

INTERFACES ABSTRACT CLASSES

OUTLINE

1. ABSTRACT CLASSES
2. INTERFACES

1) ABSTRACT CLASSES

- The following code is based on our Animal and Swan description:

```
public abstract class Animal {  
    protected int age;  
    public void eat() {  
        System.out.println("Animal is eating");  
    }  
    public abstract String getName();  
}  
  
public class Swan extends Animal {  
    public String getName() {  
        return "Swan";  
    }  
}
```

- The first thing to notice about this sample code is that the Animal class is declared **abstract** and Swan is not.
- Next, the member age and the method eat() are marked as **protected** and **public**, respectively; therefore, they are inherited in subclasses such as Swan.
- Finally, the **abstract** method getName() is terminated with a semicolon and doesn't provide a body in the parent class Animal.
- This method is implemented with the same name and signature as the parent method in the Swan class.

1) ABSTRACT CLASSES

1.1) DEFINING AN ABSTRACT CLASS

- The previous sample code illustrates a number of important rules about abstract classes.
- For example, an **abstract class** may include non-abstract methods and variables, as we saw with the variable `age` and the method `eat()`.
- In fact, an **abstract class** is not required to include any abstract methods.
- For example, the following code compiles without issue even though it doesn't define any abstract methods.

```
public abstract class Cow {}
```

- Although an abstract class doesn't have to implement any abstract methods, an abstract method may only be defined in an abstract class.
- For example, the following code won't compile because an abstract method is not defined within an abstract class.

```
public class Chicken {  
    // DOES NOT COMPILE  
    public abstract void peck();  
}
```

- Although we can't provide a default implementation to an abstract method in an abstract class, we can still define a method with a body - we just can't mark it as abstract.
- As long as we do not mark it as final, the subclass still has the option to override it, as explained in the previous section.

1) ABSTRACT CLASSES

1.1) DEFINING AN ABSTRACT CLASS

- Next, we note that an abstract class cannot be marked as final for a somewhat obvious reason.
- **By definition, an abstract class is one that must be extended by another class to be instantiated, whereas a final class can't be extended by another class.**
- By marking an abstract class as `final`, we're saying the class can never be instantiated, so the compiler refuses to process the code.
- For example, the following code snippet will not compile:
- Likewise, an abstract method may not be marked as final for the same reason that an abstract class may not be marked as final.
- Once marked as final, the method can never be overridden in a subclass, making it impossible to create a concrete instance of the abstract class.

```
// DOES NOT COMPILE
```

```
public final abstract class Tortoise {  
}
```

```
public abstract class Goat {  
    // DOES NOT COMPILE  
    public abstract final void chew();
```

1) ABSTRACT CLASSES

1.1) DEFINING AN ABSTRACT CLASS

- Finally, a method may not be marked as both **abstract** and **private**.
- How would we define a subclass that implements a required method if the method is not accessible by the subclass itself ?
- The answer is we can't, which is why the compiler will complain if you try to do the following:
- In this example, the abstract method `sing()` defined in the parent class `Whale` is not visible to the subclass `BlueWhale`.
- Even though `BlueWhale` does provide an implementation, it is not considered an override of the abstract method since the abstract method is unreachable.
- The compiler recognizes this in the parent class and throws an exception as soon as **private** and **abstract** are applied to the same method.

```
public abstract class Whale {  
    // DOES NOT COMPILE  
    private abstract void sing();  
}  
public class BlueWhale extends Whale {  
    private void sing() {  
        System.out.println("BlueWhale signs");  
    }  
}
```

1) ABSTRACT CLASSES

1.1) DEFINING AN ABSTRACT CLASS

- If we changed the access modifier from **private** to **protected** in the parent class Whale, would the code compile? Let's take a look:

```
public abstract class Whale {  
    protected abstract void sing();  
  
    public class BlueWhale extends Whale {  
        //DOES NOT COMPLY  
        private void sing() {  
            System.out.println("BlueWhale signs");  
        }  
    }  
}
```

- In this modified example, the code will still not compile but for a completely different reason.

