Qt Documentation

Google Search Documentation

Contents

Resource Center

Find webinars, use cases, tutorials, videos & more at resources.qt.io

Reference

QVariant Class

The QVariant class acts like a union for the most common Qt data types. More...

Header:	#include <QVariant>
qmake:	QT += core

- List of all members, including inherited members
- Obsolete members

Public Types

class	**Handler**
class	**Private**
class	**PrivateShared**

Public Functions

	QVariant()
	QVariant(QVariant::Type *type*)
	QVariant(const QRegularExpression &*re*)
	QVariant(const QUrl &*val*)
	QVariant(const QEasingCurve &*val*)
	QVariant(const QUuid &*val*)
	QVariant(const QJsonValue &*val*)
	QVariant(const QJsonObject &*val*)

	QVariant(const QJsonArray &*val*)
	QVariant(const QJsonDocument &*val*)
	QVariant(const QModelIndex &*val*)
	QVariant(const QPersistentModelIndex &*val*)
	QVariant(QVariant &&*other*)
	QVariant(int *typeId*, const void **copy*)
	QVariant(const QVariant &*p*)
	QVariant(QDataStream &*s*)
	QVariant(int *val*)
	QVariant(uint *val*)
	QVariant(qlonglong *val*)
	QVariant(qulonglong *val*)
	QVariant(bool *val*)
	QVariant(double *val*)
	QVariant(float *val*)
	QVariant(const char **val*)
	QVariant(const QByteArray &*val*)
	QVariant(const QBitArray &*val*)
	QVariant(const QString &*val*)
	QVariant(QLatin1String *val*)
	QVariant(const QStringList &*val*)
	QVariant(QChar *c*)

	QVariant(const QDate &*val*)
	QVariant(const QTime &*val*)
	QVariant(const QDateTime &*val*)
	QVariant(const QList<QVariant> &*val*)
	QVariant(const QMap<QString, QVariant> &*val*)
	QVariant(const QHash<QString, QVariant> &*val*)
	QVariant(const QSize &*val*)
	QVariant(const QSizeF &*val*)
	QVariant(const QPoint &*val*)
	QVariant(const QPointF &*val*)
	QVariant(const QLine &*val*)
	QVariant(const QLineF &*val*)
	QVariant(const QRect &*val*)
	QVariant(const QRectF &*val*)
	QVariant(const QLocale &*l*)
	QVariant(const QRegExp &*regExp*)
	~QVariant()
bool	canConvert(int *targetTypeId*) const
bool	canConvert() const
void	clear()
bool	convert(int *targetTypeId*)
bool	isNull() const

bool	**isValid**() const
void	**setValue**(const T &*value*)
void	**swap**(QVariant &*other*)
QBitArray	**toBitArray**() const
bool	**toBool**() const
QByteArray	**toByteArray**() const
QChar	**toChar**() const
QDate	**toDate**() const
QDateTime	**toDateTime**() const
double	**toDouble**(bool **ok* = nullptr) const
QEasingCurve	**toEasingCurve**() const
float	**toFloat**(bool **ok* = nullptr) const
QHash<QString, QVariant>	**toHash**() const
int	**toInt**(bool **ok* = nullptr) const
QJsonArray	**toJsonArray**() const
QJsonDocument	**toJsonDocument**() const
QJsonObject	**toJsonObject**() const
QJsonValue	**toJsonValue**() const
QLine	**toLine**() const
QLineF	**toLineF**() const
QList<QVariant>	**toList**() const
QLocale	**toLocale**() const

qlonglong	**toLongLong**(bool **ok* = nullptr) const
QMap<QString, QVariant>	**toMap**() const
QModelIndex	**toModelIndex**() const
QPersistentModelIndex	**toPersistentModelIndex**() const
QPoint	**toPoint**() const
QPointF	**toPointF**() const
qreal	**toReal**(bool **ok* = nullptr) const
QRect	**toRect**() const
QRectF	**toRectF**() const
QRegExp	**toRegExp**() const
QRegularExpression	**toRegularExpression**() const
QSize	**toSize**() const
QSizeF	**toSizeF**() const
QString	**toString**() const
QStringList	**toStringList**() const
QTime	**toTime**() const
uint	**toUInt**(bool **ok* = nullptr) const
qulonglong	**toULongLong**(bool **ok* = nullptr) const
QUrl	**toUrl**() const
QUuid	**toUuid**() const
QVariant::Type	**type**() const
const char *	**typeName**() const

int	**userType**() const
T	**value**() const
bool	**operator!=**(const QVariant &*v*) const
bool	**operator<**(const QVariant &*v*) const
bool	**operator<=**(const QVariant &*v*) const
QVariant &	**operator=**(const QVariant &*variant*)
QVariant &	**operator=**(QVariant &&*other*)
bool	**operator==**(const QVariant &*v*) const
bool	**operator>**(const QVariant &*v*) const
bool	**operator>=**(const QVariant &*v*) const

Static Public Members

QVariant	**fromStdVariant**(const int &*value*)
QVariant	**fromValue**(const T &*value*)
QVariant::Type	**nameToType**(const char **name*)
const char *	**typeToName**(int *typeId*)

Related Non-Members

typedef	**QVariantHash**
typedef	**QVariantList**

typedef	**QVariantMap**
T	**qvariant_cast**(const QVariant &*value*)
bool	**operator!=**(const QVariant &*v1*, const QVariant &*v2*)
bool	**operator==**(const QVariant &*v1*, const QVariant &*v2*)

Detailed Description

The QVariant class acts like a union for the most common Qt data types.

Because C++ forbids unions from including types that have non-default constructors or destructors, most interesting Qt classes cannot be used in unions. Without QVariant, this would be a problem for QObject::property() and for database work, etc.

