***Creating Abstract Classes:***

* *You have learnt that a subclass can override an inherited method defined in the parent class. Overriding a method potentially changes the behavior of a method in the parent class. For example, Bird class and its Stork subclass*

*class Bird {*

*public String getName() { return null; }*

*public void printName() {*

*System.out.print(getName());*

*}}*

*public class Stork extends Bird {*

*public String getName() { return “Stork”; }*

*public String void main(String args[]) {*

*new Stork().printName();*

*}}*

*This program prints Stork at runtime. Notice that the getName method is overridden in the subclass. Even through the implementation of printName() is defined in the Bird class. The fact that the getName() is overridden in the subclass means it is replaced everywhere even in the parent class.*

*Suppose you want to define the Bird class that other developers extend and use but you want the developer to specify the particular type of Bird. Also, rather than having the Bird version of getName() return null or throw an exception you want to ensure every class that extends Bird is required to provide its own overridden version of the getName() method.*

*An abstract class is a class that cannot be instantiated and may contain abstract methods. An abstract method is a method that doesn’t define an implementation when it is declared.*

*Both abstract classes and abstract methods are denoted with the abstract modifier.*

*abstract class Bird {*

*public abstract String getName();*

*public void printName() {*

*System.out.print(getName());*

*}}*

*public class Stork extends Bird {*

*public String getName() { return “Stork!!”; }*

*public static void main(String args[]){*

*new Stork().printName();*

*}}*

*First, the Bird class is marked abstract. Next, the getName() method in Bird is also marked as abstract. Finally, the implementation of getName(), including the braces have been replaced with a single semicolon;*

*Stork class is exactly the same before. While it might look the same, though the rules around how the class must be implemented have changed. In particular, the Stork class must now override the abstract method getName(). The Stork class doesn’t compile if the abstract method isn’t implemented in the subclass.*

*An abstract class is most commonly used when you want another class to inherit properties of a particular class but you want the subclass to fill in some of the implementation details.*

*Override vs Implement:*

* *Oftentimes, when an abstract method is overridden in a subclass it is referred to as implementing the method. It is described this way because the subclass is providing an implementation for a method that doesn’t have one. While we tend to use the terms implementing and override interchangeably for abstract methods, the term override is more accurate. When overriding an abstract method, all of the rules you follow for overriding methods are applicable. For example, you can override an abstract method with a covariant return type. Likewise, you can declare new unchecked exceptions but not checked exceptions in the overridden methods. Furthermore, you can override an abstract method in one class and then override it again in subclass of that class.*
* *The method override rules apply whether the abstract method is declared in an abstract class. Providing an implementation for an abstract method is considered a method override and all of the associated rules for overriding methods apply.*
* *An abstract class is one that cannot be instantiated. This means that if you attempt to instantiate it, the compiler will report an exception.*

*abstract class Alligator {*

*public static void main(String args[]){*

*var a = new Alligator(); //Doesn’t compile*

*}}*

*An abstract class can be initialized but only as a part of the instantiation of non abstract subclass.*

*Defining abstract methods:*

* *An abstract class may include nonabstract methods. In fact an abstract class can include all of the same members as non abstract class including variables, static and instance methods and inner classes. Abstract classes can also include constructors.*
* *One of the important features of an abstract class is that it is not actually required to include any abstract methods. For example, the following code compiles even though it doesn’t define any abstract methods.*

*public abstract class Llama{*

*public void chew() {}*

*}*

*Although an abstract class doesn’t have to declare any abstract methods, an abstract method can only be defined in an abstract class. For example, the following code won’t compile because the class is not marked as abstract:*

*public class Eqret { //Doesn’t compile*

*public abstract void peck();*

*}*

* *Like the final modifier, the abstract modifier can be placed before or after the access modifier in class and method declarations are shown in Tiger class:*

*abstract public class Tiger {*

*abstract public int claw();*

*}*

*There are some restrictions on the placement of the abstract modifier. The abstract modifier cannot be placed after the class keyword in a class declaration, nor after the return type in a method declaration.*