- If we remember the rules earlier for overriding a method, the subclass cannot reduce the visibility of the parent method, `sing()`.
- Because the method is declared **protected** in the parent class, it must be marked as **protected** or **public** in the child class.
- Even with abstract methods, the rules for overriding methods must be followed.

1) ABSTRACT CLASSES

1.2) CREATING A CONCRETE CLASS

- When working with abstract classes, it is important to remember that by themselves, they cannot be instantiated and therefore do not do much other than define variables and methods.
- For example, the following code will not compile as it is an attempt to instantiate an abstract class.

```
public abstract class Whale {  
    public static void main(String[] args) {  
        // DOES NOT COMPILE  
        Whale whale = new Whale();  
    }  
}
```

- An abstract class becomes useful when it is extended by a concrete subclass.
- **A concrete class is the first non-abstract subclass that extends an abstract class and is required to implement all inherited abstract methods.**

```
public abstract class Animal {  
    public abstract String getName();  
}  
  
public class Dog extends Animal {  
    // DOES NOT COMPILE  
}
```

- First, we note that Animal is marked as abstract and Dog is not.
- In this example, Dog is considered the first concrete subclass of Animal.
- Second, since Dog is the first concrete subclass, it must implement all inherited abstract methods, getName() in this example.
- Because it doesn't, the compiler rejects the code.

1) ABSTRACT CLASSES

1.2) CREATING A CONCRETE CLASS

- Let's expand our discussion of abstract classes by introducing the concept of extending an abstract class with another abstract.
- We'll repeat our previous Dog example with one minor variation:

```
public abstract class Animal {  
    public abstract String getName();  
  
public class Dog extends Animal {  
    // DOES NOT COMPILE  
  
public abstract class Cat  
    extends Animal {
```

- In this example, we again have an abstract class Animal with a concrete subclass Dog that doesn't compile since it doesn't implement a getName() method.
- We also have an abstract class Cat, which like Dog extends Animal and doesn't provide an implementation for getName().
- **In this situation, Cat does compile because it is marked as abstract.**

1) ABSTRACT CLASSES

1.2) CREATING A CONCRETE CLASS

- As we saw in the previous example, abstract classes can extend other abstract classes and are not required to provide implementations for any of the abstract methods.
- It follows, then, that a concrete class that extends an abstract class must implement all inherited abstract methods.
- For example, the following concrete class `Lion` must implement two methods, `getName()` and `roar()`.

```
public abstract class Animal {  
    public abstract String getName();  
  
    public abstract class BigCat  
        extends Animal {  
        public abstract void roar();  
    }  
}
```

- As we saw in the previous example, abstract classes can extend other abstract classes and are not required to provide implementations for any of the abstract methods.
- It follows, then, that a concrete class that extends an abstract class must implement all inherited abstract methods.
- For example, the following concrete class `Lion` must implement two methods, `getName()` and `roar()`.

```
public class Lion extends BigCat {  
    public String getName() {  
        return "Lion";  
    }  
    public void roar() {  
        System.out.println("ROAR!");  
    }  
}
```

1) ABSTRACT CLASSES

1.3) EXTENDING AN ABSTRACT CLASS

- In this sample code, BigCat extends Animal but is marked as **abstract**; therefore, it is not required to provide an implementation for the getName() method.
- The class Lion is not marked as **abstract**, and as the first concrete subclass, **it must implement all inherited abstract methods not defined in a parent class.**
- There is one exception to the rule for abstract methods and concrete classes:
- **A concrete subclass is not required to provide an implementation for an abstract method if an intermediate abstract class provides the implementation.**
- For example, let's take a look at the following variation on our previous example.

```
public abstract class Animal {  
    public abstract String getName();  
}  
  
public abstract class BigCat  
    extends Animal {  
    public String getName() {  
        return "BigCat";  
    }  
    public abstract void roar();  
}  
  
public class Lion extends BigCat {  
    public void roar() {  
        System.out.println("ROAR!");  
    }  
}
```

- In this example, BigCat provides an implementation for the abstract method getName() defined in the abstract Animal class.
- Therefore, Lion **inherits only one abstract method**, roar(), and is not required to provide an implementation for the method getName().

