# CSCI 2302 Test 2 Concepts

## Chapters: Object & Classes, Object-Oriented Thinking, UMLs, Inheritance & Polymorphism, Abstract Classes & Interfaces

## Pillars of OOP:

**Data encapsulation** – What is it? Why do we use it?

- To protect the data by using the `private` modifier
- To prevent data from being tampered with
  - To access or modify, have to use the setters & getters
- makes the class/object easy to maintain as each part is compartmentalized

**Class Abstraction** - What is it? Why do we use it?

- It is the separation of class implementation and the use of the class.
- A concept where the user does not need to know how the object is defined/implemented to be able use & the programmer does not need to know how the user will use the object
  - An end-user using the application need not be concerned about how a particular feature is implemented. He/she can just use the features as required.
  - the user will only know "what it does" rather than "how it does"
- By implementing data encapsulation, we get class abstraction.
- One advantage of this approach is that we can change the implementation anytime without changing the behavior that is exposed to the user.

**Inheritance**: What is it?

- Defining a new class from existing classes; super/parent – sub/child classes
- What kind does Java have?
  - single
- Constructor chaining – all objects inherit from the defined Java Object class
- When constructing a new object of the base class, it constructs an object of the super class first (and so on along the chain of inheritance) before constructing the object
- when constructing an object of a subclass, the superclass' constructor is invoked and continues up the superclasses until the last constructor is called
- `super`, `extends`
  - `super`: constructor, calling the super/parent class
  - `extends`: keyword for inheritance

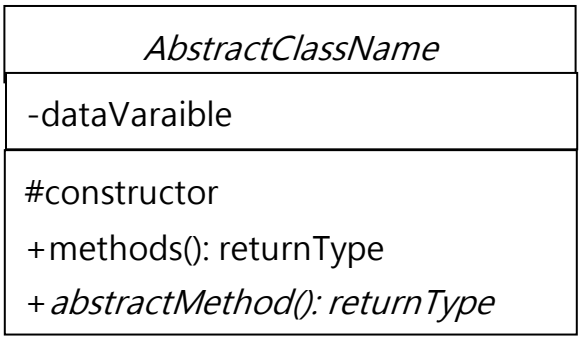
- Overriding methods
    - Overriding: using the same method signature as defined in the superclass in the subclass
        - `@Override`
            - `toString`, `equals` methods
    - Overloading: having the same name but different arguments
    - `final` – if added to class prevents extending and if added to method header prevents overriding

**Polymorphism**: that a variable of a supertype can refer to a subtype; an inheritance relationship that enables a subclass to inherit features from its superclass with additional new features → can pass an instance of a subclass to a parameter of its superclass

- Be able to identify polymorphic behavior, polymorphic calls, and methods
- `instanceof`
    - be able to identify the chain of inheritance – what it is an instanceof *AND USE* it
- Casting
    - implicit casting
        - `SuperClass superclassObject = new SubClass();`
    - explicit casting
        - `SubClass subclassObject = (SubClass) superclassObject;`
- dynamic binding vs static binding: : a method can be implemented in several classes along the inheritance chain. The JVM decides at runtime which method is invoked based on the actual class of the object
    - dynamic: runtime – e.g. the JVM matching the `toString` method from the inheritance chain of classes to use
    - static: compile time – i.e the compiler matching the method header with the invoking statements

**Abstract Classes**: cannot be used to create objects; identifies common behavior for related subclasses

- What is it?
    - A superclass that contains abstract methods
- UML diagram of it



- Abstract method – where is it on the UML diagram
    - On the Abstract Class' UML
    - abstract classes are *italicized*
- Superclasses define common behavior for related subclasses
- Subclasses become more specific and concrete with each new subclass
- Abstract classes is a super class (related) that cannot create any specific instance
- Concrete classes = classes that are not abstract
- CANNOT create any specific instance – CANNOT be instantiated using the `new` operator
    - but can be a data type (think polymorphism)

