**Object Oriented Development**

Casting with Primitives and Class References

Tutorial

© FDM Group Ltd 2020. All Rights Reserved.

Any unauthorised reproduction or distribution in part  
or in whole will constitute an infringement of copyright.

# What does this tutorial cover?

This tutorial will introduce you to casting with primitives and class references, and how to handle the problems that can arise from casting.

# How long will the tutorial take to complete?

Around 2-3 hours

# What should you have already completed?

You should have completed the slides on 4 Pillars of OOP, data types (primitives and class references)

# What do you need?

In order to complete this tutorial you will need:

* Java Development Kit 1.8 or above
* Eclipse IDE Kepler or above

# What does this tutorial cover?

* How to use variable widening and narrowing
* An overview of implicit and explicit casting
* How to cast class references to subclasses and superclasses
* The ClassCastException: what it is, why it happens, and what to do to prevent it

# Casting

Casting is the act of storing one data type into a different data type. There are several reasons to use casting, such as allowing for multiple data types to be used for one method, accessing specific versions of a method in subclasses/superclasses, and storing multiple types of data in a collection. This tutorial will discuss the various types of casting, the problems that can arise from it, and how to fix the problems.

# Primitive Casting (Variable Widening and Narrowing)

Primitives are variables that hold raw data, and have a certain number of bytes that can be stored into the data type. Larger data types are capable of storing smaller data type values inside of it, because no matter how big the value is for the smaller data type, the number of bytes that it takes up will not be more than the larger data type.

For example, a short has a storage capacity of 16 bits (2 bytes), while an int has a storage capacity of 32 bits (4 bytes). To put a smaller primitive into a wider primitive is known as *variable widening.* To put a larger data type into a smaller data type is known as *variable narrowing.*

## Variable Widening

Let’s see this in action by creating a new class, and calling it Person. The Person class should look like this:

**public** **class** Person {

**public** **static** **void** main (String [] args) {

**short** age = 50;

System.***out***.println (age);

**int** intAge = age;

System.***out***.println (intAge);

}

}

We are able to set the int variable to contain the short variable because the short variable’s storage capacity is smaller than the int variable’s storage capacity. Both values, when printed out, will equal 500.

Try setting the value for maxNumUsers to be Short.MAX\_VALUE. What is the new value that is printed out? Can intMaxNumUsers contain it?

## Implicit and Explicit Casting

The compiler knows that it can store the smaller value in the larger data type without the programmer telling it so. This is known as an *implicit cast*. An implicit cast requires no extra work from the coder on how to make the value fit in the class. If we wanted to show the cast happening, the code would look something like this:

**short** maxNumUsers = 500;

System.***out***.println (maxNumUsers);

**int** intMaxNumUsers = (**short**) maxNumUsers;

System.***out***.println (intMaxNumUsers);

Notice the short keyword wrapped up in parenthesis. This is us telling the compiler that we intend for the following value to be cast as a short. This type of cast is known as an *explicit cast*. As we will see later in this tutorial, while the explicit cast is optional here, it will be mandatory for other casting needs.

## Variable Narrowing

Sometimes, we need to take a larger value, and put it in a smaller data type. This kind of cast is known as variable narrowing. Let’s take our previous example, and add some more code.

**long** grade = 80L;

System.***out***.println(grade);

**int** intGrade = grade;

System.***out***.println(intGrade);

Long is a 64 bit primitive variable, and int is a 32 bit variable. There is no way for an int variable to be able to store a variable of that size, so the compiler throws an error. The only way for this code to compile is if we apply an explicit cast.

**long** grade = 80L;

System.***out***.println(grade);

**int** intGrade = (int) grade;

System.***out***.println(intGrade);

As you can see, the code now compiles. We are telling the compiler that we recognize the fact that the long datatype is larger than the int datatype, and that is okay. But, if the value is larger than the maximum value the smaller data type can hold, the result can be completely different.  
  
For example, casting the long value 1\_000\_000\_000\_000L to an int variable will set the int variable as -727379968.  
  
To explain what is happening behind the scenes in more detail, we will look at another example.

**int** valueLargerThanByteMaximum = 500;

**byte** byteValue = (**byte**) valueLargerThanByteMaximum;

System.***out***.println(byteValue);

The value printed out is -12. That is because in order to make the value fit into the smaller datatype, it must shorten the data type by cutting out digits on the right side when it is converted into binary.

500 = 111110100 – The red 0 will be cut out to make the number fit. After that, the number will be converted back into base 10 via its two’s complement.

500 -> 11111010 -> -12 (two’s complement)   
  
Feel free to read more about the two’s complement here:   
  
<https://en.wikipedia.org/wiki/Two's_complement>

## Variable Widening and Method Overloading

For this lesson, we will be creating a User Class. The User class will be as follows:

**public** **class** User {

**private** **long** age;

**public** **void** setAge(**byte** age) {

System.***out***.println("Byte method");

**this**.age = age;

}

**public** **void** setAge(**int** age) {

System.***out***.println("Int method");

**this**.age = age;

}

**public** **static** **void** main(String[] args) {

**short** age = 5;

User user = **new** User();

user.setAge(age);

}

}

**Question: What will be printed out?**

**Answer: ‘Int method’ will be printed out.**

Why is this? It is because the short variable cannot be put into the byte parameter without being cast, but it can be put into the int parameter without needing to be cast. (Remember, the short data is 2 bytes, and the int is 4 bytes).

