

Luca Ottaviano

Tech lead

# Qt Framework

ISIS Gobetti Volta

GENNAIO 2021

develer

## COS'È QT QUICK

Qt Quick è un sistema per sviluppare UI fluide

Permette di interagire con il sistema grazie all'integrazione con C++

È di proprietà di The Qt Company ed è rilasciato con doppia licenza open source e commerciale

## PERCHÈ QT QUICK

È un framework “batterie incluse”

È uno standard di fatto in ambito Linux embedded industriale

È una soluzione cross-platform efficace anche su desktop e microcontrollori

Obiettivo per le mie lezioni

## Cruscotto



## PROGRAMMA

- Introduzione al linguaggio QML
- Montaggio della UI a partire da asset grafici
- Cenni di Javascript
- Integrazione con C++

## QT QUICK: OVERVIEW

Insieme di tecnologie per lo sviluppo rapido di applicazioni

- QML: linguaggio dichiarativo per scrivere componenti UI
- Runtime: esegue il codice QML e fornisce il motore Javascript per eseguire il codice
- C++: backend per l'integrazione con la macchina

## QML

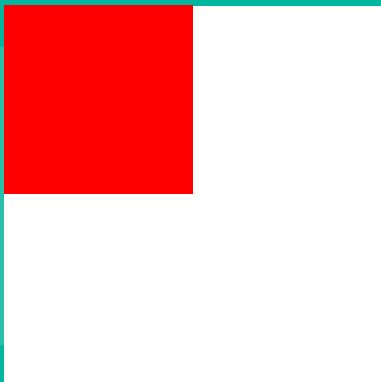
- Descrive i componenti che fanno parte della UI
- Gestisce l'input utente (tramite mouse, touchscreen o tastiera)
- Esegue in una sandbox

## C++

- Comunica con il QML tramite oggetti “promossi”
- Interagisce con la macchina
- Codice unrestricted



## QML: CONCETTI DI BASE



```
import QtQuick 2.11

Rectangle {
    id: root
    color: "white"
    width: 200
    height: 200
    Rectangle {
        color: "red"
        width: root.width / 2
        height: root.height / 2
    }
}
```

## QML: CONCETTI DI BASE

### Blocco degli import

```
import QtQuick 2.11
```

### Istanziazione degli oggetti

```
Rectangle {  
    id: root  
    color: "white"  
    width: 200  
    height: 200  
    Rectangle {  
        color: "red"  
        width: root.width / 2  
        height: root.height / 2  
    }  
}
```

## QML: CONCETTI DI BASE

Questo codice istanzia due oggetti di tipo Rectangle

I due oggetti sono in una scena

La posizione di istanziazione definisce anche la posizione in gerarchia

```
import QtQuick 2.11
```

```
Rectangle {
```

```
    id: root
```

```
    color: "white"
```

```
    width: 200
```

```
    height: 200
```

```
    Rectangle {
```

```
        color: "red"
```

```
        width: root.width / 2
```

```
        height: root.height / 2
```

```
    }
```

```
}
```

## QML: CONCETTI DI BASE

Attenzione, questi non sono comandi di disegno, ma istanziazioni

Un file QML viene interpretato per definire la scena

La scena viene poi disegnata quando serve

```
import QtQuick 2.11

Rectangle {
    id: root
    color: "white"
    width: 200
    height: 200
    Rectangle {
        color: "red"
        width: root.width / 2
        height: root.height / 2
    }
}
```

## QML: CONCETTI DI BASE

Ogni oggetto può avere  
al più un parent

L'oggetto senza parent è  
l'oggetto root

1 file QML = 1 oggetto  
root

```
import QtQuick 2.11

Rectangle {
    id: root
    color: "white"
    width: 200
    height: 200
    Rectangle {
        color: "red"
        width: root.width / 2
        height: root.height / 2
    }
}
```

