|  |
| --- |
| **Set 24: Abstract Classes and Interfaces** |

|  |
| --- |
| **Skill 24.01: Explain the purpose of an abstract class**  **Skill 24.02: Implement an abstract class**  **Skill 23.03: Implement the constructor of a super class using the keyword super**  **Skill 23.04: Invoke the methods of a super class in a subclass**  **Skill 23.05: Create objects from super and sub classes**  **Skill 23.06: Pass a sub class object as a parameter** |

|  |
| --- |
| **Skill 24.01: Explain the purpose of an abstract class** |

**Skill 24.01 Concepts**

Suppose that for some reason, we had a programming task that was to model animals. For example, we want to have a class for a cat, a class for a dog, a class for a horse, and so forth. As you can imagine, there could be lots of redundancy among those classes because animals have some common characteristics. For instance, all animals have, say, two eyes, or two ears, and make sounds, and so forth. Or at least all the animals that we would like to model do.

We could create a super class called animal to capture all these common characteristics and behaviors. And then we could have all the specific animal classes derived from the animal class. So far so good…

The question though is, what would new animal mean? In other words, if we create an object of the animal class, what kind of object would that be? What would we try to model by creating an object of the animal class?

You may be thinking there is no point in creating objects from the animal class, so there is no need to ever invoke the animal constructor. But subclasses could benefit from extending such a class.

Java prohibits, under some circumstances, creating objects from specific classes. Such classes are referred to as abstract classes. Abstract classes are classes that cannot be instantiated – that is, objects cannot be created from abstract classes. While the methods abstract classes carry may be implemented, methods declared as abstract cannot be implemented by the abstract class, they must instead be implemented by the subclass. Such methods can provide a framework for a subclass.

* Objects cannot be created from abstract classes
* Methods declared as abstract cannot be implemented by the abstract class, they must instead be implemented by the subclass

|  |
| --- |
| **Skill 24.02: Implement an abstract class** |

**Skill 24.02 Concepts**

Abstract classes are implemented using the keyword *abstract*

In the below example, we have created an abstract class called *animal*

A close-up of a grey rectangular object

Description automatically generated

As previously mentioned, abstract classes can provide the framework for a subclass. For example, consider the following subclasses which inherit the animal class,

A close-up of a grey rectangular object

Description automatically generated

Each of the classes that inherit the parent class, Animal, have some common characteristics. For example, a Cow, a Dog, and a Cat, all have two eyes, two ears, and make sounds, and so forth. But these characteristics are different for each animal we want to model.

Suppose we want to ensure that every animal we model can speak. To do this, we would write an abstract method in our abstract class.

A close-up of a grey rectangular object

Description automatically generated

Notice we have only declared the abstract method, but we did not implement it. In fact, Java does not allow for this. However, the classes that inherit the Animal class *must* implement the speak method.

This is illustrated below,

|  |  |
| --- | --- |
| **Abstract class Animal** | **Class that inherit Animal** |
| public abstract class Animal  {      public abstract String speak();      public abstract String type();  } | public class Cat extends Animal3 {  private String name;    public Cat(String n){         name = n;  }  //overriden  public String speak(){  return "Meow!";  }    public String type(){  return "Mammal";  }    public String toString(){         String msg = "My name is " + name  + " " + speak();          return msg;  }  } |
| **Output** | |
|  |  |

|  |
| --- |
| **A screenshot of a video  Description automatically generated** |
| <https://www.youtube.com/watch?v=5nUPGlN2Ovo> |

In the below example of inheritance, class *Bicycle* is a base class, class *MountainBike* is a derived class which extends Bicycle class and class Main is a driver class to run program.

|  |
| --- |
| **Bicycle** |
| class Bicycle  {    public int gear;    public int speed;      public Bicycle(int gear, int speed)    {        this.gear = gear;        this.speed = speed;    }  } |
| **MountainBike** |
| class MountainBike extends Bicycle  {    //the MountainBike subclass adds one more field    public int seatHeight;      // the MountainBike subclass has one constructor    public MountainBike(int startHeight, int g, int s)    {          // invoking base-class(Bicycle) constructor          super(g, s);          seatHeight = startHeight;  }  } |

In the above example, the *Bicycle* class is referred to as our superclass. The *MountainBike* class is referred to as our subclass. By using the work "extends" in the *MountainBike* class, the *MountainBike* class inherits all the methods and state variables of the *Bicycle* class.