*public class abstract Jackal {//Doesn’t compile*

*public int abstract howl(); //doesn’t compile*

*}*

* *It isn’t possible to define an abstract method that has a body or a default implementation. You can still define a method with a body you just cant mark it as abstract. As long as you don’t mark the method as final, the subclass has the option to override an inherited method.*

*Constructors in Abstract classes:*

* *Even though abstract classes cannot be instantiated, they are still initialized through constructors by their subclasses.*

*abstract class Bear {*

*abstract CharSequence chew();*

*public Bear() {*

*System.out.println(chew());*

*}}*

*public class Panda extends Bear{*

*String chew() {return “yummy!!”;}*

*public static void main(String[] args) {*

*new Panda();*

*}}*

*The compiler inserts a no default constructor into the Panda class which first calls super() in the Bear class. The Bear constructor is only called when the abstract class is being initialized through a subclass and therefore there is an implementation of chew() at the time the constructor is called. The code compiles and prints yummy!! At runtime.*

*The abstract classes are initialized with constructors the same way as non abstract classes. For instance, if an abstract class doesn’t provide a constructor, the compiler will automatically insert a default no argument constructor.*

*The primary difference between a constructor in an abstract class and a non abstract class is that a constructor in abstract class can be called only when it is being initialized by a non abstract subclass. This makes sense as abstract classes cannot be instantiated.*

***Invalid Abstract Method Declarations:***

*public abstract class Turtle {*

*public abstract long eat() //Doesn’t compile*

*public abstract void swim() {}; //Doesn’t compile*

*public abstract int getAge() { return 10; } // Doesn’t compile*

*public void sleep; //Doesn’t compile*

*public void goInShell(); //Doesn’t compile*

*The first method doesn’t compile because it is marked as abstract but doesn’t end with a semicolon. The next two methods swim and getAge donot compile because they are marked as abstract, but they provide an implementation block enclosed in braces. The abstract method declaration must end in semicolon without any braces. The next method sleep doesn’t compile because it has a missing braces. The last method doesn’t compile because it isn’t marked as abstract and therefore must provide a body enclosed in braces.*

***Invalid Modifiers:***

***Abstract and final modifers:***

* *If you mark something abstract, you are intending for someone else to extend or implement it. But if you mark something as final, you are preventing anyone to extend or implement it. Due to this incompatibility, Java doesn’t permit a class or a method to be marked both as abstract and final.*

*public abstract final class Tortoise { // Doesn’t compile*

*public abstract final void walk(); //Doesn’t compile*

*}*

*Neither the class or method declarations will compile because they are marked as both abstract and final.*

***abstract and private Modifiers:***

* *A method can’t be marked as both abstract and private. You cant define a subclass that implements a required method if the method isn’t inherited by subclass.*

*public abstract class Whale{*

*private abstract void sing(); //Doesn’t compile*

*}*

*public class HumpbackWhale extends Whale{*

*private void sing() {*

*System.out.println(“Humpback Whale is singing…”);*

*}}*

*The abstract method sing() defined in the parent class Whale isn’t visible to the subclass HumbackWhale. Even though HumpbackWhale doesn’t provide an implementation, it isn’t considered as an override of the abstract method since the abstract method isn’t inherited. The compiler recognizes this in the parent class and reports this as an error as soon as private and abstract are applied to the same method.*

*While it isn’t possible to declare a method abstract and private, it is possible to declare a method final and private.*

*public abstract class Whale{*

*protected abstract void sing();*

*}*

*public class HumpbackWhale extends Whale {*

*private void sing() { /Doesn’t compile*

*System.out.println(“Humpback Whale while singing…”);*

*}}*

*In this modified example, the code will still not compile but for a completely different reason. The subclass cannot reduce the visibility of the parent method. Because the method is declared protected in the parent class, it must be marked protected or public in the child class. Even with abstract methods, the rules for overriding methods must be followed.*

*Abstract and static modifiers:*

* *A static method cant be overridden. It is defined as belonging to the class, not an instance of the class. If a static method cant be overridden, then it follows that it also cannot be marked abstract since it can never be implemented.*