4) ABSTRACT CLASSES

- The following are lists of rules for abstract classes and abstract methods that we have covered in this section.

ABSTRACT CLASS DEFINITION RULES

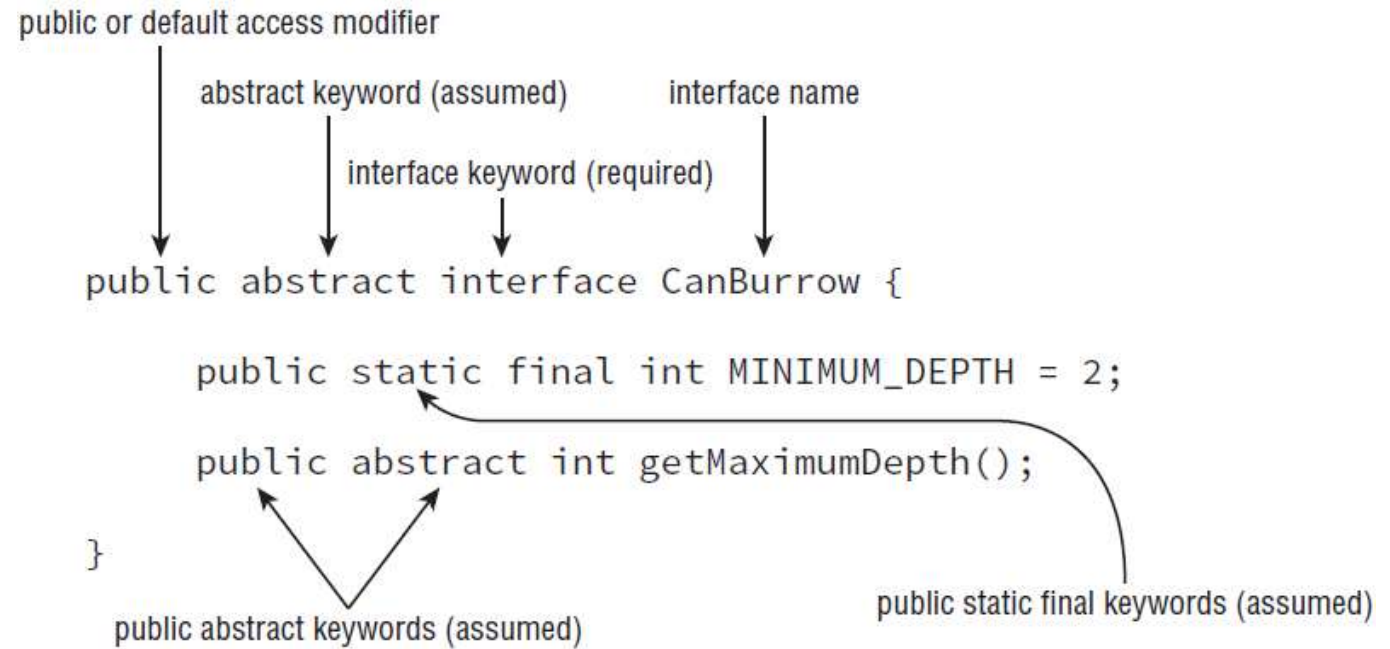
1. Abstract classes cannot be instantiated directly.
2. Abstract classes may be defined with any number, including zero, of abstract and non-abstract methods.
3. Abstract classes may not be marked as private or final.
4. An abstract class that extends another abstract class inherits all of its abstract methods as its own abstract methods.
5. The first concrete class that extends an abstract class must provide an implementation for all of the inherited abstract methods.

ABSTRACT METHOD DEFINITION RULES

1. Abstract methods may only be defined in abstract classes.
2. Abstract methods may not be declared private or final.
3. Abstract methods must not provide a method body/implementation in the abstract class for which is it declared.
4. Implementing an abstract method in a subclass follows the same rules for overriding a method. For example, the name and signature must be the same, and the visibility of the method in the subclass must be at least as accessible as the method in the parent class, ... etc.

