**The Java Main Class**

If only a single Java class in your Java program contains a main() method, then the class containing the main() method is often referred to as the *main class*.

You can have as many classes as you want in your project with a main() method in. But, the Java Virtual Machine can only be instructed to run one of them at a time. You can still call the other main() methods from inside the main() method the Java Virtual Machine executes (you haven't seen how yet) and you can also start up multiple virtual machines which each execute a single main() method.

**Java Local-Variable Type Inference**

From Java 10 it is possible to have the Java compiler infer the type of a local variable by looking at what actual type that is assigned to the variable when the variable is declared. This enhancement is restricted to local variables, indexes in for-each loops and local variables declared in for-loops.

To see how the Java local-variable type inference works, here is first an example of a pre Java 10 String variable declaration:

String myVar = "A string!";

From Java 10 it is no longer necessary to specify the type of the variable when declared, if the type can be inferred from the value assigned to the variable. Here is an example of declaring a variable in Java 10 using local-variable type inference:

var myVar = "A string!";

Notice the var keyword used in front of the variable name, instead of the type String. The compiler can see from the value assigned that the type of the variable should be String, so you don't have to write it explicitly.

Here are a few additional Java local-variable type inference examples:

var list = new ArrayList();

var myNum = new Integer(123);

var myClassObj = new MyClass();

**String Literals as Constants or Singletons**

If you use the same string (e.g. "Hello World") in other String variable declarations, the Java virtual machine may only create a single String instance in memory. The string literal thus becomes a de facto constant or singleton. The various different variables initialized to the same constant string will point to the same String instance in memory. Here is a Java String constant / singleton example:

String myString1 = "Hello World";

String myString2 = "Hello World";

In this case the Java virtual machine will make both myString1 and myString2 point to the same String object.

More precisely, objects representing Java String literals are obtained from a constant String pool which the Java virtual machine keeps internally. That means, that even classes from different projects compiled separately, but which are used in the same application may share constant String objects. The sharing happens at runtime. It is not a compile time feature.

If you want to be sure that two String variables point to separate String objects, use the new operator like this:

String myString1 = new String("Hello World");

String myString2 = new String("Hello World");

Even though the value (text) of the two Java Strings created is the same, the Java virtual machine will create two different objects in memory to represent them.

**Concatenating Strings**

Concatenating Strings means appending one string to another. Strings in Java are immutable meaning they cannot be changed once created. Therefore, when concatenating two Java String objects to each other, the result is actually put into a third String object.

Here is a Java String concatenation example:

String one = "Hello";

String two = "World";

String three = one + " " + two;

The content of the String referenced by the variable three will be Hello World; The two other Strings objects are untouched.

**Converting Numbers to Strings With valueOf()**

The Java String class contains a set of overloaded static methods named valueOf() which can be used to convert a number to a String. Here are some simple Java String valueOf() examples:

String intStr = String.valueOf(10);

System.out.println("intStr = " + intStr);

String flStr = String.valueOf(9.99);

System.out.println("flStr = " + flStr);

The output printed from this code would be:

intStr = 10

flStr = 9.99

**Initial Field Value**

A Java field can have be given an initial value. This value is assigned to the field when the field is created in the JVM. Static fields are created when the class is loaded. A class is loaded the first time it is referenced in your program. Non-static fields are created when the object owning them are created.

Here is an example of a Java field being declared with an initial value:

public class Customer {

String customerType = "OnlineCustomer";

}**Class Access Modifiers**

It is important to keep in mind that the Java access modifier assigned to a Java class takes precedence over any access modifiers assigned to fields, constructors and methods of that class. If the class is marked with the default access modifier, then no other class outside the same Java package can access that class, including its constructors, fields and methods. It doesn't help that you declare these fields public, or even public static.

The Java access modifiers private and protected cannot be assigned to a class. Only to constructors, methods and fields inside classes. Classes can only have the default (package) and public access modifier assigned to them.

########## IMP POINTS ##########

**What is Inherited?**

When a subclass extends a superclass in Java, all protected and public fields and methods of the superclass are inherited by the subclass. By *inherited* is meant that these fields and methods are part of of the subclass, as if the subclass had declared them itself. protected and public fields can be called and referenced just like the methods declared directly in the subclass.

Fields and methods with default (package) access modifiers can be accessed by subclasses only if the subclass is located in the same package as the superclass. Private fields and methods of the superclass can never be referenced directly by subclasses. They can, however, be referenced indirectly via methods reachable from the subclass (e.g default (package), protected and public methods).

Constructors are not inherited by subclasses, but a subclass constructor must call a constructor in the superclass. This will be explained in detail in a later section.

