Qt Essentials - Basic Types Module

Training Course

Visit us at `http://qt.digia.com`

Produced by Digia Plc.

Material based on Qt 5.0, created on September 27, 2012

Digia Plc.

digia

- Qt's Object Model
  - QObject
  - QWidget
  - Variants
  - Properties
- String Handling
- Container Classes

Module Objectives

- Learn
  - what the Qt object model is
  - the basics of the widget system in C++
  - which utility classes exists to help you in C++

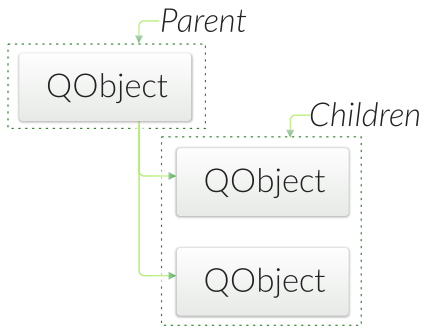
- Qt's Object Model
  - QObject
  - QWidget
  - Variants
  - Properties
- String Handling
- Container Classes

Qt's C++ Object Model - `QObject`

- `QObject` is the heart of Qt's object model
- Include these features:
  - Memory management
  - Object properties
  - Signals and slots
  - Event handling
- `QObject` has no visual representation

Object Tree

- `QObjects` organize themselves in object trees
  - Based on parent-child relationship
- `QObject(QObject *parent = 0)`
  - Parent adds object to list of children
  - Parent owns children
- Widget Centric
  - Used intensively with QtWidget
  - Less so when using Qt/C++ from QML



Note: Parent-child relationship is NOT inheritance

Creating Objects - General Guideline

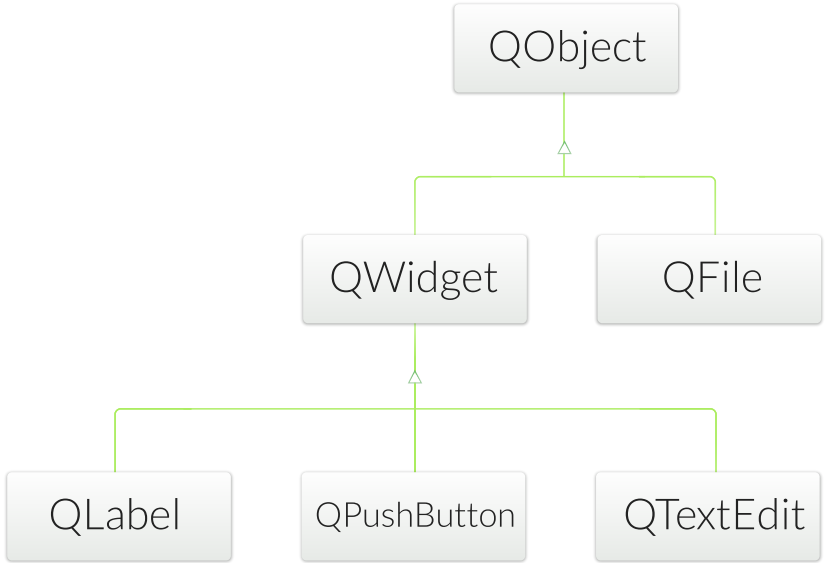
- **On Heap -** `QObject` with parent

```
QTimer* timer = new QTimer(this);
```

- **On Stack -** `QObject` without parent:
  - `QFile`, `QApplication`
  - Top level `QWidgets`: `QMainWindow`
- **On Stack** - value types
  - QString, QStringList, QColor
- **Stack or Heap -** `QDialog` - depending on lifetime

Qt's Widget Model - `QWidget`

- Derived from `QObject`
  - Adds visual representation
- Receives events
  - e.g. mouse, keyboard events
- Paints itself on screen
  - Using styles

