**OOPS Concepts**

In this section of the tutorials, we cover basic and advanced OOPS concepts for beginners:

- Object

- Class

- Abstraction

- Encapsulation

- Inheritance

- Polymorphism

- Composition

- Aggregation

- Association

- Cohesion

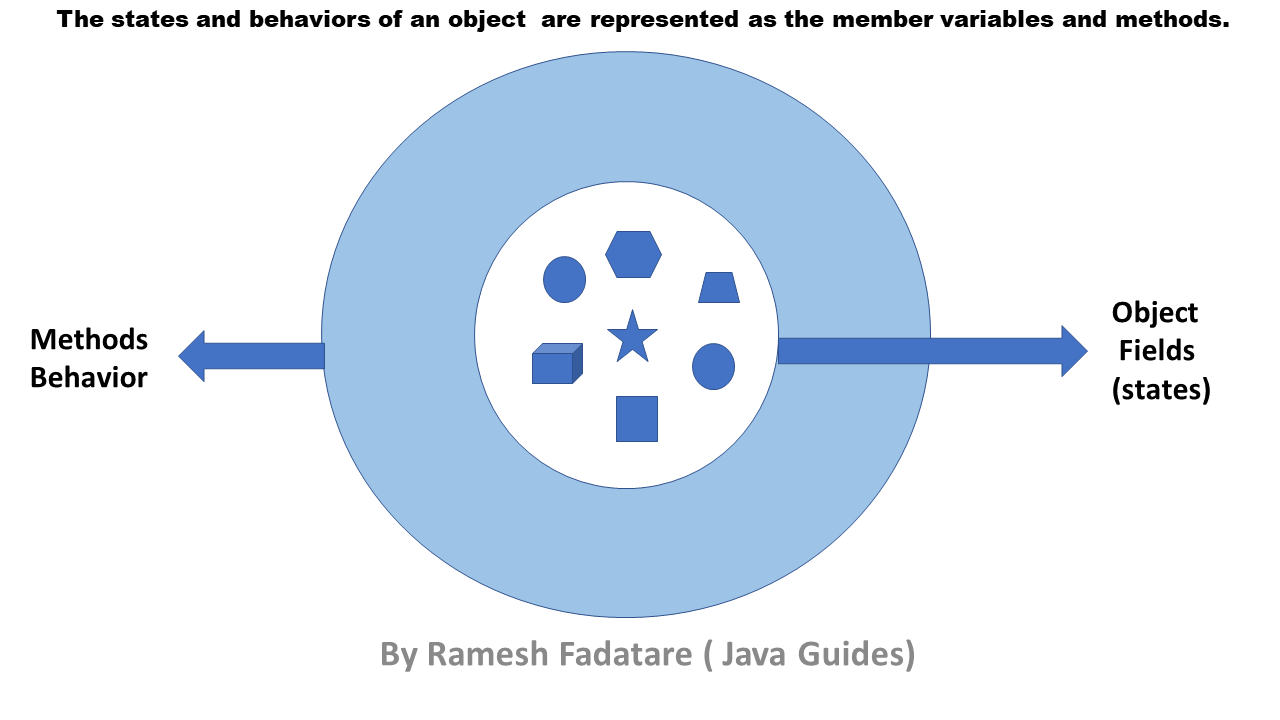
- Coupling

- Delegation

* What is OOPS?
* What is the difference between Procedural programming and OOPS?
* What is an Object?
* What are the advantages of using Software Objects
* How to Declare, Create and Initialize an Object in Java
* 5 Different Ways to Create an Object in Java?
* 1. Using a new keyword
* 2. Using newInstance() method of Class class
* 3. Using newInstance() method of Constructor class
* 4. Using Object Deserialization
* 5. Using Object Cloning – clone() method
* java.lang.Object Class in Java
* java.lang.Object Class Methods with Examples

## What is OOPS?

**Object-Oriented Programming System** is the programming technique to write programs based on the real world objects. The states and behaviors of an object are represented as the member variables and methods. In OOPS programming programs are organized around objects and data rather than actions and logic.

**[](https://1.bp.blogspot.com/-y-EBZmHTwGw/X66scVpnotI/AAAAAAAAIVI/fEYX-4lJd_cfPRxDxWvJ4CZKWCJlGmoYwCPcBGAYYCw/s1280/java+object.png)**

## What is the difference between Procedural programming and OOPS?

## **A procedural language is based on functions object-oriented language is based on real-world objects.**

## **Procedural language gives importance to the sequence of function execution but object-oriented language gives importance on states and behaviors of the objects.**

## **Procedural language exposes the data to the entire program but object-oriented language encapsulates the data.**

## **Procedural language follows a top-down programming paradigm but object-oriented language follows a bottom-up programming paradigm.**

## **A procedural language is complex in nature so it is difficult to modify, extend and maintain but an object-oriented language is less complex in nature so it is easier to modify, extend and maintain.**

## **Procedural language provides less scope of code reuse but object-oriented language provides more scope of code reuse.**

## **Now, you are familiar with what is OOPS and difference between OOPS and Procedural Programming.**

## 

## **Now let's discuss what is Object, how to use Object and what are benefits of using Objects in Java.**

## 

## **Let's get started with what is an object in Java with real-world examples.**

## What is an Object?

## **The Object is the real-time entity having some state and behavior. In Java, Object is an instance of the class having the instance variables as the state of the object and the methods as the behavior of the object. The object of a class can be created by using the new keyword in Java Programming language.**