**Inheritance and Type Casting**

It is possible to reference a subclass as an instance of one of its superclasses. For instance, using the class definitions from the example in the previous section it is possible to reference an instance of the Car class as an instance of the Vehicle class. Because the Car class extends (inherits from) the Vehicle class, it is also said to **be** a Vehicle.

Here is a Java code example:

Car car = new Car();

Vehicle vehicle = car;

First a Car instance is created. Second, the Car instance is assigned to a variable of type Vehicle. Now the Vehicle variable (reference) points to the Car instance. This is possible because the Car class inherits from the Vehicle class.

As you can see, it is possible to use an instance of some subclass as if it were an instance of its superclass. That way, you don't need to know exactly what subclass the object is an instance of. You could treat e.g. both Truck and Car instances as Vehicle instances.

The process of referencing an object of class as a different type than the class itself is called type casting. You cast an object from one type to another.

In Java you cannot override private methods from a superclass. If the superclass calls a private method internally from some other method, it will continue to call that method from the superclass, even if you create a private method in the subclass with the same signature

**Interface Methods**

A Java interface can contain one or more method declarations. As mentioned earlier, the interface cannot specify any implementation for these methods. It is up to the classes implementing the interface to specify an implementation.

All methods in an interface are public, even if you leave out the public keyword in the method declaration.

**Interface Static Methods**

A Java interface can have static methods. Static methods in a Java interface must have implementation. Here is an example of a static method in a Java interface:

public interface MyInterface {

public static void print(String text){

System.out.print(text);

}

}

Calling a static method in an interface looks and works just like calling a static method in a class. Here is an example of calling the static print() method from the above MyInterface interface:

MyInterface.print("Hello static method!");

Static methods in interfaces can be useful when you have some utility methods you would like to make available, which fit naturally into an interface related to the same responsibility. For instance, a Vehicle interface could have a printVehicle(Vehicle v) static method.

**Interfaces and Inheritance**

It is possible for a Java interface to inherit from another Java interface, just like classes can inherit from other classes. You specify inheritance using the extends keyword. Here is a simple interface inheritance example:

public interface MySuperInterface {

public void saiHello();

}

public interface MySubInterface **extends MySuperInterface** {

public void sayGoodbye();

}

The interface MySubInterface extends the interface MySuperInterface. That means, that the MySubInterface inherits all field and methods from MySuperInterface. That then means, that if a class implements MySubInterface, that class has to implement all methods defined in both MySubInterface and MySuperInterface.

It is possible to define methods in a subinterface with the same signature (name + parameters) as methods defined in a superinterface, should you find that desirable in your design, somehow.

Unlike classes, interfaces can actually inherit from multiple superinterfaces. You specify that by listing the names of all interfaces to inherit from, separated by comma. A class implementing an interface which inherits from multiple interfaces must implement all methods from the interface and its superinterfaces.

Here is an example of a Java interface that inherits from multiple interfaces:

public interface MySubInterface extends

SuperInterface1, SuperInterface2 {

public void sayItAll();

}

As when implementing multiple interfaces, there are no rules for how you handle the situation when multiple superinterfaces have methods with the same signature (name + parameters).

**Inheritance and Default Methods**

Interface default methods add a bit complexity to the rules of interface inheritance. While it is normally possible for a class to implement multiple interfaces even if the interfaces contain methods with the same signature, this is not possible if one or more of these methods are default methods. In other words, if two interfaces contain the same method signature (name + parameters) and one of the interfaces declare this method as a default method, a class cannot automatically implement both interfaces.

The situation is the same if an interface extends (inherits from) multiple interfaces, and one or more of these interfaces contain methods with the same signature, and one of the superinterfaces declare the overlapping method as a default method.

In both of the above situations the Java compiler requires that the class implementing the interface(s) explicitly implements the method which causes the problem. That way there is no doubt about which implementation the class will have. The implementation in the class takes precedence over any default implementations.

**Interfaces and Polymorphism**

Java interfaces are a way to achieve polymorphism. Polymorphism is a concept that takes some practice and thought to master. Basically, polymorphism means that an instance of an class (an object) can be used as if it were of different types. Here, a type means either a class or an interface.

Look at this simple class diagram:

|  |
| --- |
| Two parallel class hierarchies used in the same application. |
| **Two parallel class hierarchies used in the same application.** |

The classes above are all parts of a model representing different types of vehicles and drivers, with fields and methods. That is the responsibility of these classes - to model these entities from real life.

Now imagine you needed to be able to store these objects in a database, and also serialize them to XML, JSON, or other formats. You want that implemented using a single method for each operation, available on each Car, Truck or Vehicle object. A store() method, a serializeToXML() method and a serializeToJSON() method.