# Reference Casting

Classes can be cast just like primitives. One of the benefits of casting is that you can access specific versions of methods. For this next example, we will create three classes: Animal.java,

Dog.java, and Client.java. The code for the classes is as follows:

Animal.java:

**public** **class** Animal {

**public** **void** makeNoise()

{

System.***out***.println("Generic noise");

}

}

Dog.java:

**public** **class** Dog **extends** Animal {

**public** **void** makeNoise()

{

System.***out***.println("Woof!");

}

**public** **void** doTrick()

{

System.***out***.println("The dog rolled over!");

}

}

Client.java:

**public** **class** Client {

**public** **static** **void** main(String[] args) {

Animal animal = new Animal();

animal.makeNoise();

}

}

As we can see here, the Animal and the Dog class both have the makeNoise() method. The Animal class is the parent class of the Dog class.  
  
**Question: What is printed out when you run the Client class?  
  
Answer: Generic Sound**

## Upcasting

After the makeNoise method call, add the following code:

Animal dog = **new** Dog();

dog.makeNoise();

**Question: What will the output be after these two lines are added in?**

**Answer: The output will be**:   
Generic sound  
Woof!

Why is this? Because at line 3, the Animal variable has been implicitly casted to hold the data type of Dog. It is the equivalent of the following line:   
  
animal = (Animal) (**new** Dog());

We do not need to explicitly cast this initialization because the compiler can guarantee that the Dog class is an instance of the Animal class, because the Dog class is a subclass of Animal. This is called *upcasting.* According to the rules of inheritance of Java, all subclasses are instances of their parent classes.

You can confirm that the animal variable is initialized as a dog with the following line:   
  
System.***out***.println(animal.getClass());

**(Note: This is not the recommended way to check the runtime initialization for safety checks. That will be discussed later.)**

## Downcasting

Place this code in your Client’s main method.

Dog anotherDog = new Animal();

anotherDog.performTrick();

You will see that the code does not compile. That is because Animal is a superclass of the Dog class. Trying to cast a subclass to a superclass is known as *downcasting*. Now, why does this not compile? Because the compiler cannot guarantee that the code here is safe. It cannot guarantee that the value on the right actually is an instance of a Dog object implicitly, which is why we need to cast it.

Change the above code to now say:

Dog anotherDog = (Dog) (new Animal());

anotherDog.performTrick();

The code should compile.

**Question: What happens when you run it? What is the output?**

**Answer: It should throw a ClassCastException exception.**

## ClassCastException

ClassCastException, according to the Oracle Java API, is “Thrown to indicate that the code has attempted to cast an object to a subclass of which it is not an instance” (<https://docs.oracle.com/javase/8/docs/api/java/lang/ClassCastException.html>). What this means is that we tried to tell a class to be something it is not. In this case, Animal is not Dog in the way Dog is an Animal. Here is another example to show a different instance of a ClassCastException, this time using a Cat class.

Create a new class called Cat.java:

**public** **class** Cat **extends** Animal {

**public** **void** makeNoise()

{

System.***out***.println("Meow!");

}

}

The Cat class extends the Animal class as well. Which means this code will compile. Feel free to put this code in your Client class:  
  
 Animal cat = **new** Cat();

cat.makeNoise();

Dog dogOrCat = (Animal) (cat);   
  
So, why does this compile? It compiles because at compile time, the cat variable is declared as an Animal. Thus, the dog does not see a cat- it sees an animal class. Only at runtime is the class initialization checked, wherein it is found that the value is a Cat, which is not a Dog.

How do we solve this problem, and properly check if the value is what we are looking for?

## instanceof keyword

The instanceof keyword allows us to check the data type, and see if it is a specified type. It would be used like this:

if (variable instanceof Class) {}

Let’s use this to fix our previous code. Using instanceof, our code should look like this:

**if** (cat **instanceof** Dog) {

System.***out***.println("It is a dog!");

Dog dogOrCat = (Dog) (cat);

} **else** {

System.***out***.println("Value passed in is NOT a dog");

}

**Question: what will be printed out?**

**Answer: “Value passed in is NOT a dog”.**

## Differences between Upcasting and Downcasting

|  |  |
| --- | --- |
| UPCASTING | DOWNCASTING |
| Goes from Parent to Child | Goes from Child to Parent |
| Cast is implicit, but can be explicit | Cast must be explicit |
| Allows you to use child version of methods for methods that are shared by both classes | Allows you to use methods only in child class |
| Does not throw ClassCastException | Can throw ClassCastException |
| Animal a = new Dog(); | Dog d = (Dog) new Animal(); |

# Conclusion:

As with many things in coding, casting is a tricky topic that you may not get on your first go around. It will take time and practice to remember and understand the various rules of casting. However, once you do understand them, you will find your code more flexible and safe when handling situations involving polymorphism.