## **A class is a template or blueprint from which objects are created. So, an object is the instance(result) of a class.**

## I found various Object Definitions:

## **1. An object is a real-world entity.**

## **2. An object is a runtime entity.**

## **3. The object is an entity which has state and behavior.**

## **4. The object is an instance of a class.**

### Real-world examples

## **Dogs have state (name, color, breed, hungry) and behavior (barking, fetching, wagging tail). Chair, Bike, Marker, Pen, Table, Car, Book, Apple, Bag etc. It can be physical or logical (tangible and intangible).**

## <https://3.bp.blogspot.com/-KBL3m5iZghA/W_D9lrv_VnI/AAAAAAAAExI/qc91jvwIq9gHck81Xe0C4fJ2t6Pbpgj-gCLcBGAs/s1600/objects-examples.jpg>

## **Bicycles also have state (current gear, current pedal cadence, current speed) and behavior (changing gear, changing pedal cadence, applying brakes).**

## <https://2.bp.blogspot.com/-fq5NrAxwQTU/W_DpZEAEMhI/AAAAAAAAEw8/nWHUsQ2Xufww2gJxpyREQ0wx2dbD5OhUwCLcBGAs/s1600/concepts-bicycleObject.gif>

## What are the advantages of using Software Objects

Bundling code into individual software objects provides a number of benefits, including:

**Modularity:** The source code for an object can be written and maintained independently of the source code for other objects. Once created, an object can be easily passed around inside the system.

**Information-hiding:**By interacting only with an object's methods, the details of its internal implementation remain hidden from the outside world.

**Code re-use:** If an object already exists (perhaps written by another software developer), you can use that object in your program. This allows specialists to implement/test/debug complex, task-specific objects, which you can then trust to run in your own code.

**Pluggability and debugging ease:** If a particular object turns out to be problematic, you can simply remove it from your application and plug in a different object as its replacement. This is analogous to fixing mechanical problems in the real world. If a bolt breaks, you replace it, not the entire machine.

## How to Declare, Create and Initialize an Object in Java

A class is a blueprint for Object, you can create an object from a class. Let's take Student class and try to create Java object for it.  
  
Let's create a simple *Student* class which has *name* and *college* fields. Let's write a program to create declare, create and initialize a *Student* object in Java.

package net.javaguides.corejava.oops;

public class Student {

private String name;

private String college;

public Student(String name, String college) {

super();

this.name = name;

this.college = college;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getCollege() {

return college;

}

public void setCollege(String college) {

this.college = college;

}

public static void main(String[] args) {

Student student = new Student("Ramesh", "BVB");

Student student2 = new Student("Prakash", "GEC");

Student student3 = new Student("Pramod", "IIT");

}

}

From the above program, the *Student*objects are:

Student student = new Student("Ramesh", "BVB");

Student student2 = new Student("Prakash", "GEC");

Student student3 = new Student("Pramod", "IIT");

Each of these statements has three parts (discussed in detail below):

**Declaration:**The code *Student student;* declarations that associate a variable name with an object type.

**Instantiation:**The *new* keyword is a Java operator that creates the object.

**Initialization:** The *new* operator is followed by a call to a constructor, which initializes the new object.

### Declaring a Variable to Refer to an Object

General syntax:

type name;

This notifies the compiler that you will use a *name* to refer to data whose type is a *type*. With a primitive variable, this declaration also reserves the proper amount of memory for the variable.

From the above program, we can declare variables to refer to an object as:

Student student;

Student student2;

Student student3;

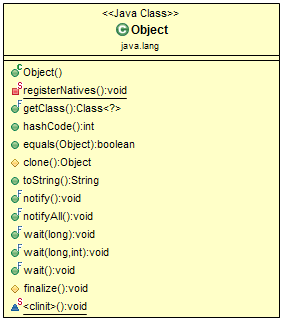
## Object Class Methods

1. **protected Object clone()**
2. **boolean equals(Object obj)**
3. **protected void finalize()**
4. **Class<?> getClass()**
5. **int hashCode()**
6. **void notify()**
7. **void notifyAll()**
8. **void wait()**
9. **String toString()**

The *notify, notifyAll*, and *wait*methods of *Object*all play a part in synchronizing the activities of independently running threads in a program. There are five of these methods:

* public final void notify()
* public final void notifyAll()
* public final void wait()
* public final void wait(long timeout)
* public final void wait(long timeout, int nanos)

The below diagram is a Object class diagram shows a list of methods it provides.

**[](https://1.bp.blogspot.com/-a0Df6omk6Bw/W45SpvecpKI/AAAAAAAADiA/jgMJBECV-24Y4x8f6fcU3TmZWdwg-dzHACLcBGAs/s1600/object-class-diagram.png)**

Let's discuss above each method with examples.

## 1. protected Object clone() Method

*clone()*method creates and returns a copy of this object. The precise meaning of "copy" may depend on the class of the object. The general intent is that for any object x, the expression:

x.clone() != x

will be true, and that the expression:

x.clone().getClass() == x.getClass()

will be true, but these are not absolute requirements. While it is typically the case that:

x.clone().equals(x)

will be true, this is not an absolute requirement.