Please forget for a while, that implementing this functionality as methods directly on the objects may lead to a messy class hierarchy. Just imagine that this is how you want the operations implemented.

Where in the above diagram would you put these three methods, so they are accessible on all classes?

One way to solve this problem would be to create a common superclass for the Vehicle and Driver class, which has the storage and serialization methods. However, this would result in a conceptual mess. The class hierarchy would no longer model vehicles and drivers, but also be tied to the storage and serialization mechanisms used in your application.

A better solution would be to create some interfaces with the storage and serialization methods on, and let the classes implement these interfaces. Here are examples of such interfaces:

public interface Storable {

public void store();

}

public interface Serializable {

public void serializeToXML(Writer writer);

public void serializeToJSON(Writer writer);

}

When each class implements these two interfaces and their methods, you can access the methods of these interfaces by casting the objects to instances of the interface types. You don't need to know exactly what class a given object is of, as long as you know what interface it implements. Here is an example:

Car car = new Car();

Storable storable = (Storable) car;

storable.store();

Serializable serializable = (Serializable) car;

serializable.serializeToXML (new FileWriter("car.xml"));

serializable.serializeToJSON(new FileWriter("car.json"));

As you can probably imagine by now, interfaces provide a cleaner way of implementing cross cutting functionality in classes than inheritance.

**Generic Interfaces**

A generic Java interface is an interface which can be typed - meaning it can be specialized to work with a specific type (e.g. interface or class) when used. Let me first create a simple Java interface that contains a single method:

public interface MyProducer() {

public Object produce();

}

This interface represents an interface which contains a single method called produce() which can produce a single object. Since the return value of produce() is Object, it can return any Java object.

Here is a class that implements the MyProducer interface:

public class CarProducer implements MyProducer{

public Object produce() {

return new Car();

}

}

The above class CarProducer implements the MyProducer interface. The implementation of the produce() method returns a new Car object every time it is called. Here is how it looks to use the CarProducer class:

MyProducer carProducer = new CarProducer();

Car car = (Car) carProducer.produce();

Notice how the object returned from the carProducer.produce() method call has to be cast to a Car instance, because the produce() method return type is Object. Using Java Generics you can *type* the MyProducer interface so you can specify what type of object it produces when you use it. Here is first a generic version of the MyProducer interface:

public interface MyProducer <T>{

public T produce();

}

Now when I implement the MyProducer interface in the CarProducer class, I have to include the generic type declaration too, like this:

public class CarProducer<T> implements MyProducer<T>{

@Override

public T produce() {

return (T) new Car();

}

}

Now, when creating a CarProducer I can specify its generic interface type, like this:

MyProducer<Car> myCarProducer = new CarProducer<Car>();

Car produce = myCarProducer.produce();

As you can see, since the generic type for the CarProducer instance is set to Car, it is no longer necessary to cast the object returned from the produce() method, since the original method declaration in the MyProducer interface states, that this method returns the same type as is specified in the generic type when used.

But - now it is actually possible to specify another generic type for a CarProducer instance than the type it actually returns from it's produce() method implementation. If you scroll up, you can see that the CarProducer.produce() implementation returns a Car object no matter what generic type you specify for it when you create it. So, the following declaration is possible, but would return in a ClassCastException when executed:

MyProducer<String> myStringProducer = new CarProducer<String>();

String produce1 = myStringProducer.produce();

Instead, you can lock down the generic type of the MyProducer interface already when you implement it, in the CarProducer class. Here is an example of specifying the generic type of a generic interface when implementing it:

public class CarProducer implements MyProducer<Car>{

@Override

public Car produce() {

return new Car();

}

}

Now you cannot specify the generic type of the CarProducer when using it. It is already typed to Car. Here is how using the CarProducer looks:

MyProducer<Car> myCarProducer = new CarProducer();

Car produce = myCarProducer.produce();

As you can see, it is still not necessary to cast the object returned by produce(), as the CarProducer implementation declares that to be a Car instance.

Java generics is covered in more detail in my [**Java Generics Tutorial**](https://jenkov.com/java-generics/index.html) .

**Functional Interfaces**

From Java 8 a new concept was introduced called *functional interfaces*. In short, a functional interface is an interface with a single, unimplemented method (non-default, non-static method). I have explained functional interfaces in my [**Java functional interface tutorial**](https://jenkov.com/java-functional-programming/functional-interfaces.html), which is part of my [**Java Functional Programming Tutorial**](https://jenkov.com/java-functional-programming/index.html). .

Functional interfaces are often intended to be implemented by a [**Java Lambda Expression**](https://jenkov.com/java/lambda-expressions.html).

