

---

# Conversions and Casting

Taken and modified slightly from the book *The Java<sup>TM</sup> Language Specification*, Second Edition. Written by Sun Microsystems.

---

Conversion of one reference type to another is divided into two categories:

- Widening reference conversion
- Narrowing reference conversion

An explicit cast may or may not be required when converting one reference type to another. The following sections will describe each category in greater detail.

## Widening Reference Conversions

The following conversions are called the *widening reference conversions*:

- From any class type *S* to any class type *T*, provided that *S* is a subclass of *T*. (An important special case is that there is a widening conversion to the class type `Object` from any other class type.)
- From any class type *S* to any interface type *K*, provided that *S* implements *K*.
- From the null type to any class type, interface type, or array type.
- From any interface type *J* to any interface type *K*, provided that *J* is a subinterface of *K*.
- From any interface type-to-type `Object`.
- From any array type-to-type `Object`.
- From any array type-to-type `Cloneable`.
- From any array type to type `java.io.Serializable`
- From any array type *SC*[] to any array type *TC*[], provided that *SC* and *TC* are reference types and there is a widening conversion from *SC* to *TC*.

Such conversions never require a special action at run time (e.g. no cast is required in this case!) and therefore never throw an exception at run time. They consist simply in regarding a reference as having some other type in a manner that can be proved correct at compile time.

## Narrowing Reference Conversions

The following conversions are called the *narrowing reference conversions*:

- From any class type *S* to any class type *T*, provided that *S* is a superclass of *T*. (An important special case is that there is a narrowing conversion from the class type `Object` to any other class type.)
- From any class type *S* to any interface type *K*, provided that *S* is not `final` and does not implement *K*. (An important special case is that there is a narrowing conversion from the class type `Object` to any interface type.)
- From type `Object` to any array type.
- From type `Object` to any interface type.
- From any interface type *J* to any class type *T* that is not `final`.
- From any interface type *J* to any class type *T* that is `final`, provided that *T* implements *J*.
- From any interface type *J* to any interface type *K*, provided that *J* is not a subinterface of *K* and there is no method name *m* such that *J* and *K* both contain a method named *m* with the same signature but different return types.
- From any array type *SC*[] to any array type *TC*[], provided that *SC* and *TC* are reference types and there is a narrowing conversion from *SC* to *TC*.

Such conversions require a test at run time to find out whether the actual reference value is a legitimate value of the new type. If not, then a `ClassCastException` is thrown.

## Forbidden Conversions

- There is no permitted conversion from any reference type to any primitive type.
- Except for the string conversions, there is no permitted conversion from any primitive type to any reference type.
- There is no permitted conversion from the null type to any primitive type.
- There is no permitted conversion to the null type other than the identity conversion.
- There is no permitted conversion from class type *S* to interface type *K* if *S* is `final` and does not implement *K*.
- There is no permitted conversion from class type *S* to any array type if *S* is not `Object`.
- There is no permitted conversion from interface type *J* to interface type *K* if *J* and *K* contain methods with the same signature but different return types.
- There is no permitted conversion from any array type to any class type other than `Object` or `String`.
- There is no permitted conversion from any array type to any interface type, except to the interface types `java.io.Serializable` and `Cloneable`, which are implemented by all arrays.

A value of the null type (the null reference is the only such value) may be assigned to any reference type, resulting in a null reference of that type.

Here is a sample program illustrating assignments of references:

```
public class Point {
    int x, y;
}

public class Point3D extends Point {
    int z;
}

public interface Colorable {
    void setColor(int color);
}

public class ColoredPoint extends Point implements Colorable {
    int color;
    public void setColor(int color) { this.color = color; }
}

class Test {
    public static void main(String[] args) {
        // Assignments to variables of class type:
        Point p = new Point();
        p = new Point3D();           // ok: because Point3D is a
                                   // subclass of Point

        Point3D p3d = p;           // error: will require a cast because a
                                   // Point might not be a Point3D
                                   // (even though it is, dynamically,
                                   // in this example.)

        // Assignments to variables of type Object:
        Object o = p;              // ok: any object to Object
        int[] a = new int[3];
        Object o2 = a;             // ok: an array to Object

        // Assignments to variables of interface type:
        ColoredPoint cp = new ColoredPoint();
        Colorable c = cp;          // ok: ColoredPoint implements
                                   // Colorable

        // Assignments to variables of array type:
        byte[] b = new byte[4];
    }
}
```

```

        a = b;                    // error: these are not arrays
                                // of the same primitive type
        Point3D[] p3da = new Point3D[3];
        Point[] pa = p3da;        // ok: since we can assign a
                                // Point3D to a Point
        p3da = pa;                // error: (cast needed) since a Point
                                // can't be assigned to a Point3D
    }
}