## QML: CONCETTI DI BASE

Identificatore univoco  
oggetto QML

Inizia per lettera  
lowercase o \_

È univoco all'interno del  
file QML

Serve agli oggetti per  
riferirsi tra loro

```
import QtQuick 2.11

Rectangle {
    id: root
    color: "white"
    width: 200
    height: 200
    Rectangle {
        color: "red"
        width: root.width / 2
        height: root.height / 2
    }
}
```

## QML: CONCETTI DI BASE

Ogni tipo QML espone  
delle property

Le property:

- hanno un tipo
- controllano aspetto e comportamento dell'oggetto

```
import QtQuick 2.11

Rectangle {
    id: root
    color: "white"
    width: 200
    height: 200
    Rectangle {
        color: "red"
        width: root.width / 2
        height: root.height / 2
    }
}
```

## QML: CONCETTI DI BASE

Le property possono essere in binding tra loro

Il binding descrive la relazione tra la property e un'espressione

Ogni volta che l'espressione a destra cambia, la property viene rivalutata

```
import QtQuick 2.11
```

```
Rectangle {
```

```
    id: root
```

```
    color: "white"
```

```
    width: 200
```

```
    height: 200
```

```
    Rectangle {
```

```
        color: "red"
```

```
        width: root.width / 2
```

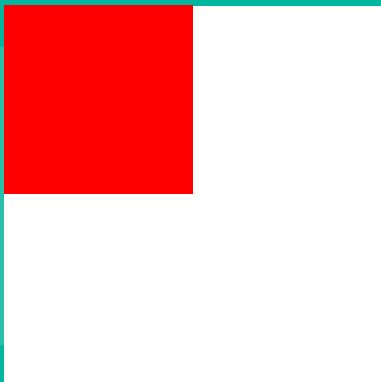
```
        height: root.height / 2
```

```
    }
```

```
}
```



## QML: CONCETTI DI BASE



```
import QtQuick 2.11

Rectangle {
    id: root
    color: "white"
    width: 200
    height: 200
    Rectangle {
        color: "red"
        width: root.width / 2
        height: root.height / 2
    }
}
```

## TIPI DELLE PROPERTY

### Forniti da Javascript

- int, bool, real, double
- string, url, list, var

### Forniti da Qt Quick

- color, font, date, time, point, size, rect
- matrix4x4, vector2d, vector3d

## TIPI DI OGGETTI QML

### Tipi visuali

- Item (ha una superficie ma non un aspetto)
- Rectangle, Text, Image
- Row, Column (non hanno aspetto proprio)

### Tipi di input

- MouseArea

Tramite questi tipi si può creare circa il 50% di tutte le UI QML :)

In generale, se non specificate, le property hanno valore nullo per il rispettivo tipo:

- 0 per le property numeriche
- "" per le property stringa o url
- undefined per le property var

La documentazione di tutte le property si trova qua: [Qt 5.15](#)

Esempio: Text{}

È un Item che renderizza del testo

La property principale è "text", di tipo stringa

Di solito non è necessario specificare la dimensione (width e height) perché è la dimensione del testo renderizzato con il font selezionato

```
Text {  
    text: "Hello world!"  
}
```

```
Text {  
    text: "Hello" + " " + "world!"  
}
```

```
Text {  
    id: text1  
    text: "Hello world!"  
}  
Text {  
    text: text1.text  
    // text: text1.width // ERRORE!  
}
```

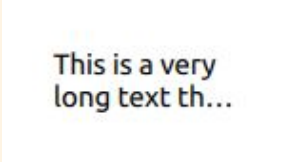
Esempio: Text{}

Per “andare a capo” devo dare una dimensione all’elemento Text

width + wrapMode: mandano a capo il testo

height + elide: in aggiunta alle due sopra troncano il testo se necessario

```
Text {  
    text: "This is a very long text that  
    overflows"  
  
    width: 100  
    wrapMode: Text.Wrap  
  
    height: 50  
    elide: Text.ElideRight  
}
```



This is a very  
long text th...

