**TypeCasting :**

TypeCasting is converting the object or the variable or primitive or interface or class of type into another type.

Syntax : dataType variableName = (dataType) variableToConvert;

Type Casting is done in two ways

* Explicit TypeCasting : Narrowing(Larger to smaller type) – must be done explicitly
* Implicit TypeCasting : Widening(Smaller to larger) – can be automatically done

Implicit TypeCasting (Narrowing) :

In this type the variable or an object is converted to smaller type.The target type is larger than source type.So this can be done directly.

Byte->short->int->long->float->double

Eg : byte i;

Short j = i;

Long k = j;

When Super class and sub class are there, a subclass object can be referenced to super class automatically whereas a super class object cannot be converted into sub class without explicit type casting.It throws compile time error or ClassCastException

**Upcasting**

**Casting from a subclass to a superclass is called upcasting**. Typically, the upcasting is implicitly performed by the compiler.

Upcasting is closely related to inheritance – another core concept in Java. It’s common to use reference variables to refer to a more specific type. And every time we do this, implicit upcasting takes place.

To demonstrate upcasting let’s define an *Animal* class:

[?](https://www.baeldung.com/java-type-casting)

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public class Animal {        public void eat() {          // ...      }  } |

Now let’s extend *Animal*:

[?](https://www.baeldung.com/java-type-casting)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | public class Cat extends Animal {        public void eat() {           // ...      }        public void meow() {           // ...      }  } |

Now we can create an object of *Cat* class and assign it to the reference variable of type *Cat*:

[?](https://www.baeldung.com/java-type-casting)

|  |  |
| --- | --- |
| 1 | Cat cat = new Cat(); |

And we can also assign it to the reference variable of type *Animal*:

[?](https://www.baeldung.com/java-type-casting)

|  |  |
| --- | --- |
| 1 | Animal animal = cat; |

In the above assignment, implicit upcasting takes place. We could do it explicitly:

[?](https://www.baeldung.com/java-type-casting)

|  |  |
| --- | --- |
| 1 | animal = (Animal) cat; |

But there is no need to do explicit cast up the inheritance tree. The compiler knows that *cat* is an *Animal* and doesn’t display any errors.

Note, that reference can refer to any subtype of the declared type.

Using upcasting, we’ve restricted the number of methods available to *Cat* instance but haven’t changed the instance itself. Now we can’t do anything that is specific to *Cat –* we can’t invoke *meow()* on the *animal* variable.

Although *Cat* object remains *Cat* object, calling *meow()* would cause the compiler error:

[?](https://www.baeldung.com/java-type-casting)

|  |  |
| --- | --- |
| 1 | // animal.meow(); The method meow() is undefined for the type Animal |

To invoke *meow()* we need to downcast *animal*, and we’ll do this later.

But now we’ll describe what gives us the upcasting. Thanks to upcasting, we can take advantage of polymorphism.

**Downcasting**

What if we want to use the variable of type *Animal* to invoke a method available only to *Cat* class? Here comes the downcasting. **It’s the casting from a superclass to a subclass.**

Let’s take an example:

[?](https://www.baeldung.com/java-type-casting)

|  |  |
| --- | --- |
| 1 | Animal animal = new Cat(); |

We know that *animal* variable refers to the instance of *Cat*. And we want to invoke *Cat*’s *meow()* method on the *animal*. But the compiler complains that *meow()* method doesn’t exist for the type *Animal*.

To call *meow()* we should downcast *animal* to *Cat*:

[?](https://www.baeldung.com/java-type-casting)

|  |  |
| --- | --- |
| 1 | ((Cat) animal).meow(); |

The inner parentheses and the type they contain are sometimes called the cast operator. Note that external parentheses are also needed to compile the code.

Let’s rewrite the previous *AnimalFeeder* example with *meow()* method:

[?](https://www.baeldung.com/java-type-casting)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | public class AnimalFeeder {        public void feed(List<Animal> animals) {          animals.forEach(animal -> {              animal.eat();              if (animal instanceof Cat) {                  ((Cat) animal).meow();              }          });      }  } |

**Final Keyword :**

Final keyword implies that the value of the variable cannot be changed.Final variable must be initialized only once either via initializer or by an assignment statement.

If a final variable is not initialized at the point of declaration it is call blank variable but it must be definitely assigned in every constructor if the class in which it is declared.

If a method is declared as final that cannot be overridden.

If a class is declared as final it cannot be subclassed.

If a **final** variable holds a reference to an object, then the state of the object may be changed by operations on the object, but the variable will always refer to the same object (this property of **final** is called *non-transitivity*[[1]](https://en.wikipedia.org/wiki/Final_(Java)#cite_note-1)). This applies also to arrays, because arrays are objects; if a **final** variable holds a reference to an array, then the components of the array may be changed by operations on the array, but the variable will always refer to the same array.[[2]](https://en.wikipedia.org/wiki/Final_(Java)#cite_note-2)

**Super Keyword :**

Super keyword is used inside a subclass method definition to call a method defined in super class.Private methods of super class cannot be called, only public and protected methods can be called.

It is also used by class constructors to invoke constructors in super class.

Super keyword is not used in static methods.

Syntax : super.<methodname>();

Or

Super();

**Exception Handling :**

Java Exception Handling is a framework that is used to handle the runtime errors only, compile time errors are not handled by exception handling in java.

Keyword in exception :

Throw : This is used to throw exception in java like if the password is wrong it has to throw as exception.

Throws : Whne we throw an exception and doesn’t handle, then we need to use throws in the method signature to let the caller know that the exceptions might be thrown by the method.

**try-catch** – We use try-catch block for exception handling in our code. try is the start of the block and catch is at the end of try block to handle the exceptions. We can have multiple catch blocks with a try and try-catch block can be nested also. catch block requires a parameter that should be of type Exception.

**finally** – finally block is optional and can be used only with try-catch block. Since exception halts the process of execution, we might have some resources open that will not get closed, so we can use finally block. finally block gets executed always, whether exception occurred or not.

* We can’t have catch or finally clause without a try statement.
* A try statement should have either catch block or finally block, it can have both blocks.
* We can’t write any code between try-catch-finally block.
* We can have multiple catch blocks with a single try statement.
* try-catch blocks can be nested similar to if-else statements.
* We can have only one finally block with a try-catch statement.

Java Exception Hierarchy :

Throwable is the parent class of Java Exceptions.Throwable class implements Serializable interface. It has two child objects – Error and Exception

Exceptions are further divided into Checked and runtime exceptions.

Errors : These are exceptional scenarios that are out of scope and cannot be handled like harware issues, JVM crash, out of memory error i.e OutOfMemoryError or StackOverFlowError.

**Checked Exceptions**: Checked Exceptions are exceptional scenarios that we can anticipate in a program and try to recover from it, for example FileNotFoundException. We should catch this exception and provide useful message to user and log it properly for debugging purpose. Exception is the parent class of all Checked Exceptions and if we are throwing a checked exception, we must catch it in the same method or we have to propagate it to the caller using throws keyword.

**Runtime Exception**: Runtime Exceptions are cause by bad programming, for example trying to retrieve an element from the Array. We should check the length of array first before trying to retrieve the element otherwise it might throw ArrayIndexOutOfBoundException at runtime. RuntimeException is the parent class of all runtime exceptions. If we are throwing any runtime exception in a method, it’s not required to specify them in the method signature throws clause. Runtime exceptions can be avoided with better programming.