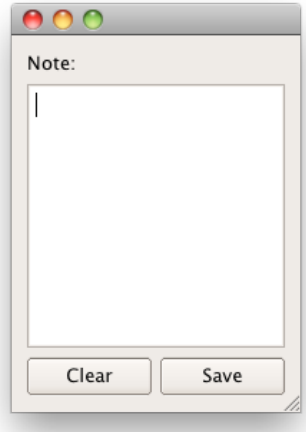


Object Tree and QWidget

- `new QWidget(0)`
  - Widget with no parent = "window"
- QWidget's children
  - Positioned in parent's coordinate system
  - Clipped by parent's boundaries
- QWidget parent
  - Propagates state changes
    - hides/shows children when it is hidden/shown itself
    - enables/disables children when it is enabled/disabled itself

Widgets that contain other widgets

- Container Widget
  - Aggregates other child-widgets
- Use layouts for aggregation
  - In this example: `QHBoxLayout` and `QVBoxLayout`
  - Note: Layouts are *not* widgets
- Layout Process
  - Add widgets to layout
  - Layouts may be nested
  - Set layout on container widget

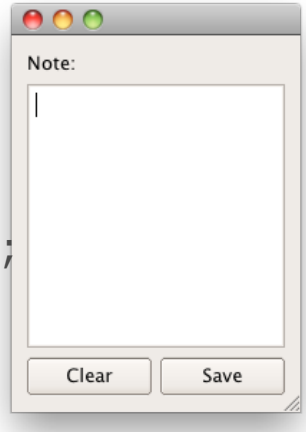


Example Container Widget

```
// container (window) widget creation
QWidget* container = new QWidget;
QLabel* label = new QLabel("Note:", container);
QTextEdit* edit = new QTextEdit(container);
QPushButton* clear = new QPushButton("Clear", container);
QPushButton* save = new QPushButton("Save", container);

// widget layout
QVBoxLayout* outer = new QVBoxLayout();
outer->addWidget(label);
outer->addWidget(edit);
QHBoxLayout* inner = new QHBoxLayout();
inner->addWidget(clear);
inner->addWidget(save);

container->setLayout(outer);
outer->addLayout(inner); // nesting layouts
```



Demo coretypes/ex-simplelayout

- `QVariant`
  - Union for common Qt "value types" (copyable, assignable)
  - Supports implicit sharing (fast copying)
  - Supports user types

- Use cases:

```
QVariant property(const char* name) const;
void setProperty(const char* name, const QVariant &value);
```

```
class QAbstractItemModel {
  virtual QVariant data( const QModelIndex& index, int role );
  ...
}
```

- For `QtCore` types

```
QVariant variant(42);
int value = variant.toInt(); // read back
qDebug() << variant.typeName(); // int
```

- For non-core and custom types:

```
QVariant variant = QVariant::fromValue(QColor(Qt::red));
QColor color = variant.value<QColor>(); // read back
qDebug() << variant.typeName(); // "QColor"
```

See QVariant Documentation

Custom data types in variants

```
#include <QMetaType>

class Contact
{
  public:
    void setName(const QString & name);
    QString name() const;
  ...
};

Q_DECLARE_METATYPE(Contact);
```

- Type must support default construction, copy and assignment.
- Q_DECLARE_METATYPE shoud after class definition in header file.

See Q_DECLARE_METATYPE Documentation

```
#include "Contact.h"
#include <QDebug>
#include <QVariant>

int main(int argc, char* argv[])
{
    Contact contact;
    contact.setName("Peter");

    const QVariant variant = QVariant::fromValue(contact);

    const Contact otherContact = variant.value<Contact>();
    qDebug() << otherContact.name(); // "Peter"
    qDebug() << variant.typeName();  // prints "Contact"

    return 0;
}
```