```

Assignment of a value of compile-time reference type *S* (source) to a variable of compile-time reference type *T* (target) is checked as follows:

- If *S* is a class type:
  - If *T* is a class type, then *S* must either be the same class as *T*, or *S* must be a subclass of *T*, or a compile-time error occurs.
  - If *T* is an interface type, then *S* must implement interface *T*, or a compile-time error occurs.
  - If *T* is an array type, then a compile-time error occurs.
- If *S* is an interface type:
  - If *T* is a class type, then *T* must be `Object`, or a compile-time error occurs.
  - If *T* is an interface type, then *T* must be either the same interface as *S* or a superinterface of *S*, or a compile-time error occurs.
  - If *T* is an array type, then a compile-time error occurs.
- If *S* is an array type *SC*[], that is, an array of components of type *SC*:
  - If *T* is a class type, then *T* must be `Object`, or a compile-time error occurs.
  - If *T* is an interface type, then a compile-time error occurs unless *T* is the type `java.io.Serializable` or the type `Cloneable`, the only interfaces implemented by arrays.
  - If *T* is an array type *TC*[], that is, an array of components of type *TC*, then a compile-time error occurs unless one of the following is true:
    - *TC* and *SC* are the same primitive type.
    - *TC* and *SC* are both reference types and type *SC* is assignable to *TC*, as determined by a recursive application of these compile-time rules for assignability.

The following test program illustrates assignment conversions on reference values, but fails to compile because it violates the preceding rules, as described in its comments. This example should be compared to the preceding one.

```

public class Point {
    int x, y;
}

public interface Colorable {
    void setColor(int color);
}

public class ColoredPoint extends Point implements Colorable {
    int color;
    public void setColor(int color) { this.color = color; }
}

class Test {
    public static void main(String[] args) {
        Point p = new Point();
        ColoredPoint cp = new ColoredPoint();
        // Okay because ColoredPoint is a subclass of Point:
        p = cp;
        // Okay because ColoredPoint implements Colorable:
        Colorable c = cp;
        // The following cause compile-time errors because
        // we cannot be sure they will succeed, depending on
        // the run-time type of p; a run-time check will be
        // necessary for the needed narrowing conversion and
        // must be indicated by including a cast:
        cp = p;                // p might be neither a ColoredPoint
                               // nor a subclass of ColoredPoint
        c = p;                // p might not implement Colorable
    }
}

```

## Explicit Casting

Some casts can be proven incorrect at compile time; such casts result in a compile-time error. Other casts cannot be determined incorrect at compile time and as such, during run-time they will cause an Exception.

A value of a primitive type cannot be cast to a reference type by casting, nor can a value of a reference type be cast to a primitive type.

The detailed rules for compile-time correctness checking of a casting of a value of compile-time reference type *S* (source) to a compile-time reference type *T* (target) are as follows:

- If  $S$  is a class type:
  - If  $T$  is a class type, then  $S$  and  $T$  must be related classes—that is,  $S$  and  $T$  must be the same class, or  $S$  a subclass of  $T$ , or  $T$  a subclass of  $S$ ; otherwise a compile-time error occurs.
  - If  $T$  is an interface type:
    - If  $S$  is not a `final` class, then the cast is always correct at compile time (because even if  $S$  does not implement  $T$ , a subclass of  $S$  might).
    - If  $S$  is a `final` class, then  $S$  must implement  $T$ , or a compile-time error occurs.
  - If  $T$  is an array type, then  $S$  must be the class `Object`, or a compile-time error occurs.
- If  $S$  is an interface type:
  - If  $T$  is an array type, then  $T$  must implement  $S$ , or a compile-time error occurs.
  - If  $T$  is a class type that is not `final`, then the cast is always correct at compile time (because even if  $T$  does not implement  $S$ , a subclass of  $T$  might).
  - If  $T$  is an interface type and if  $T$  and  $S$  contain methods with the same signature but different return types, then a compile-time error occurs.
- If  $S$  is an array type  $SC[]$ , that is, an array of components of type  $SC$ :
  - If  $T$  is a class type, then if  $T$  is not `Object`, then a compile-time error occurs (because `Object` is the only class type to which arrays can be assigned).
  - If  $T$  is an interface type, then a compile-time error occurs unless  $T$  is the type `java.io.Serializable` or the type `Cloneable`, the only interfaces implemented by arrays.
  - If  $T$  is an array type  $TC[]$ , that is, an array of components of type  $TC$ , then a compile-time error occurs unless one of the following is true:
    - $TC$  and  $SC$  are the same primitive type.
    - $TC$  and  $SC$  are reference types and type  $SC$  can be cast to  $TC$  by a recursive application of these compile-time rules for casting.