*abstract class Hippo{*

*abstract static void swim(); //Doesn’t compile*

*}*

*Creating a Concrete class:*

* *An abstract class becomes usable when it is extended by a concrete subclass. A concrete class is a non-abstract class. The first concrete subclass that extends abstract class is required to implement all inherited abstract methods. This includes implementing any inherited abstract methods from inherited abstract methods from inherited interfaces.*

*public abstract class Animal {*

*public abstract String getName() ;*

*}*

*public class Walrus extends Animal{} //Doesn’t compile*

*Animal is marked as abstract and Walrus isn’t making a concrete subclass of Animal. Since, Walrus is first concrete subclass it must implement all the inherited abstract methods. Because it doesn’t the compiler reports an error at the time of declaration of Walrus.*

*abstract class Mammal {*

*abstract void showHorn();*

*abstract void eatLeaf();*

*}*

*abstract class Rhino extends Mammal{*

*void showHorn() {};*

*}*

*public class BlackRhino extends Rhino{*

*void eatLeaf() {}*

*}*

*The BlackRhino is the first concrete subclass while the Mammal and Rhino are abstract. The BlackRhino class inherits the eatLeaf() method as abstract and is therefore required to provide an implementation which it does.*

*Since the parent class Rhino provides an implementation of showHorn() the method is inherited in the BlackRhino as a nonabstract method. For this reason, the BlackRhino class is permitted but not required to override the showHorn() method.*

*public abstract class Animal {*

*abstract String getName();*

*}*

*public abstract class BigCat extends Animal {*

*protected abstract void roar();*

*}*

*public class Lion extends BigCat {*

*public String getName() { return “Lion”; }*

*public void roar() {*

*System.out.println(“The Lion lets out a loud roar!!”);*

*}}*

*BigCat extends Animal but it is marked as abstract therefore it isn’t required to provide an implementation for the getName() method. The class Lion is not marked as an abstract and as the first concrete subclass, it must implement all of the inherited abstract methods not defined in a parent class. All of the three classes compile successfully.*

*Reviewing Abstract class rules:*

* *Abstract classes cannot be instantiated.*
* *All top-level types including abstract classes cannot be marked as protected or private.*
* *Abstract classes cannot be marked as final*
* *Abstract classes may include zero or more abstract and non abstract methods.*
* *An abstract class that extends another abstract class inherits all of its abstract methods.*
* *The first concrete class that extends an abstract class must provide an implementation for all of the inherited abstract methods.*
* *Abstract class constructors follow the same rules for initialization as regular constructors except they can be called only as part of the initialization of a subclass.*

*These rules for abstract methods apply regardless of whether the abstract method is defined in an abstract class or interface.*

*Abstract Method Definition Rules:*

* *Abstract methods can be defined only in the abstract classes or interfaces.*
* *Abstract methods cannot be declared as private or final.*
* *Abstract methods must not provide a method body or implementation in the abstract class in which they are declared.*
* *Implementing an abstract method in a subclass follows the same rules for overriding a method, including covariant return types, exception declarations etc*

*Implementing Interfaces:*

* *Although Java doesn’t allow multiple inheritance of state, it does allow a class to implement any number of interfaces. An interface is an abstract data type that declares a list of abstract methods that any class implementing the interface must provide. An interface can also include constant variables. Both abstract methods and constant variables included with an interface are implicitly assumed to be public.*

*Interfaces and Non-Abstract methods:*

* *With Java8, interfaces were updated to include static and default methods. A default method is one in which the interface method has a body and is not marked as abstract. It was added for backward compatibility, allowing an older class to use a new version of an interface that contains a new method, without having to modify the existing class.*
* *In Java9, interfaces were updated to support private and private static methods. both of these types were added for code reusability within an interface declaration and cannot be called outside the interface definition.*
* *In Java, an interface is defined with the interface keyword analogous to the class keyword analogous to the class keyword when defining a class.*

*public abstract interface CanBurrow {*

*public abstract float getSpeed(int age); //Abstract interface method*

*public static final int MINIMUM\_DEPTH=2;*

*Our interface declaration includes a constant variable and an abstract method. Interface variables are referred to as constants because they are assumed to be public static and final. They are initialized with a constant value when they are declared. Since they are public and static they can be used outside the interface declaration without requiring an instance of the interface. For brevity, sometimes we say an instance of the interface to mean an instance of a class that implements the interface.*