A QVariant object holds a single value of a single type() at a time. (Some type()s are multi-valued, for example a string list.) You can find out what type, T, the variant holds, convert it to a different type using convert(), get its value using one of the toT() functions (e.g., toSize()) and check whether the type can be converted to a particular type using canConvert().

The methods named toT() (e.g., toInt(), toString()) are const. If you ask for the stored type, they return a copy of the stored object. If you ask for a type that can be generated from the stored type, toT() copies and converts and leaves the object itself unchanged. If you ask for a type that cannot be generated from the stored type, the result depends on the type; see the function documentation for details.

Here is some example code to demonstrate the use of QVariant:

```
QDataStream out(...);
QVariant v(123);                // The variant now contains an int
int x = v.toInt();              // x = 123
out << v;                       // Writes a type tag and an int to out
v = QVariant("hello");          // The variant now contains a QByteArray
v = QVariant(tr("hello"));      // The variant now contains a QString
int y = v.toInt();              // y = 0 since v cannot be converted to an int
QString s = v.toString();       // s = tr("hello")  (see QObject::tr())
out << v;                       // Writes a type tag and a QString to out
out << v;                       // Writes a type tag and a QString to out
...
QDataStream in(...);            // (opening the previously written stream)
in >> v;                        // Reads an Int variant
int z = v.toInt();              // z = 123
qDebug("Type is %s",            // prints "Type is int"
        v.typeName());
v = v.toInt() + 100;            // The variant now hold the value 223
v = QVariant(QStringList());
```

You can even store QList<QVariant> and QMap<QString, QVariant> values in a variant, so you can easily construct arbitrarily complex data structures of arbitrary types. This is very powerful and versatile, but may prove less memory and speed efficient than storing specific types in standard data structures.

QVariant also supports the notion of null values, where you can have a defined type with no value set. However, note that QVariant types can only be cast when they have had a value set.

```
QVariant x, y(QString()), z(QString(""));
x.convert(QVariant::Int);
// x.isNull() == true
// y.isNull() == true, z.isNull() == false
```

QVariant can be extended to support other types than those mentioned in the Type enum. See Creating Custom Qt Types for details.

A Note on GUI Types

Because QVariant is part of the Qt Core module, it cannot provide conversion functions to data types defined in Qt GUI, such as QColor, QImage, and QPixmap. In other words, there is no `toColor()` function. Instead, you can use the QVariant::value() or the qvariant_cast() template function. For example:

```
    QVariant variant;
    ...
    QColor color = variant.value<QColor>();
```

The inverse conversion (e.g., from QColor to QVariant) is automatic for all data types supported by QVariant, including GUI-related types:

```
    QColor color = palette().background().color();
    QVariant variant = color;
```

Using canConvert() and convert() Consecutively

When using canConvert() and convert() consecutively, it is possible for canConvert() to return true, but convert() to return false. This is typically because canConvert() only reports the general ability of QVariant to convert between types given suitable data; it is still possible to supply data which cannot actually be converted.

For example, canConvert(Int) would return true when called on a variant containing a string because, in principle, QVariant is able to convert strings of numbers to integers. However, if the string contains non-numeric characters, it cannot be converted to an integer, and any attempt to convert it will fail. Hence, it is important to have both functions return true for a successful conversion.

See also QMetaType.

Member Function Documentation

QVariant::QVariant()

Constructs an invalid variant.

QVariant::QVariant(QVariant::Type *type*)

Constructs an uninitialized variant of type *type*. This will create a variant in a special null state that if accessed will return a default constructed value of the *type*.

See also isNull().

QVariant::QVariant(const QRegularExpression &*re*)

Constructs a new variant with the regular expression value *re*.

This function was introduced in Qt 5.0.

QVariant::QVariant(const QUrl &*val*)

Constructs a new variant with a url value of *val*.

QVariant::QVariant(const QEasingCurve &*val*)

Constructs a new variant with an easing curve value, *val*.

This function was introduced in Qt 4.7.

QVariant::QVariant(const QUuid &*val*)

QVariant::QVariant(const QUuid &*val*)

Constructs a new variant with an uuid value, *val*.

This function was introduced in Qt 5.0.

QVariant::QVariant(const QJsonValue &*val*)

Constructs a new variant with a json value, *val*.

This function was introduced in Qt 5.0.

QVariant::QVariant(const QJsonObject &*val*)

Constructs a new variant with a json object value, *val*.

This function was introduced in Qt 5.0.

QVariant::QVariant(const QJsonArray &*val*)

Constructs a new variant with a json array value, *val*.

This function was introduced in Qt 5.0.

QVariant::QVariant(const QJsonDocument &*val*)

Constructs a new variant with a json document value, *val*.

This function was introduced in Qt 5.0.

QVariant::QVariant(const QModelIndex &*val*)

Constructs a new variant with a QModelIndex value, *val*.

This function was introduced in Qt 5.0.

QVariant::QVariant(const QPersistentModelIndex &*val*)

Constructs a new variant with a QPersistentModelIndex value, *val*.

This function was introduced in Qt 5.5.

QVariant::QVariant(QVariant &&*other*)

Move-constructs a QVariant instance, making it point at the same object that *other* was pointing to.

This function was introduced in Qt 5.2.

QVariant::QVariant(int *typeId*, const void **copy*)

Constructs variant of type *typeId*, and initializes with *copy* if *copy* is not 0.

Note that you have to pass the address of the variable you want stored.

Usually, you never have to use this constructor, use QVariant::fromValue() instead to construct variants from the pointer types represented by `QMetaType::VoidStar`, and `QMetaType::QObjectStar`.

See also QVariant::fromValue() and QMetaType::Type.