By convention, the returned object should be obtained by calling super.clone. If a class and all of its superclasses (except Object) obey this convention, it will be the case that x.clone().getClass() == x.getClass().

*clone()* method throws a CloneNotSupportedException if the object's class does not support the Cloneable interface. Subclasses that override the clone method can also throw this exception to indicate that an instance cannot be cloned.

### protected Object clone() Method Example

This example shows the usage of*clone()* method:

public class ObjectClass {

public static void main(String[] args) {

Date date = new Date();

System.out.println(date.toString());

Date date2 = (Date) date.clone();

System.out.println(date2.toString());

}

}

Output:

Tue Sep 04 14:15:00 IST 2018

Tue Sep 04 14:15:00 IST 2018

Let's discuss one more example using the *Closable*interface and *clone()* method together.

First, create a *Person*class which implements a *Closable*interface.

public class Person implements Cloneable {

private String firstName;

private String lastName;

public String getFirstName() {

return firstName;

}

public void setFirstName(String firstName) {

this.firstName = firstName;

}

public String getLastName() {

return lastName;

}

public void setLastName(String lastName) {

this.lastName = lastName;

}

@Override

public Object clone() throws CloneNotSupportedException {

Person person = (Person) super.clone();

return person;

}

@Override

public String toString() {

return "Person [firstName=" + firstName + ", lastName=" + lastName + "]";

}

public static void main(String[] args) throws CloneNotSupportedException {

Person person = new Person();

person.setFirstName("Ramesh");

person.setLastName("Fadatare");

System.out.println(person.toString());

Person person2 = (Person) person.clone();

System.out.println(person2.toString());

}

}

Output:

Person [firstName=Ramesh, lastName=Fadatare]

Person [firstName=Ramesh, lastName=Fadatare]

Note that we have to override the *clone()* method from *Object*class and made a call to Object *clone()* method using *super.clone().*

@Override

public Object clone() throws CloneNotSupportedException {

Person person = (Person) super.clone();

return person;

}

## 2. boolean equals(Object obj)

The java.lang.Object.equals(Object obj) indicates whether some other object is "equal to" this one.

The equals method implements an equivalence relation on non-null object references:

* **It is reflexive: for any non-null reference value x, *x.equals(x)* should return true.**
* **It is symmetric: for any non-null reference values x and y, *x.equals(y)* should return true if and only if y.equals(x) returns true.**
* **It is transitive: for any non-null reference values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true.**
* **It is consistent: for any non-null reference values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equals comparisons on the objects is modified.**
* **For any non-null reference value x, x.equals(null) should return false.**

### boolean equals(Object obj) Example

Let's create a simple example to demonstrate the usage of equals method:

// get an integer, which is an object

Integer x = new Integer(50);

// get a float, which is an object as well

Float y = new Float(50f);

// check if these are equal,which is

// false since they are different class

System.out.println("" + x.equals(y));

// check if x is equal with another int 50

System.out.println("" + x.equals(50));

Output:

false

true

The equals() method for Person class is:

@Override

public boolean equals(Object obj) {

if (this == obj)

return true;

if (obj == null)

return false;

if (getClass() != obj.getClass())

return false;

Person other = (Person) obj;

if (firstName == null) {

if (other.firstName != null)

return false;

} else if (!firstName.equals(other.firstName))

return false;

if (lastName == null) {

if (other.lastName != null)

return false;

} else if (!lastName.equals(other.lastName))

return false;

return true;

}

Let's test above equals method:

public class Person implements Cloneable {

private String firstName;

private String lastName;

public String getFirstName() {

return firstName;

}

public void setFirstName(String firstName) {

this.firstName = firstName;

}

public String getLastName() {

return lastName;

}

public void setLastName(String lastName) {

this.lastName = lastName;

}

@Override

public boolean equals(Object obj) {

if (this == obj)

return true;

if (obj == null)

return false;

if (getClass() != obj.getClass())

return false;

Person other = (Person) obj;

if (firstName == null) {

if (other.firstName != null)

return false;

} else if (!firstName.equals(other.firstName))

return false;

if (lastName == null) {

if (other.lastName != null)

return false;

} else if (!lastName.equals(other.lastName))

return false;

return true;

}

@Override

public String toString() {

return "Person [firstName=" + firstName + ", lastName=" + lastName + "]";

}

public static void main(String[] args) throws CloneNotSupportedException {

Person person = new Person();

person.setFirstName("Ramesh");

person.setLastName("Fadatare");

Person person1 = new Person();

person1.setFirstName("Ramesh");

person1.setLastName("Fadatare");

boolean hasEqual = person.equals(person1);

System.out.println("Both objects equal :: " + hasEqual);

}

}

Output:

Both objects equal :: true

**Note: If you override *equals()*, you must override *hashCode()* as well.**

## 3. protected void finalize() Method

The *java.lang.Object.finalize()* is called by the garbage collector on an object when garbage collection determines that there are no more references to the object. A subclass overrides the finalize method to dispose of system resources or to perform other cleanups.

The finalize() method may be called automatically by the system, but when it is called, or even if it is called, is uncertain. Therefore, you should not rely on this method to do your cleanup for you. For example, if you don't close file descriptors in your code after performing I/O and you expect finalize() to close them for you, you may run out of file descriptors.