* *One aspect o an interface declaration that differs from an abstract class is that it contains implicit modifiers. An implicit modifier is a modifier that compiler automatically adds to a class, interface, method or variables declaration. An interface is always considered to be abstract even if it is not marked so.*

*public abstract interface WalksOnTwoLegs {}*

*It compiles because interfaces aren’t required to define any methods. The abstract modifier in this example is optional for interfaces with the compiler inserting it if it isn’t provided.*

*public class Biped {*

*public static void main(String args[]) {*

*var e = new WalksOnTwoLegs(); //Doesn’t compile*

*}}*

*public final interface WalksOnEightLegs {} //Doesn’t compile*

*The first example doesn’t compile as WalksOnTwoLegs is an interface and cannot be instantiated. The second example WalksonEightLegs doesn’t compile because interfaces cant be marked as final for the same reason that abstract classes cannot be marked as final. In other words, marking an interface final implies no class could ever implement it.*

*interface Climb {*

*Number getSpeed(int age); }*

*Next, we have a concrete class FieldMouse that invokes the Climb interface by using the implements keyword in its class declaration.*

*public class FieldMouse implements Climb, CanBurrow {*

*public Float getSpeed(int age){*

*return 11f;*

*}}*

*The FieldMouse class declares that it implements the Climb interface and includes an overridden version of getSpeed() matches exactly and the return type is covariant.*

*The access modifier of the interface method is assumed to be public in Climb, although the concrete class FieldMouse must explicitly declare it.*

* *A class can implement multiple interfaces each separated by a comma. If any of the interfaces define abstract methods then the concrete class is required to override them. In this case, FieldMouse also implements the CanBurrow interface. In this manner, the class overrides two abstract methods at the same time with one method declaration.*
* *Like a class an interface can extend another interface using the extends keyword:*

*interface Nocturnal {*

*public interface HasBigEyes extends Nocturnal {}*

*Unlike a class, which extends only one class, an interface can extend multiple interfaces.*

*interface Nocturnal {*

*public int hunt();*

*}*

*interface CanFly {*

*public void flap();*

*}*

*interface HasBigEyes extends Nocturnal, CanFly {}*

*public class Owl implements Nocturnal, CanFly {*

*public int hunt() { return 5; }*

*public void flap() { System.out.println(“Flap!!!”); }*

*In this example, the Owl class implements the HasBigEyes interface and must implement the hunt() and flap() methods. Extending two interfaces is permitted because interfaces are not initialized as part of a class hierarchy. Unlike abstract classes, they do not contain constructors and are not part of instance initialization. Interfaces simply define a set of rules that a class implementing them must follow. They also include various static members, including constants that do not require an instance of the class to use.*

*Many of the rules for the class declarations also apply to interfaces including the following:*

* *A Java file may have at most one public top-level class or interface and it must watch the name of the file.*
* *A top level class or interface can only be declared with public or package-private access.*

*Interface is a specialized abstract class, as many of the rules carry over. An interface doesn’t follow the same rules for single inheritance and instance initialization with constructors as a class does.*

*Enums:*

* *A Java class can have at most one public top level element, a class or interface. The public top- level element could also be an enumeration or enum for short. An enum is a specialized type that defines a set of fixed values. It is declared with the enum keyword. The following demonstrates a simple example for an enum for Color:*

*public enum Color {*

*RED, YELLOW, BLUE, GREEN, ORANGE, PURPLE*

*}*

* *Like classes and interfaces, enums can have more complex formations including methods, private contructors and instance variables.*

*Inserting Implicit Modifiers:*

* *An implicit modifier is one that compiler will automatically insert. It’s a reminiscent of the compiler inserting a default no-argument constructor if you do not define a constructor. You can choose to insert these implicit modifiers*
* *The following list includes the implicit modifiers for interfaces that you need to know:*
* *Interfaces are assumed to be abstract.*
* *Interface variables are assumed to be public, static and final.*
* *Interface methods without a body are assumed to be abstract and public.*