QVariant::QVariant(const QVariant &*p*)

Constructs a copy of the variant, *p*, passed as the argument to this constructor.

QVariant::QVariant(QDataStream &*s*)

Reads the variant from the data stream, *s*.

QVariant::QVariant(int *val*)

Constructs a new variant with an integer value, *val*.

QVariant::QVariant(uint *val*)

Constructs a new variant with an unsigned integer value, *val*.

QVariant::QVariant(qlonglong *val*)

Constructs a new variant with a long long integer value, *val*.

QVariant::QVariant(qulonglong *val*)

Constructs a new variant with an unsigned long long integer value, *val*.

QVariant::QVariant(bool *val*)

Constructs a new variant with a boolean value, *val*.

QVariant::QVariant(double *val*)

Constructs a new variant with a floating point value, *val*.

QVariant::QVariant(float *val*)

Constructs a new variant with a floating point value, *val*.

This function was introduced in Qt 4.6.

QVariant::QVariant(const char **val*)

Constructs a new variant with a string value of *val*. The variant creates a deep copy of *val* into a QString assuming UTF-8 encoding on the input *val*.

Note that *val* is converted to a QString for storing in the variant and QVariant::userType() will return QMetaType::QString for the variant.

You can disable this operator by defining `QT_NO_CAST_FROM_ASCII` when you compile your applications.

QVariant::QVariant(const QByteArray &*val*)

Constructs a new variant with a bytearray value, *val*.

QVariant::QVariant(const QBitArray &*val*)

Constructs a new variant with a bitarray value, *val*.

QVariant::QVariant(const QString &*val*)

Constructs a new variant with a string value, *val*.

QVariant::QVariant(QLatin1String *val*)

Constructs a new variant with a string value, *val*.

QVariant::QVariant(const QStringList &*val*)

Constructs a new variant with a string list value, *val*.

QVariant::QVariant(QChar *c*)

Constructs a new variant with a char value, *c*.

QVariant::QVariant(const QDate &*val*)

Constructs a new variant with a date value, *val*.

QVariant::QVariant(const QTime &*val*)

Constructs a new variant with a time value, *val*.

QVariant::QVariant(const QDateTime &*val*)

Constructs a new variant with a date/time value, *val*.

QVariant::QVariant(const QList<QVariant> &*val*)

Constructs a new variant with a list value, *val*.

QVariant::QVariant(const QMap<QString, QVariant> &*val*)

Constructs a new variant with a map of QVariants, *val*.

QVariant::QVariant(const QHash<QString, QVariant> &*val*)

Constructs a new variant with a hash of QVariants, *val*.

QVariant::QVariant(const QSize &*val*)

Constructs a new variant with a size value of *val*.

QVariant::QVariant(const QSizeF &*val*)

Constructs a new variant with a size value of *val*.

QVariant::QVariant(const QPoint &*val*)

Constructs a new variant with a point value of *val*.

QVariant::QVariant(const QPointF &*val*)

Constructs a new variant with a point value of *val*.

QVariant::QVariant(const QLine &*val*)

Constructs a new variant with a line value of *val*.

QVariant::QVariant(const QLineF &*val*)

Constructs a new variant with a line value of *val*.

QVariant::QVariant(const QRect &*val*)

Constructs a new variant with a rect value of *val*.

QVariant::QVariant(const QRectF &*val*)

Constructs a new variant with a rect value of *val*.

QVariant::QVariant(const QLocale &*l*)

Constructs a new variant with a locale value, *l*.

QVariant::QVariant(const QRegExp &*regExp*)

Constructs a new variant with the regexp value *regExp*.

QVariant::~QVariant()

Destroys the QVariant and the contained object.

Note that subclasses that reimplement clear() should reimplement the destructor to call clear(). This destructor calls clear(), but because it is the destructor, QVariant::clear() is called rather than a subclass's clear().

bool QVariant::canConvert(int *targetTypeId*) const

Returns `true` if the variant's type can be cast to the requested type, *targetTypeId*. Such casting is done automatically when calling the toInt(), toBool(), ... methods.

The following casts are done automatically:

Type	Automatically Cast To
QMetaType::Bool	QMetaType::QChar, QMetaType::Double, QMetaType::Int, QMetaType::LongLong, QMetaType::QString, QMetaType::UInt, QMetaType::ULongLong

Type	Automatically Cast To
QMetaType::QByteArray	QMetaType::Double, QMetaType::Int, QMetaType::LongLong, QMetaType::QString, QMetaType::UInt, QMetaType::ULongLong, QMetaType::QUuid
QMetaType::QChar	QMetaType::Bool, QMetaType::Int, QMetaType::UInt, QMetaType::LongLong, QMetaType::ULongLong
QMetaType::QColor	QMetaType::QString
QMetaType::QDate	QMetaType::QDateTime, QMetaType::QString
QMetaType::QDateTime	QMetaType::QDate, QMetaType::QString, QMetaType::QTime
QMetaType::Double	QMetaType::Bool, QMetaType::Int, QMetaType::LongLong, QMetaType::QString, QMetaType::UInt, QMetaType::ULongLong
QMetaType::QFont	QMetaType::QString
QMetaType::Int	QMetaType::Bool, QMetaType::QChar, QMetaType::Double, QMetaType::LongLong, QMetaType::QString, QMetaType::UInt, QMetaType::ULongLong
QMetaType::QKeySequence	QMetaType::Int, QMetaType::QString
QMetaType::QVariantList	QMetaType::QStringList (if the list's items can be converted to QStrings)
QMetaType::LongLong	QMetaType::Bool, QMetaType::QByteArray, QMetaType::QChar, QMetaType::Double, QMetaType::Int, QMetaType::QString, QMetaType::UInt, QMetaType::ULongLong
QMetaType::QPoint	QMetaType::QPointF
QMetaType::QRect	QMetaType::QRectF
QMetaType::QString	QMetaType::Bool, QMetaType::QByteArray, QMetaType::QChar, QMetaType::QColor, QMetaType::QDate, QMetaType::QDateTime, QMetaType::Double, QMetaType::QFont, QMetaType::Int, QMetaType::QKeySequence, QMetaType::LongLong, QMetaType::QStringList, QMetaType::QTime, QMetaType::UInt, QMetaType::ULongLong, QMetaType::QUuid
QMetaType::QStringList	QMetaType::QVariantList, QMetaType::QString (if the list contains exactly one item)
QMetaType::QTime	QMetaType::QString
QMetaType::UInt	QMetaType::Bool, QMetaType::QChar, QMetaType::Double, QMetaType::Int, QMetaType::LongLong, QMetaType::QString, QMetaType::ULongLong