Demo coretypes/ex_custom_types

qRegisterMetaType

```
int main(int argc, char* argv[])
{
    // Register string typename:
    const int typeId = qRegisterMetaType<Contact>();

    Contact contact;
    contact.setName("Peter");

    // Create copy of object in a generic way
    void *object = QMetaType::construct(typeId, &contact);

    Contact *otherContact = reinterpret_cast<Contact*>(object);
    qDebug() << otherContact->name();

    return 0;
}
```

See qRegisterMetaType Documentation | See construct Documentation

- Qt Quick example

```
Rectangle {
    objectName: "myRect"
    height: 100
    ...
}
```

- Direct access (Broken, due to private headers)

```
QQuickRectangle* rectangle
  = root->findChild<QQuickRectangle*>("myRect");
int height = rectangle->height();
```

- Generic property access:

```
QObject* rectangle = root->findChild<QObject*>("myRect");
int height = rectangle->property("height").value<int>();
```

Using findChild is almost always a bad idea!

- The root object is the top level element in the QML document

```
QQuickItem *root = view.rootItem()->childItems().first();

// From previous slide:
QObject* rectangle = root->findChild<QObject*>("myRect");
int height = rectangle->property("height").value<int>();
```

Providing properties from QObject

```
class Customer : public QObject
{
    Q_OBJECT

    Q_PROPERTY(QString id READ getId WRITE setId NOTIFY idChanged);

  public:
     QString getId() const;
     void setId(const QString& id);

  signals:
     void idChanged();

  ...
};
```

Enum properties

```
class Customer : public QObject
{
    Q_OBJECT

    Q_PROPERTY(CustomerType type READ getType WRITE setType
               NOTIFY typeChanged);

  public:
    enum CustomerType {
      Corporate, Individual, Educational, Government
    };

    Q_ENUMS(CustomerType);

  ...
};
```

- Q_Property is a macro:

```
Q_PROPERTY( type name READ getFunction [WRITE setFunction]
[RESET resetFunction] [NOTIFY notifySignal] [DESIGNABLE bool]
[SCRIPTABLE bool] [STORED bool] )
```

- Property access methods:

```
QVariant property(const char* name) const;
void setProperty(const char* name, const QVariant &value);
```

- If name is not declared as a `Q_PROPERTY`
  - -> **dynamic property**
  - Not accessible from Qt Quick.
- Note:
  - Q_OBJECT macro required for properties to work
  - `QMetaObject` knows nothing about dynamic properties

- `QMetaObject` support property introspection

```
const QMetaObject *metaObject = object->metaObject();
const QString className = metaObject->className();
const int propertyCount = metaObject->propertyCount();
for ( int i=0; i<propertyCount; ++i ) {
  const QMetaProperty metaProperty = metaObject->property(i);
  const QString typeName = metaProperty.typeName()
  const QString propertyName = metaProperty.name();
  const QVariant value = object->property(metaProperty.name());
}
```

Demo coretypes/ex-properties

- Qt's Object Model
  - QObject
  - QWidget
  - Variants
  - Properties
- String Handling
- Container Classes

Strings can be created in a number of ways:

- Conversion constructor and assignment operators:

```
QString str("abc");
str = "def";
```

- From a number using a static function:

```
QString n = QString::number(1234);
```

- From a char pointer using the static functions:

```
QString text = QString::fromLatin1("Hello Qt");
QString text = QString::fromUtf8(inputText);
QString text = QString::fromLocal8Bit(cmdLineInput);
QString text = QStringLiteral("Literal string"); (Assumed to be UTF-8)
```

- From char pointer with translations:

```
QString text = tr("Hello Qt");
```

- operator+ and operator+=

```
QString str = str1 + str2;
fileName += ".txt";
```

- simplified() // removes duplicate whitespace
- left(), mid(), right() // part of a string
- leftJustified(), rightJustified() // padded version

```
QString s = "apple";
QString t = s.leftJustified(8, '.'); // t == "apple..."
```

Data can be extracted from strings.

