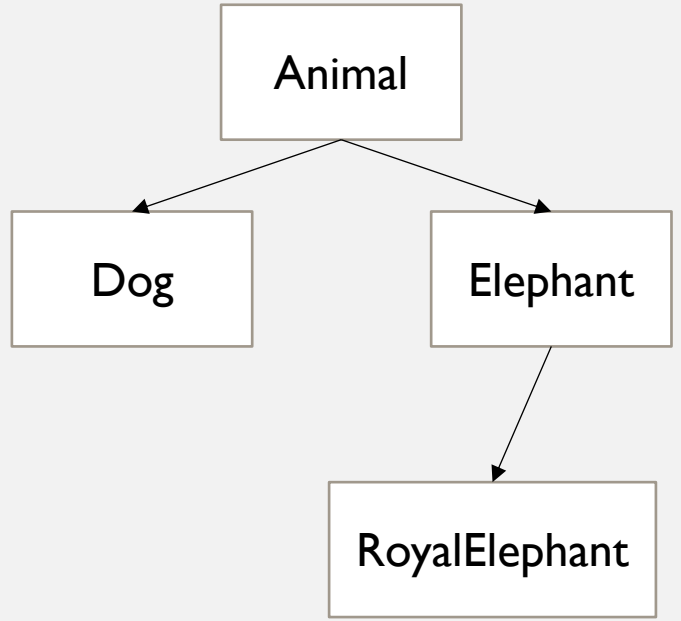
JAVA

Lecture VI – Inheritance and Interface

RECAP: INHERITANCE

- A class can inherit methods and data from another class.
- The more general class is called "superclass".
- The more specific one is called "subclass".
- Subclass inherit all the data and methods from the superclass.
- This relationship is declared by the **extends** keyword.
- A subclass can have additional members.
- However, a superclass has no knowledge of its subclasses.
- Each superclass can have multiple subclasses.



RECAP: CONSTRUCTOR

- The constructor of a subclass will automatically call the default constructor of its superclass

```
class A{
      int value;
      A(){
            value = 1;
      }

}
```

```
class B extends A{
      B(){
            //value = 1;
            value = 2;
      }

}
```

Which constructor will be used? What is the value of `b.value;`

RECAP: SUPER

- How to call the constructor of its superclass?

```
super(parameter-list)
```

- super() must be called as the first statement.

```
class A{
      int value;
      A(int n){
            value = n;
      }
}
```

```
class B extends A{
      int value2;
      B(int n, int m){
            super(n);
            value2 = m
      }
}
```

SUPER KEYWORD

- ```
  class Parent{
      int i;
  }
  ```

```
class Child extends Parent{
    int i;
    Child(int n){
        i = n; // which i?
    }
}
```
```

SUPER KEYWORD

- What if we want to change the value of i in Parent?

```
class Parent{
    int i;
}
```

```
class Child extends Parent{
    int i;
    Child(int n, int m){
        i = m;
    }
}
```

- Remember how **this** is used, can we do the following:

```
super.member
```

SUPER KEYWORD

- What if how we want to change the value of i in Parent?

```
class Parent{
    int i;
}
```

```
class Child extends Parent{
    int i;
    Child(int n, int m){
        super.i =n;
        i = m;
    }
}
```

SUPERCLASS AND SUBCLASS OBJECTS

- Java is a strongly typed language.

```
X x = new X();
X x2;
Y y = new Y();

x2 = x;  // will it work?
x2 = y;  // will it work?
```

SUPERCLASS AND SUBCLASS OBJECTS

- Java is a strongly typed language.

```
X x = new X();
X x2;
Y y = new Y();

x2 = x;  // will it work?
x2 = y;  // will it work?
```

- It works only when Y is a subclass of X.

SUPERCLASS AND SUBCLASS OBJECTS

- Java is a strongly typed language.

```
X[] xs   = new X[2];
xs[0] = x;
xs[1] = y;

Y y2 = xs[1];          // will it work?
xs[1].b = 27;          // will it work?
```

SUPERCLASS AND SUBCLASS OBJECTS

- Java is a strongly typed language.

```
X[] xs   = new X[2];
xs[0] = x;
xs[1] = y;

Y y2 = (Y) xs[1];          // will it work?
y2.b = 27;                 // will it work?