QMetaType::ULongLong	QMetaType::Bool, QMetaType::QChar, QMetaType::Double, QMetaType::Int, QMetaType::LongLong, QMetaType::QString, QMetaType::UInt
QMetaType::QUuid	QMetaType::QByteArray, QMetaType::QString

A QVariant containing a pointer to a type derived from QObject will also return true for this function if a qobject_cast to the type described by *targetTypeId* would succeed. Note that this only works for QObject subclasses which use the Q_OBJECT macro.

A QVariant containing a sequential container will also return true for this function if the *targetTypeId* is QVariantList. It is possible to iterate over the contents of the container without extracting it as a (copied) QVariantList:

```
QList<int> intList = {7, 11, 42};

QVariant variant = QVariant::fromValue(intList);
if (variant.canConvert<QVariantList>()) {
    QSequentialIterable iterable = variant.value<QSequentialIterable>();
    // Can use foreach:
    foreach (const QVariant &v, iterable) {
        qDebug() << v;
    }
    // Can use C++11 range-for:
    for (const QVariant &v : iterable) {
        qDebug() << v;
    }
    // Can use iterators:
    QSequentialIterable::const_iterator it = iterable.begin();
    const QSequentialIterable::const_iterator end = iterable.end();
    for ( ; it != end; ++it) {
        qDebug() << *it;
    }
}
```

This requires that the value_type of the container is itself a metatype.

Similarly, a QVariant containing a sequential container will also return true for this function the *targetTypeId* is QVariantHash or QVariantMap. It is possible to iterate over the contents of the container without extracting it as a (copied) QVariantHash or QVariantMap:

```
QHash<int, QString> mapping;
mapping.insert(7, "Seven");
mapping.insert(11, "Eleven");
mapping.insert(42, "Forty-two");

QVariant variant = QVariant::fromValue(mapping);
if (variant.canConvert<QVariantHash>()) {
    QAssociativeIterable iterable = variant.value<QAssociativeIterable>();
    // Can use foreach over the values:
    foreach (const QVariant &v, iterable) {
        qDebug() << v;
    }
    // Can use C++11 range-for over the values:
    for (const QVariant &v : iterable) {
        qDebug() << v;
    }
    // Can use iterators:
    QAssociativeIterable::const_iterator it = iterable.begin();
    const QAssociativeIterable::const_iterator end = iterable.end();
    for ( ; it != end; ++it) {
        qDebug() << *it; // The current value
        qDebug() << it.key();
        qDebug() << it.value();
    }
}
```

See also convert(), QSequentialIterable, Q_DECLARE_SEQUENTIAL_CONTAINER_METATYPE(), QAssociativeIterable, and Q_DECLARE_ASSOCIATIVE_CONTAINER_METATYPE().

bool QVariant::canConvert() const

Returns `true` if the variant can be converted to the template type T, otherwise false.

Example:

```
QVariant v = 42;

v.canConvert<int>();              // returns true
v.canConvert<QString>();          // returns true

MyCustomStruct s;
v.setValue(s);

v.canConvert<int>();              // returns false
v.canConvert<MyCustomStruct>();   // returns true
```

A QVariant containing a pointer to a type derived from QObject will also return true for this function if a qobject_cast to the template type T would succeed. Note that this only works for QObject subclasses which use the Q_OBJECT macro.

See also convert().

void QVariant::clear()

Convert this variant to type QMetaType::UnknownType and free up any resources used.

bool QVariant::convert(int *targetTypeId*)

Casts the variant to the requested type, *targetTypeId*. If the cast cannot be done, the variant is still changed to the requested type, but is left in a cleared null state similar to that constructed by QVariant(Type).

Returns `true` if the current type of the variant was successfully cast; otherwise returns `false`.

A QVariant containing a pointer to a type derived from QObject will also convert and return true for this function if a qobject_cast to the type described by *targetTypeId* would succeed. Note that this only works for QObject subclasses which use the Q_OBJECT macro.

Note: converting QVariants that are null due to not being initialized or having failed a previous conversion will always fail, changing the type, remaining null, and returning `false`.

See also canConvert(int targetTypeId) and clear().

QVariant QVariant::fromStdVariant(const int &*value*) `[static]`

Returns a QVariant with the type and value of the active variant of *value*. If the active type is std::monostate a default QVariant is returned.

Note: With this method you do not need to register the variant as a Qt metatype, since the std::variant is resolved before being stored. The component types should be registered however.

This function was introduced in Qt 5.11.

See also fromValue().

QVariant QVariant::fromValue(const T &*value*) `[static]`

Returns a QVariant containing a copy of *value*. Behaves exactly like setValue() otherwise.

Example:

```
MyCustomStruct s;
return QVariant::fromValue(s);
```

Note: If you are working with custom types, you should use the Q_DECLARE_METATYPE() macro to register your custom type.