- Numbers:

```
QString text = ...;
int value = text.toInt();
float value = text.toFloat();
```

- Strings:

```
QString text = ...;
QByteArray bytes = text.toLatin1();
QByteArray bytes = text.toUtf8();
QByteArray bytes = text.toLocal8Bit();
```

Formatted output with QString::arg()

```
int i = ...;
int total = ...;
QString fileName = ...;
QString status = tr("Processing file %1 of %2: %3")
                 .arg(i).arg(total).arg(fileName);
double d = 12.34;
QString str = QString::fromLatin1("delta: %1").arg(d,0,'E',3);
// str == "delta: 1.234E+01"
```

- Safer: `arg(QString,...,QString)` (``multi-arg()'').
  - But: only works with `QString` arguments.

- Obtaining raw character data from a QByteArray:

```
char *str = bytes.data();
const char *str = bytes.constData();
```

WARNING:

- Character data is only valid for the lifetime of the byte array.
- Calling a non-const member of bytes also invalidates ptr.
- Either copy the character data or keep a copy of the byte array.

- `length()`
- `endsWith()` and `startsWith()`
- `contains()`, `count()`
- `indexOf()` and `lastIndexOf()`

Expression can be characters, strings, or regular expressions

Text Processing with `QStringList`

- `QString::split()`, `QStringList::join()`
- `QStringList::replaceInStrings()`
- `QStringList::filter()`

Working with Regular Expressions

- `QRegExp` supports
  - Regular expression matching
  - Wildcard matching
- `QString cap(int)`
  `QStringList capturedTexts()`

```
QRegExp rx("^\\d\\d?$");     // match integers 0 to 99
rx.indexIn("123");           // returns -1 (no match)
rx.indexIn("-6");            // returns -1 (no match)
rx.indexIn("6");             // returns 0 (matched as position 0)
```

See QRegExp Documentation

- Qt's Object Model
  - QObject
  - QWidget
  - Variants
  - Properties
- String Handling
- Container Classes

General purpose template-based container classes

- `QList<QString>` - *Sequence Container*
  - Other: QLinkedList, QStack , QQueue ...
- `QMap<int, QString>` - *Associative Container*
  - Other: QHash, QSet, QMultiMap, QMultiHash

Qt's Container Classes compared to STL

- Lighter, safer, and easier to use than STL containers
- If you prefer STL, feel free to continue using it.
- Methods exist that convert between Qt and STL
  - e.g. you need to pass `std::list` to a Qt method

- Using `QList`

```
QList<QString> list;
list << "one" << "two" << "three";
QString item1 = list[1]; // "two"
for(int i=0; i<list.count(); i++) {
  const QString &item2 = list.at(i);
}
int index = list.indexOf("two"); // returns 1
```

- Using `QMap`

```
QMap<QString, int> map;
map["Norway"] = 5; map["Italy"] = 48;
int value = map["France"]; // inserts key if not exists
if(map.contains("Norway")) {
  int value2 = map.value("Norway"); // recommended lookup
}
```

Concern

How fast is a function when number of items grow

- Sequential Container

	Lookup	Insert	Append	Prepend
QList	O(1)	O(n)	O(1)	O(1)
QVector	O(1)	O(n)	O(1)	O(n)
QLinkedList	O(n)	O(1)	O(1)	O(1)

- Associative Container

	Lookup	Insert
QMap	O(log(n))	O(log(n))
QHash	O(1)	O(1)

all complexities are amortized

Storing Classes in Qt Container

- Class must be an *assignable data type*
- Class is *assignable*, if:

```
class Contact {
public:
  Contact() {} // default constructor
  Contact(const Contact &other); // copy constructor
  // assignment operator
  Contact &operator=(const Contact &other);
};
```

- *If copy constructor or assignment operator is not provided*
  - C++ will provide one (uses member copying)
- *If no constructors provided*
  - Empty default constructor provided by C++

Requirements on Container Keys

- Type `K` as key for `QMap`:
  - `bool K::operator<( const K& )` or
    `bool operator<( const K&, const K& )`

```
bool Contact::operator<(const Contact& c);
bool operator<(const Contact& c1, const Contact& c2);
```