## POSIZIONAMENTO

Per creare maschere complesse si assemblano insieme oggetti di tipo più semplice

Ci sono tre modi principali di posizionare oggetti a video:

- Contenitori (oggetti Row, Column)
- Ancoraggi
- ...anche posizionamento assoluto (x, y)

## CONTENITORI



```
import QtQuick 2.11

Row {
    spacing: 10
    Image {
        source: "images/D.png"
    }
    Image {
        source: "images/N.png"
    }
    Image {
        source: "images/P.png"
    }
}
```

## CONTENITORI

Controllano posizione degli elementi figli

Nessuna autorità su dimensioni

I contenitori sono oggetti “logici”, non hanno aspetto visuale

Per “funzionare” è necessario che gli oggetti abbiano delle dimensioni specificate

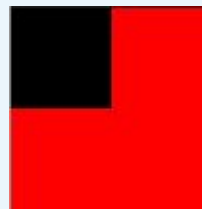
```
import QtQuick 2.11

Row {
    spacing: 10
    Image {
        source: "images/D.png"
    }
    Image {
        source: "images/N.png"
    }
    Image {
        source: "images/P.png"
    }
}
```



## QUIZ

```
Rectangle {  
    width: 100; height: 100; color: "red"  
  
    Rectangle {  
        width: 50; height: 50; color: "black"  
    }  
}
```



```
Rectangle {  
    width: 100; height: 100; color: "red"  
  
    Rectangle {  
        width: 50; height: 50; color: "black"  
        x: 50  
        y: 50  
    }  
}
```

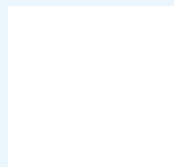


## QUIZ

```
Column {  
    spacing: 10  
    Text {  
        text: "32 km/h"  
    }  
    Text {  
        text: "48" + " %"  
    }  
}
```

32 km/h  
48 %

```
Row {  
    Rectangle {  
        color: "red"  
    }  
    Rectangle {  
        color: "green"  
    }  
    Rectangle {  
        color: "blue"  
    }  
}
```



## QUIZ

```
Row {  
    Rectangle {  
        width: 10; height: 10  
        color: "red"  
    }  
    Rectangle {  
        width: 10; height: 10  
        color: "green"  
    }  
    Rectangle {  
        width: 10; height: 10  
        color: "blue"  
    }  
}
```



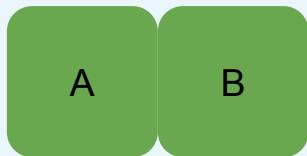
## ANCORAGGI

Sono il secondo metodo di posizionamento

Controllano la posizione e la dimensione degli elementi

Sono basati sul concetto di **property binding**

## ANCORAGGI

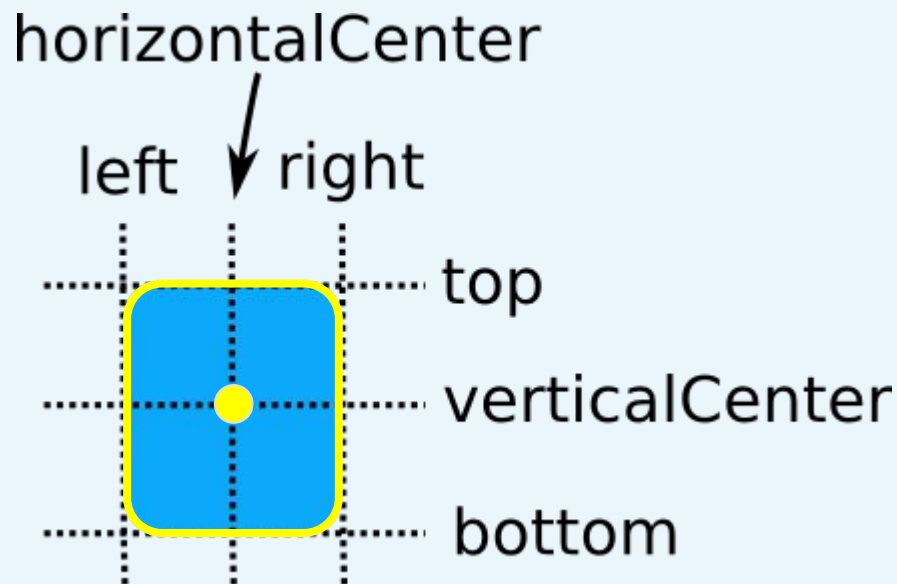


```
Item {  
    id: b  
    anchors.left: a.right  
}
```



```
Item {  
    id: small  
    anchors.centerIn: big  
}
```