### protected void finalize() Method Example

Let's override finalize() method from Object class into Person class and test it using main() method.

public class Person {

private String firstName;

private String lastName;

public String getFirstName() {

return firstName;

}

public void setFirstName(String firstName) {

this.firstName = firstName;

}

public String getLastName() {

return lastName;

}

public void setLastName(String lastName) {

this.lastName = lastName;

}

// This method is called just before an object is garbage collected

@Override

protected void finalize() throws Throwable {

// TODO Auto-generated method stub

super.finalize();

}

@Override

public String toString() {

return "Person [firstName=" + firstName + ", lastName=" + lastName + "]";

}

public static void main(String[] args) {

Person person = new Person();

person.setFirstName("Ramesh");

person.setLastName("Fadatare");

System.out.println("Before Finalize");

try {

person.finalize();

} catch (Throwable e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

System.out.println("After Finalize");

}

}

Let us compile and run the above program, this will produce the following result −

Before Finalize

After Finalize

## 4. Class<?> getClass() Method

The *java.lang.Object.getClass()* method returns the runtime class of an object. That Class object is the object that is locked by static *synchronized*methods of the represented class.

### Class<?> getClass() Method Example

The following example shows the usage of lang.Object.getClass() method.

public class Person {

private String firstName;

private String lastName;

public String getFirstName() {

return firstName;

}

public void setFirstName(String firstName) {

this.firstName = firstName;

}

public String getLastName() {

return lastName;

}

public void setLastName(String lastName) {

this.lastName = lastName;

}

public static void main(String[] args) {

Person person = new Person();

System.out.println(person.getClass());

}

}

Let us compile and run the above program, this will produce the following result −

class com.javaguides.corejava.lang.Person

## 4. int hashCode() Method

The java.lang.Object.hashCode() method returns a hash code value for the object. This method is supported for the benefit of hash tables such as those provided by java.util.HashMap.

**The general contract of hashCode is:**

* **Whenever it is invoked on the same object more than once during an execution of a Java application, the hashCode method must consistently return the same integer, provided no information used in equals comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.**
* **If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result.**
* **It is not required that if two objects are unequal according to the java.lang.Object.equals(java.lang.Object) method, then calling the hashCode method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.**

### int hashCode() Method Example

The following example shows the usage of lang.Object.hashCode() method:

public class Person {

private String firstName;

private String lastName;

public String getFirstName() {

return firstName;

}

public void setFirstName(String firstName) {

this.firstName = firstName;

}

public String getLastName() {

return lastName;

}

public void setLastName(String lastName) {

this.lastName = lastName;

}

@Override

public int hashCode() {

final int prime = 31;

int result = 1;

result = prime \* result + ((firstName == null) ? 0 : firstName.hashCode());

result = prime \* result + ((lastName == null) ? 0 : lastName.hashCode());

return result;

}

public static void main(String[] args) {

Person person = new Person();

person.setFirstName("Ramesh");

person.setLastName("Fadatare");

System.out.println(person.hashCode());

Person person1 = new Person();

person1.setFirstName("Ramesh");

person1.setLastName("Fadatare");

System.out.println(person1.hashCode());

}

}

Let us compile and run the above program, this will produce the following result −

-1066062211

-1066062211

**By definition, if two objects are equal, their hashcode must also be equal. If you override the equals() method, you change the way two objects are equated and Object's implementation of hashCode() is no longer valid. Therefore, if you override the equals() method, you must also override the hashCode() method as well.**

## 6. notify() , 7. notifyAll() and 8. wait() Methods

* void notify() - This method wakes up a single thread that is waiting on this object's monitor.
* void notifyAll() - This method wakes up all threads that are waiting on this object's monitor.
* void wait() - This method causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.

### notify() , 7. notifyAll() and 8. wait() Methods Example

The following example shows the usage of notify() , notifyAll() and wait() Methods:

import java.util.Collections;

import java.util.LinkedList;

import java.util.List;

public class ObjectClassNotifyNotifyAllAndWaitExample {

private List<String> synchedList;

public ObjectClassNotifyNotifyAllAndWaitExample() {

// create a new synchronized list to be used

synchedList = Collections.synchronizedList(new LinkedList<>());

}

// method used to remove an element from the list

public String removeElement() throws InterruptedException {

synchronized (synchedList) {

// while the list is empty, wait

while (synchedList.isEmpty()) {

System.out.println("List is empty...");

synchedList.wait();

System.out.println("Waiting...");

}

String element = synchedList.remove(0);

return element;

}

}

// method to add an element in the list

public void addElement(String element) {

System.out.println("Opening...");

synchronized (synchedList) {

// add an element and notify all that an element exists

synchedList.add(element);

System.out.println("New Element added:'" + element + "'");

synchedList.notifyAll();

System.out.println("notifyAll called!");

}

System.out.println("Closing in AddElement method...");

}

public static void main(String[] args) {

final ObjectClassNotifyNotifyAllAndWaitExample demo = new ObjectClassNotifyNotifyAllAndWaitExample();

Runnable runA = () -> {

try {

String item = demo.removeElement();

System.out.println("" + item);

} catch (InterruptedException ix) {

System.out.println("Interrupted Exception!");

} catch (Exception x) {

System.out.println("Exception thrown.");

}

};

Runnable runB = () -> {

// run adds an element in the list and starts the loop

demo.addElement("Hello!");

};

try {

Thread threadA1 = new Thread(runA, "A");

threadA1.start();

Thread.sleep(500);

Thread threadA2 = new Thread(runA, "B");

threadA2.start();

Thread.sleep(500);

Thread threadB = new Thread(runB, "C");

threadB.start();

Thread.sleep(1000);

threadA1.interrupt();

threadA2.interrupt();

} catch (InterruptedException x) {

}

}

}

Output:

List is empty...