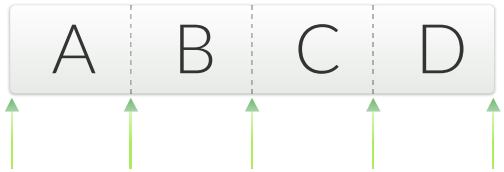
See QMap Documentation

- Type `K` as key for `QHash` or `QSet`:
  - `bool K::operator==( const K& )` or
    `bool operator==( const K&, const K& )`
  - `uint qHash( const K& )`

See QHash Documentation

- Allow reading a container's content sequentially
- **Java-style iterators**: simple and easy to use
  - `QListIterator<...>` for read
  - `QMutableListIterator<...>` for read-write
- **STL-style iterators** slightly more efficient
  - `QList::const_iterator` for read
  - `QList::iterator()` for read-write
- Same works for QSet, QMap, QHash, ...

Iterators Java style



- Example QList iterator

```
QList<QString> list;
list << "A" << "B" << "C" << "D";
QListIterator<QString> it(list);
```

- Forward iteration

```
while(it.hasNext()) {
  qDebug() << it.next();    // A B C D
}
```

- Backward iteration

```
it.toBack(); // position after the last item
while(it.hasPrevious()) {
  qDebug() << it.previous(); // D C B A
}
```

See QListIterator Documentation

Modifying During Iteration

- Use *mutable* versions of the iterators
  - e.g. `QMutableListIterator`.

```
QList<int> list;
list << 1 << 2 << 3 << 4;
QMutableListIterator<int> i(list);
while (i.hasNext()) {
  if (i.next() % 2 != 0)
    i.remove();
}
// list now 2, 4
```

- `remove()` and `setValue()`
  - Operate on items just jumped over using next()/previous()
- `insert()`
  - Inserts item at current position in sequence
  - `previous()` reveals just inserted item

Iterating Over QMap and QHash

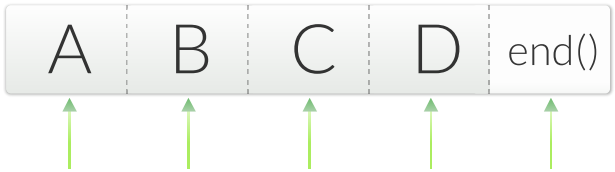
- `next()` and `previous()`
  - Return Item class with `key()` and `value()`
- Alternatively use `key()` and `value()` from iterator

```
QMap<QString, QString> map;
map["Paris"] = "France";
map["Guatemala City"] = "Guatemala";
map["Mexico City"] = "Mexico";
map["Moscow"] = "Russia";

QMutableMapIterator<QString, QString> i(map);
while (i.hasNext()) {
  if (i.next().key().endsWith("City"))
    i.remove();
}
// map now "Paris", "Moscow"
```

Demo coretypes/ex-containers

- Example QList iterator




```
QList<QString> list;
list << "A" << "B" << "C" << "D";
QList<QString>::iterator i;
```

- Forward mutable iteration

```
for (i = list.begin(); i != list.end(); ++i) {
    *i = (*i).toLower();
}
```

- Backward mutable iteration

```
i = list.end();
while (i != list.begin()) {
    --i;
    *i = (*i).toLower();
}
```

- `QList<QString>::const_iterator` for read-only

The `foreach` Keyword

- It is a macro, feels like a keyword

`foreach` (*variable*, *container*) *statement*

```
foreach (const QString& str, list) {
  if (str.isEmpty())
    break;
  qDebug() << str;
}
```

- `break` and `continue` as normal
- Modifying the container while iterating
  - results in container being copied
  - iteration continues in unmodified version
- Not possible to modify item
  - iterator variable is a const reference.