*For example, the following two interface definitions are equivalent as the compiler will convert them both to the second declaration:*

*public interface Soar {*

*int MAX\_HEIGHT = 10;*

*final static boolean UNDERWATER = true;*

*void fly(int speed);*

*abstract void takeoff();*

*public abstract double dive();*

*}*

*public* ***abstract*** *interface Soar {*

***public static final*** *int MAX\_HEIGHT = 10;*

***public*** *final static boolean UNDERWATER = true;*

***public abstract*** *void fly(int speed);*

***public*** *abstract void takeoff();*

*public abstract double dive();*

*}*

*In this example, we have marked the implicit modifiers that the compiler automatically inserts. First, the abstract keyword is added to the interface declaration. Next, the public, static and final keywords are added to the interface variables if they do not exist. Finally, each abstract method is prepended with the abstract and public keywords if they donot contain them already.*

*Conflicting Modifiers:*

* *What happens if a developer marks a method or variables with a modifier that conflicts an implicit modifier? For example, if an abstract method is assumed to be public, then can it be explicitly marked as protected or private?*

*public interface Dance {*

*private int count = 4; //Doesn’t compile*

*protected void step(); //Doesn’t compile*

*}*

*Neither of these interface member declarations compiles, as the compiler will apply the public modifier to both, resulting in conflict.*

*While issues with private and protected access modifiers in interfaces are easy to spot, what about package-private access? For example, what is the access level of the following two elements volume and start()?*

*public interface Sing {*

*float volume = 10;*

*abstract void start();*

*}*

*If you said public, then you are correct!!!. When working with class members, omitting the access modifier indicates default access. When working with interface members though, the lack of access modifier always indicates public access.*

*private final interface Crawl {*

*String distance;*

*private int MAXIMUM\_DEPTH = 100;*

*protected abstract boolean UNDERWATER = false;*

*private void dig(int depth);*

*protected abstract double depth();*

*public final void surface(); }*

*Every single line of this example, including the interface declaration doesn’t compile!. Line 1 doesn’t compile for two reasons. First it is marked as final, which cannot be applies to an interface since it conflicts with the implicit abstract keyword. Next, it is marked as private which conflicts with the public or package-private access for top-level interfaces.*

*Line 2 doesn’t compile because the distance variable is not initialized. Remember that interface variables are assumed to be static final constants and initialized when they are declared. Lines 3 and 4 donot compile because interface variables are not assumed to be public, and the access modifiers on these lines conflict with this. Line 4 doesn’t compile because variables cannot be marked as abstract. Lines 5 and 6 donot compile because all the interface abstract methods are assumed to be public and marking them as private or protected is not permitted. Finally, the last line doesn’t compile because the method is marked as final, and since interface methods without a body are assumed to be abstract, the compiler throws an exception for using both abstract and final keywords on method.*

*Differences between Interfaces and Abstract classes:*

* *Even though abstract classes and interfaces are both considered abstract types, only interfaces make use of implicit modifiers. This means that an abstract class and interface with similar declarations may have different properties.*

*abstract class Husky {*

*abstract void play();*

*}*

*interface Poodle {*

*void play();*

*}*

*Both of these method definitions are considered abstract. That said Husky class will not compile if the play() method is not marked abstract whereas the method in the Poodle interface will compile with or without abstract modifier.*

*Even though neither has an access modifier, they donot have the same access level. The play() method in Husky class is considered default(package-private) whereas the method in the Poodle interface is assumed to be public. This is especially important when you create classes that inherit these definitions.*

*class Webby extends Husky {*

*void play() {}*

*}*

*Class Georgette implements Poodle {*

*void play() {}*

*}*

*The Webby class compiles, but the Georgette class doesn’t. Even though the two method implementations are identical, the method in the Georgette class breaks the rules of the method overriding. From the Poodle interface, the inherited abstract method is assumed to be public. The definition of play() in the Georgette class therefore reduces the visibility of a method from public to package-private resulting in compiler error. Following is correct implementation*

*class Georgette implements Poddle {*

*public void play() {}*

*}*

***Inheriting an Interface:***

*An interface can be inherited in one of three ways:*

* *An interface can extend another interface*
* *A class can implement an interface*
* *A class can extend another class whose ancestor implements an interface*