// to check the real type
if( xs[1] instanceof Y){
   Y y2 = (Y) xs[1];
}
```

OVERRIDING

- Polymorphism: overload and override.
- Method Overriding:
- When? a method is declared in both a superclass and a subclass. i.e., same name, return type and same input parameters.
- What happens? When this method is called from the subclass, it will refer to the one defined in the subclass.
- The decision is made at runtime.

EXAMPLE

```
class Vehicle{
	public void move(){
		System.out.println(“All vehicles move”);
	}
}
class Boat extends Vehicle{
	public void move(){
		System.out.println(“Boats float”);
	}
}
```

EXAMPLE

```
class Vehicle{
	public void move(){
		System.out.println("All vehicles move");
	}
}
class Boat extends Vehicle{
	public void move(String s){
		System.out.println("Boats " + s);
	} // method override here?
}
```

EXAMPLE

```
class VehicleDemo{
      public static void main(String[] args)
            Vehicle a = new Vehicle();
            Vehicle b = new Boat();
            Boat c = new Boat();

            a.move();
            b.move();
            c.move();
      }     }
```

EXAMPLE

```
class VehicleDemo{
      public static void main(String[] args)
            Vehicle a = new Vehicle();
            Vehicle b = new Boat();
            Boat c = new Boat();

            a.move(); // “All vehicles move”
            b.move(); // “Boats float”
            c.move(); // “Boats float”
      }     // What are the differences?
}
```

EXAMPLE

Compile-time:

- You have a dog and you tell the compiler this is an animal.

```
Animal x = new Dog();    // It succeeds as a dog is indeed an
                            animal.
```

- You then ask the compiler to access the value of the dog type from x.

EXAMPLE

Compile-time:

- You have a dog and you tell the compiler this is an animal.

```
Animal x = new Dog();    // It succeeds as a dog is indeed an
                            animal.
```

- You then ask the compiler to access the value of dog type from x.
- It fails (errors at compile-time), as the compiler does not know where to find the dog type for an animal.
- Each type of animals has its own voice.

```
x.getVoice();
```

EXAMPLE

Compile-time:

- You have a dog and you tell the compiler this is an animal.

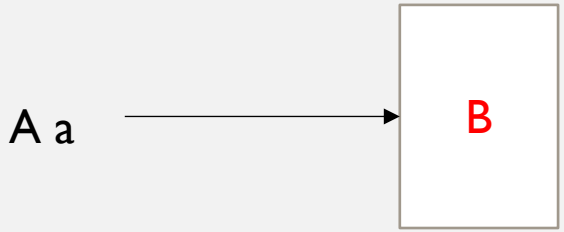
```
Animal x = new Dog();    // It succeeds as a dog is indeed an
                            animal.
```

- You then ask the compiler to access the value of dog type from x.
- It fails (errors at compile-time), as the compiler does not know where to find the dog type for an animal.
- Each type of animals has its own voice.

```
x.getVoice(); // method overridden at run time
              // dog barks, if we want its voice
```

OVERRIDING

- Dynamic method dispatch: a call to an overridden method is resolved at run time rather than compile time. (runtime polymorphism).
- B extends A
- `A a = new B();`
- `a.method()`
- When an overridden method is called through a superclass, Java determine which version of the object being referred to at run time. (i.e., the call determined by the type it refers to but not the type of the reference variable)



OVERLOADING VS OVERRIDING

When	Compile-time	Runtime
Where	Within a class	Two classes with inheritance relationship
Inheritance	Not necessary	Always required
Parameters	Must be different	Must be the same
Return type	Doesn't matter	Must be the same

STATIC VS NON-STATIC

- Static method: is called according to the type of class, not according to the type of object it refers to, i.e., decided at compile time.

- Non-static method: is called through the object, not according to the class type, i.e., decided at run-time

- Thus, we CANNOT override static method.

ABSTRACTION

- Abstraction: provide high-level specifications but hide the detail implementations.
- Why we need abstractions?
- Abstraction: when a superclass is unable to create a meaningful implementation

- Example: 2D shape is an abstract concept, a method to calculate its area does not make any sense.
- However, we can have methods to calculate the area of a triangle.

- Both the classes and the methods can be abstract in Java.

ABSTRACT METHODS

- Abstraction: declare methods that must be overridden by its subclass.
- Keywords **abstract** is used to declare abstract classes and methods.
- Abstract method: only specify the return type, input parameters and the name.

```
abstract type name(parameter-list);
```

- Properties:
  - An abstract method can only be declared within an abstract class.
  - Static methods can not be declared as abstract (Why?)
  - Abstract methods must be implemented in its subclass (True or false?)

ABSTRACT CLASSES

- Abstract Class:
  - Can contain both abstract and concrete methods.
  - Cannot be instantiated (no object of such class can be generated).
  - When a class inherits an abstract class, it must implement all the abstract methods. (What happens if it does not)

```
abstract class TwoDShape{

      …

      abstract double area();

}
```

FINAL KEYWORDS

- How to prevent overriding? Use the keywords **final**.

