JAVA CHALLENGERS
By Rafael del Nero, Java Developer, InfoWorld
JUN 28, 2022 3:00 AM PDT

About
Tease your mind and test your learning, with these quick introductions to challenging concepts in Java programming.

Abstract classes vs. interfaces in Java

When should you choose an abstract class over an interface in Java? Take the challenge! Learn the difference between these Java language elements and how to use them in your programs.

Abstract classes and interfaces are plentiful in Java code, and even in the Java Development Kit (JDK) itself. Each code element serves a fundamental purpose:

- *Interfaces* are a kind of *code contract*, which must be implemented by a concrete class.
- *Abstract classes* are similar to normal classes, with the difference that they can include *abstract methods*, which are methods without a body. Abstract classes cannot be instantiated.

Many developers believe that interfaces and abstract classes are similar, but they are actually quite different. Let's explore the main differences between them.

The essence of an interface

At heart, an interface is a contract, so it depends on an implementation to serve its purpose. An interface *can never have a state*, so it cannot use mutable instance variables. An interface can only use final variables.

[Also on InfoWorld: 7 reasons Java is still great]

When to use interfaces

Interfaces are very useful for decoupling code and implementing polymorphism. We can see an example in the JDK, with the `List` interface:

```
public interface List<E> extends Collection<E> {

    int size();
    boolean isEmpty();
    boolean add(E e);
    E remove(int index);
    void clear();
}
```

As you likely noticed, this code is short and very descriptive. We can easily see the *method signature*, which we'll use to implement the methods in the interface using a concrete class.

The `List` interface contains a contract that can be implemented by the `ArrayList`, `Vector`, `LinkedList`, and other classes.

To use polymorphism, we can simply declare our variable type with `List`, and then choose any of the available instantiations. Here's an example:


```
List list = new ArrayList();
System.out.println(list.getClass());

 List list = new LinkedList();
 System.out.println(list.getClass());
```

Here is the output from this code:

```
class java.util.ArrayList
class java.util.LinkedList
```

In this case, the implementation methods for `ArrayList`, `LinkedList`, and `Vector` are all different, which is a great scenario for using an interface. If you notice that many classes belong to a parent class with the same method actions but different behavior, then it's a good idea to use an interface.

Next, let's look at a few of the things we can do with interfaces.

Overriding an interface method

Remember that an interface is a kind of contract that must be implemented by a concrete class. Interface methods are implicitly abstract, and also require a concrete class implementation.

Here's an example:

```
public class OverridingDemo {
  public static void main(String[] args) {
    Challenger challenger = new JavaChallenger();
    challenger.doChallenge();
  }
}

interface Challenger {
  void doChallenge();
}

class JavaChallenger implements Challenger {
  @Override
  public void doChallenge() {
    System.out.println("Challenge done!");
  }
}
```

Here's the output from this code:

```
Challenge done!
```

Notice the detail that interface methods are *implicitly abstract*. This means we don't need to explicitly declare them as abstract.

Constant variables

Another rule to remember is that an interface can only contain constant variables. Thus, the following code is fine:

```
public interface Challenger {

  int number = 7;
  String name = "Java Challenger";

}
```

Notice that both variables are implicitly `final` and `static`. This means they are constants, do not depend on an instance, and can't be changed.

If we try to change the variables in the `Challenger` interface, say, like this:

```
Challenger.number = 8;
Challenger.name = "Another Challenger";
```

we will trigger a compilation error, like this one:

```
Cannot assign a value to final variable 'number'
Cannot assign a value to final variable 'name'
```

Default methods

When default methods were introduced in Java 8, some developers thought they would be the same as abstract classes. That's not true, however, because interfaces can't have state.

A default method can have an implementation, whereas abstract methods can't. Default methods are the result of great innovations with lambdas and streams, but we should use them with caution.

A method in the JDK that uses a default method is `forEach()`, which is part of the `Iterable` interface. Instead of copying code to every `Iterable` implementation, we can simply reuse the `forEach` method:

```
default void forEach(Consumer<? super T> action) {
   // Code implementation here…
```

Any `Iterable` implementation can use the `forEach()` method without requiring a new method implementation. Then, we can reuse the code with a default method.

Let's create our own default method:

```
public class DefaultMethodExample {

  public static void main(String[] args) {
    Challenger challenger = new JavaChallenger();
    challenger.doChallenge();
  }

}

class JavaChallenger implements Challenger { }

interface Challenger {

  default void doChallenge() {
    System.out.println("Challenger doing a challenge!");
  }
}
```

Here's the output:

```
Challenger doing a challenge!
```

The important thing to notice about default methods is that each default method needs an implementation. A default method cannot be static.

Now, let's move on to abstract classes.