See also setValue() and value().

bool QVariant::isNull() const

Returns `true` if this is a null variant, false otherwise. A variant is considered null if it contains no initialized value, or the contained value is a null pointer or is an instance of a built-in type that has an isNull method, in which case the result would be the same as calling isNull on the wrapped object.

Warning: Null variants is not a single state and two null variants may easily return `false` on the == operator if they do not contain similar null values.

See also convert(int).

bool QVariant::isValid() const

Returns `true` if the storage type of this variant is not QMetaType::UnknownType; otherwise returns `false`.

QVariant::Type QVariant::nameToType(const char **name*) `[static]`

Converts the string representation of the storage type given in *name*, to its enum representation.

If the string representation cannot be converted to any enum representation, the variant is set to `Invalid`.

void QVariant::setValue(const T &*value*)

Stores a copy of *value*. If T is a type that QVariant doesn't support, QMetaType is used to store the value. A compile error will occur if QMetaType doesn't handle the type.

Example:

```
QVariant v;

v.setValue(5);
int i = v.toInt();         // i is now 5
QString s = v.toString()   // s is now "5"

MyCustomStruct c;
v.setValue(c);

...

MyCustomStruct c2 = v.value<MyCustomStruct>();
```

See also value(), fromValue(), and canConvert().

void QVariant::swap(QVariant &*other*)

Swaps variant *other* with this variant. This operation is very fast and never fails.

This function was introduced in Qt 4.8.

QBitArray QVariant::toBitArray() const

Returns the variant as a QBitArray if the variant has userType() QMetaType::QBitArray; otherwise returns an empty bit array.

See also canConvert(int targetTypeId) and convert().

bool QVariant::toBool() const

Returns the variant as a bool if the variant has userType() Bool.

Returns `true` if the variant has userType() QMetaType::Bool, QMetaType::QChar, QMetaType::Double, QMetaType::Int, QMetaType::LongLong, QMetaType::UInt, or QMetaType::ULongLong and the value is non-zero, or if the variant has type QMetaType::QString or QMetaType::QByteArray and its lower-case content is not one of the following: empty, "0" or "false"; otherwise returns `false`.

See also canConvert(int targetTypeId) and convert().

QByteArray QVariant::toByteArray() const

Returns the variant as a QByteArray if the variant has userType() QMetaType::QByteArray or QMetaType::QString (converted using QString::fromUtf8()); otherwise returns an empty byte array.

See also canConvert(int targetTypeId) and convert().

QChar QVariant::toChar() const

Returns the variant as a QChar if the variant has userType() QMetaType::QChar, QMetaType::Int, or QMetaType::UInt; otherwise returns an invalid QChar.

See also canConvert(int targetTypeId) and convert().

QDate QVariant::toDate() const

Returns the variant as a QDate if the variant has userType() QMetaType::QDate, QMetaType::QDateTime, or QMetaType::QString; otherwise returns an invalid date.

If the type() is QMetaType::QString, an invalid date will be returned if the string cannot be parsed as a Qt::ISODate format date.

See also canConvert(int targetTypeId) and convert().

QDateTime QVariant::toDateTime() const

Returns the variant as a QDateTime if the variant has userType() QMetaType::QDateTime, QMetaType::QDate, or QMetaType::QString; otherwise returns an invalid date/time.

If the type() is QMetaType::QString, an invalid date/time will be returned if the string cannot be parsed as a Qt::ISODate format date/time.

See also canConvert(int targetTypeId) and convert().

double QVariant::toDouble(bool **ok* = nullptr) const

Returns the variant as a double if the variant has userType() QMetaType::Double, QMetaType::Float, QMetaType::Bool, QMetaType::QByteArray, QMetaType::Int, QMetaType::LongLong, QMetaType::QString, QMetaType::UInt, or QMetaType::ULongLong; otherwise returns 0.0.

QMetaType::Int, QMetaType::LongLong, QMetaType::QString, QMetaType::UInt, or QMetaType::ULongLong; otherwise returns 0.0.

If *ok* is non-null: **ok* is set to true if the value could be converted to a double; otherwise **ok* is set to false.

See also canConvert(int targetTypeId) and convert().

QEasingCurve QVariant::toEasingCurve() const

Returns the variant as a QEasingCurve if the variant has userType() QMetaType::QEasingCurve; otherwise returns a default easing curve.

This function was introduced in Qt 4.7.

See also canConvert(int targetTypeId) and convert().

float QVariant::toFloat(bool **ok* = nullptr) const

Returns the variant as a float if the variant has userType() QMetaType::Double, QMetaType::Float, QMetaType::Bool, QMetaType::QByteArray, QMetaType::Int, QMetaType::LongLong, QMetaType::QString, QMetaType::UInt, or QMetaType::ULongLong; otherwise returns 0.0.

If *ok* is non-null: **ok* is set to true if the value could be converted to a double; otherwise **ok* is set to false.

This function was introduced in Qt 4.6.

See also canConvert(int targetTypeId) and convert().

QHash<QString, QVariant> QVariant::toHash() const

Returns the variant as a QHash<QString, QVariant> if the variant has type() QMetaType::QVariantHash; otherwise returns an empty map.

See also canConvert(int targetTypeId) and convert().

See also canConvert(int targetTypeId) and convert().

int QVariant::toInt(bool **ok* = nullptr) const

Returns the variant as an int if the variant has userType() QMetaType::Int, QMetaType::Bool, QMetaType::QByteArray, QMetaType::QChar, QMetaType::Double, QMetaType::LongLong, QMetaType::QString, QMetaType::UInt, or QMetaType::ULongLong; otherwise returns 0.