2) INTERFACES

- Although Java doesn't allow multiple inheritance, it does allow classes to implement any number of interfaces.
- An interface is an abstract data type that defines a list of abstract public methods that any class implementing the interface must provide.
- As we see in this example, an interface is not declared as an abstract class, although it has many of the same properties of abstract class.
- Notice that the method modifiers in this example, **abstract** and **public**, are assumed.
- In other words, whether or not we provide them, the compiler will automatically insert them as part of the method definition.



The diagram illustrates the syntax of a Java interface declaration with annotations. The code is as follows:

```
public abstract interface CanBurrow {  
    public static final int MINIMUM_DEPTH = 2;  
    public abstract int getMaximumDepth();  
}
```

Annotations with arrows pointing to the code:

- public or default access modifier** points to `public`.
- abstract keyword (assumed)** points to `abstract`.
- interface keyword (required)** points to `interface`.
- interface name** points to `CanBurrow`.
- public static final keywords (assumed)** points to `public static final` in the field declaration.
- public abstract keywords (assumed)** points to `public abstract` in the method declaration.

2) INTERFACES

- In Java, an interface is defined with the **interface** keyword, analogous to the **class** keyword used when defining a class.
- A class invokes the interface by using the **implements** keyword in its class definition.

```
public class FieldMouse implements CanBurrow {  
    public int getMaximumDepth() {  
        return 10;  
    }  
}
```

signature matches interface method

2) INTERFACES

- A class may implement multiple interfaces, each separated by a comma, such as in the following example:
- **public class** Elephant **implements** WalksOnFourLegs, HasTrunk, Herbivore {}
- In the example, if any of the interfaces defined abstract methods, the concrete class **Elephant** would be required to implement those methods.
- New to Java 8 is the notion of default and static interface methods, which we'll also cover.

2) INTERFACES

2.1) DEFINING AN INTERFACE

- 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 following is a list of rules for creating an interface, many of which we should recognize as adaptations of the rules for defining abstract classes.
1. Interfaces cannot be instantiated directly.
 2. An interface is not required to have any methods.
 3. An interface may not be marked as `final`.
 4. All top-level interfaces are assumed to have **public** or **default** access. Therefore, marking an interface as **private**, **protected**, or **final** will trigger a compiler error, since this is incompatible with these assumptions.
 5. All non-default methods in an interface are assumed to have the modifiers **abstract** and **public** in their definition. Therefore, marking a method as **private**, **protected**, or **final** will trigger compiler errors as these are incompatible with the **abstract** and **public** keywords.

2) INTERFACES

2.1) DEFINING AN INTERFACE

- Let's say we have an interface `WalksOnTwoLegs`, defined as follows:

```
public interface WalksOnTwoLegs {  
}
```

- It compiles without issue, since interfaces are not required to define any methods.
- Now let's consider the following two examples, which do not compile:

```
public class TestClass {  
    // DOES NOT COMPILE  
    WalksOnTwoLegs example = new WalksOnTwoLegs();  
  
    public final interface WalksOnTwoLegs {  
        // DOES NOT COMPILE
```

- The first example doesn't compile, as `WalksOnTwoLegs` is an interface and cannot be instantiated directly.
- The second example, `WalksOnEightLegs`, doesn't compile since interfaces may not be marked as `final` for the same reason that abstract classes cannot be marked as `final`.

2) INTERFACES

2.1) DEFINING AN INTERFACE

- The fourth and fifth rule about “assumed keywords” might be new to us, but we can think of these in the same light as the compiler inserting a default no-argument constructor or **super()** statement into our constructor.
- We may provide these modifiers, although the compiler will insert them automatically if we do not.
- For example, the following two interface definitions are equivalent, as the compiler will convert them both to the second example.
- In this example, the **abstract** keyword is first automatically added to the interface definition.
- Then, each method is prepended with **abstract** and **public** keywords.
- If the method already has either of these keywords, then no change is required.

```
public interface CanFly {  
    void fly(int speed);  
    abstract void takeoff();  
    public abstract double dive();  
}
```

```
public abstract interface CanFly {  
    public abstract void fly(int speed);  
    public abstract void takeoff();  
    public abstract double dive();  
}
```