The essence of an abstract class

Abstract classes can have state with instance variables. This means that an instance variable can be used and mutated. Here's an example:

```
public abstract class AbstractClassMutation {

  private String name = "challenger";

  public static void main(String[] args) {
    AbstractClassMutation abstractClassMutation = new AbstractClassImpl();
    abstractClassMutation.name = "mutated challenger";
    System.out.println(abstractClassMutation.name);
  }

}

class AbstractClassImpl extends AbstractClassMutation { }
```

Here is the output:

```
mutated challenger
```

Abstract methods in abstract classes

Just like interfaces, abstract classes can have abstract methods. An *abstract method* is a method without a body. Unlike in interfaces, abstract methods in abstract classes must be explicitly declared as abstract. Here's an example:

```
public abstract class AbstractMethods {

  abstract void doSomething();

}
```

Attempting to declare a method without an implementation, and without the `abstract` keyword, like this:

```
public abstract class AbstractMethods {
   void doSomethingElse();
}
```

results in a compilation error, like this:

```
Missing method body, or declare abstract
```

When to use abstract classes

It's a good idea to use an abstract class when you need to implement mutable state. As an example, the Java Collections Framework includes the AbstractList class, which uses the state of variables.

In cases where you don't need to maintain the state of the class, it's usually better to use an interface.

Abstract classes in practice
The design pattern template method is good example of using abstract classes. The template method pattern manipulates instance variables within concrete methods.

Differences between abstract classes and interfaces

From an object-oriented programming perspective, the main difference between an interface and an abstract class is that an interface *cannot* have state, whereas the abstract class can have state with instance variables.

Another key difference is that classes can implement more than one interface, but they can extend only one abstract class. This is a design decision based on the fact that multiple inheritance (extending more than one class) can cause code deadlocks. Java's engineers decided to avoid that.

Another difference is that interfaces can be implemented by classes or extended by interfaces, but classes can be only extended.

It's also important to note that lambda expressions can only be used with a functional interface (meaning an interface with only one method), while abstract classes with only one abstract method *cannot* use lambdas.

Table 1 summarizes the differences between abstract classes and interfaces.

Table 1. Comparing interfaces and abstract classes

Interfaces	Abstract classes
Can only have final static variables. An interface can never change its own state.	Can have any kind of instance or static variables, mutable or immutable.
A class can implement multiple interfaces.	A class can extend only one abstract class.
Can be implemented with the `implements` keyword. An interface can also `extend` interfaces.	Can only be extended.
Can only use static final fields, parameters, or local variables for methods.	Can have instance mutable fields, parameters, or local variables.
Only functional interfaces can use the lambda feature in Java.	Abstract classes with only one abstract method cannot use lambdas.
Can't have constructor.	Can have constructor.

Interfaces	Abstract classes
Can have abstract methods. Can have default and static methods (introduced in Java 8). Can have private methods with the implementation (introduced in Java 9).	Can have any kind of methods.

Take the Java code challenge!

Let's explore the main differences between interfaces and abstract classes with a Java code challenge. We have the code challenge below, or you can view the abstract classes vs. interfaces challenge in a video format.

In the following code, both an interface and an abstract class are declared, and the code also uses lambdas.

```
public class AbstractResidentEvilInterfaceChallenge {
  static int nemesisRaids = 0;
  public static void main(String[] args) {
    Zombie zombie = () -> System.out.println("Graw!!! " + nemesisRaids++);
    System.out.println("Nemesis raids: " + nemesisRaids);
    Nemesis nemesis = new Nemesis() { public void shoot() { shoots = 23; }};

    Zombie.zombie.shoot();
    zombie.shoot();
    nemesis.shoot();
    System.out.println("Nemesis shoots: " + nemesis.shoots +
        " and raids: " + nemesisRaids);
  }
}
interface Zombie {
  Zombie zombie = () -> System.out.println("Stars!!!");
  void shoot();
}
abstract class Nemesis implements Zombie {
   public int shoots = 5;
}
```

What do you think will happen when we run this code? Choose one of the following:

Option A

```
    Compilation error at line 4
```

Option B

```
    Graw!!! 0
    Nemesis raids: 23
    Stars!!!
    Nemesis shoots: 23 and raids:1
```

Option C

```
Nemesis raids: 0
Stars!!!
Graw!!! 0
Nemesis shoots: 23 and raids: 1
```