      Ex:
      ```
      GeoObject[] obj = new GeoObject[10];
      Obj[0] = new Circle();
      ```
      Where `GeoObject` is abstract and `Circle` is concrete
- a subclass can be abstract even if the superclass is concrete
- can have concrete methods (methods with a body)
- since it allows the concrete methods, it may be stated as partial abstraction
    - since it allows the concrete methods, it may be stated as partial abstraction (interfaces provide 100% abstraction)
- have abstract methods without implementation
- do not have to have methods (other than the constructor)
- the constructor is protected (because it is used by subclasses only)
    - when an instance of the concrete subclass constructor is invoked the superclass's constructor is invoked to initialize data fields defined in the superclass
- state abstract: `public abstract class NewObjectClass`

```
public abstract class Ball{
    public abstract int hit(int batSpeed);
}
```

**Interfaces**: contains (only constants) and abstract methods; identifies common behavior for ***UN***related subclasses

- What is it?
    - a class-like construct that contains only constants and abstract methods
    - common behavior
- implements
    - keyword to inherit from an Interface
- `Cloneable`: interface that makes clones of the objects – specifies that an object can be cloned * have to state implements `Cloneable` ** deep v shallow copy

- `Comparable`: interface that compares objects - defines the `compareTo` method for comparing objects * have to state implements Comparable

**Abstract Classes & Interfaces**

- Superclasses define common behavior for related subclasses
- Subclasses become more specific and concrete with each new subclass
- Interfaces define common behavior for classes, related or not
- Abstract classes is a super class (related) that cannot create any specific instance
- Concrete classes = classes that are not abstract – do not have abstract methods
- CANNOT create any specific instance – CANNOT be instantiated using the `new` operator
    - but the Abstract Class can be a data type

      Ex:
      ```
      GeoObject[] obj = new GeoObject[10];
      obj[0] = new Circle();
      ```
      Where `GeoObject` is abstract and `Circle` is concrete
- an Abstract class subclass can be abstract even if the superclass is concrete
- an Abstract class can have concrete methods (methods with a body)
- have abstract methods without implementation
- In an Abstract class the constructor is protected (because it is used by subclasses only)
    - when an instance of the concrete subclass constructor is invoked the superclass's constructor is invoked to initialize data fields defined in the superclass
- Abstract class has to state abstract: `public abstract class NewObjectClass`

```
public abstract class Ball{
    public abstract int hit(int batSpeed);
}
```

- in the UML diagram – abstract class names are *italicized* and abstract methods are *italicized*

**Abstract methods:**

- are common methods used in the subclasses
- have no implementation/no body
    - the implementation is in the subclass that uses it **overridden**
        - all abstract methods must be overridden
    - the JVM dynamically determines which method to invoke at runtime, depending on the actual object that invokes the method (dynamic polymorphism)
    - Always end the declaration with a **semicolon**(;)
- if there is an abstract method – then the class HAS TO BE abstract class

- an abstract method CANNOT be contained in a non-abstract class

- if a subclass of an abstract class does not implement all the abstract methods, the subclass MUST BE defined as abstract

```
public class BaseBall extends Ball{
    public int hit(int batSpeed)    {
       // code that implements the hit method goes here   }}
```

- in the UML diagram – abstract methods are *italicized*
  - Superclass methods are generally omitted in the UML diagram for subclasses

**Remember two rules for Abstract Classes & Interfaces**:

1) If the class is having few abstract methods and few concrete methods: declare it as abstract class.

2) If the class is having only abstract methods: declare it as interface.

- A class can implement multiple interfaces but only extend 1 superclass (inheritance)

| | *Variables* | *Constructors* | *Methods* |
|---|---|---|---|
| Abstract Class | No restrictions | Constructors are invoked by subclasses through constructor chaining. | No restrictions |
| Interface | All variables must be `public static final` | No constructors | Must be `public abstract` instance methods |

- Strong *is-a* relationship = classes
- Weak *is-a* relationship = interfaces (also *can-do* relationship)