List is empty...

Opening...

New Element added:'Hello!'

notifyAll called!

Waiting...

Closing in AddElement method...

Hello!

Waiting...

List is empty...

Interrupted Exception!

## 9. String toString() Method

The *java.lang.Object.toString()* method returns a string representation of the object. In general, the toString method returns a string that "textually represents" this object. The result should be a concise but informative representation that is easy for a person to read. It is recommended that all subclasses override this method.

### String toString() Method Example

The following example shows the usage of *lang.Object.toString()* method.

public class Person {

private String firstName;

private String lastName;

public String getFirstName() {

return firstName;

}

public void setFirstName(String firstName) {

this.firstName = firstName;

}

public String getLastName() {

return lastName;

}

public void setLastName(String lastName) {

this.lastName = lastName;

}

@Override

public String toString() {

return "Person [firstName=" + firstName + ", lastName=" + lastName + "]";

}

public static void main(String[] args) {

Person person = new Person();

person.setFirstName("Ramesh");

person.setLastName("Fadatare");

System.out.println(person.toString());

}

}

Let us compile and run the above program, this will produce the following result −

Person [firstName=Ramesh, lastName=Fadatare]

### Instantiating a Class

## **The *new* operator instantiates a class by allocating memory for a new object and returning a reference to that memory. The *new* operator also invokes the object constructor.**

## **For example:**

Student student = new Student("Ramesh", "BVB");

Student student2 = new Student("Prakash", "GEC");

Student student3 = new Student("Pramod", "IIT");

## **Note that we have used a *new* keyword to create *Student*** **objects.**

### Initializing an Object

The *new* keyword is followed by a call to a constructor, which initializes the new object. For example:

Student student = new Student("Ramesh", "BVB");

Student student2 = new Student("Prakash", "GEC");

Student student3 = new Student("Pramod", "IIT");

From above code will call below constructor in **Student**class.

public class Student {

private String name;

private String college;

public Student(String name, String college) {

super();

this.name = name;

this.college = college;

}

}

## Different Ways to Create an Object in Java?

## 1. Using a new keyword

This is the most popular way of creating an object in Java using a *new* keyword. This approach every Java Developer knows.

package net.javaguides.corejava.oops;

public class Student {

private String name;

private String college;

public Student(String name, String college) {

super();

this.name = name;

this.college = college;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getCollege() {

return college;

}

public void setCollege(String college) {

this.college = college;

}

public static void main(String[] args) {

Student student = new Student("Ramesh", "BVB");

Student student2 = new Student("Prakash", "GEC");

Student student3 = new Student("Pramod", "IIT");

}

}

From the above code, we are creating *Student* object using *new*keyword:

Student student = new Student("Ramesh", "BVB");

Student student2 = new Student("Prakash", "GEC");

Student student3 = new Student("Pramod", "IIT");

## 2. Using newInstance() method of Class class

*Class.forName()* will load the class dynamically and it indirectly will give you “Class class” object. Once the class is loaded we will be using *newInstance()* method to create the object dynamically.

Let's create a Java object for the *Student*class here:

package net.javaguides.corejava.oops;

public class Student {

private String name = "Ramesh";

private String college = "ABC";

public Student() {

super();

}

public Student(String name, String college) {

super();

this.name = name;

this.college = college;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getCollege() {

return college;

}

public void setCollege(String college) {

this.college = college;

}

public static void main(String[] args) {

try {

String className = "net.javaguides.corejava.oops.Student";

Class clasz = Class.forName(className);

Student student = (Student) clasz.newInstance();

System.out.println(student.getName());

System.out.println(student.getCollege());

} catch (InstantiationException | IllegalAccessException | ClassNotFoundException e) {

e.printStackTrace();

}

}

}

Output:

Ramesh

ABC

The *forName()* method returns the Class object associated with the class or interfaces with the given string name.

Class clasz = Class.forName(className);

*newInstance()*method creates a new instance of the class represented by this Class object.

Student student = (Student) clasz.newInstance();

System.out.println(student);

## 3. Using newInstance() method of Constructor class

Similar to the *newInstance()* method of **Class**class, There is one *newInstance()* method in the *java.lang.reflect.Constructor* class which we can use to create objects. We can also call a parameterized constructor, and private constructor by using this *newInstance()* method.