Option D

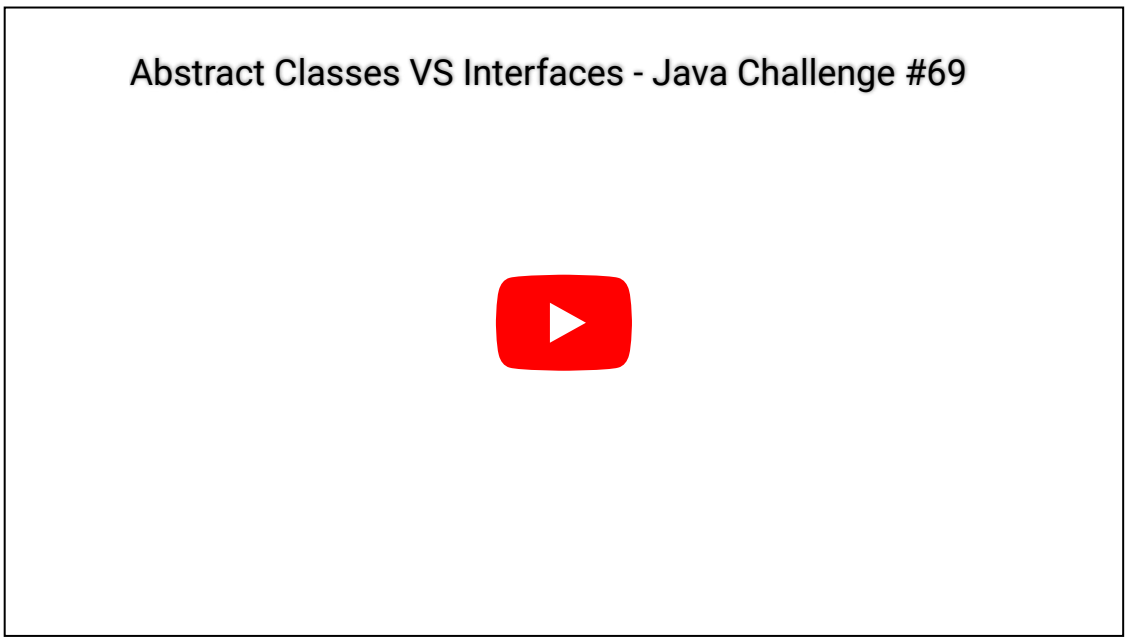
```
Nemesis raids: 0
Stars!!!
Graw!!! 1
Nemesis shoots: 23 and raids:1
```

Option E

```
Compilation error at line 6
```

Java code challenge video

Have you selected the correct output for this challenge? Watch the video or keep reading to find out.



Understanding interfaces and abstract classes and methods

This Java code challenge demonstrates many important concepts about interfaces, abstract methods, and more. Stepping through the code line by line will teach us a lot about what is happening in the output.

The first line of the code challenge includes a lambda expression for the `Zombie` interface. Notice that in this lambda we are incrementing a static field. An instance field would also work here, but a local variable declared outside of a lambda would not. Therefore, so far, the code will compile fine. Also notice that the lambda expression has not yet executed, so the `nemesisRaids` field won't be incremented just yet.

At this point, we will print the `nemesisRaids` field, which is not incremented because the lambda expression hasn't yet been invoked, only declared. Therefore, the output from this line will be:

```
Nemesis raids: 0
```

Another interesting concept in this Java code challenge is that we are using an *anonymous inner class*. This basically means any class that will implement the methods from the `Nemesis` abstract class. We're not really instantiating the `Nemesis` abstract class because it's actually an annonymous class. Also notice that the first concrete class will always be obliged to implement the abstract methods when extending them.

Inside the `Zombie` interface, we have the `zombie static Zombie` interface declared with a lambda expression. Therefore, when we invoke the `zombie shoot` method, we print the following:

```
Stars!!!
```

The next line of code invokes the lambda expression we created at the start. Therefore, the `nemesisRaids` variable will be incremented. However, because we are using the post-increment operator, it will be incremented only after this code

statement. The next output will be:

```
Graw!!! 0
```

Now, we will invoke the `shoot` method from `nemesis` which will change its `shoots` instance variable to `23`. Note that this part of the code demonstrates the biggest difference between an interface and an abstract class.

Finally, we print the value of `nemesis.shoots` and `nemesisRaids`. Therefore, the output will be:

```
Nemesis shoots: 23 and raids: 1
```

In conclusion, the correct output is option C:

```
    Nemesis raids: 0
    Stars!!!
    Graw!!! 0
    Nemesis shoots: 23 and raids: 1
```

Learn more about Java

- Get more quick code tips: Read all of Rafael's articles in the InfoWorld Java Challengers series.
- See the Java 101 Java interfaces tutorial for a more in-depth introduction to using interfaces in your Java programs, including where and where not to use default, static, and private methods.
- If you liked the video for this Java code challenger, check out other videos in Rafael's Java Challengers video playlist.
- Find even more Java Challengers on Rafael's Java Challengers blog and in his book, with more than 70 code challenges.

Next read this:

- *Cloud computing is no longer a slam dunk*

- *What is generative AI? Artificial intelligence that creates*
- *Coding with AI: Tips and best practices from developers*
- *Python moves to remove the GIL and boost concurrency*
- *7 reasons Java is still great*
- *The open source licensing war is over*

How to choose a low-code development platform