## ANCORAGGI



Margini: `topMargin`, `leftMargin`

`fill` e `centerIn`: scorciatoie

## ANCORAGGI



0 %

```
import QtQuick 2.11

Item {
    id: batt
    width: 200
    height: 200
    Image {
        id: batt_img
        anchors.top: batt.top
        anchors.horizontalCenter: batt.horizontalCenter
        source: "assets/images/batteria_vuota.png"
    }
    Text {
        anchors.top: batt_img.bottom
        anchors.topMargin: 10
        anchors.horizontalCenter: batt.horizontalCenter
        text: "0 %"
    }
}
```

## ANCORAGGI

Le linee di ancoraggio vanno in binding con le linee di ancoraggio.

Linee orizzontali vanno con linee orizzontali; linee verticali vanno con linee verticali

```
import QtQuick 2.11

Item {
    id: batt
    width: 200
    height: 200
    Image {
        id: batt_img
        anchors.top: batt.top
        anchors.horizontalCenter: batt.horizontalCenter
        source: "assets/images/batteria_vuota.png"
    }
    Text {
        anchors.top: batt_img.bottom
        anchors.topMargin: 10
        anchors.horizontalCenter: batt.horizontalCenter
        text: "0 %"
    }
}
```



## ANCORAGGI

I margini sono attivi solo se è definita la rispettiva linea di ancoraggio

```
import QtQuick 2.11

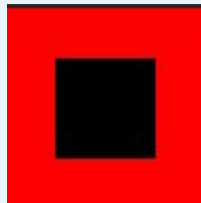
Item {
    id: batt
    width: 200
    height: 200
    Image {
        id: batt_img
        anchors.top: batt.top
        anchors.horizontalCenter: batt.horizontalCenter
        source: "assets/images/batteria_vuota.png"
    }
    Text {
        anchors.top: batt_img.bottom
        anchors.topMargin: 10
        anchors.horizontalCenter: batt.horizontalCenter
        text: "0 %"
    }
}
```

## QUIZ

```
Rectangle {  
    width: 100; height: 100; color: "red"  
  
    Rectangle {  
        width: parent.width / 2; height: parent.height / 2; color: "black"  
        anchors.right: parent.right  
        anchors.bottom: parent.bottom  
    }  
}
```



```
Rectangle {  
    width: 100; height: 100; color: "red"  
  
    Rectangle {  
        width: parent.width / 2; height: parent.height / 2; color:  
"black"  
        anchors.centerIn: parent  
    }  
}
```



## ESEMPIO

```
Rectangle {  
    width: 100  
    height: 100  
    color: "red"  
  
    Rectangle {  
        anchors.fill: parent  
        anchors.leftMargin: 10  
        anchors.rightMargin: 10  
        color: "black"  
    }  
}
```

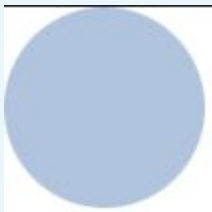


## ESEMPIO

```
Rectangle {  
    width: 100  
    height: 100  
    radius: 20  
    color: "red"  
  
    Rectangle {  
        anchors.centerIn: parent  
        width: 50  
        height: 50  
        anchors.verticalCenterOffset : 25  
        color: "black"  
    }  
}
```