[**Skill 23.02: Exercise 1**](file:///C:\Users\PLUSKH01\Desktop\APCompSciA\ticketOutTheDoor\set19\Set19TicketOutTheDoorAPCompSciA.pdf)

|  |
| --- |
| **Skill 23.03: Implement the constructor of a super class using the keyword super** |

**Skill 23.03 Concepts**

As previously mentioned, the keyword **extends** causes a subclass to inherit all the methods and state variables of its super class. This is also true for the constructor.

To invoke the constructor of a super class the keyword **super** is used. This concept is illustrated below.

|  |
| --- |
| **The use of super to invoke a super class** |
| A screenshot of a computer  Description automatically generated |

There are some significant features of inheritance illustrated above. In the absence of *super(gear, speed)*, the *MountainBike* constructor would have tried to automatically call the Bicycle constructor and would have failed, since the Bicycle constructor requires two parameters and we would not have supplied these. By making *super(gear, speed)* the first line of code, we are able to supply the needed parameters. When used, *super()* must be the first line of code in the constructor of the subclass.

[**Skill 23.03: Exercises 1**](file:///C:\Users\PLUSKH01\Desktop\APCompSciA\ticketOutTheDoor\set19\Set19TicketOutTheDoorAPCompSciA.pdf)

|  |
| --- |
| **Skill 23.04: Invoke the methods of a super class in a subclass** |

**Skill 23.04 Concepts**

Because the *MountainBike* class extends the Bicycle class, *MountainBike* is able to access all the public methods and variables associated with Bicycle. This concept is illustrated below,

|  |
| --- |
| **Invoking the methods of a super class** |
| A screen shot of a computer program  Description automatically generated |

[**Skill 23.04: Exercises 1**](file:///C:\Users\PLUSKH01\Desktop\APCompSciA\ticketOutTheDoor\set19\Set19TicketOutTheDoorAPCompSciA.pdf)

|  |
| --- |
| **Skill 23.05: Create objects from super and sub classes** |

**Skill 23.05 Concepts**

The inheritance of one class from another follows an "IS A" relationship. That is, a mountain bike "IS A" bicycle. The reverse is not true, however. For example, a bike is not necessarily a mountain bike. When creating objects from super and sub classes, this relationship becomes important.

Consider the following hierarchy of inherited classes between bicycles.

A diagram of a bicycle

Description automatically generated

The above hierarchy shows the relationship among the classes in a program. According to the hierarchy

* CrossCountryBike "IS A" MountainBike
* DownhillBike "IS A" MountainBike
* MountainBike "IS A" Bicycle

When creating objects from super and sub classes this relationship is enforced. This is illustrated below,

A blue arrows with red x in the middle

Description automatically generated

[**Skill 23.05: Exercises 1**](file:///C:\Users\PLUSKH01\Desktop\APCompSciA\ticketOutTheDoor\set19\Set19TicketOutTheDoorAPCompSciA.pdf)

|  |
| --- |
| **Skill 23.06: Pass a sub class object as a parameter** |

**Skill 23.06 Concepts**

Any time when a parameter is expecting to receive an object of a particular type, it is acceptable to send it an object of the same class or a subclass, but never of a superclass. This is because the passed class object inherits all the methods of the object. If you attempt to pass an object of a parent class, the expected object may not have all the expected methods. Consider the following hierarchy of classes where each class is a subclass of the class immediately above it.

A screen shot of a diagram

Description automatically generated

Suppose there is a method with the following signature,



The method *theMethod* is clearly expecting a *Male* object; therefore, the following calls to this method would be legal since we are either sending a Male object or an object of a subclass,

A close-up of a white background

Description automatically generated

Since *theMethod* is expecting a *Male* object, we can’t send an object of a superclass.

A close-up of a grey background

Description automatically generated

[**Skill 23.06: Exercises 1**](file:///C:\Users\PLUSKH01\Desktop\APCompSciA\ticketOutTheDoor\set19\Set19TicketOutTheDoorAPCompSciA.pdf)