Algorithms

- STL-style iterators are compatible with the STL algorithms
  - Defined in the STL <algorithm> header
- Qt has own algorithms
  - Defined in <QtAlgorithms> header
- *If STL is available on all your supported platforms you can choose to use the STL algorithms*
  - The collection is much larger than the one in Qt.

Algorithms

- **qSort**(begin, end) sort items in range
- **qFind**(begin, end, value) find value
- **qEqual**(begin1, end1, begin2) checks two ranges
- **qCopy**(begin1, end1, begin2) from one range to another
- **qCount**(begin, end, value, n) occurrences of value in range
- and more ...

See QtAlgorithms Documentation

Counting 1's in list

```
QList<int> list;
list << 1 << 2 << 3 << 1;
int count = 0;
qCount(list, 1, count); // count the 1's
qDebug() << count;      // 2  (means 2 times 1)
```

- For parallel (ie. multi-threaded) algorithms
  - See `QtConcurrent` Documentation

- Copy list to vector example

```
QList<QString> list;
list << "one" << "two" << "three";
QVector<QString> vector(3);
qCopy(list.begin(), list.end(), vector.begin());
// vector: [ "one", "two", "three" ]
```

- Case insensitive sort example

```
bool lessThan(const QString& s1, const QString& s2) {
    return s1.toLower() < s2.toLower();
}
// ...
QList<QString> list;
list << "AlPha" << "beTA" << "gamma" << "DELTA";
qSort(list.begin(), list.end(), lessThan);
// list: [ "AlPha", "beTA", "DELTA", "gamma" ]
```

Implicit Sharing

If an object is copied, then its data is copied *only when* the data of one of the objects is changed

- Shared class has a pointer to shared data block
  - Shared data block = reference counter and actual data
- Assignment is a shallow copy
- Changing results into deep copy (detach)

```
QList<int> l1, l2; l1 << 1 << 2;
l2 = l1; // shallow-copy: l2 shares date with l1
l2 << 3; // deep-copy: change triggers detach from l1
```

Important to remember when inserting items into a container, or when returning a container.

- Explain how object ownership works in Qt?
- What are the key responsibilities of `QObject`?
- What does it mean when a QWidget has no parent?
- What is the purpose of the class `QVariant`?
- What do you need to do to support properties in a class?
- Name the different ways to go through the elements in a list, and discuss advantages and disadvantages of each method.

Questions And Answers

- Explain how object ownership works in Qt?
- What are the key responsibilities of `QObject`?
- What does it mean when a QWidget has no parent?
- What is the purpose of the class `QVariant`?
- What do you need to do to support properties in a class?
- Name the different ways to go through the elements in a list, and discuss advantages and disadvantages of each method.

Questions And Answers

- Explain how object ownership works in Qt?
- What are the key responsibilities of `QObject`?
- What does it mean when a QWidget has no parent?
- What is the purpose of the class `QVariant`?
- What do you need to do to support properties in a class?
- Name the different ways to go through the elements in a list, and discuss advantages and disadvantages of each method.

Questions And Answers

- Explain how object ownership works in Qt?
- What are the key responsibilities of `QObject`?
- What does it mean when a QWidget has no parent?
- What is the purpose of the class `QVariant`?
- What do you need to do to support properties in a class?
- Name the different ways to go through the elements in a list, and discuss advantages and disadvantages of each method.

Questions And Answers

- Explain how object ownership works in Qt?
- What are the key responsibilities of `QObject`?
- What does it mean when a QWidget has no parent?
- What is the purpose of the class `QVariant`?
- What do you need to do to support properties in a class?
- Name the different ways to go through the elements in a list, and discuss advantages and disadvantages of each method.

- Explain how object ownership works in Qt?
- What are the key responsibilities of `QObject`?
- What does it mean when a QWidget has no parent?
- What is the purpose of the class `QVariant`?
- What do you need to do to support properties in a class?
- Name the different ways to go through the elements in a list, and discuss advantages and disadvantages of each method.