Let's demonstrate this approach by creating *Student* class object using *newInstance()* method of *java.lang.reflect.Constructor* class:

package net.javaguides.corejava.oops;

import java.lang.reflect.Constructor;

import java.lang.reflect.InvocationTargetException;

public class Student {

private String name = "Ramesh";

private String college = "ABC";

public Student() {

super();

}

public Student(String name, String college) {

super();

this.name = name;

this.college = college;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getCollege() {

return college;

}

public void setCollege(String college) {

this.college = college;

}

public static void main(String args[]) {

Constructor < Student > constructor;

try {

constructor = Student.class.getConstructor();

Student student = constructor.newInstance();

System.out.println(student.getName());

System.out.println(student.getCollege());

} catch (InstantiationException | IllegalAccessException | IllegalArgumentException | InvocationTargetException |

NoSuchMethodException | SecurityException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

}

Output:

Ramesh

ABC

## 4. Using Object Deserialization

In this approach, we will be using *Serializable* interface in Java which is a marker interface(interface with no fields or methods within it) for serializing a Java *Student* Object *s1* into a text file (sample.txt) and using object deserialization we will be reading and assigning it to a new *Student*object *s2*.

package net.javaguides.corejava.oops;

import java.io.Serializable;

public class Student implements Serializable{

private String name;

private String college;

public Student() {

super();

}

public Student(String name, String college) {

super();

this.name = name;

this.college = college;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getCollege() {

return college;

}

public void setCollege(String college) {

this.college = college;

}

}

package net.javaguides.corejava.oops;

import java.io.FileInputStream;

import java.io.FileOutputStream;

import java.io.ObjectInputStream;

import java.io.ObjectOutputStream;

public class StudentDemo {

public static void main(String[] args) {

// Path to store the Serialized object

String filePath = "sample.txt";

Student s1 = new Student("Ramesh", "ABC");

try {

FileOutputStream fileOutputStream = new FileOutputStream(filePath);

ObjectOutputStream outputStream = new ObjectOutputStream(fileOutputStream);

outputStream.writeObject(s1);

outputStream.flush();

outputStream.close();

FileInputStream fileInputStream = new FileInputStream(filePath);

ObjectInputStream inputStream = new ObjectInputStream(fileInputStream);

Student s2 = (Student) inputStream.readObject();

inputStream.close();

System.out.println(s2.getName());

System.out.println(s2.getCollege());

} catch (Exception ee) {

ee.printStackTrace();

}

}

}

Output:

Ramesh

ABC

## 5. Using Object Cloning – clone() method

The *clone()* method is used to create a copy of an existing object, in order to the *clone()* method the corresponding class should have implemented a *Cloneable* interface which is again a Marker Interface.

In this approach we will be creating an object for *Student*class “student1” and using *clone()* method we will be cloning it to “student2” object

package net.javaguides.corejava.oops;

import java.io.Serializable;

public class Student implements Cloneable {

private String name;

private String college;

public Student() {

super();

}

public Student(String name, String college) {

super();

this.name = name;

this.college = college;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getCollege() {

return college;

}

public void setCollege(String college) {

this.college = college;

}

public static void main(String args[]) {

Student student1 = new Student("Ramesh", "ABC");

try {

Student student2 = (Student) student1.clone();

System.out.println(student2.getName());

System.out.println(student2.getCollege());

} catch (CloneNotSupportedException e) {

e.printStackTrace();

}

}

}

Output:

Ramesh

ABC

## java.lang.Object Class in Java

The Object class, in the *java.lang* package sits at the top of the class hierarchy tree. Every class is a descendant, direct or indirect, of the *Object* class. Every class you use or write inherits the instance methods of *Object*. You need not to use any of these methods, but, if you choose to do so, you may need to override them with code that is specific to your class.

### Object Class Methods

1. **protected Object clone()**
2. **boolean equals(Object obj)**
3. **protected void finalize()**
4. **Class<?> getClass()**
5. **int hashCode()**
6. **void notify()**
7. **void notifyAll()**
8. **void wait()**
9. **String toString()**

The *notify*, *notifyAll*, and *wait* methods of *Object* all play a part in synchronizing the activities of independently running threads in a program. There are five of these methods:

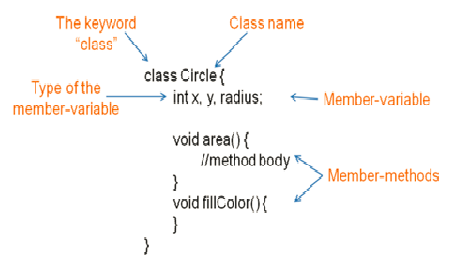
* public final void notify()
* public final void notifyAll()
* public final void wait()
* public final void wait(long timeout)
* public final void wait(long timeout, int nanos)

The below diagram is an *Object* class diagram shows a list of methods it provide

## What is the Class?

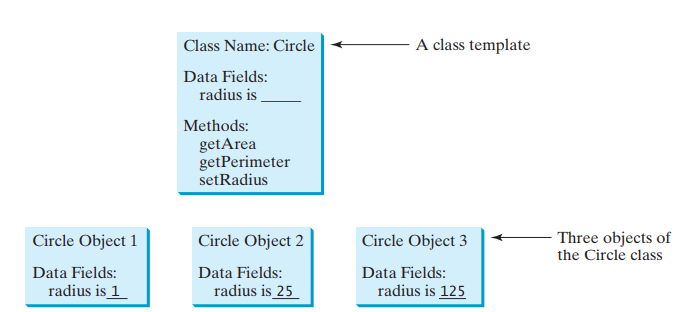
A class is a group of objects which have common properties. It is a template or blueprint from which objects are created. In short, a class is the **specification or template of an object**.