```
class A{
        final void method(){} // this method cannot be overridden
}
```

- Moreover, **final** can be used to prevent inheritance, i.e., final class.
- All methods in a final class are declared (implicitly) as final
- **final** can also be applied to variables. (What does it means?)

ABSTRACTION

- Abstract Classes: What a class must do
- Concreate Classes: How it will be done
- What are the problems of the abstract classes?

- Interface: fully separate the specification from the implementation.
  - Does not allow concreate methods.
  - An interface can be implemented by multiple classes.
  - One class can implement multiple interfaces, e.g., Comparable, Cloneable, …

INTERFACE

- How to create an interface:

```
access interface name{
        return-type method-name(param-list);
}
```

- The access modifier private cannot be used (how about abstract classes?)
- Methods are mostly abstract (what are the exceptions?).

INTERFACE

- How to create an interface:

```
access interface name{
        return-type method-name(param-list);
}
```

- The access modifier private cannot be used (how about abstract classes?)
- Methods are mostly abstract (what are the exceptions?).
- Static method, SE 8 or later, must have method body

INTERFACE

- How to implement an interface:

```
class name extends superclass implements interface1, interface2{
        // class-body
}
```

- All the non-static and non-default method specified in the interface must be implemented in any of the concrete classes that implements the interface.
- All the implemented methods must be declared as public. (all abstract methods in interfaces are public, so we cannot reduce the visibility)
- Return type and the input parameters must match the specification.

EXAMPLE

```
public interface Series{
    int getNext();
    void reset();
}
```

```
class EvenNum implements Series{
    int start, val;
    EvenNum(){
      start = 0;
    }
    public int getNext(){
      val += 2;
      return val;
    }
    public void reset(){
      val = start;
    }
}
```

INTERFACE REFERENCES

- When we create an object, an instance of such class is generated.
- `class_type name = new class_type();`
- What happens if this class implements a particular interface?
- `interface name = new class_type();`
- The new object is also an instance of the interface;
- `interface name;`
- name here is an interface reference variable which can be assigned with an object of any class that implements this interface.
- However, you cannot new an interface

INTERFACE REFERENCES

- Why the interface type useful?
- Example: class ByThree implements Series
- Array can be used to store objects that implements the same interface.
- What about abstract classes?
- If an object of class X is declared as an interface that X implements, can it call the methods unique to X?
- Given a list of objects that implements one particular interface, how to recognise them?
    - Instanceof

MULTIPLE INTERFACES

- Java class cannot inherit multiple classes. (Why?)
- However, a class can implement multiple interface.

```
public class Being implements canJump, canFly{
        ...
}
```

- Is it safe to implement multiple interface? Any ambiguous?

CONSTANTS AND EXTENDED INTERFACE

- Interface can only have variables but they are implicitly **public**, **final** and **static** ,and must be initialized.
- Interface inherit another interface by using the keyword **extends**.

```
public interface A{
        public void doA();
}
public interface B extends A{
        public void doB();
}
```

ABSTRACT CLASS VS INTERFACE

	Abstract Class	Interface
Methods Type	Both abstract and concrete methods	Only abstract, from java 8 default and static methods
Final Variable	May contain non-final variables	Final, by default
Variable Type	Final, non-final, static, non-static	Static and final
Implementation	Can provide implementation of interface	Interface cannot provide implementation of abstract class
keywords	extends	implements
Multiple implementation	A class can extends one class	A class can implement multiple interfaces
Accessibility	Can have private members	Members are pubic by default

DESIGNING A CLASS

- Coherence:

  A class should describe a single entity, and all the class operations should logically fit together to support a coherent purpose.

  You can use a class for students, for example, but you should usually not combine students and elephant in the same class, because students and elephants are different entities

DESIGNING A CLASS

- Separating responsibilities:
  - A single entity with too many responsibilities can be broken into several classes to separate responsibilities.

- The classes String, StringBuilder, and StringBuffer all deal with strings, for example, but have different responsibilities.
  - The String class deals with immutable strings;
  - The StringBuilder class is for creating mutable strings;
  - StringBuffer class is similar to StringBuilder except that StringBuffer contains synchronized methods for updating strings.

DESIGNING A CLASS

- Follow standard Java programming style and naming conventions.
- Choose informative names for classes, data fields, and methods.
- Always place the data declaration before the constructor, and place constructors before methods.
- Although java provides a default constructor you should always* provide a constructor and initialize variables to avoid programming errors.

DESIGNING A CLASS

- A class should use the **`private`** modifier to hide its data from direct access by clients.
- You can use get methods and set methods to provide users with access to the private data, but only to private data you want the user to see or to modify.
- A class should also hide methods not intended for client use.

SOLID

- There are several guides and sets of principles on how best to design Classes
- **S.O.L.I.D** is one

- **Single Responsibility Principle**
- **Open Closed Principle**
- **Liskov's Substitution Principle**
- **Interface Segregation Principle**
- **Dependency Inversion Principle**

SINGLE RESPONSIBILITY PRINCIPLE

- “One class should have one and only one responsibility”
- “A class should have only one reason to change.”
- A class is written and exists for one purpose
- Don’t try and achieve too much through a single class
- ..this feeds into other principles

OPEN CLOSED PRINCIPLE

- “Software components should be open for extension, but closed for modification”
- Changes to the flow of control in an application need to be achieved by extending the class and overriding some functions and that’s it.
- Don’t modify - instead write a new class

LISKOV'S SUBSTITUTION PRINCIPLE

- "Derived types must be completely substitutable for their base types"
- A method should work for every possible subclass object of SuperClass that is passed to it.
- Or
- "if S is a subtype of T, then objects of type T may be replaced with objects of type S" – Wikipedia
- Or
- A class can be replaced by any of its children (refinement)

INTERFACE SEGREGATION PRINCIPLE

- “Clients should not be forced to implement unnecessary methods which they will not use
- Keep interfaces as small as possible
- For instance
  - I have a print driver with 2 methods
  - One to print from pdf
  - One to print from Word

  - What’s wrong with this?

DEPENDENCY INVERSION PRINCIPLE

- “Entities should depend on abstractions, not on concretions”
- Abstractions should not depend on details.
- Don’t design abstract classes with details in mind
- The details should depend on abstractions
- Refinement
- Classes can change as long as they keep they adhere to the parental boundaries

STRING

- One of the most important data structure in Java.
  - Strings are objects in Java, not primitive type.

- Construct a String:
- `String str = new String("Happy"); // like an object`
- `String str2 = new String(str); // from another string`

- Alternatively
- `String str = "Happy";`

STRING METHODS

- Useful methods that operate on String:
- `boolean equals(Object str)//` return true, if they contains the same character sequence
- `int length() //` return the number of characters
- `char charAt(int index) //` return character at a specified index
- `int compareTo(String str) //` comparison based on Unicode of each character
- `int indexOf(char ch/ string str) //` return the index of the first occurrence of the given character or substring
- `int lastIndexOf(char ch/ string str) //` last index of..

- You can find more here
- https://docs.oracle.com/javase/8/docs/api/java/lang/String.html

IMMUTABLE STRING

- The contents of a String object are immutable.
- Example, replace returns a string resulting from replacing all occurences of oldChar in this string with newChar

