

# 15-440

# Distributed Systems

# Recitation 1

Tamim Jabban

# Office Hours



Office 1004



Sunday, Tuesday: 9:30 - 11:59 AM

Appointment: send an e-mail

Open door policy

# Java: Object Oriented Programming

- A programming paradigm based on **objects**
- An example of an Object template:

```
public class Animal {  
}
```

# Java: Object Oriented Programming

- A programming paradigm based on **objects**
- An **Object** can contain data/*attributes/fields*:

```
public class Animal {  
    String name;  
    Integer age;  
    ...  
}
```

# Java: Object Oriented Programming

- A programming paradigm based on **objects**
- An **Object** can contain *methods (behavior)*:

```
public class Animal {  
    ...  
    String name;  
    public String getName() {  
        return name;  
    }  
}
```

# Java: Object Oriented Programming

- A programming paradigm based on **objects**
- To create an **Animal** Object:

```
Animal bear = new Animal();
```

# Constructors

- Constructors take in **zero or more** variables to create an **Object**:

```
public class Animal {  
    String name;  
    int age;  
    public Animal () {  
    }  
}
```

← Constructor

```
Animal bear = new Animal();
```

# Constructors

- Constructors take in **zero or more** variables to create an **Object**:

```
public class Animal {  
    String name;  
    int age;  
    public Animal(String name, int sAge) {  
        name = name; ← Problem!  
        age = sAge;  
    }  
}
```



# Constructors

- Constructors take in **zero or more** variables to create an **Object**:

```
public class Animal {
    String name;
    int age;
    public Animal(String name, int sAge) {
        this.name = name;
        age = sAge;
    }
}

Animal bear = new Animal("Bear", "21");
```

# Inheritance

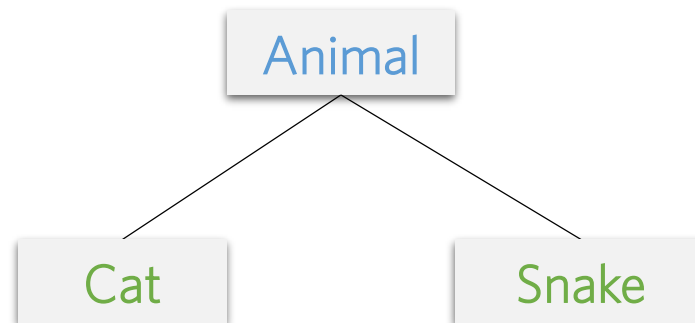
- Enables one object to inherit *methods* (behavior) and *attributes* from another object.
- For example, a **Cat** class can **extend** an **Animal** class:

```
public class Cat extends Animal    {  
    int numOfMugsPushed;  
    String favoriteHuman;  
}
```

- **Cat** inherits **name**, **age** & **getName** from **Animal**.

# Inheritance : Class Hierarchy

- This introduces **subclasses** and **superclasses**.
- A class that *inherits* from another class is called a **subclass**:
  - **Cat** *inherits* from **Animal**, and therefore **Cat** is a **subclass**.
- The class that is *inherited* is called a **superclass**:
  - **Animal** is *inherited*, and is the **superclass**.



# Inheritance

- Organizes related classes in a hierarchy:
  - This allows reusability and extensibility of common code
- Subclasses extend the functionality of a superclass
- Subclasses inherit all the methods of the superclass (*excluding constructors and privates*)
- Subclasses can **override** methods from the superclass (*more on this later*)

# Java Workspace Hierarchy

*Project name*

*Package name*

*Class name*

The screenshot shows the Eclipse IDE interface. The Package Explorer on the left displays a project named 'Automobiles' (highlighted with a green box). Inside it is a package named 'src' (highlighted with a brown box), which contains a class named 'BMW' (highlighted with a brown box). Under 'BMW' is a file named 'BMW\_i8.java' (highlighted with a blue box). The main editor window shows the code for 'BMW\_i8.java' with the following content:

```
1 package BMW;  
2  
3 public class BMW_i8 {  
4  
5 }  
6
```

# Access Control

Access modifiers describe the accessibility (scope) of data like:

- Attributes:

```
public String name;
```

- Methods:

```
public String getName() { ... }
```

- Constructors:

```
private Student(String name, int sAge) { ... }
```

# Access Control

- Access modifiers include:
  - Public
  - Protected
  - Private

# Access Control

- Access modifiers include:
  - Public
  - Protected
  - Private



# Access Control

- Access modifiers include:

- Public:

Allows the **access** of the object/attributes/methods from **any other** program that is using this object:

```
public class Animal {  
    ...  
    public void setName(String newName) {  
        this.name = newName;  
    }  
}  
  
public class Test {  
    public static void main(String[] args) {  
        Animal foobar = new Animal();  
        foobar.setName("Foo Bar");  
    }  
}
```

# Access Control

- Access modifiers include:
  - Public
  - Protected
  - Private

# Access Control

- Access modifiers include:
  - Protected:
    - You can use this only in the following
      - Same class as the variable,
      - Any subclasses of that class,
      - Or classes in the same **package**.
  - A **package** is a group of related classes that serve a common purpose.

# Access Control

- Access modifiers include:
  - Public
  - Protected
  - Private

# Access Control

- Access modifiers include:
  - Private:

Restricted even further than a protected variable: you can use it **only in the same class**:

```
public class Animal {  
    ...  
    private void setName(String newName) {  
        this.name = newName;  
    }  
    public Student(String name) {  
        setName(name);  
    }  
}  
public class Test {  
    public static void main(String[] args) {  
        Animal foobar = new Animal();  
        foobar.setName("Foo Bar"); // Not accessible anymore!  
    }  
}
```

# Object & Class Variables

- Each **Animal** object has its own **name**, **age**, etc...
  - **name** and **age** are examples of Object Variables.
- When an attribute should describe an **entire class** of objects instead of a specific object, we use **Class Variables** (or **Static Variables**).

# Object & Class Variables

- A Class Variable Example:

```
public class Animal {  
    public static final String currentPlanet = "EARTH";  
}
```

```
public class Test() {  
    public static void main(String[] args) {  
        Animal foobar = new Animal();  
        String planet = foobar.currentPlanet;  
    }  
}
```

# Object & Class Variables

- A Class Variable Example:

```
public class Animal {  
    public static final String currentPlanet = "EARTH";  
}
```

```
public class Test() {  
    public static void main(String[] args) {  
        String planet = Animal.currentPlanet;  
    }  
}
```



# Encapsulation

- Encapsulation is restricting access to an object's components.
- How can we change or access `name` now?:

```
public class Animal {  
    private String name;  
    private int age;  
  
}  
  
Animal foobar = new Animal();
```

# Encapsulation

- Encapsulation is restricting access to an object's components.
- Using getters and setters:

```
public class Animal {  
    private String name;  
    private int age;  
    public void setName(String newName) {  
        this.name = newName;  
    }  
    public String getName() {  
        return name;  
    }  
}  
  
Animal foobar = new Animal();  
foobar.setName("Foo Bar");
```

# Overloading Methods

- Methods overload one another when they have the same method name but:
  - The **number of parameters** is different for the methods
  - The parameter **types** are different

- **Example:**

```
public void changeDate(int year) {  
    // do cool stuff here  
}
```

```
public void changeDate(int year, int month) {  
    // do cool stuff here  
}
```

# Overloading Methods

- Methods overload one another when they have the same method name but:
  - The **number of parameters** is different for the methods
  - The parameter **types** are different
- Another Example:

```
public void addSemesterGPA(float newGPA) {  
    // process newGPA  
}
```

```
public void addSemesterGPA(double newGPA) {  
    // process newGPA  
}
```

# Overloading Methods

- Methods overload one another when they have the same method name but:
  - The **number of parameters** is different for the methods
  - The parameter **types** are different

- **Another Example:**

```
public void changeDate(int year) {  
    // do cool stuff here  
}
```

```
public void changeDate(int month) {  
    // do cool stuff here  
}
```

# Overloading Methods

- Methods overload one another when they have the same method name but:
  - The **number of parameters** is different for the methods
  - The parameter **types** are different

- **Another Example:**

```
public void changeDate(int year) {  
    // do cool stuff here  
}
```

```
public void changeDate(int month) {  
    // do cool stuff here  
}
```

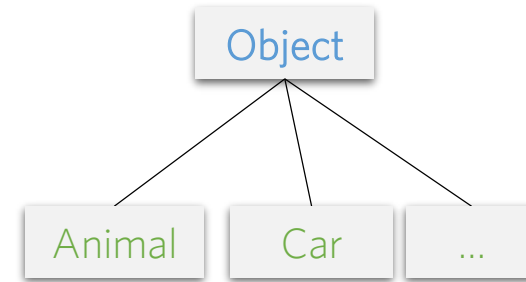
We can't overload methods by just changing the parameter name!