2) INTERFACES

2.1) DEFINING AN INTERFACE

- Let's take a look at an example that violates the assumed keywords.
- Every single line of this example doesn't compile.
- The first line doesn't compile for two reasons.
- First, it is marked as **final**, which cannot be applied to an interface since it conflicts with the assumed **abstract** keyword.
- Next, it is marked as **private**, which conflicts with the **public** or **default** required access for interfaces.
- The second and third line do not compile because all interface methods are assumed to be public and marking them as **private** or **protected** throws a compiler error.
- Finally, the last line doesn't compile because the method is marked as **final** and since interface methods are assumed to be **abstract**, the compiler throws an exception for using both **abstract** and **final** keywords on a method.

```
// DOES NOT COMPILE
private final interface CanCrawl {
    // DOES NOT COMPILE
    private void dig(int depth);
    // DOES NOT COMPILE
    protected abstract double depth();
    // DOES NOT COMPILE
    public final void surface();
}
```

2) INTERFACES

2.2) INHERITING AN INTERFACE

- There are two inheritance rules we should keep in mind when extending an interface:
 1. An interface that extends another interface, as well as an abstract class that implements an interface, inherits all of the abstract methods as its own abstract methods.
 2. The first concrete class that implements an interface, or extends an abstract class that implements an interface, must provide an implementation for all of the inherited abstract methods.
- Like an abstract class, an interface may be extended using the **extends** keyword. In this manner, the new child interface inherits all the abstract methods of the parent interface.
- **Unlike an abstract class, though, an interface may extend multiple interfaces.**

2) INTERFACES

2.2) INHERITING AN INTERFACE

- Consider the following example:

```
public interface HasTail {  
    public int getTailLength();  
}  
  
public interface HasWhiskers {  
    public int getNumberOfWhiskers();  
}  
  
public interface Seal extends HasTail,  
                               HasWhiskers {
```

- Any class that implements the **Seal** interface must provide an implementation for all methods in the parent interfaces - in this case, **getTailLength()** and **getNumberOfWhiskers()**.

- What about an **abstract class** that implements an **interface**?
- In this scenario, the abstract class is treated in the same way as an interface extending another interface.
- In other words, the abstract class inherits the abstract methods of the interface but is not required to implement them.
- That said, like an abstract class, the first concrete class to extend the abstract class must implement all the inherited abstract methods of the interface.

2) INTERFACES

2.2) INHERITING AN INTERFACE

- We illustrate this in the following example:

```
public interface HasTail {  
    public int getTailLength();  
}  
  
public interface HasWhiskers {  
    public int getNumberOfWhiskers();  
}  
  
public abstract class HarborSeal  
    implements HasTail, HasWhiskers {  
  
    // DOES NOT COMPILE  
    public class LeopardSeal  
        implements HasTail, HasWhiskers {
```

- In this example, we see that HarborSeal is an abstract class and compiles without issue.
- Any class that extends HarborSeal will be required to implement all of the methods in the HasTail and HasWhiskers interface.
- Alternatively, LeopardSeal is not an abstract class, so it must implement all the interface methods within its definition.
- In this example, LeopardSeal doesn't provide an implementation for the interface methods, so the code doesn't compile.

2) INTERFACES

2.2) INHERITING AN INTERFACE

- Since Java allows for multiple inheritance via interfaces, we might be wondering what will happen if we define a class that inherits from two interfaces that 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 compatible, so we 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");  
    }  
}
```

- Remember that interface methods in this example are abstract and define the “behavior” that the class implementing the interface must have.
- If two abstract interface methods have identical behaviors - or in this case the same method signature - creating a class that implements one of the two methods automatically implements the second method.
- In this manner, the interface methods are considered duplicates since they have the same signature.

2) INTERFACES

2.2) INHERITING AN INTERFACE

- What happens if the two 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.
- In this example, we see that the class that implements both interfaces must provide implements of both versions of eatPlants(), since they are considered separate methods.
- Notice that it doesn't matter if the return type of the two methods is the same or different, because the compiler treats these methods as independent.

