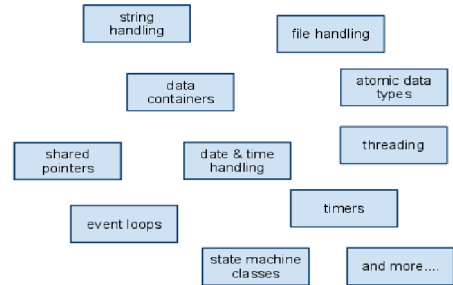


Qt Fundamentals: Core Classes

1

QtCore

The Qt Core module contains core non-GUI functionality such as:



2

Managing text using QString

- Stores strings as *Unicode* characters
 - Can represent almost all writing systems in use today.
- Support conversions from one locale to another
- Provides convenient API for string inspection and modification:
 - **replace**, **insert**, **compare**, **remove**, etc.
- QString can be created in a number of ways:

constructors e.g.: `QString hello("hello you");`

operator+() e.g.: `QString helloworld = "hello " + world;`

string builder e.g.: `QString answer = "hello" % world;`

3

QStringBuilder

- Using the **operator+** to join strings results in numerous memory allocations and checks for string length
- A better way to do it is to include `QStringBuilder` and use the `%` operator.

Good: `QString answer = "hello" % world % anotherString % andAnother;`

Bad: `QString answer = "hello";
answer = answer % world;
answer = answer % anotherString;
answer = answer % andAnother;`

Can be a little tricky to use, one of the strings must be a QString

4

QString::arg

- The `arg` method replaces `%1-99` with values

```
QString answer = QString("%1 plus %2 equal %3").arg(4).arg(2).arg(6);
```

- Can handle strings, chars, integers and floats
- Can convert between number bases

```
...).arg(value, width, base, fillchar);  
...).arg(42, 3, 16, QChar('0')); // Results in 02a
```

```
...).arg(qulonglong a)    ...).arg(QString, ... QString)  
...).arg(short a)       ...).arg(int a)  
...).arg(ushort a)      ...).arg(uint a)  
...).arg(QChar a)       ...).arg(long a)  
...).arg(char a)        ...).arg(ulong a)  
...).arg(double a)      ...).arg(qulonglong a)
```

5

Substrings

- Access substrings using **left**, **right** and **mid**

```
QString s = "Hello world";  
r = s.left(5); // "Hello"  
r = s.right(5); // "world"  
r = s.mid(6,5); // world
```

- By not specifying a length to **mid**, the rest of the string is returned:

Use `r = s.mid(6); // world`

```
r = s.replace("world", "universe"); // Hello universe
```

Empty and null strings

- A string can be null, i.e. contain nothing

```
QString s();  
s.isNull(); // true  
s.isEmpty(); // true
```

- It can also be empty i.e. contain an empty string

```
QString s("");  
s.isNull(); // false  
s.isEmpty(); // true
```

7

Handling Binary Data

QByteArray provides a container for binary data

Advantages over using a raw `char *`:

1. You do not have to handle memory allocation/deallocation
 - Rich API for manipulation
 - Tight integration with other Qt APIs

API very similar to `QString`, but instead of 16-bit (unicode) data the `QByteArray` stores 8-bit (ascii or raw bytes).

8

Qt Collection Classes

- `QList` is one of many of Qt's container template classes
- `QLinkedList` – quick insert in the middle, access through iterators
- `QVector` – uses continuous memory, slow inserts
- `QStack` – LIFO, last in – first out
- `QQueue` – FIFO, first in – first out
- `QSet` – unique values
- `QMap` – associative array
- `QHash` – associative array, faster than `QMap`, but requires hash
- `QMultiMap` – associative array with multiple values per key
- `QMultiHash` – associative array with multiple values per key

Specialization for popular combinations of value and containers exist e.g.: `QStringList` is derived from `QList<QString>`

QList

The `QList` class is a template class that provides lists.

- Use the stream operator to populate the list

```
QList<QString> list;  
list << "one" << "two" << "three";  
// list: ["one", "two", "three"]
```

- Or use the `QStringList` which inherits `QList<QString>`

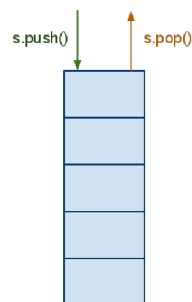
QVector

- Similar functionality to `QList`

Which to use? Depends on the type of items stored. In general `QList` is recommended for most use cases.