*When an interface is inherited, all of the abstract methods are inherited. If the type inheriting the interface is also abstract, such an interface or abstract class is not required to implement the interface methods. On the other hand, the first concrete subclass that inherits the interface must implement all of the inherited abstract methods.*

*public interface HasTail {*

*public int getTailLength();*

*}*

*public interface HasWhiskers {*

*public int getNumberOfWhiskers();*

*}*

*public abstract class HarborSeal implements HasTail, HasWhiskers {*

*public class CommonSeal extends HarborSeal { //Doesn’t compile*

*}*

*The HarborSeal class is not required to implement any of the abstract methods it herits from the HasTail and HasWhiskers because it is marked as abstract. The concrete class CommonSeal, which extends HarborSeal is required to implement all inherited abstract methods. In this example, CommonSeal doesn’t provide an implementation for the inherited abstract interface methods so CommonSeal doesn’t compile.*

*Mixing Class and Interface keywords:*

* *Although a class can implement an interface, a class cannot extend an interface. Likewise, while a interface ca extend another interface, an interface cannot implement another interface.*

*public interface CanRun {}*

*public class Cheetah extends CanRun {} //Doesn’t compile*

*public class Hyena {}*

*public interface HasFur extends Hyena {} //Doesn’t compile*

*The first example shows a class trying to extend an interface that doesn’t compile. The second example shows an interface that is trying to extend a class which doesn’t compile.*

*class1 extends class2*

*interface1 extends interface2, interface3, …….*

*class1 implements interface2, interface3,….*

*Duplicate Interface Method Declarations:*

* *Since Java allows for multiple inheritance via interfaces might be wondering what will happen if you define a class that inherits from two interfaces tat contain the same abstract method.*

*public interface Herbivore {*

*public void eatPlants() ;*

*}*

*public interface Omnivore {*

*public void eatPlants();*

*public void eatMeat();*

*}*

* *In this scenario, the signatures for the two interface methods eatPlants() are duplicates. As they have identical method declarations, they are also compatible. By compatibility, we mean that the compiler can resolve the differences between the two declarations without finding any conflicts. You can define a class that fulfills both interfaces simultaneously.*

*public class Bear implements Herbivore, Omnivore {*

*public void eatMeat() {*

*System.out.println(“Eating Meat…”);*

*}*

*public void eatPlants() {*

*System.out.println(“Eating Plants…”);*

*}}*

* *As we said earlier, interfaces simply define a set of rules that a class implementing them must follow. If two abstract interface methods have identical behaviors or in this case the same method declaration- you just need to be able to create a single method that overrides both inherited abstract methods at the same time.*
* *What if the duplicate methods have different signatures? If the method name is the same but the input parameters are different, there is no conflict because this is considered a method overload.*

*public interface Herbivore {*

*public int eatPlants(int quantity);*

*}*

*public interface Omnivore {*

*public void eatPlants();*

*}*

*public class Bear implements Herbivore, Omnivore {*

*public int eatPlants(int quantity) {*

*System.out.println(“Eating plants: “+quantity);*

*return quantity;*

*}*

*public void eatPlants() {*

*System.out.println(“Eating Plants….”);*

*}}*

*In this example, we see that the class implements both interfaces must provide implementations of both versions of eatPlants() since they are considered separate methods.*

*What if the duplicate methods have the same signature but different return types? In that case, you need to review the rules for overriding methods.*

*interface Dances {*

*String swingArms();*

*}*

*Interface EatsFish {*

*CharSequence swingArms();*

*}*

*public class Penguin implements Dances, EatsFish {*

*public String swingArms() {*

*return “swing!!!”;*

*}}*

*In this example, the Penguin class compiles. The Dances version of the swingArms() method is trivially overridden in the Penguin class as the declaration in Dances and Penguin have the same method declarations. The EatsFish version of the swingArms() is also overridden as String and CharSequence are covariant return types.*