```
String replace(char oldChar, char newChar)
String str1 = “Apple”;
str1.replace(’p’, ‘b’); // will it change the value of str1?
String str2 = str1.replace(’p’, ‘b’);
```

IMMUTABLE STRING

- The contents of a String object are immutable.
- Example, replace returns a string resulting from replacing all occurences of oldChar in this string with newChar

```
String replace(char oldChar, char newChar)
String str1 = "Apple";
str1.replace('p', 'b');
String str2 = str1.replace('p', 'b');
```

replace

Once String "Apple" is created, it cannot be changed.

```
str1 = str1.replace('p', 'b');
// update the reference, "Apple" Garbage Collected
```

What if we need to manipulate a string in several steps? StringBuffer, StringBuilder

STRING AND CHAR[]

- Both String and char[] represent a collection of characters, are they the same?

```
String str1 = "Happy";
for(char c : str1){
      ...
} // Will it work?

char[] cs1 = {'a', 'b', 'c'};
char[] cs2 = {'d', 'e', 'f'};
char[] cs3 = cs1 + cs2; // Will it work?
```

STRING VS CHAR[]

- Other differences:

- Data type?
- Immutable?
- Build-in functions?
- Accessing each character?
- Conversions?

STRING VS CHAR[]

- Data type: Single data type vs collections
- Immutable: Immutable vs mutable
- Build-in functions: String has a lot of build-in functions, char[] not.
- Accessing each character: charAt() vs var_name[index]
- Conversions:

```
String s = "Happy";
char[] cs = s.toCharArray();
cs = {'H', 'A', 'P', 'P', 'Y'};
s = new String(cs);
```

TYPE CONVERSION

- Can we read integer and floating numbers from the command-line?

```
java Args 1 2
```

- We want 1 and 2 as integer number!

- Automatic type conversion:
  - Two types are compatible.
  - The destination type is larger than the source type.
  - E.g., byte to int, int to long, long to double, …
- Cast: an instruction to the compiler to convert one type into another

```
(target-type) expression
```

TYPE CONVERSION

```
double x, y;
// …
int z = (int) (x / y);
```

- Narrowing conversion: information might be lost.
- E.g., information lost when we convert long to short

- Example: CastDemo

- How to convert string into integer, double, …

TYPE WRAPPER

- Type Wrapper: classes that encapsulate the primitive types.
- Primitive types: are not objects, e.g., cannot be passed by reference.
- Double, Integer, Float, …

- Numeric wrappers provide methods to convert a string into corresponding number.
- Double.parseDouble(String)
- Integer.parseInt(String)
- Short.parseShort(String)
- …
- Boolean values:
- Boolean.parseBoolean(String)