```
public interface Herbivore {  
    public int eatPlants(int nrPlants);
```

```
public interface Omnivore {  
    public void eatPlants();
```

```
public class Bear implements  
    Herbivore, Omnivore {  
    public int eatPlants(int nrPlants) {  
        System.out.println("Eating plants:"  
            + " " + nrPlants);  
        return nrPlants;  
    }  
    public void eatPlants() {  
        System.out.println("Eating plants");
```


2) INTERFACES

2.2) INHERITING AN INTERFACE

- Unfortunately, if the method name and input parameters are the same but the return types are different between the two methods, the class or interface attempting to inherit both interfaces will not compile.
- The reason the code doesn't compile has less to do with interfaces and more to do with class design. It is not possible in Java to define two methods in a class with the same name and input parameters but different return types.

```
public interface Herbivore {  
    public int eatPlants();  
  
    public interface Omnivore {  
        public void eatPlants();  
    }  
}
```

```
public class Bear  
    implements Herbivore, Omnivore {  
    // DOES NOT COMPILE  
    public int eatPlants() {  
        System.out.println("Eating plants: 10");  
        return 10;  
    }  
    // DOES NOT COMPILE  
    public void eatPlants() {  
        System.out.println("Eating plants");  
    }  
}
```

- The code doesn't compile, as the class defines two methods with the same name and input parameters but different return types.
- If we were to remove either definition of eatPlants(), the compiler would stop because the definition of Bear would be missing one of the required methods.
- In other words, there is no implementation of the Bear class that inherits from Herbivore and Omnivore that the compiler would accept.

2) INTERFACES

2.3) INHERITING VARIABLES

- Let's expand our discussion of interfaces to include interface variables, which can be defined within an interface.
- Like interface methods, interface variables are assumed to be **public**.
- Unlike interface methods, though, interface variables are also assumed to be **static** and **final**.
- Here are two interface variables rules.
 1. Interface variables are assumed to be **public**, **static**, and **final**. Therefore, marking a variable as **private** or **protected** will trigger a compiler error, as will marking any variable as **abstract**.
 2. The value of an interface variable must be set when it is declared since it is marked as **final**.
- In this manner, interface variables are essentially constant variables defined on the interface level.
- Because they are assumed to be **static**, they are accessible even without an instance of the interface.

2) INTERFACES

2.3) INHERITING VARIABLES

- The following two interface definitions are equivalent, because the compiler will automatically convert them both to the second example.

```
public interface CanSwim {  
    int MAXIMUM_DEPTH = 100;  
    final static boolean UNDERWATER = true;  
    public static final String TYPE = "Frog";  
}
```

```
public interface CanSwim {  
    public static final int MAXIMUM_DEPTH = 100;  
    public static final boolean UNDERWATER = true;  
    public static final String TYPE = "Frog";  
}
```

- As we see in this example, the compiler will automatically insert **public static final** to any constant interface variables it finds missing those modifiers.
- Also note that it is a common coding practice to use uppercase letters to denote constant values within a class.

2) INTERFACES

2.3) INHERITING VARIABLES

- Based on these rules, it should come as no surprise that the following entries will not compile:

```
public interface CanDig {  
    // DOES NOT COMPILE  
    private int MAXIMUM_DEPTH = 100;  
    // DOES NOT COMPILE  
    protected abstract boolean UNDERWATER = false;  
    // WILL NOT COMPILE  
    public static String TYPE;
```

- The first example, `MAXIMUM_DEPTH`, doesn't compile because the `private` modifier is used, and all interface variables are assumed to be `public`.
- The second line, `UNDERWATER`, doesn't compile for two reasons. It is marked as `protected`, which conflicts with the assumed modifier `public`, and it is marked as `abstract`, which conflicts with the assumed modifier `final`.
- Finally, the last example, `TYPE`, doesn't compile because it is missing a value. Unlike the other examples, the modifiers are correct, but as you may remember from earlier, we must provide a value to a `final` member of the class it's instantiated.

2) INTERFACES

2.4) DEFAULT METHODS

- With the release of Java 8, the authors of Java have introduced a new type of method to an interface, referred to as a **default** method.
- A **default** method is a method defined within an interface with the **default** keyword in which a method body is provided.
- Contrast **default** methods with “regular” methods in an interface, which are assumed to be abstract and may not have a method body.
- A **default** method within an interface defines an abstract method with a default implementation.
- In this manner, classes have the option to override the **default** method if they need to, but they are not required to do so.
- If the class doesn't override the method, the **default** implementation will be used.
- In this manner, the method definition is concrete, not abstract.
- The purpose of adding **default** methods to the Java language was in part to help with code development and backward compatibility.
- Imagine we have an interface that is shared among dozens or even hundreds of users that you would like to add a new method to.
- If we just update the interface with the new method, the implementation would break among all of our subscribers, who would then be forced to update their code.
- In practice, this might even discourage us from making the change altogether.

2) INTERFACES

2.4) DEFAULT METHODS

- By providing a **default** implementation of the method, though, the interface becomes backward compatible with the existing codebase, while still providing those individuals who do want to use the new method with the option to override it.
- The following is an example of a default method defined in an interface:

```
public interface ISwarmBlooded {  
    boolean hasScales();  
  
    public default double getTemperature() {  
        return 10.0;  
    }  
}
```

- This example defines two interface methods, one is a normal abstract method and the other a default method.
- Note that both methods are assumed to be **public**, as all methods of an interface are all **public**.
- The first method is terminated with a semicolon and doesn't provide a body, whereas the second default method provides a body.
- Any class that implements ISwarmBlooded may rely on the default implementation of getTemperature() or override the method and create its own version.
- Note that the default access modifier as defined earlier is completely different from the **default** interface method.
- Because all methods within an interface are assumed to be **public**, the access modifier for a default method is therefore **public**.

2) INTERFACES

2.4) DEFAULT METHODS

- The following are the **default** interface method rules you need to be familiar with:
 1. A default method may only be declared within an interface and not within a class or abstract class.
 2. A default method must be marked with the **default** keyword. If a method is marked as default, it must provide a method body.
 3. A default method is not assumed to be static, final, or abstract, as it may be used or overridden by a class that implements the interface.
 4. Like all methods in an interface, a default method is assumed to be public and will not compile if marked as private or protected.
- The first rule should give us some comfort in that we'll only see default methods in interfaces.
- The second rule just denotes syntax, as default methods must use the **default** keyword.

```
public interface Carnivore {  
    // DOES NOT COMPILE  
    public default void eatMeat();  
    // DOES NOT COMPILE  
    public int getRequiredFoodAmount() {  
        return 13;  
    }  
}
```
- In this example, the first method, `eatMeat()`, doesn't compile because it is marked as **default** but doesn't provide a method body.
- The second method, `getRequiredFoodAmount()`, also doesn't compile because it provides a method body but is not marked with the **default** keyword.

2) INTERFACES

2.4) DEFAULT METHODS

- Unlike interface variables, which are assumed static class members, default methods cannot be marked as static and require an instance of the class implementing the interface to be invoked.
- They can also not be marked as **final** or **abstract**, because they are allowed to be overridden in subclasses but are not required to be overridden.
- When an interface extends another interface that contains a default method, it may choose to ignore the default method, in which case the default implementation for the method will be used.
- Alternatively, the interface may override the definition of the default method using the standard rules for method overriding.
- Finally, the interface may redeclare the method as **abstract**, requiring classes that implement the new interface to explicitly provide a method body.
- Analogous options apply for an abstract class that implements an interface.

2) INTERFACES

2.4) DEFAULT METHODS

- For example, the following class overrides one default interface method and redeclares a second interface method as **abstract**:

```
public interface HasTeeth {  
    public default int getNrOfTeeth() {  
        return 4;  
    }  
    public default double getBiggest() {  
        return 20.0;  
    }  
    public default boolean doCleanTeeth() {  
        return true;  
    }  
}  
  
public interface Shark extends HasTeeth {  
    public default int getNrOfTeeth() {  
        return 8;  
    }  
    public double getBiggest();  
    // DOES NOT COMPILE  
    public boolean doCleanTeeth() {  
        return false;  
    }  
}
```

- In this example, the first interface, **HasTeeth**, defines three default methods: **getNrOfTeeth()**, **getBiggest()**, and **doCleanTeeth()**.
- The second interface, **Shark** extends **HasTeeth** and overrides the default method **getNrOfTeeth()** with a new method that returns a different value.
- Next, the **Shark** interface replaces the default method **getBiggest()** with a new abstract method, forcing any class that implements the **Shark** interface to provide an implementation of the method.
- Finally, the **Shark** interface overrides the **doCleanTeeth()** method but doesn't mark the method as default.
- Since interfaces may only contain methods with a body that are marked as default, the code will not compile.

2) INTERFACES

2.4) DEFAULT METHODS

- We may have realized that by allowing default methods in interfaces, coupled with the fact a class may implement multiple interfaces, Java has essentially opened the door to multiple inheritance problems.

