. In that example, we created another top-level class (and so, a new file) just to define how to compare two strings, which is a little bit silly.

We can actually just define the `NameComparator` class when we need it, just before we sort:

```java
void sortNames(List<String> names) {

  class NameComparator implements Comparator<String> {
    public int compare(String s1, String s2) {
      return s1.length() - s2.length();
    }
  }

  names.sort(new NameComparator());
}
```

This makes the code easier to read since we keep the definition of the class and its usage closer together.

Classes like `NameComparator` that are declared inside a method (to be more precise, inside a block of code between `{` and `}`) is called a *local class.* Just like a local variable, a local class is scoped within the method. Like a nested class, a local class has access to the variables of the enclosing class. Further, it can access the variables of the local variables of the enclosing method.

```java
class A {
  int x = 1;

  void f() {
    int y = 1;

    class B {
      void g() {
        x = y; // accessing x and y is OK.
      }
    }

    new B().g();
  }
}
```

### Variable Capture

Recall that when a method returns, all local variables of the methods are removed from the stack. But, an instance of that local class might still exist. Consider the following example:

```java
interface C {
  void g();
}
class A {
  int x = 1;

  C f() {
    int y = 1;

    class B implements C {
      void g() {
        x = y; // accessing x and y is OK.
      }
    }

    B b = new B();
    return b;
  }
}
```

Calling

```
A a = new A();
C b = a.f();
b.g();
```

will give us a reference to an object of type `B` now. But, if we call `b.g()`, what is the value of `y`?

For this reason, even though a local class can access the local variables in the enclosing method, the local class makes *a copy of local variables* inside itself. We say that a local class *captures* the local variables.

## Effectively `final`

Variable captures could be confusing. Consider the following code:

```
void sortNames(List<String> names) {
  boolean ascendingOrder = true;
  class NameComparator implements Comparator<String> {
    public int compare(String s1, String s2) {
      if (ascendingOrder)
        return s1.length() - s2.length();
      else
        return s2.length() - s1.length();
    }
  }

  ascendingOrder = false;
  names.sort(new NameComparator());
}
```

Will `sort` sorts in ascending order or descending order?

To avoid confusing code like this, Java only allows a local class to access variables that are explicitly declared `final` or implicitly final (or *effectively* final). An implicitly final variable is one that does not change after initialization. Therefore, Java saves us from such hair-pulling situation and disallow such code -- `ascendingOrder` is not effectively final so the code above does not compile.

**Variable Capture in Javascript**
Those of you who did CS1101S or otherwise familiar with Javascript might want to note that this is different from Javascript, which does not enforce the final/effectively final restriction in variable captures.

I do not see a good use case for local class -- if you have information and behavior inside a block of code that is so complex that you need to encapsulate it within a local class, it is time to rethink your design.

What about the use case of `NameComparator` above? There are actually better ways to write the class. We will look at *anonymous class* today.

## Anonymous Class

An anonymous class is one where you declare a class and instantiate it in a single statement. It's anonymous since We do not even have to give the class a name.

```
names.sort(new Comparator<String>() {
  public int compare(String s1, String s2) {
    return s1.length() - s2.length();
  }
});
```

The example above removes the need to declare a class just for the purpose of comparing two strings.

An anonymous class has the following format: `new X (arguments) { body }`, where:

- *X* is a class that the anonymous class extends or an interface that the anonymous class implements. X cannot be empty. This syntax also implies an anonymous class cannot extend another class and implement an interface at the same time. Furthermore, an anonymous class cannot implement more than one interface.
- *arguments* are the arguments that you want to pass into the constructor of the anonymous class. If the anonymous class is extending an interface, then there is no constructor, but we still need `()`.
- *body* is the body of the class as per normal, except that we cannot have a constructor for an anonymous class.

The syntax might look overwhelming at the beginning, but we can also write it as:

```
Comparator<String> cmp = new Comparator<String>() {
  public int compare(String s1, String s2) {
    return s1.length() - s2.length();
  }
};
names.sort(cmp);
```

Line 1 above looks just like what we do when we instantiate a class, except that we are instantiating an interface with a `{ .. }` body.

An anonymous class is just like a local class, it captures the variables of the enclosing scope as well -- the same rules to variable access as local class applies.

## 3. Enum

An `enum` is a special type of class in Java. Variable of an enum type can only be one of the predefined constants. Using enum has one advantage over the use of `int` for predefined constant -- it is type safe. Consider how we have been defining different event types in Lab 1a [../lab1a/index.html].

```
  public static final int CUSTOMER_ARRIVE = 1;
  public static final int CUSTOMER_DONE = 2;
```

But, we cannot prevent someone from creating an event `new Event(time, 100)`, passing in an invalid event type (type 100).

If we define the event type as enum, then we can write like this:

```
enum EventType {
  CUSTOMER_ARRIVE,
  CUSTOMER_DONE
}
```

and the field `eventType` in `Event` now has a type `EventType` instead of `int`:

```
class Event {
  private double time;
  private EventType eventType;
}
```

Trying to assign anything other than the two predefined event type to `eventType` would result in compilation error. Remember, an error caught at compile time is much better than an error caught during run time, so this is good!