# Overriding Methods

- Example:

```
public class ClassA {  
    public Integer someMethod() {  
        return 3;  
    }  
}  
  
public class ClassB extends ClassA {  
  
    // this is method overriding:  
    public Integer someMethod() {  
        return 4;  
    }  
}
```

# Overriding Methods



- Any class extends the **Java** superclass “**Object**”.
- The Java “**Object**” class has 3 important methods:
  - `public boolean equals(Object obj);`
  - `public int hashCode();`
  - `public String toString();`
- The **hashCode** is just a number that is generated by any object:
  - It **shouldn't** be used to compare two objects!
  - Instead, **override** the `equals`, `hashCode`, and `toString` methods.



# Overriding Methods

- Example: Overriding the `toString` and `equals` methods in our `Animal` class:

```
public class Animal {  
    ...  
    public String toString() {  
        return this.name;  
    }  
}
```

# Overriding Methods

- Example: Overriding the `toString` and `equals` methods in our `Animal` class:

```
public class Animal {  
    ...  
    public boolean equals(Object obj) {  
        if (obj.getClass() != this.getClass())  
            return false;  
        else {  
            Animal s = (Animal) obj;  
            return (s.getName().equals(this.name));  
        }  
    }  
}
```

# Abstract Classes

- A class that is **not completely implemented**.
- Contains one or more *abstract* methods (methods with no bodies; *only signatures*) that subclasses must implement
- Cannot be used to instantiate objects
- Abstract class header:

```
accessModifier abstract class className  
public abstract class Car
```

- Abstract method signature:

```
accessModifier abstract returnType methodName ( args );  
public abstract int max_speed ();
```

- Subclass signature:

```
accessModifier class subclassName extends className  
public class Mercedes extends Car
```

# Interfaces

- A **special abstract class** in which *all the methods are abstract*
- Contains only abstract methods that subclasses **must implement**
- Interface header:

```
accessModifier interface interfaceName  
public          interface Car
```

- Abstract method signature:

```
accessModifier abstract returnType methodName ( args );  
public         abstract String      CarType   ( args );
```

- Subclass signature:

```
accessModifier class subClassName implements someInterface  
public          class BMW          implements Car
```

# Generic Methods

- *Generic* or *parameterized* methods receive the data-type of elements as a parameter
- E.g.: a generic method for sorting elements in an array (be it **Integers, Doubles, Objects** etc.)

# A Simple Box Class

- Consider this non-generic **Box** class:

```
public class Box {  
    private Object object;  
    public void set(Object object) {  
        this.object = object;  
    }  
    public Object get() {  
        return object;  
    }  
}
```

# A Simple Box Class

- A *generic class* is defined with the following format:

```
class my_generic_class <T1, T2, ..., Tn>
{
    /* ... */
}
```

Type parameters

# A Simple Box Class

- Now to make our **Box** class *generic*:

```
public class Box<T> {  
    // T stands for "Type"  
    private T t;  
    public void set(T t) {  
        this.t = t;  
    }  
    public T get() {  
        return t;  
    }  
}
```

- To create, for example, an **Integer** "Box":

```
Box<Integer> integerBox;
```



# Java Generic Collections

- Classes that represent data-structures
- *Generic* or *parameterized* since the elements' data-type is given as a parameter
- E.g.: LinkedList, Queue, ArrayList, HashMap, Tree
- They provide methods for:
  - Iteration
  - Bulk operations
  - Conversion to/from arrays

## Class LinkedList<E>

```
java.lang.Object
  java.util.AbstractCollection<E>
    java.util.AbstractList<E>
      java.util.AbstractSequentialList<E>
        java.util.LinkedList<E>
```

### Type Parameters:

E - the type of elements held in this collection

### All Implemented Interfaces:

Serializable, Cloneable, Iterable<E>, Collection<E>, Deque<E>, List<E>, Queue<E>

```
public class LinkedList<E>
  extends AbstractSequentialList<E>
  implements List<E>, Deque<E>, Cloneable, Serializable
```