If *ok* is non-null: **ok* is set to true if the value could be converted to an int; otherwise **ok* is set to false.

Warning: If the value is convertible to a QMetaType::LongLong but is too large to be represented in an int, the resulting arithmetic overflow will not be reflected in *ok*. A simple workaround is to use QString::toInt().

See also canConvert(int targetTypeId) and convert().

QJsonArray QVariant::toJsonArray() const

Returns the variant as a QJsonArray if the variant has userType() QJsonArray; otherwise returns a default constructed QJsonArray.

This function was introduced in Qt 5.0.

See also canConvert(int targetTypeId) and convert().

QJsonDocument QVariant::toJsonDocument() const

Returns the variant as a QJsonDocument if the variant has userType() QJsonDocument; otherwise returns a default constructed QJsonDocument.

This function was introduced in Qt 5.0.

See also canConvert(int targetTypeId) and convert().

QJsonObject QVariant::toJsonObject() const

Returns the variant as a QJsonObject if the variant has userType() QJsonObject; otherwise returns a default constructed QJsonObject.

This function was introduced in Qt 5.0.

See also canConvert(int targetTypeId) and convert().

QJsonValue QVariant::toJsonValue() const

Returns the variant as a QJsonValue if the variant has userType() QJsonValue; otherwise returns a default constructed QJsonValue.

This function was introduced in Qt 5.0.

See also canConvert(int targetTypeId) and convert().

QLine QVariant::toLine() const

Returns the variant as a QLine if the variant has userType() QMetaType::QLine; otherwise returns an invalid QLine.

See also canConvert(int targetTypeId) and convert().

QLineF QVariant::toLineF() const

Returns the variant as a QLineF if the variant has userType() QMetaType::QLineF; otherwise returns an invalid QLineF.

Returns the variant as a QLineF if the variant has userType() QMetaType::QLineF; otherwise returns an invalid QLineF.

See also canConvert(int targetTypeId) and convert().

QList<QVariant> QVariant::toList() const

Returns the variant as a QVariantList if the variant has userType() QMetaType::QVariantList or QMetaType::QStringList; otherwise returns an empty list.

See also canConvert(int targetTypeId) and convert().

QLocale QVariant::toLocale() const

Returns the variant as a QLocale if the variant has userType() QMetaType::QLocale; otherwise returns an invalid QLocale.

See also canConvert(int targetTypeId) and convert().

qlonglong QVariant::toLongLong(bool **ok* = nullptr) const

Returns the variant as a long long int if the variant has userType() QMetaType::LongLong, QMetaType::Bool, QMetaType::QByteArray, QMetaType::QChar, QMetaType::Double, QMetaType::Int, QMetaType::QString, QMetaType::UInt, or QMetaType::ULongLong; otherwise returns 0.

If *ok* is non-null: *ok is set to true if the value could be converted to an int; otherwise *ok is set to false.

See also canConvert(int targetTypeId) and convert().

QMap<QString, QVariant> QVariant::toMap() const

Returns the variant as a QMap<QString, QVariant> if the variant has type() QMetaType::QVariantMap; otherwise returns an empty map.

See also canConvert(int targetTypeId) and convert().

QModelIndex QVariant::toModelIndex() const

Returns the variant as a QModelIndex if the variant has userType() QModelIndex; otherwise returns a default constructed QModelIndex.

This function was introduced in Qt 5.0.

See also canConvert(int targetTypeId), convert(), and toPersistentModelIndex().

QPersistentModelIndex QVariant::toPersistentModelIndex() const

Returns the variant as a QPersistentModelIndex if the variant has userType() QPersistentModelIndex; otherwise returns a default constructed QPersistentModelIndex.

This function was introduced in Qt 5.5.

See also canConvert(int targetTypeId), convert(), and toModelIndex().

QPoint QVariant::toPoint() const

Returns the variant as a QPoint if the variant has userType() QMetaType::QPoint or QMetaType::QPointF; otherwise returns a null QPoint.

See also canConvert(int targetTypeId) and convert().

QPointF QVariant::toPointF() const

Returns the variant as a QPointF if the variant has userType() QMetaType::QPoint or QMetaType::QPointF; otherwise returns a null QPointF.

See also canConvert(int targetTypeId) and convert().

qreal QVariant::toReal(bool **ok* = nullptr) const

Returns the variant as a qreal if the variant has userType() QMetaType::Double, QMetaType::Float, QMetaType::Bool, QMetaType::QByteArray, QMetaType::Int, QMetaType::LongLong, QMetaType::QString, QMetaType::UInt, or QMetaType::ULongLong; otherwise returns 0.0.

If *ok* is non-null: **ok* is set to true if the value could be converted to a double; otherwise **ok* is set to false.

This function was introduced in Qt 4.6.

See also canConvert(int targetTypeId) and convert().

QRect QVariant::toRect() const

Returns the variant as a QRect if the variant has userType() QMetaType::QRect; otherwise returns an invalid QRect.

See also canConvert(int targetTypeId) and convert().

QRectF QVariant::toRectF() const

Returns the variant as a QRectF if the variant has userType() QMetaType::QRect or QMetaType::QRectF; otherwise returns an invalid QRectF.

See also canConvert(int targetTypeId) and convert().

QRegExp QVariant::toRegExp() const

Returns the variant as a QRegExp if the variant has userType() QMetaType::QRegExp; otherwise returns an empty QRegExp.

This function was introduced in Qt 4.1.

See also canConvert(int targetTypeId) and convert().

QRegularExpression QVariant::toRegularExpression() const

Returns the variant as a QRegularExpression if the variant has userType() QRegularExpression; otherwise returns an empty QRegularExpression.

This function was introduced in Qt 5.0.

See also canConvert(int targetTypeId) and convert().