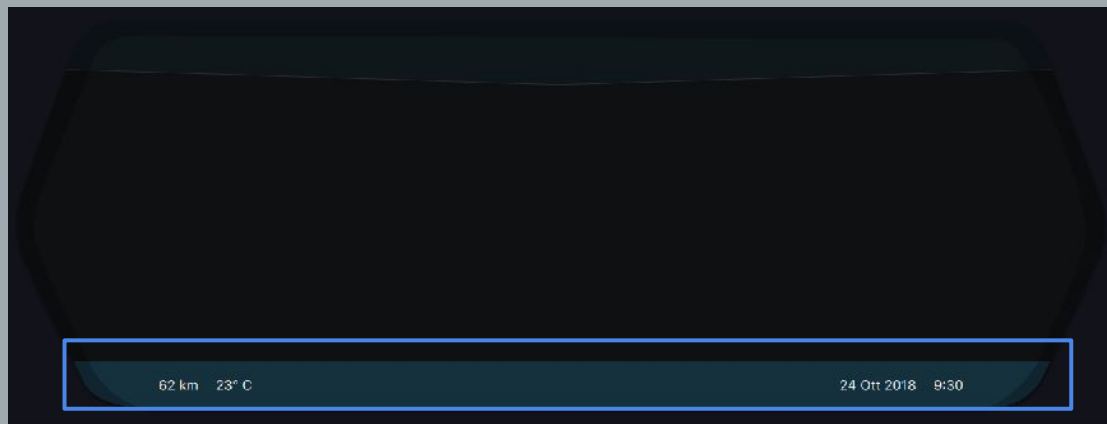
## QUIZ: COME SI FA UN CERCHIO?



```
Rectangle {  
    width: 100  
    height: 100  
    radius: width / 2  
    color: "lightsteelblue"  
}
```

## CRUSCOTTO

Realizziamo insieme il primo componente



## IMMAGINI



0 %

L'elemento `Image{}` carica un'immagine alla sua dimensione "naturale" → `width` e `height` sono già impostate

Il percorso dell'immagine va specificato nella property **source**

L'immagine è un `Item`, quindi la possiamo posizionare come tutti gli altri elementi

```
Image {  
    id: batt_img  
    anchors.top: batt.top  
    anchors.horizontalCenter: batt.horizontalCenter  
    source: "assets/images/batteria_vuota.png"  
}
```

## ESERCIZIO

Realizzate la ghiera della velocità, usando queste immagini:

- contagiri\_back
- contagiri\_top
- ghiera+numeri
- centrale





Javascript è un linguaggio di programmazione comunemente usato per la programmazione lato client in pagine web [Wikipedia]

Ha una sintassi molto simile a Java e C++

In QML è usato in tutti i binding e i signal handlers

Esempio:

```
width: parent.width / 2
```

Il binding è un'espressione Javascript

### Pico-guida alla sintassi:

- Per quello che ci interessa è più o meno come C e C++
- È un linguaggio debolmente tipizzato, cioè ci sono molte conversioni automatiche tra tipi
  - Es. una stringa "42" può essere assegnata ad un numero senza errori
- Cicli for, while come in C
- Le variabili si dichiarano con "let"
  - for (let i = 0; i < 10; i++)
- In fondo ad una riga è opzionale il ";"
- **ATTENZIONE!** Non scrivere mai  
return <a capo> 42;

### Pico-guida alla sintassi (cont):

- L'uguaglianza tra stringhe e numeri avviene per valore
- L'uguaglianza tra array e dictionaries avviene per puntatore (NON elemento per elemento)
- Il confronto si esprime con "==="
- I commenti si fanno con `//` ad inizio riga
- Debug: potete stampare con `console.log("qualcosa")`

La libreria del linguaggio fornisce le strutture dati di base con cui lavorare:

- stringhe
- array
- dizionari
- data e ora

### [Stringhe su MDN](#)

Inoltre Qt fornisce alcuni metodi di utilità per formattare un numero in una stringa

#### API Javascript:

- `"Hello" + "World" → "HelloWorld"`
- `"Hello".length === 5`
- `"7".padStart(3, "0") → "007"`

#### API Qt

- `"There are %1 files inside %2 directory".arg(4).arg("Downloads")`

[Array su MDN](#)

API Javascript:

- `let fruits = ['Apple', 'Banana']` //costruzione
- `fruits.length === 2`
- `fruits[0] === 'Apple'`
- `fruits[fruits.length - 1] === 'Banana'`
- `fruits.push('Mango')` // Aggiunge un elemento

[Data e ora su MDN](#)

[Qt.formatDateTime](#)

API Javascript:

- `let now = new Date();` // data e ora corrente
- `now.getDate()` // giorno del mese 1-31
- `now.getMonth()` // mese dell'anno 0-11
- `now.getFullYear()` // anno in 4 cifre es. 1999

API Qt:

- `Qt.formatDate(date, format)` → stringa
- `Qt.formatTime(date, format)` → stringa
- `Qt.formatDateTime(date, format)` → stringa

Il parametro **date** è l'oggetto data, il parametro **format** è il formato in cui vogliamo la stringa

`Qt.formatDateTime(new Date(), "dd-MM-yyyy")` → "05-01-2021"

`Qt.formatDateTime(new Date(), "yyyy MM:dd")` → "2021 01:05"

## TIMER

Come si esegue un'azione dopo un po' di tempo?

In Javascript abbiamo "setInterval()", ma QML si usa l'elemento Timer{}

```
Timer {  
    running: true  
    interval: 2000  
    repeat: true  
    onTriggered: {  
        console.log(Qt.formatDateTime(new Date(),  
"dd-MM-yyyy hh:mm:ss"))  
    }  
}
```

qml: 05-01-2021 18:51:06

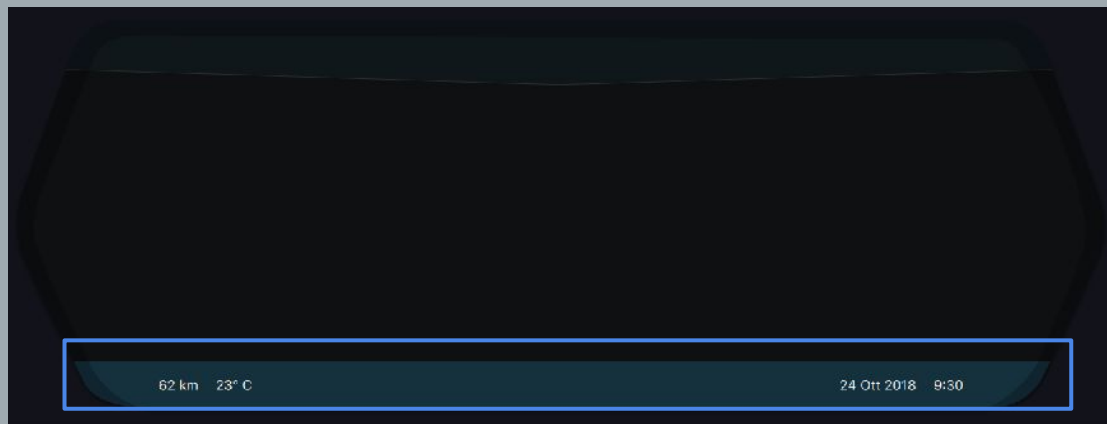
qml: 05-01-2021 18:51:08

qml: 05-01-2021 18:51:10



## ESERCIZIO

Facciamo in modo che la data si aggiorni in modo automatico



## QT E C++

Il C++ fornisce classi, derivazione, costruttori e operatori, tuttavia le funzionalità del C++ sono relativamente di basso livello.

Non sono presenti funzionalità per la gestione del lifetime degli oggetti, per comunicare fra di essi o fra processi separati, per fare introspezione, ecc..

Qt è un framework scritto in C++ che nasce proprio con l'obiettivo di elevare il linguaggio ad un livello più alto per rendere più semplice la programmazione.

