# Announcements

-

# Quiz time ;)

# Generics

What is the point of Generics if we do not know what type of object we are working with? (i.e. we do not know what methods to call or even how to compare two different objects)

- For example:
  - E obj1;
  - E obj2;

How do we compare obj1 to obj2? The **“==” (equality) operator** won’t work because obj1 and obj2 are reference types and therefore each have different addresses stored in their respective memory locations.

Let’s look at an example...

# Generics (cont.)

The answer: we can enforce inheritance and interfaces on generic types.

How do we do this:

- class Foo<E extends SomeInterface> { …

In this case: “SomeInterface” is just some arbitrary interface. Notice the **extends** keyword. Remember extends is the keyword for inheritance.  Yet what is being extended is an interface.

You cannot use implements instead of extends, the compiler will get mad at you.

# Generics (cont.)

Revisiting the question: How do we compare obj1 to obj2? Java provides common interface called: **Comparable**.

- Comparable requires the implementation of a method called: **compareTo**
  - The method signature is:
    - int compareTo(T o)
  - Notice the input parameter is a generic type
- To implement the comparable interface, you must also specify what you are comparing
  - This is because the Comparable interface also requires a generic type specification
- Let's look at Java documentation...

# Generics (cont.)

An example of using the comparable interface with generics is:

- class Foo<E extends Comparable<E>> { …

Let’s look at another example...

# Generics (cont.)

We can require that a generic type inherits from or implements any: Class, Abstract Class, and / or Interface.

Remember you can inherit from **only one class ,** but you can implement multiple interfaces.  Here is an example of the syntax for doing that for a generic type:

- class Foo<E extends SomeClass & SomeInterface1 & SomeInterface2> {...

# A quick review

The next half this course will begin to introduce you to **Data Structures and Algorithms**.

You really do need to understand everything in the first half of this course in order for the second half to make any sense.  So we are going to review key concepts.

# Review

- The **class** construct defines a new type that can group data and methods to form an object.
  - Here is the syntax to create a class:
    - public class Foo {}
    - This create a class called Foo
- An **object** is an instance of a class.
  - Here is the syntax to create a Foo object (also known as an instance of Foo):
    - Foo myVarName = new Foo();

# Review (cont.)

- The **new** operator in the previous example allocates memory for an object of the specified class type, and returns a reference type to that allocated memory.
    - Foo myVarName = new Foo();
- Remember, a **reference type** is a variable type that refers to an object, it can be thought of as storing the memory address of an object

# Review (cont.)

- Class can contain methods and fields
- A **method** is a statement, or more specifically a block of code, that performs a specific task and (usually) returns the result to the caller of the method.
    - An example of a method with in a class could be:

```
class Foo {
        public int addFive( int x ) {
                return x + 5;
        }
}
```

# Review (cont.)

- To access a field or invoke a method you must use the **"." (dot)** operator, known as the **member access operator,** on an object.
  - For example:
    - First: (using the example on the previous slide) create a Foo object:
      - Foo myVarName = new Foo();
    - Second: Using the Foo object you just created, invoke the “addFive” method using the “.” (dot) operator:
      - myVarName.addFive( 5 );
    - The result returned is: 10

# Review (cont.)

- A **constructor** is a block of code with in a class that is called when an object is created, and is only called once.

```
class Foo {
	public int x; // x is declared but not initialized
	public Foo( int x ) {
		this.x = x; // This line only runs once, when an object is first created
	}
}
```

Foo myVarName = new Foo( 5 );  // This line creates a new object, and therefore calls the above constructor...

# Review (cont.)

- **Encapsulation** is defined as the wrapping up of data and code (variables and methods) under a single unit (a class)
    - Information hiding is apart of this concept of encapsulation, and involves the the hiding of that wrapped up data
    - An example of information hiding is a private variable within a class
- **Abstraction** means to have a user interact with an item at a high-level, with lower-level internal details hidden from the user
    - Essentially it is how the user knows how to use the classes that you create
    - A specific abstraction that relates to hidden information, is Accessors / Mutators (i.e. Getters, and Setters)

# Review (cont.)

- A getter and a setter for a variable might look like this:

```
class Foo {
	private int x = 5;

	public int getX( ) { return x; }
	public void setX( int x ) { this.x = x; }
}
```

Notice the x variable is private (**information hiding**) , and the getter and setter for that x variable are public (**Abstraction**, the abstraction of getters and setters has provided the user of your class access to the hidden variable: x )

# Review (cont.)

- **Derived class** (or **subclass**) refers to a class that is derived from another class that is known as a **base class** (or **superclass**)
- The derived class is said to inherit the properties of its base class, concept commonly called **inheritance**
- “Inherit the properties” means the derived class has access to all of the same public members as its corresponding base class

# Review (cont.)

Here is an example of inheritance:

```
class Foo {
        protected int x = 5;
        public void print() {
                System.out.println( x );
        }
}
```

```
class Bar extends Foo {
        /* Bar has now inherited the field: x  ,
         * and the method: print()
}

class Main {
        public static void main (String[] args) {
                Bar bar = new Bar();
                bar.print();
        }
}
```

# Review (cont.)

- A derived class may provide its own definition for a base class method, this is known as: **Overriding**, it does this by using the annotation: **@Override**