Let’s look at an example of a class and analyze its various parts in a below diagram. This example declares the class Circle, which has the member-variables x, y, and radius of type Integer and the two member-methods, *area()* and *fillColor()*.

**[](https://4.bp.blogspot.com/--G21pab-qsc/W_FAxykVW_I/AAAAAAAAExU/JE_tU9pf8cg-NDwFlBahnb_7Bi46nqM3ACLcBGAs/s1600/class-diagram.png)**

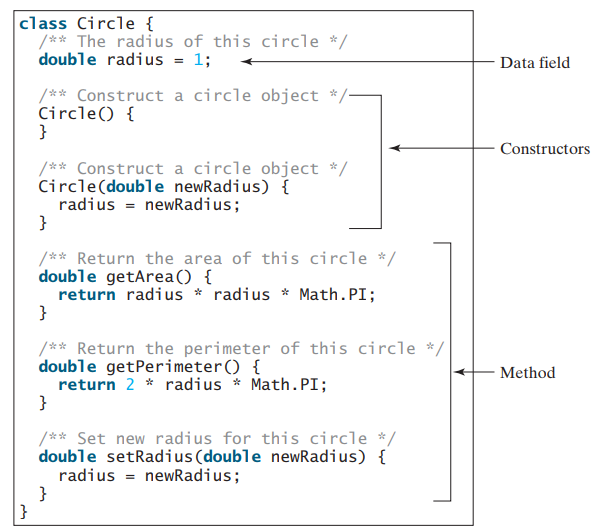
## A class is a template for creating objects

Below diagram shows a Circle class which is a template to create three objects:

**[](https://4.bp.blogspot.com/-xy7oBGHZEV8/XFBhiD0hzUI/AAAAAAAAFfk/6owMziFT_2gCjyejJUBXanGKUMQtb8W6gCLcBGAs/s1600/A+class+is+a+template+for+creating+objects.PNG)**

## A class is a construct that defines objects of the same type

The below diagram shows a Circle class with it's data fields, constructors and methods.

**[](https://2.bp.blogspot.com/-RiB2SMaDXYM/XFBhxZq6s-I/AAAAAAAAFfo/1N-EPzsp0dAXiAVSCfLZtyuTDT6KN_pTgCLcBGAs/s1600/A+class+is+a+construct+that+defines+objects+of+the+same+type.PNG)**

## Examples: Creating Student Class

Let's demonstrate how to create *Class* in Java with an example. Here is a **Student**class:

package net.javaguides.corejava.oops;

import java.lang.reflect.Constructor;

import java.lang.reflect.InvocationTargetException;

public class Student {

private String name = "Ramesh";

private String college = "ABC";

public Student() {

super();

}

public Student(String name, String college) {

super();

this.name = name;

this.college = college;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getCollege() {

return college;

}

public void setCollege(String college) {

this.college = college;

}

}

## Declaring Classes

You've seen classes defined in the following way:

class MyClass {

// field, constructor, and

// method declarations

}

This is a class declaration. The *class body* (the area between the braces) contains all the code that provides for the life cycle of the objects created from the class: *constructors* for initializing new objects, *declarations* for the fields that provide the state of the class and its objects, and methods to implement the behavior of the class and its objects.

A Class can extend another class or implement an interface like:

class MyClass extends MySuperClass implements YourInterface {

// field, constructor, and

// method declarations

}

means that *MyClass* is a subclass of *MySuperClass* and that it implements the *YourInterface*interface.

In general, class declarations can include these components, in order:

1. Modifiers such as [**public, private, protected, default.**](http://www.javaguides.net/2018/10/java-access-modifiers-public-private-protected-default.html)
2. The class name, with the initial letter capitalized by convention.
3. The name of the class's parent (superclass), if any, preceded by the keyword *extends*. A class can only *extend*(subclass) one parent.
4. A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword *implements*. A class can implement more than one interface.
5. The class body, surrounded by braces, {}.

## Declaring Member Variables

There are several kinds of variables:

* Member variables in a class—these are called *fields*.
* Variables in a method or block of code—these are called *local variables*.
* Variables in method declarations—these are called *parameters*.

The Bicycle class uses the following lines of code to define its fields:

public class Bicycle {

private int cadence;

private int gear;

private int speed;

}

Field declarations are composed of three components, in order:

* Zero or more modifiers, such as *public or private.*
* The field's type.
* The field's name.

Read more at [**http://www.javaguides.net/2018/10/variables-in-java-local-variable-class-variable-instance-variable.html**](http://www.javaguides.net/2018/10/variables-in-java-local-variable-class-variable-instance-variable.html)

## Access Modifiers

Java supports four types of access modifiers:

1. Private
2. Default (no access modifier specified)
3. Protected
4. public

### 1. Private Access Modifier

A private class member cannot be accessed from outside the class; only members of the same class can access these private members.

### 2. Default Access Modifier (no access modifier specified)

When we do not mention any access modifier, it is called default access modifier. The scope of this modifier is limited to the package only. This means that if we have a class with the default access modifier in a package, only those classes that are in this package can access this class. No other class outside this package can access this class. Similarly, if we have a default method or data member in a class, it would not be visible in the class of another package.

### 3. Protected Access Modifier

If a class or its members are declared as protected are only accessible by the classes of the same package and the subclasses present in any package. You can also say that the protected access modifier is similar to default access modifier with one exception that it has visibility in subclasses.