**Interface Default Methods**

Before Java 8 Java interfaces could not contain an implementation of the methods, but only contain the method signatures. However, this results in some problems when an API needs to add a method to one of its interfaces. If the API just adds the method to the desired interface, all classes that implements the interface must implement that new method. That is fine if all implementing classes are located within the API. But if some implementing classes are located in client code of the API (the code that uses the API), then that code breaks.

Let me illustrate this with an example. Look at this interface and imagine that it is part of e.g. an open source API which many applications are using internally:

public interface ResourceLoader {

Resource load(String resourcePath);

}

Now imagine that a project uses this API and has implemented the ResourceLoader interface like this:

public class FileLoader implements ResourceLoader {

public Resource load(String resourcePath) {

// in here is the implementation +

// a return statement.

}

}

If the developer of the API wants to add one more method to the ResourceLoader interface, then the FileLoader class will be broken when that project upgrades to the new version of the API.

To alleviate this Java interface evolution problem, Java 8 has added the concept of *interface default methods* to Java interfaces. An interface default method can contain a default implementation of that method. Classes that implement the interface but which contain no implementation for the default interface will then automatically get the default method implementation.

You mark a method in an interface as a default method using the default keyword. Here is an example of adding a default method to the ResourceLoader interface:

public interface ResourceLoader {

Resource load(String resourcePath);

default Resource load(Path resourcePath) {

// provide default implementation to load

// resource from a Path and return the content

// in a Resource object.

}

}

This example adds the default method load(Path). The example leaves out the actual implementation (inside the method body) because this is not really interesting. What matters is how you declare the interface default method.

A class can override the implementation of a default method simply by implementing that method explicitly, as is done normally when implementing a Java interface. Any implementation in a class takes precedence over interface default method implementations.

**Important Points about Default Methods:**

* **Multiple Default Methods**: If a class implements multiple interfaces that each have a default method with the same signature, the class must explicitly override the method, otherwise, it will cause a compilation error due to the ambiguity.

class Point {

int x, y;

private Point() { reset(); }

Point(int x, int y) { this.x = x; this.y = y; }

private void reset() { this.x = 0; this.y = 0; }

}

class ColoredPoint extends Point {

int color;

void clear() { reset(); } // error

}

class Test {

public static void main(String[] args) {

ColoredPoint c = new ColoredPoint(0, 0); // error

c.reset(); // error

}

}

*This program causes four compile-time errors.*

*One error occurs because ColoredPoint has no constructor declared with two int parameters, as requested by the use in main. This illustrates the fact that ColoredPoint does not inherit the constructors of its superclass Point.*

*Another error occurs because ColoredPoint declares no constructors, and therefore a default constructor for it is implicitly declared (*[*§8.8.9*](https://docs.oracle.com/javase/specs/jls/se23/html/jls-8.html#jls-8.8.9)*), and this default constructor is equivalent to:*

ColoredPoint() { super(); }

*which invokes the constructor, with no arguments, for the direct superclass of the class ColoredPoint. The error is that the constructor for Point that takes no arguments is private, and therefore is not accessible outside the class Point, even through a superclass constructor invocation (*[*§8.8.7*](https://docs.oracle.com/javase/specs/jls/se23/html/jls-8.html#jls-8.8.7)*).*

*Two more errors occur because the method reset of class Point is private, and therefore is not inherited by class ColoredPoint. The method invocations in method clear of class ColoredPoint and in method main of class Test are therefore not correct.*

**Example 8.2-2. Inheritance of Class Members with Package Access**

*Consider the example where the points package declares two compilation units:*

package points;

public class Point {

int x, y;

public void move(int dx, int dy) { x += dx; y += dy; }

}

*and:*

package points;

public class Point3d extends Point {

int z;

public void move(int dx, int dy, int dz) {

x += dx; y += dy; z += dz;

}

}

*and a third compilation unit, in another package, is:*

import points.Point3d;

class Point4d extends Point3d {

int w;

public void move(int dx, int dy, int dz, int dw) {

x += dx; y += dy; z += dz; w += dw; // compile-time errors

}

}

*Here both classes in the points package compile. The class Point3d inherits the fields x and y of class Point, because it is in the same package as Point. The class Point4d, which is in a different package, does not inherit the fields x and y of class Point or the field z of class Point3d, and so fails to compile.*

*A better way to write the third compilation unit would be:*

import points.Point3d;

class Point4d extends Point3d {

int w;

public void move(int dx, int dy, int dz, int dw) {

super.move(dx, dy, dz); w += dw;

}

}

*using the move method of the superclass Point3d to process dx, dy, and dz. If Point4d is written in this way, it will compile without errors.*