QSize QVariant::toSize() const

Returns the variant as a QSize if the variant has userType() QMetaType::QSize; otherwise returns an invalid QSize.

See also canConvert(int targetTypeId) and convert().

QSizeF QVariant::toSizeF() const

Returns the variant as a QSizeF if the variant has userType() QMetaType::QSizeF; otherwise returns an invalid QSizeF.

See also canConvert(int targetTypeId) and convert().

QString QVariant::toString() const

Returns the variant as a QString if the variant has a userType() including, but not limited to:

QMetaType::QString, QMetaType::Bool, QMetaType::QByteArray, QMetaType::QChar, QMetaType::QDate, QMetaType::QDateTime, QMetaType::Double, QMetaType::Int, QMetaType::LongLong, QMetaType::QStringList, QMetaType::QTime, QMetaType::UInt, or QMetaType::ULongLong.

Calling QVariant::toString() on an unsupported variant returns an empty string.

See also canConvert(int targetTypeId) and convert().

QStringList QVariant::toStringList() const

Returns the variant as a QStringList if the variant has userType() QMetaType::QStringList, QMetaType::QString, or QMetaType::QVariantList of a type that can be converted to QString; otherwise returns an empty list.

See also canConvert(int targetTypeId) and convert().

QTime QVariant::toTime() const

Returns the variant as a QTime if the variant has userType() QMetaType::QTime, QMetaType::QDateTime, or QMetaType::QString; otherwise returns an invalid time.

If the type() is QMetaType::QString, an invalid time will be returned if the string cannot be parsed as a Qt::ISODate format time.

See also canConvert(int targetTypeId) and convert().

uint QVariant::toUInt(bool **ok* = nullptr) const

Returns the variant as an unsigned int if the variant has userType() QMetaType::UInt, QMetaType::Bool, QMetaType::QByteArray, QMetaType::QChar, QMetaType::Double, QMetaType::Int, QMetaType::LongLong, QMetaType::QString, or QMetaType::ULongLong; otherwise returns 0.

If *ok* is non-null: **ok* is set to true if the value could be converted to an unsigned int; otherwise **ok* is set to false.

Warning: If the value is convertible to a QMetaType::ULongLong but is too large to be represented in an unsigned int, the resulting arithmetic overflow will not be reflected in *ok*. A simple workaround is to use QString::toUInt().

See also canConvert(int targetTypeId) and convert().

qulonglong QVariant::toULongLong(bool **ok* = nullptr) const

Returns the variant as an unsigned long long int if the variant has type() QMetaType::ULongLong, QMetaType::Bool, QMetaType::QByteArray, QMetaType::QChar, QMetaType::Double, QMetaType::Int, QMetaType::LongLong, QMetaType::QString, or QMetaType::UInt; otherwise returns 0.

If *ok* is non-null: **ok* is set to true if the value could be converted to an int; otherwise **ok* is set to false.

See also canConvert(int targetTypeId) and convert().

QUrl QVariant::toUrl() const

Returns the variant as a QUrl if the variant has userType() QMetaType::QUrl; otherwise returns an invalid QUrl.

See also canConvert(int targetTypeId) and convert().

QUuid QVariant::toUuid() const

Returns the variant as a QUuid if the variant has type() QMetaType::QUuid, QMetaType::QByteArray or QMetaType::QString; otherwise returns a default-constructed QUuid.

This function was introduced in Qt 5.0.

See also canConvert(int targetTypeId) and convert().

QVariant::Type QVariant::type() const

Returns the storage type of the value stored in the variant. Although this function is declared as returning QVariant::Type, the return value should be interpreted as QMetaType::Type. In particular, QVariant::UserType is returned here only if the value is equal or greater than QMetaType::User.

Note that return values in the ranges QVariant::Char through QVariant::RegExp and QVariant::Font through QVariant::Transform correspond to the values in the ranges QMetaType::QChar through QMetaType::QRegExp and QMetaType::QFont through QMetaType::QQuaternion.

Pay particular attention when working with char and QChar variants. Note that there is no QVariant constructor specifically for type char, but there is one for QChar. For a variant of type QChar, this function returns QVariant::Char, which is the same as QMetaType::QChar, but for a variant of type `char`, this function returns QMetaType::Char, which is *not* the same as QVariant::Char.

Also note that the types `void*`, `long`, `short`, `unsigned long`, `unsigned short`, `unsigned char`, `float`, `QObject*`, and `QWidget*` are represented in QMetaType::Type but not in QVariant::Type, and they can be returned by this function. However, they are considered to be user defined types when tested against QVariant::Type.

To test whether an instance of QVariant contains a data type that is compatible with the data type you are interested in, use canConvert().

const char *QVariant::typeName() const

const char *QVariant::typeName() const

Returns the name of the type stored in the variant. The returned strings describe the C++ datatype used to store the data: for example, "QFont", "QString", or "QVariantList". An Invalid variant returns 0.

const char *QVariant::typeToName(int *typeId*) `[static]`

Converts the int representation of the storage type, *typeId*, to its string representation.

Returns a null pointer if the type is QMetaType::UnknownType or doesn't exist.

int QVariant::userType() const

Returns the storage type of the value stored in the variant. For non-user types, this is the same as type().

See also type().

T QVariant::value() const

Returns the stored value converted to the template type T. Call canConvert() to find out whether a type can be converted. If the value cannot be converted, a default-constructed value will be returned.

If the type T is supported by QVariant, this function behaves exactly as toString(), toInt() etc.