QStack

- A stack is a LIFO container
last in, first out
- Items are pushed onto the stack
- Items are popped off the stack
- The top item can be seen using `top()`

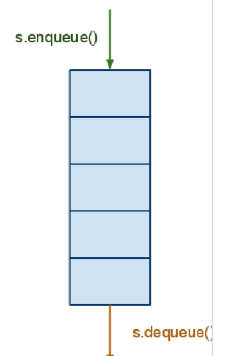


```
QStack<int> stack;  
stack.push(1);  
stack.push(2);  
stack.push(3);  
QDebug("Top: %d", stack.top()); // 3  
QDebug("Pop: %d", stack.pop()); // 3  
QDebug("Pop: %d", stack.pop()); // 2  
QDebug("Pop: %d", stack.pop()); // 1  
QDebug("isEmpty? %s", stack.isEmpty()? "yes": "no");
```

11

QQueue

- A queue is a FIFO container
first in, first out
- Items are enqueued into the queue
- Items are dequeued from the queue
- The first item can be seen using `head()`



```
QQueue<int> queue;  
queue.enqueue(1);  
queue.enqueue(2);  
queue.enqueue(3);  
QDebug("Head: %d", queue.head()); // 1  
QDebug("Pop: %d", queue.dequeue()); // 1  
QDebug("Pop: %d", queue.dequeue()); // 2  
QDebug("Pop: %d", queue.dequeue()); // 3  
QDebug("isEmpty? %s", queue.isEmpty()? "yes": "no");
```

12

QSet

- A set contains values, but only one instance of each value.
- It is possible to determine if a value is a part of the set or not

```
QSet<int> primes;
primes << 2 << 3 << 5 << 7 << 11 << 13;
for(int i=1; i<=10; ++i)
    qDebug("%d is %sprime", i, primes.contains(i)?"": "not ");
```

- You can also iterate over a set, to see all values

```
foreach(int prime, primes)
    qDebug("Prime: %d", prime);
```

- It is possible to convert a QList to a QSet

```
QList<int> list;
list << 1 << 1 << 2 << 2 << 2 << 3 << 3 << 5;
QSet<int> set = list.toSet();
qDebug() << list; // (1, 1, 2, 2, 2, 3, 3, 5)
qDebug() << set; // (1, 2, 3, 5)
```

13

Key/value collections aka. QMap & QHash

- The QMap and QHash classes let you create associative arrays

```
QMap<QString, int> map;
map["Helsinki"] = 1310755;
map["Oslo"] = 1403266;
map["Copenhagen"] = 1892233;
map["Stockholm"] = 2011047;

foreach(const QString &key, map.keys())
    qDebug("%s", qPrintable(key));

if(map.contains("Oslo"))
{
    qDebug("Oslo: %d",
        map.value("Oslo"));
}
```

```
QHash<QString, int> hash;
hash["Helsinki"] = 1310755;
hash["Oslo"] = 1403266;
hash["Copenhagen"] = 1892233;
hash["Stockholm"] = 2011047;

foreach(const QString &key, hash.keys())
    qDebug("%s", qPrintable(key));
```

- QHash faster lookup
- QHash requires specification of hash function for user-defined types

14

QVariant

- QVariant is an implementation of a *Single Value, Multiple Types* class.
- What does this mean?
 - A QVariant allows us to store and retrieve a number of different arguments in a generic value.
- Maybe we need an example:
- You will encounter it in an increasing number of places in the Qt APIs

```
QList<QVariant> m_list;
m_list << QVariant(QString("hello world"));
m_list << QVariant(5);

if(m_list[0].type() == QVariant::String)
{
    qDebug() << "I'm a string: " << m_list[0].toString();
}

if(m_list[1].type() == QVariant::Int)
{
    qDebug() << "I'm a int " << m_list[1].value<int>();
}
```

15

Files and file systems

- Referring to files and directories in a cross platform manner poses a number of problems
 - Does the system have drives, or just a root?
 - Are paths separated by "/" or "\"?
 - Where does the system store temporary files?
 - Where does the user store documents?
 - Where is the application stored?

