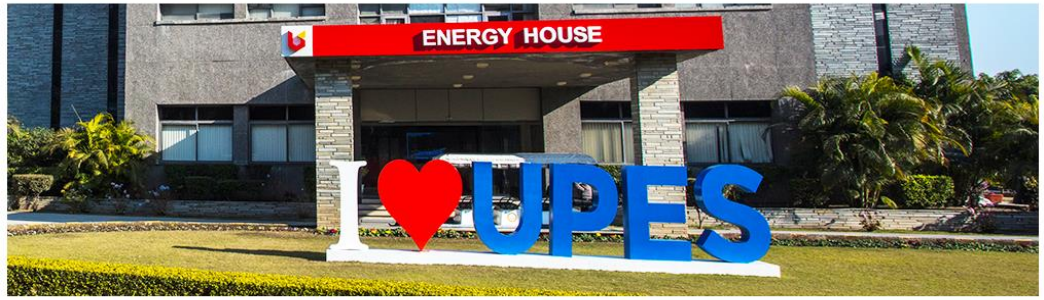

UPES

UNIVERSITY WITH A PURPOSE

Abstract Classes

- There are situations in which we want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method.
- Sometimes we want to create a superclass that only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details.
- Such a class determines the nature of the methods that the subclasses must implement.
- One way this situation can occur is when a superclass is unable to create a meaningful implementation for a method.

Abstract Methods

- Certain methods can be overridden by subclasses by specifying the **abstract** type modifier.
- These methods are sometimes referred to as **subclasser responsibility** because they have no implementation specified in the superclass.
- Thus, a subclass must override them—it cannot simply use the version defined in the superclass.

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

- Any class that contains one or more abstract methods must also be declared abstract.
- To declare a class abstract, you simply use the abstract keyword in front of the class keyword at the beginning of the class declaration.

Abstract Methods

- There can be no objects of an abstract class. That is, an abstract class **cannot be directly instantiated** with the new operator.
- Such objects would be useless, because an abstract class is not fully defined.
- Also, you cannot declare abstract constructors, or abstract static methods.
- Any subclass of an abstract class must either implement all of the abstract methods in the superclass, or be declared abstract itself.

Abstract class and method

```
// A Simple demonstration of abstract.
abstract class A {
abstract void callme();
// concrete methods are still allowed in abstract classes
void callmetoo() {
System.out.println("This is a concrete method."); } }
class B extends A {
void callme() {
System.out.println("B's implementation of callme."); } }
class AbstractDemo {
public static void main(String args[]) {
B b = new B();
b.callme();
b.callmetoo();
}
}
```

Note:

- No objects of class **A** are declared in the program. As mentioned, it is not possible to instantiate an abstract class.
- Class **A** implements a concrete method called **callmetoo( )**. This is perfectly acceptable.
- Although abstract classes cannot be used to instantiate objects, they can be used to create **object references**, because Java's approach to run-time polymorphism is implemented through the use of superclass references.
- Thus, it must be possible to create a reference to an abstract class so that it can be used to point to a subclass object.

UPES
UNIVERSITY WITH A PURPOSE

Abstract class and method (cont.)

```
// Using abstract methods and classes.
abstract class Figure {
double dim1; double dim2;
Figure(double a, double b) {
dim1 = a; dim2 = b; }
// area is now an abstract method
abstract double area();
}
class Rectangle extends Figure {
Rectangle(double a, double b) {
super(a, b); }
// override area for rectangle
double area() {
System.out.println("Inside Area for
Rectangle.");
return dim1 * dim2; }  }
class Triangle extends Figure {
Triangle(double a, double b) {
super(a, b);
}
```

```
// override area for right triangle
double area() {
System.out.println("Inside Area for Triangle.");
return dim1 * dim2 / 2; }  }

class AbstractAreas {
public static void main(String args[]) {
// Figure f = new Figure(10, 10); // illegal now
Rectangle r = new Rectangle(9, 5);
Triangle t = new Triangle(10, 8);
Figure figref; // this is OK, no object is created
figref = r;
System.out.println("Area is " + figref.area());
figref = t;
System.out.println("Area is " + figref.area());
}
}
```

UPES
UNIVERSITY WITH A PURPOSE

Abstract class and method (cont.)

Note:

- Comment inside **main( )** indicates, it is not possible to declare objects of type **Figure**, since it is abstract.
- All subclasses of **Figure** must override **area( )**.
- To prove this to yourself, try creating a subclass that does not override **area( )**. You will receive a **compile-time error**.
- Although it is not possible to create an object of type **Figure**, you can create a reference variable of type **Figure**.
- The variable **figref** is declared as a reference to **Figure**, which means that it can be used to refer to an object of any class derived from **Figure**.
- As explained, it is through superclass reference variables that overridden methods are resolved at run time.

The Object Class

- There is one special class, **Object**, defined by Java.
- All other classes are subclasses of **Object class**. That is, **Object** is a superclass of all other classes.
- This means that a reference variable of type **Object** can refer to an object of any other class.
- Also, since arrays are implemented as classes, a variable of type **Object** can also refer to any array.
- **Object** defines the certain methods, which means that they are available in every object.
- All objects, including arrays, implement the methods of **Object** class.

The Object Class

Method	Purpose
Object clone( )	Creates a new object that is the same as the object being cloned.
boolean equals(Object *object*)	Determines whether one object is equal to another.
void finalize( )	Called before an unused object is recycled.
Class<?> getClass( )	Obtains the class of an object at run time.
int hashCode( )	Returns the hash code associated with the invoking object.
void notify( )	Resumes execution of a thread waiting on the invoking object.
void notifyAll( )	Resumes execution of all threads waiting on the invoking object.
String toString( )	Returns a string that describes the object.
void wait( ) void wait(long *milliseconds*) void wait(long *milliseconds*, int *nanoseconds*)	Waits on another thread of execution.

clone() method

```
protected Object clone() throws CloneNotSupportedException
```

- The **clone() method** saves the extra processing task for creating the exact copy of an object. If we perform it by using the new keyword, it will take a lot of processing time to be performed that is why we use object cloning.

```
class Student implements Cloneable{
int rollno;
String name;
Student(int rollno,String name){
this.rollno=rollno;
this.name=name;
}
public Object clone()throws CloneNotS
upportedException{
return super.clone();
}
```

```
public static void main(String args[]){
try{
Student s1=new Student(101,"amit");

Student s2=(Student)s1.clone();

System.out.println(s1.rollno+" "+s1.name);
System.out.println(s2.rollno+" "+s2.name);

}catch(CloneNotSupportedException c){}
}
}
```

UPES
UNIVERSITY WITH A PURPOSE

References

Schildt, H. (2014). *Java: the complete reference*. McGraw-Hill Education Group.