Example:

```
QVariant v;
QVariant v;

MyCustomStruct c;
if (v.canConvert<MyCustomStruct>())
    c = v.value<MyCustomStruct>();

v = 7;
int i = v.value<int>();                        // same as v.toInt()
QString s = v.value<QString>();                // same as v.toString(), s is now "7"
MyCustomStruct c2 = v.value<MyCustomStruct>(); // conversion failed, c2 is empty
```

If the QVariant contains a pointer to a type derived from QObject then T may be any QObject type. If the pointer stored in the QVariant can be qobject_cast to T, then that result is returned. Otherwise a null pointer is returned. Note that this only works for QObject subclasses which use the Q_OBJECT macro.

If the QVariant contains a sequential container and T is QVariantList, the elements of the container will be converted into QVariants and returned as a QVariantList.

```
QList<int> intList = {7, 11, 42};

QVariant variant = QVariant::fromValue(intList);
if (variant.canConvert<QVariantList>()) {
    QSequentialIterable iterable = variant.value<QSequentialIterable>();
    // Can use foreach:
    foreach (const QVariant &v, iterable) {
        qDebug() << v;
    }
    // Can use C++11 range-for:
    for (const QVariant &v : iterable) {
        qDebug() << v;
    }
    // Can use iterators:
    QSequentialIterable::const_iterator it = iterable.begin();
    const QSequentialIterable::const_iterator end = iterable.end();
    const QSequentialIterable::const_iterator end = iterable.end();
    for ( ; it != end; ++it) {
        qDebug() << *it;
    }
}
```

See also setValue(), fromValue(), canConvert(), and Q_DECLARE_SEQUENTIAL_CONTAINER_METATYPE().

bool QVariant::operator!=(const QVariant &*v*) const

Compares this QVariant with *v* and returns `true` if they are not equal; otherwise returns `false`.

Warning: To make this function work with a custom type registered with qRegisterMetaType(), its comparison operator must be registered using QMetaType::registerComparators().

bool QVariant::operator<(const QVariant &*v*) const

Compares this QVariant with *v* and returns `true` if this is less than *v*.

Note: Comparability might not be availabe for the type stored in this QVariant or in *v*.

Warning: To make this function work with a custom type registered with qRegisterMetaType(), its comparison operator must be registered using QMetaType::registerComparators().

bool QVariant::operator<=(const QVariant &*v*) const

Compares this QVariant with *v* and returns `true` if this is less or equal than *v*.

Note: Comparability might not be available for the type stored in this QVariant or in *v*.

Warning: To make this function work with a custom type registered with qRegisterMetaType(), its comparison operator must be registered using QMetaType::registerComparators().

QVariant &QVariant::operator=(const QVariant &*variant*)

Assigns the value of the variant *variant* to this variant.

QVariant &QVariant::operator=(QVariant &&*other*)

Move-assigns *other* to this QVariant instance.

This function was introduced in Qt 5.2.

bool QVariant::operator==(const QVariant &*v*) const

Compares this QVariant with *v* and returns `true` if they are equal; otherwise returns `false`.

QVariant uses the equality operator of the type() it contains to check for equality. QVariant will try to convert() *v* if its type is not the same as this variant's type. See canConvert() for a list of possible conversions.

Warning: To make this function work with a custom type registered with qRegisterMetaType(), its comparison operator must be registered using QMetaType::registerComparators().

bool QVariant::operator>(const QVariant &*v*) const

bool QVariant::operator>(const QVariant &*v*) const

Compares this QVariant with *v* and returns `true` if this is larger than *v*.

Note: Comparability might not be available for the type stored in this QVariant or in *v*.

Warning: To make this function work with a custom type registered with qRegisterMetaType(), its comparison operator must be registered using QMetaType::registerComparators().

bool QVariant::operator>=(const QVariant &*v*) const

Compares this QVariant with *v* and returns `true` if this is larger or equal than *v*.

Note: Comparability might not be available for the type stored in this QVariant or in *v*.

Warning: To make this function work with a custom type registered with qRegisterMetaType(), its comparison operator must be registered using QMetaType::registerComparators().

Related Non-Members

typedef QVariantHash

Synonym for QHash<QString, QVariant>.

This typedef was introduced in Qt 4.5.

typedef QVariantList

Synonym for QList<QVariant>.

typedef QVariantMap

Synonym for QMap<QString, QVariant>.

T qvariant_cast(const QVariant &*value*)

Returns the given *value* converted to the template type T.

This function is equivalent to QVariant::value().

See also QVariant::value().

bool operator!=(const QVariant &*v1*, const QVariant &*v2*)

Returns `false` if *v1* and *v2* are equal; otherwise returns `true`.

Warning: To make this function work with a custom type registered with qRegisterMetaType(), its comparison operator must be registered using QMetaType::registerComparators().

bool operator==(const QVariant &*v1*, const QVariant &*v2*)

Returns `true` if *v1* and *v2* are equal; otherwise returns `false`.

If *v1* and *v2* have the same type(), the type's equality operator is used for comparison. If not, it is attempted to convert() *v2* to the same type as *v1*. See canConvert() for a list of possible conversions.

The result of the function is not affected by the result of QVariant::isNull, which means that two values can be equal even if one of them is null and another is not.

Warning: To make this function work with a custom type registered with qRegisterMetaType(), its comparison operator must be registered using QMetaType::registerComparators().

Download	Product	Services	Developers	About us
Start for Free	Qt in Use	Technology Evaluation	Documentation	Training & Events
Qt for Application Development	Qt for Application Development	Proof of Concept	Examples & Tutorials	Resource Center
Qt for Device Creation	Qt for Device Creation	Design & Implementation	Development Tools	News
Qt Open Source	Commercial Features	Productization	Wiki	Careers
Terms & Conditions	Qt Creator IDE	Qt Training	Forums	Locations
Licensing FAQ	Qt Quick	Partner Network	Contribute to Qt	Contact Us

Qt The Qt Company