## FUNZIONALITÀ SPECIFICHE DI QT

- Meccanismi di message passing (signal / slot)
- Proprietà
- Timer e callback
- ...e tanti altri

## META OBJECT COMPILER (MOC)

Tutte le funzionalità aggiuntive sono fornite dal MOC, che a partire da alcune annotazioni nel sorgente in C++ genera il codice necessario

```
#include <QObject>

class ControlUnit : public QObject {
    Q_OBJECT
    // ...
}
```

## LOOP DEGLI EVENTI

Tutte le applicazioni Qt si appoggiano sul loop degli eventi

È un loop senza fine all'interno del quale vengono processati gli eventi del sistema operativo e quelli interni all'applicazione

Tutta l'applicazione deve essere scritta in modo asincrono, le funzioni utente sono chiamate in reazione ad eventi (callback)

- Anche chiamato “Principio di Hollywood” (“Non chiamarci, ti chiameremo noi”)

## SIGNALS E SLOTS

Ogni QObject dichiara un insieme di **signals** e **slots**

- **signals**: notifiche che è accaduto un evento che l'oggetto vuole comunicare
- **slots**: azioni che un oggetto intraprende quando arriva un segnale

Importante! Un oggetto può emettere un segnale quando vuole, anche se non è collegato a nessuno.

## SIGNALS E SLOTS

Uno slot è un'azione che può essere connessa ad un segnale

Gli slot possono essere anche utilizzati direttamente, come funzioni o metodi standard di un oggetto

I segnali possono avere argomenti, e gli slot connessi ad essi devono avere argomenti di tipo compatibile con quelli del segnale

## CONNECT

Si usa il metodo `QObject::connect()` per connettere un segnale ad uno slot

Parametri:

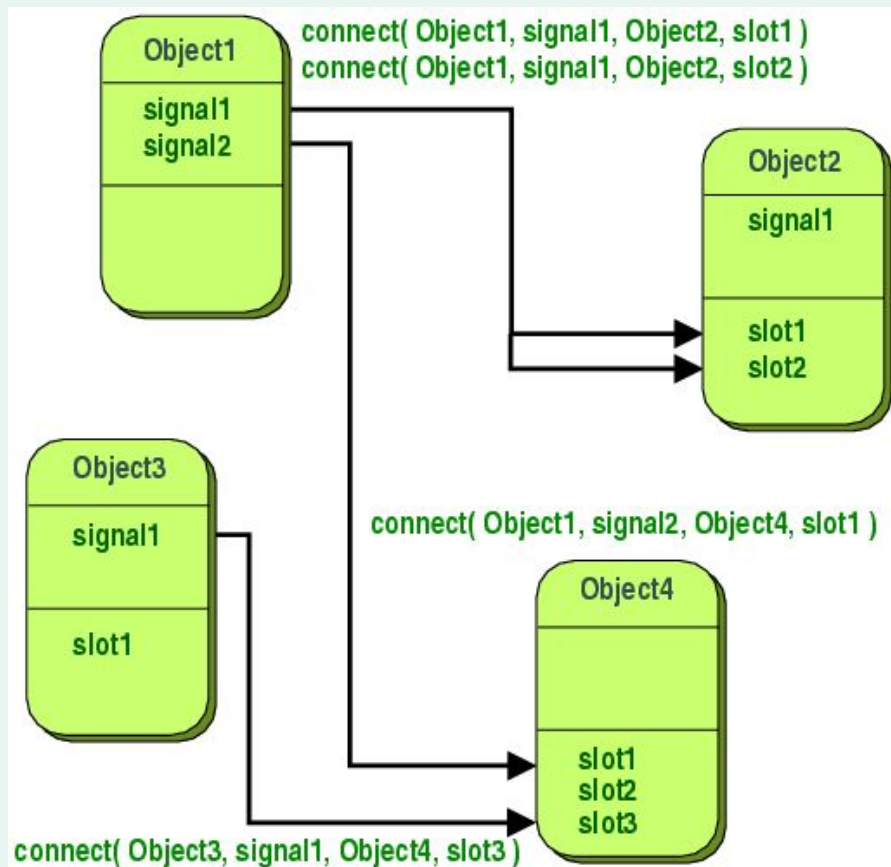
1. puntatore dell'oggetto che emette il segnale
2. nome del segnale
3. puntatore dell'oggetto che riceve il segnale
4. nome dello slot

```
connect(manager, SIGNAL(stateChanged(int)), this, SLOT(updateUi(int)));
```

```
connect(manager, &ManagerClass::stateChanged, this, &MyClass::updateUi);
```