*interface Dances {*

*int countMoves();*

*}*

*interface EatsFish {*

*boolean countMoves();*

*}*

*public class Penguin implements Dances, EatsFish { //doesn’t compile*

*…*

*}*

* *Since it isn’t possible to define a version of countMoves() that returns both int and boolean , there is no implementation of the Penguin that will allow this declaration to compile. It is the equivalent of trying to define two methods in the same class with the same signature and different return types.*

*The compiler would also throw an exception if you define an abstract class or interface that inherits from two conflicting abstract types*

*interface LongEars {*

*int softSkin();*

*}*

*interface LongNose {*

*void softSkin();*

*}*

*interface Donkey extends LongEars, LongNose {} //Doesn’t compile*

*abstract class Aardvark implements LongEars, LongNose {} //Doesn’t compile*

* *All of these types in this example are abstract, with none being concrete. Despite the fact they are all abstract, the compiler detects that Donkey and Aardvark contain incompatible methods and prevents them from compiling.*

*Polymorphism and Interfaces:*

* *The fact that a class can inherit multiple interfaces limits some of the checks the compiler can perform.*

*Abstract Reference Types:*

* *When working with abstract types, you may prefer to work with the abstract reference types, rather than the concrete class. This is especially common when defining method parameters.*

*import java.util.\*;*

*public class Zoo {*

*public void sortAndPrintZooAnimals(List<String> animals) {*

*Collections.sort(animals);*

*For(String a : animals) {*

*System.out.println(a);*

*}}}*

* *This class defines a method that sorts and prints animals in alphabetical order. At no point in this class interested in what the actual underlying object for animals is. It might be an ArrayList which you have seen before, but it may also be a LinkedList or a Vector.*

*Casting Interfaces:*

* *Lets say we have an abstract reference type variable which has been instantiated by a concrete subclass. If you need access to a method that is only declared in the concrete subclass then you will need to cast the interface reference to that type, assuming that the cast is supported at runtime. For example, the following is not permitted as the compiler detects that the String and Long class cannot be related*

*String lion = “Bert”;*

*Long tiger = (Long) lion;*

* *With interfaces, there are limitations to what the compiler can validate.*

*interface Canine {}*

*class Dog implements Canine {}*

*class Wolf implements Canine {}*

*public class BadCasts {*

*public static void main(String[] args) {*

*Canine canine = new Wolf();*

*Canine badDog = (Dog) canine;*

*}}*

* *In this program, a Wolf object is created and then assigned to a Canine reference type. Because of polymorphism, Java cannot be sure which specific class type the canine instance. Therefore, it allows the invalid cast to the Dog reference type, even though the Dog and Wolf are not related. The code compiles but throws a ClassCastException at runtime.*
* *This limitation aside, the compiler can enforce one rule around interface casting. The compiler doesn’t allow a cast from an interface reference to an object reference if the object type doesn’t implement the interface.*
* *Object badDog = (String) canine; //Doesn’t compile*

*Since String doesn’t implement Canine, the compiler recognized that this cast isn’t possible.*

*Interfaces and the instanceOf operator:*

* *Compiler will report an error if you attempt to use the instanceof operator with two unrelated classes*

*Number tickets = 4;*

*If(tickets instanceof String) {} //Doesn’t compile*

*With interfaces, the compiler has limited ability to enforce this rule because even though a reference type may not implement an interface, one of it subclasses could*

*Number tickets=5;*

*If(tickets instanceOf List) {}*

* *Even though Number doesn’t inherit List, it’s possible the tickets variable may be a reference to a subclass of Number that does inherit List. the tickets variable could be assigned to an instance of the MyNumber class*
* *public class MyNumber extends Number implements List*

*That said, the compiler can check for unrelated interfaces if the reference is a class that is marked final.*

*Integer tickets = 6;*

*If(tickets instanceof List) {} //Doesn’t compile*

*The compiler rejects this code because the Integer class is marked final and doesn’t inherit List. Therefore, it isn’t possible to create subclass of Integer that inherits the List interface.*