If a cast to a reference type is not a compile-time error, there are two cases:

- The cast can be determined to be correct at compile time. A cast from the compile-time type  $S$  to compile-time type  $T$  is correct at compile time if and only if  $S$  can be converted to  $T$  by assignment conversion.
- The cast requires a run-time validity check. If the value at run time is `null`, then the cast is allowed. Otherwise, let  $R$  be the class of the object referred to by the run-time reference value, and let  $T$  be the type named in the cast operator. A cast conversion must check, at run time, that the class  $R$  is assignment compatible with the type  $T$ , using the algorithm specified in but using the class  $R$  instead of the compile-time type  $S$  as specified there. (Note that  $R$  cannot be an interface when these rules are first applied for any given cast, but  $R$  may be an interface if the rules are applied recursively because the run-time reference value may refer to an

array whose element type is an interface type.) The modified algorithm is shown here:

- If  $R$  is an ordinary class (not an array class):
  - If  $T$  is a class type, then  $R$  must be either the same class as  $T$  or a subclass of  $T$ , or a run-time exception is thrown.
  - If  $T$  is an interface type, then  $R$  must implement interface  $T$ , or a run-time exception is thrown.
  - If  $T$  is an array type, then a run-time exception is thrown.
- If  $R$  is an interface:
  - If  $T$  is a class type, then  $T$  must be `Object`, or a run-time exception is thrown.
  - If  $T$  is an interface type, then  $R$  must be either the same interface as  $T$  or a subinterface of  $T$ , or a run-time exception is thrown.
  - If  $T$  is an array type, then a run-time exception is thrown.
- If  $R$  is a class representing an array type  $RC[]$ -that is, an array of components of type  $RC$ :
  - If  $T$  is a class type, then  $T$  must be `Object`, or a run-time exception is thrown.
  - If  $T$  is an interface type, then a run-time exception is thrown unless  $T$  is the type `java.io.Serializable` or the type `Cloneable`, the only interfaces implemented by arrays (this case could slip past the compile-time checking if, for example, a reference to an array were stored in a variable of type `Object`).
  - If  $T$  is an array type  $TC[]$ , that is, an array of components of type  $TC$ , then a run-time exception is thrown unless one of the following is true:
    - $TC$  and  $RC$  are the same primitive type.
    - $TC$  and  $RC$  are reference types and type  $RC$  can be cast to  $TC$  by a recursive application of these run-time rules for casting.

If a run-time exception is thrown, it is a `ClassCastException`.

Here are some examples of casting conversions of reference types.

```
public class Point {
    int x, y;
}
```

```
public interface Colorable {
    void setColor(int color);
}
```

```
public class ColoredPoint extends Point implements Colorable {
    int color;
    public void setColor(int color) { this.color = color; }
}
```

```

final class EndPoint extends Point { }

class Test {
    public static void main(String[] args) {
        Point p = new Point();
        ColoredPoint cp = new ColoredPoint();
        Colorable c;

        // The following may cause errors at run time because
        // we cannot be sure they will succeed; this possibility
        // is suggested by the casts:
        cp = (ColoredPoint)p;           // p might not reference an
                                        // object which is a ColoredPoint
                                        // or a subclass of ColoredPoint

        c = (Colorable)p;              // p might not be Colorable

        // The following are incorrect at compile time because
        // they can never succeed as explained in the text:
        Long l = (Long)p;              // compile-time error #1
        EndPoint e = new EndPoint();
        c = (Colorable)e;              // compile-time error #2
    }
}

```

Here the first compile-time error occurs because the class types `Long` and `Point` are unrelated (that is, they are not the same, and neither is a subclass of the other), so a cast between them will always fail.

The second compile-time error occurs because a variable of type `EndPoint` can never reference a value that implements the interface `Colorable`. This is because `EndPoint` is a `final` type, and a variable of a `final` type always holds a value of the same run-time type as its compile-time type. Therefore, the run-time type of variable `e` must be exactly the type `EndPoint`, and type `EndPoint` does not implement `Colorable`.

The following example uses casts to compile, but it throws exceptions at run time, because the types are incompatible:

```

public class Point {
    int x, y;
}

public interface Colorable {
    void setColor(int color);
}

```

```
public class ColoredPoint extends Point implements Colorable {
    int color;
    public void setColor(int color) { this.color = color; }
}

class Test {
    public static void main(String[] args) {
        Point[] pa = new Point[100];
        // The following line will throw a ClassCastException:
        ColoredPoint[] cpa = (ColoredPoint[])pa;
        System.out.println(cpa[0]);
        int[] shortvec = new int[2];
        Object o = shortvec;
        // The following line will throw a ClassCastException:
        Colorable c = (Colorable)o;
        c.setColor(0);
    }
}
```