### 4. Public Access Modifier

If a class or its members are declared as public, they can be accessed from any other class regardless of the package boundary. It is comparable to a public place in the real world, such as a company cafeteria that all employees can use irrespective of their department.

Read more at [**http://www.javaguides.net/2018/10/java-access-modifiers-public-private-protected-default.html**](http://www.javaguides.net/2018/10/java-access-modifiers-public-private-protected-default.html)

## Defining Methods

Here is an example of a typical method declaration:

public double calculateAnswer(double wingSpan, int numberOfEngines,

double length, double grossTons) {

//do the calculation here

}

The only required elements of a method declaration are the method's return type, name, a pair of parentheses, (), and a body between braces, {}.

More generally, method declarations have six components, in order:

* Modifiers—such as [**public, private, protected, default.**](http://www.javaguides.net/2018/10/java-access-modifiers-public-private-protected-default.html)
* The return type—the data type of the value returned by the method, or void if the method does not return a value.
* The method name—the rules for field names apply to method names as well, but the convention is a little different.
* The parameter list in parenthesis—a comma-delimited list of input parameters, preceded by their data types, enclosed by parentheses, (). If there are no parameters, you must use empty parentheses.
* An exception list—to be discussed later.
* The method body, enclosed between braces—the method's code, including the declaration of local variables, goes here.

## Constructors

In Java, a constructor is a block of codes similar to the method. It is called when an instance of the object is created, and memory is allocated for the object.

Every class has a constructor. If you do not explicitly write a constructor for a class, the Java compiler provides a default constructor (without any parameter) for that class.

### Rules for creating Java constructor

There are two rules defined for the constructor.

* Constructor name must be the same as its class name
* A Constructor must have no explicit return type
* A Java constructor cannot be abstract, static, final, and synchronized

### Constructors Example

package net.javaguides.corejava.oops;

import java.lang.reflect.Constructor;

import java.lang.reflect.InvocationTargetException;

public class Student {

private String name = "Ramesh";

private String college = "ABC";

// default constructor

public Student() {

super();

}

// parameterized constructor

public Student(String name, String college) {

super();

this.name = name;

this.college = college;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getCollege() {

return college;

}

public void setCollege(String college) {

this.college = college;

}

}

## More Class Examples in Java

public class Circle {

private int xPos;

private int yPos;

private int radius;

// three overloaded constructors for Circle

public Circle(int x, int y, int r) {

xPos = x;

yPos = y;

radius = r;

}

public Circle(int x, int y) {

xPos = x;

yPos = y;

radius = 10; // default radius

}

public Circle() {

xPos = 20; // assume some default values for xPos and yPos

yPos = 20;

radius = 10; // default radius

}

public String toString() {

return "center = (" + xPos + "," + yPos + ") and radius = " + radius;

}

public static void main(String[] s) {

System.out.println(new Circle());

System.out.println(new Circle(50, 100));

System.out.println(new Circle(25, 50, 5));

}

}

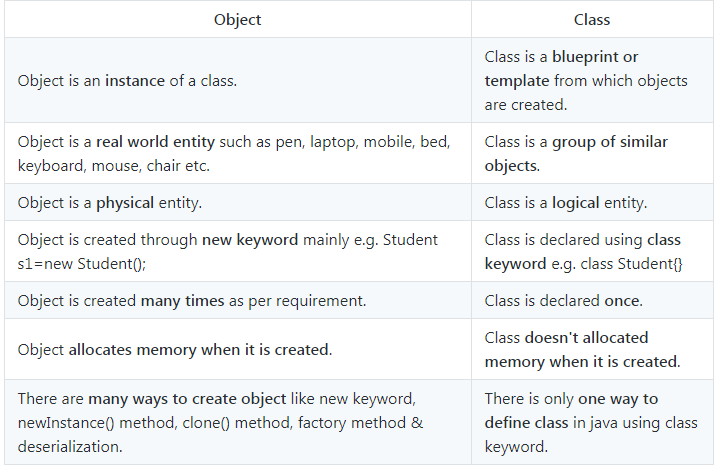
Output:

center = (20,20) and radius = 10

center = (50,100) and radius = 10

center = (25,50) and radius = 5

## Object vs Class

**[](https://4.bp.blogspot.com/-I8g1FWM5NOk/W_FEBvP0_pI/AAAAAAAAExg/NFm--mfMWrsopB8EZNOFyFiWab-MzsaPACLcBGAs/s1600/class-vs-object.PNG)**

## Summary

* A class is a **template**for objects. It defines the properties of objects and provides constructors for creating objects and methods for manipulating them.
* A class is also a data type. You can use it to declare object reference variables. An object reference variable that appears to hold an object actually contains a reference to that object.
* An object is an instance of a class. You use the **new**operator to create an object, and the dot operator (.) to access members of that object through its reference variable.
* An instance variable or method belongs to an instance of a class. Its use is associated with individual instances. A static variable is a variable shared by all instances of the same class. A static method is a method that can be invoked without using instances.
* Visibility modifiers specify how the class, method, and data are accessed. A **public**class, method, or data is accessible to all clients. A **private**method or data is accessible only inside the class.
* You can provide a **getter**(accessor) method or a **setter**(mutator) method to enable clients to see or modify the data