***Reviewing Interface Rules:***

* *Interfaces cannot be instantiated.*
* *All top-level types including interfaces cannot be marked as protected or private*
* *Interfaces are assumed to be abstract and cannot be marked final*
* *Interfaces may include zero or more abstract methods.*
* *An interface can extend any number of interfaces.*
* *An interface reference may be cast to any reference that inherits the interface, although this may produce an exception at runtime if the classes aren’t related.*
* *The compiler will only report an unrelated type error for an instanceof operation with an interface on the right side if the reference on the left side is a final class that doesn’t inherit the interface.*
* *An interface method with a body must be marked default, private, static or private static*

***Abstract Interface Method Rules:***

* *Abstract methods can be defined only in abstract classes or interfaces.*
* *Abstract methods cannot be declared private or final.*
* *Abstract methods must not provide a body/implementation in the abstract class in which it is declared.*
* *Implementing an abstract method in a subclass follows the same rules for overriding a method, including covariant return types, exception declarations etc.*
* *Interface methods without a body are assumed to be abstract and public.*

*The first four rules for abstract methods whether they be defined in abstract classes or interfaces, are exactly the same.*

*Finally, there are two rules to remember for interface variables.*

***Interface variables rules:***

* *Interface variables are assumed to be public, static and final.*
* *Because interface variables are marked final, they must be initialized with a value when they are declared.*
* *It may be helpful to think of an interface as a specialized kind of abstract class, since it shares many of the same properties and rules as an abstract class.*
* *The primary differences between the two are that interfaces include implicit modifiers, do not contain constructors donot participate in the instance initialization process and support multiple inheritance.*

***Using an Interface vs Implementing an Interface:***

* *An interface provides a way for one individual to develop code that uses another individuals code without having access to the other individual’s underlying implementation.*
* *Interfaces can facilitate rapid application development by enbling development teams to create applications in parallel, rather than being directly dependent on each other.*
* *For example, two teams can work together to develop a one page standard interface at the start of the project. One team then develops code that uses the interface while the other team develops code that uses the interface, while the other team develops code that implements the interface. The development teams can then combine their implementations toward the end of the project, and as long as both teams developed with the same interface they will be compatible.*

***Introducing Inner classes:***

* *A member inner class is a class that is defined at the member level of the class the same level as the methods, instance variables and constructors. It is the opposite of a top level class in that it cannot be declared unless it is inside another class.*
* *Developers often define a member inner class inside another class if the relationship between the two classes is very close.*
* *There are four types of nested classes you will need to know about: member inner classes, local classes, anonymous classes and static nested classes. The following is an example of an outer class Zoo with an inner class Ticket:*

*public class Zoo {*

*public class Ticket {}*

*}*

*We can expand this to include an interface:*

*public class Zoo {*

*private interface Paper {}*

*public class Ticket implements Paper {}*

*}*

* *While top level classes and interfaces can only be set with public or package-private access, member inner classes donot have same restriction.*
* *A member inner class can be declared with all of the same access modifiers as a class member such as public, protected, default or private.*
* *A member inner class can contain many of the same methods and variables as a top level class. Some members are disallowed in member inner classes such as static members.*

*public class Zoo {*

*private interface Paper {*

*public String getId();*

*}*

*public class Ticket implements Paper {*

*private String serialNumber;*

*public String getId() { return serialNumber; }*

*}}*

*Using a Member Inner class:*

* *One of the ways a member inner class can be used is by calling it in the outer class. Continuing with our previous example, lets define a method in Zoo that makes use of he member inner class with a new sellTicket() method.*

*public class Zoo {*

*private interface Paper {*

*public String getId();*

*}*

*public class Ticket implements Paper {*

*private String serialNumber;*

*public String getId() { return serialNumber; }*

*}*

*public Ticket sellTicket(String serialNumber) {*

*var t = new Ticket();*

*t.serialNumber = serialNumber;*

*return t;*

*}*

*}*

*The advantage of using a member inner class in this example is that the Zoo class completely manages the lifecycle of the Ticket class.*

*public class Zoo {*

*…*

*public static void main(String args[]) {*

*var z= new Zoo();*

*var t = z.sellTicket(“12345”);*

*System.out.println(t.getId()+ “ Ticket sold!”);*

*}}*

*This compiles and prints 12345 Ticket sold! at runtime.*