```
public interface Walk {  
    public default int getSpeed() {  
        return 5;  
    }  
  
    public interface Run {  
        public default int getSpeed() {  
            return 10;  
        }  
    }  
  
    public class Cat implements Walk, Run {  
        // DOES NOT COMPILE  
        public static void main(String[] args) {  
            System.out.println(new Cat().getSpeed());  
        }  
    }  
}
```

- In this example, Cat inherits the two **default** methods for getSpeed(), so which does it use?
- Since Walk and Run are considered siblings in terms of how they are used in the Cat class, it is not clear whether the code should output 5 or 10.
- The answer is that the code outputs neither value, it fails to compile.
- If a class implements two interfaces that have default methods with the same name and signature, the compiler will throw an error.
- There is an exception to this rule, though: if the subclass overrides the duplicate default methods, the code will compile without issue - the ambiguity about which version of the method to call has been removed.

2) INTERFACES

2.4) DEFAULT METHODS

- For example, the following modified implementation of Cat will compile and output 1:

```
public class Cat implements Walk, Run {  
  
    public int getSpeed() {  
        return 1;  
    }  
  
    public static void main(String[] args) {  
        System.out.println(new Cat().getSpeed());  
    }  
}
```

- We can see that having a class that implements or inherits two duplicate default methods forces the class to implement a new version of the method, or the code will not compile.
- This rule holds true even for abstract classes that implement multiple interfaces, because the default method could be called in a concrete method within the abstract class.

2) INTERFACES

2.5) STATIC METHODS

- Java 8 also now includes support for static methods within interfaces. These methods are defined explicitly with the `static` keyword and function nearly identically to static methods defined in classes.
- In fact, there is really only one distinction between a static method in a class and an interface.
- **A static method defined in an interface is not inherited in any classes that implement the interface.**
- Here are the static interface method rules you need to be familiar with:
 1. Like all methods in an interface, a **static** method is assumed to be **public** and will not compile if marked as **private** or **protected**.
 2. To reference the **static** method, a reference to the name of the interface must be used.

```
public interface Hop {  
    static int getJumpHeight() {  
        return 8;  
    }  
}
```

- The method `getJumpHeight()` works just like a **static** method as defined in a class.
- In other words, it can be accessed without an instance of the class using the `Hop.getJumpHeight()` syntax.
- Also, note that the compiler will automatically insert the access modifier **public** since all methods in interfaces are assumed to be **public**.

2) INTERFACES

2.5) STATIC METHODS

- The following is an example of a class Bunny that implements Hop:

```
public class Bunny implements Hop {  
    // DOES NOT COMPILE  
    public void printDetails() {  
        System.out.println(getJumpHeight());  
    }  
}
```

- As we can see, without an explicit reference to the name of the interface the code will not compile, even though Bunny **implements** Hop.
- In this manner, the static interface methods are not inherited by a class implementing the interface.
- The following modified version of the code resolves the issue with a reference to the interface name Hop:

```
public class Bunny implements Hop {  
    // DOES NOT COMPILE  
    public void printDetails() {  
        System.out.println(Hop.getJumpHeight());  
    }  
}
```

- It follows, then, that a class that implements two interfaces containing static methods with the same signature will still compile at runtime, because the static methods are not inherited by the subclass and must be accessed with a reference to the interface name.
- Contrast this with the behavior we saw for default interface methods in the previous section: the code would compile if the subclass overrode the default methods and would fail to compile otherwise.
- We can see that static interface methods have none of the same multiple inheritance issues and rules as default interface methods do.