16

Paths

- Use the QDir class to handle paths

```
QDir d = QDir("C:\\");
```

- Use the static methods to initialize

```
QDir d = QDir::root(); // "C:\\" on windows and "/" on Linux

QDir::current() // Current directory
QDir::home() // Home directory
QDir::temp() // Temporary directory

// Executable directory path
QDir(QApplication::applicationDirPath())
```

17

Finding directory contents

- The entryInfoList returns a list of information for the directory contents

```
QDir d = QDir(QApplication::applicationDirPath());
QFileInfoList infos = d.entryInfoList();
foreach(const QFileInfo &info, infos)
    qDebug() << info.fileName();
```

 this function supports filters

- You can add filters to skip files or directories

```
QDir::Dirs           ← Dirs, files or
QDir::Files          ← symbolic links?
QDir::NoSymLinks
```

```
QDir::Readable      ← Which files?
QDir::Writable
QDir::Executable
```

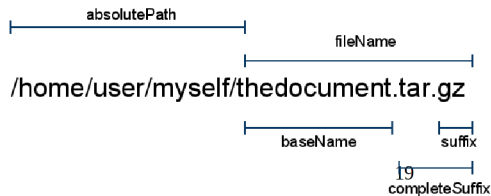
```
QDir::Hidden        ← Hidden files?
QDir::System        ← System files?
```

18

QFileInfo

Each QFileInfo object has a number of methods

- `absoluteFilePath` – full path to item
- `isDir` / `isFile` – type of item
- `isWritable` / `isReadable` / `isExecutable` – **permission for file**



Opening and reading files

- The `QFile` is used to access files

```
QFile f("/home/john/input.txt");
if (!f.open(QIODevice::ReadOnly))
    qFatal("Could not open file");

QByteArray data = f.readAll();
processData(data);

f.close();

while(!f.atEnd())
{
    QByteArray data = f.read(160);
    processData(data);
}
```

Read chunks or read everything

20

Writing to files

- When writing files, open it in **WriteOnly** mode and use the write method to add data to the file

```
QFile f("/home/john/input.txt");
if (!f.open(QIODevice::WriteOnly))
    qFatal("Could not open file");

QByteArray data = createData();
f.write(data);

f.close();
```

- Files can also be opened in **ReadWrite** mode
- The flags **Append** or **Truncate** can be used in combination with write-enabled modes to either append data to the file or to truncate it (i.e. clear the file from its previous contents)

```
if (!f.open(QIODevice::WriteOnly|QIODevice::Append)) 21
```

The QIODevice

- `QFile` is derived from `QIODevice`
- The constructors `QTextStream` and `QDataStream` take a `QIODevice` pointer as an argument, not a `QFile` pointer
- There are `QIODevice` implementations
 - `QBuffer` – for reading and writing to memory buffers
 - `QextSerialPort` – for serial (RS232) communication (3rd party)
 - `QAbstractSocket` – the base of TCP, SSL and UDP socket classes
 - `QProcess` – for reading and writing to processes' standard input and output

22

Timer Events

`int QObject::startTimer(int interval)`

- Starts the timer, which will continuously send timer events at specified intervals (ms)
- Returns the timer id
- Stops when `QObject::killTimer(id)` is called

```
class MyObject : public QObject
{
    Q_OBJECT
public:
    MyObject(QObject *parent = 0);
protected:
    void timerEvent(QTimerEvent *event);
};

MyObject::MyObject(QObject *parent)
: QObject(parent)
{
    startTimer(50); // 50-millisecond timer
    startTimer(1000); // 1-second timer
    startTimer(60000); // 1-minute timer
}

void MyObject::timerEvent(QTimerEvent *event)
{
    qDebug() << "Timer ID:" << event->timerId();
}
```

23

Timer Events

Using `QTimer` we can have repetitive and/or single-shot timers.

- Emits the `timeout()` signal when the timer expires

```
QTimer *timer = new QTimer(this);
connect(timer, SIGNAL(timeout()), this, SLOT(update()));
timer->start(1000);
```

24