**In other languages**
Enumerated types like `enum` are common in other langauges, including procedural languages like C. But, `enum` in Java is more powerful, as seen below.

## Enum's Fields and Methods

Each constant of an enum type is actually an instance of the enum class and is a field in the enum class declared with `public static final`.
Since enum in Java is a class, we can define constructors, methods, and fields in enums.

```
enum Color {
  BLACK(0, 0, 0),
  WHITE(1, 1, 1),
  RED(1, 0, 0),
  BLUE(0, 0, 1),
  GREEN(0, 1, 0),
  YELLOW(1, 1, 0),
  PURPLE(1, 0, 1);

  private final double r;
  private final double g;
  private final double b;

  Color(double r, double g, double b) {
    this.r = r;
    this.g = g;
    this.b = b;
  }

  public double luminance() {
    return (0.2126 * r) + (0.7152 * g) + (0.0722 * b);
  }

  public String toString() {
    return "(" + r + ", " + g + ", " + b + ")";
  }
}
```

In the example above, we represent a color with its RGB component. Enum values should only constants, so `r`, `g`, `b` are declared as `final`. We have a method that computes the luminance (the "brightness") of a color, and a `toString()` method.

The enum values are now written as `BLACK(0, 0, 0)`, with arguments passed into its constructor.

## Custom Methods for Each Enum

Enum in Java is more powerful than the above -- we can define custom methods for each of the enum constant, by writing *constant-specific class body*. If we do this, then each constant becomes an anonymous class that extends the enclosing enum.

Consider the enum `EventType`. You can do the following:

```
enum EventType {
  CUSTOMER_ARRIVE {
    void log(double time, Customer c) {
      System.out.println(time + " " + c + " arrives");
    }
  },
  CUSTOMER_DONE {
    void log(double time, Customer c) {
      System.out.println(time + " " + c + " done");
    }
  };

  abstract void log(double time, Customer c);
}
```

In the code above, `EventType` is an abstract class -- `log` is defined as `abstract` with no implementation. Each enum constant has its own implementation to log that particular event.

```
EventType.CUSTOMER_DONE.log(time, customer)
```

to log that particular event.

I admit that this example is rather contrived -- we can do the same with a polymorphism in a much cleaner way.

## The Class `Enum`

`enum` is a special type of class in Java. All `enum` inherits from the class `Enum` implicitly. Since `enum` is a class, we can extend `enum` from interfaces as per normal class. Unfortunately, `enum` cannot extend another class, since it already extends from `Enum`.

One implicitly declared method in `enum` is a static method:

```
public static E[] values();
```

We can call `EventType.values()` or `Color.values()` to return an array of event types or an array of colors. `E` is a type parameter, corresponding to the enum type (either `EventType`, `Color`, etc). To maintain flexibility and type safety, the class `Enum` which all enums inherit from has to be a generic class with `E` as a type parameter.

Considering `EventType`,

```
enum EventType {
  CUSTOMER_ARRIVE,
  CUSTOMER_DONE
}
```

is actually

```
public final class EventType extends Enum<EventType> {
  public static final EventType[] values { .. }
  public static EventType valueOf(String name) { .. }

  public static final EventType CUSTOMER_ARRIVE;
  public static final EventType CUSTOMER_DONE;
    :

  static {
    CUSTOMER_ARRIVE = new EventType();
    CUSTOMER_DONE = new EventType();
      :
  }
}
```

Even though we can't extend from `Enum` directly, Java wants to ensure that `E` must be a subclass of `Enum` (so that we can't do something non-sensical like `Enum<String>`. Furthermore, some methods from `Enum` (such as `compareTo()`) are inherited to the enum class, and these methods involved generic type `E`. To ensure that the generic type `E` actually inherits from `Enum<E>`, Java defines the class `Enum` to have bounded generic type `Enum<E extends Enum<E>>`.

The expansion of enum `EventType` to a class above also illustrates a few points:

- An `enum` is final. We cannot inherit from enum (those with constant-specific body are exceptions).
- A class in Java can contain fields of the same class.
- The block marked by `static { .. }` are *static initializers*, they are called when the class is first used. They are the counterpart to constructors for objects, and are useful for non-trivial initialization of static fields in a class.

## Enum-related Collections

Java Collection Frameworks provide two useful classes `EnumSet` and `EnumMap` -- they can be viewed as special cases of `HashSet` and `HashMap` respectively -- the only difference is that we can only put enum values into `EnumSet` and enum-type keys into `EnumMap`.

## Exercise

1. Explain how each of the following language features of Java ensures type safety. You can give an example.

   (a) enum (b) generics

2. Consider the program below:

```
class B {
  void f() {
    int x = 0;

    class A {
      int y = 0;
      A() {
        y = x + 1;
      }
    }

    A a = new A();
  }
}
```

Suppose that a variable `b` is an instance of class `B`, and a program calls `b.f()`. Sketch the content of the stack and heap immediately after the Line `A a = new A()` is executed. Label the values and variables / fields clearly. You can assume b is already on the heap and you can ignore all other content of the stack and the heap before `b.f()` is called.