

For example, a sports car is an object. Some of its members might be its height, weight, acceleration, and speed. An object's members just hold data about that object. Some of the methods of the sports car could be "drive", "park", "race", etc. The methods really don't mean much unless associated with the sports car, and the same goes for the members.

The blueprint that lets us build our sports car object is called a **class**. A class doesn't tell us how fast our sports car goes, or what color it is, but it does tell us that our sports car will have a member representing speed and color, and that they will be say, a number and a word (or hex color code), respectively. The class also lays out the methods for us, telling the car how to park and drive, but these methods can't take any action with just the blueprint - they need an object to have an effect.

In Java, a class is located in a file similar to its own name. If you want to have a class called `SportsCar`, its source file needs to be `SportsCar.java`. The class is created by placing the following in the source file:

```
public class SportsCar
{
    /* Insert your code here */
}
```

The class doesn't do anything yet, as you will need to add methods and member variables first.

Instantiation and Constructors

In order to get from class to object, we "build" our object by **instantiation**. Instantiation simply means to create an **instance** of a class. Instance and object are very similar terms and are sometimes interchangeable, but remember that an instance refers to a *specific object*, which was created from a class.

This instantiation is brought about by one of the class's methods, called a **constructor**. As its name implies, a constructor builds the object based on the blueprint. Behind the scenes, this means that computer memory is being allocated for the instance, and values are being assigned to the data members.

In general there are four constructor types: default, non-default, copy, and cloning.

A **default constructor** will build the most basic instance. Generally, this means assigning all the members values like null, zero, or an empty string. Nothing would stop you, however, from your default sports car color from being red, but this is generally bad programming style. Another programmer would be confused if your basic car came out red instead of say, colorless.

A **non-default constructor** is designed to create an object instance with prescribed values for most, if not all, of the object's members. The car is red, goes from 0-60 in 12 seconds, tops out at 190mph, etc.

A **copy constructor** is not included in the Java language, however one can easily create a constructor that do the same as a copy constructor. It's important to understand what it is. As the name implies, a copy constructor creates a new instance to be a duplicate of an already existing one. In Java, this can be also accomplished by creating the instance with the default constructor, and then using the assignment operator to equivocate them. This is not possible in all languages though, so just keep the terminology under your belt.

Java has the concepts of **cloning object**, and the end results are similar to copy constructor. Cloning an object is faster than creation with the `new` keyword, because all the object memory is copied at once to destination cloned object. This is possible by implementing the `Cloneable` interface, which allows the method `Object.clone()` to perform a field-by-field copy.

Type

When an object is created, a reference to the object is also created. The object can not be accessed directly in Java, only through this object reference. This object reference has a **type** assigned to it. We need this type when passing the object reference to a method as a parameter. Java does strong type checking.

Type is basically a list of features/operations, that can be performed through that object reference. The object reference type basically is a contract that guarantees that those operations will be there at run time.

When a car is created, it comes with a list of features/operations listed in the user manual that guarantees that those will be there when the car is used.

When you create an object from a class by default its type is the same as its class. It means that all the features/operations the class defined are there and available, and can be used. See below:

```
(new ClassName()).operations();
```

You can assign this to a variable having the same type as the class:

```
ClassName objRefVariable = new ClassName();  
objRefVariable.operations();
```

You can assign the created object reference to the class super class, or to an interface the class implements:

```
SuperClass objectRef = new ClassName(); // features/operations list  
are defined by the SuperClass class  
..  
Interface inter = new ClassName(); // features/operations list are  
defined by the interface
```

In the car analogy, the created car may have different **Type** of drivers. We create separate user manuals for them, Average user manual, Power user manual, Child user manual, or Handicapped user manual. Each type of user manual describes only those features/operations appropriate for the type of driver. The Power driver may have additional gears to switch to higher speeds, that are not available to other type of users...

When the car key is passed from an adult to a child we replacing the user manuals, that is called **Type Casting**.

In Java, casts can occur in three ways:

- up casting: going up in the inheritance tree, until we reach the Object
- up casting: to an interface the class implements
- down casting: until we reach the class the object was created from

Type and **Type Casting** will be covered in more details later at 'Java Programming/Types' module.

Multiple classes in a Java file

Normally, a Java file can have one and only one **public** Java class. However, a given file can contain additional non-public classes.

```
public class OuterClass  
{  
    ...  
}  
class AdditionalClass  
{  
    ...  
}
```

```
}
```

Because they have the "package (default)" access specifier, the 'AdditionalClass' can be accessed only in the same package.

These "additional" classes compile to separate ".class" bytecode files when compiled, just as if they were in separate source files. However, including multiple classes in one file may increase the difficulty in examining the structure of a given application.

External links

- Constructors, interactive Java Lesson ^[24]

Packages

Navigate **Language Fundamentals** topic:

Java Package / Name Space

Usually a Java application is built by many developers and it is common that third party modules/classes are integrated. The end product can easily contain hundreds of classes. Class name collision is likely to happen. To avoid this a Java class can be put in a "name space". This "name space" in Java is called the **package**.

The Java package needs to be unique across Vendors to avoid name collisions. For that reason Vendors usually use their domain name in reverse order. That is guaranteed to be unique. For example a company called 'Your Company Inc.', would use a package name something like this:
com.yourcompany.yourapplicationname.yourmodule.YourClass.

To put a class in a package, the package keyword is used at the top of each class file. For Example,

```
package com.yourcompany.yourapplication.yourmodule;
```

When we want to reference a Java class that is defined outside of the current package name space, we have to specify which package name space that class is in. So we could reference that class with something like *com.yourcompany.yourapplication.yourmodule.YourClass* . To avoid having to type in the package name each time when we want to reference an outside class, we can declare which package the class belongs to by using the **import** Java keyword at the top of the file. For Example,

```
import com.yourcompany.yourapplication.yourmodule.YourClass;
```

Then we can refer to that class by just using the class name *YourClass* .

In rare cases it can happen that you need to reference two classes having the same name in different packages. In those cases, you can not use the import keyword for both classes. One of them needs to be referenced by typing in the whole package name. For Example,

```
package com.mycompany.myapplication.mymodule;
...
import com.yourcompany.yourapplication.youmodule.SameClassName;
...
SameClassName yourObjectRef = new SameClassName();
com.hercompany.herapplication.hermodule.SameClassName herObjectRef =
    new com.hercompany.herapplication.hermodule.SameClassName();
```