## SIGNALS E SLOTS



## ESEMPIO

```
class MyClass : public QObject {  
    Q_OBJECT  
public:  
    void doThings() {  
        //...  
        emit stateChanged(2);  
        //...  
    }  
  
signals:  
    void stateChanged(int);  
};
```

## API DI RILIEVO

### Classi:

- [QString](#): rappresenta delle stringhe unicode, con molti metodi di utilità rispetto alle stringhe C++
- [QDate](#), `QTime`, `QDateTime`: rappresentano rispettivamente una data, un'ora e data e ora combinate insieme
- [QTimer](#): classe che emette segnali ogni X millisecondi

### Metodi:

- `QDebug()`: equivalente di `std::cout`, stampa su console e interpreta i tipi Qt

```
QDebug() << QString("Hello") << QDate::currentDate();
```

## ESERCIZIO

Creare una classe Controller che ogni secondo stampa la data e ora correnti

```
#include <QObject>
#include <QTimer>

class Controller : public QObject {
    Q_OBJECT

public:
    Controller();

public slots:
    void printDateTime();

private:
    QTimer *m_timer{nullptr};
};
```

## ESERCIZIO

### Passi:

- Creare controller.cpp e controller.h
- Nel file .h inserire il codice a destra ->
- Nel costruttore di Controller, creare il QTimer e assegnarlo a m\_timer

m\_timer = new QTimer(this);

```
#include <QObject>
#include <QTimer>

class Controller : public QObject {
    Q_OBJECT

public:
    Controller();

public slots:
    void printDateTime();

private:
    QTimer *m_timer{nullptr};
};
```

## ESERCIZIO

### Passi (cont):

- Impostare l'intervallo di timeout con `setInterval()`
- Impostare auto repeat a true con `setSingleShot()`
- connettere il signal `timeout()` di `m_timer` con lo slot `printDateTime()`

```
#include <QObject>
#include <QTimer>

class Controller : public QObject {
    Q_OBJECT

public:
    Controller();

public slots:
    void printDateTime();

private:
    QTimer *m_timer{nullptr};
};
```

## ESERCIZIO

### Passi (cont):

- Implementare la funzione `printDateTime()`
- Usare `qDebug()` e `QDateTime::currentDateTime()`
- Creare un oggetto `controller` nel `main()`

```
#include <QObject>
#include <QTimer>

class Controller : public QObject {
    Q_OBJECT

public:
    Controller();

public slots:
    void printDateTime();

private:
    QTimer *m_timer{nullptr};
};
```

## PROPERTY

La definizione di una property, ad es. quelle degli oggetti QML, avviene usando la macro `Q_PROPERTY()`

Possiamo specificare:

- tipo della property
- nome
- READ: getter
- NOTIFY: segnale di cambiamento
- [opzionale] WRITE: setter



## PROPERTY

Esempio di property read  
write da QML

```
class MyClass : public QObject {  
    Q_OBJECT  
    Q_PROPERTY(QDate date READ getDate WRITE setDate NOTIFY  
dateChanged)
```

```
public:  
    QDate getDate() const { return m_date; }  
    void setDate(const QDate &new_date) {  
        if (m_date == new_date)  
            return;  
        m_date = new_date;  
        emit dateChanged(m_date);  
    }
```

```
signals:  
    void dateChanged(QDate);
```

```
private:  
    QDate m_date;  
};
```

## PROPERTY

Esempio di property  
read-only da QML

### IMPORTANTE!

Quando `m_counter` cambia,  
dobbiamo emettere il  
segnale `counterChanged()`

```
class MyClass : public QObject {  
    Q_OBJECT  
    Q_PROPERTY(int counter READ getCounter NOTIFY counterChanged)  
public:  
    int getCounter() const { return m_counter; }  
  
signals:  
    void counterChanged(int);  
  
private:  
    int m_counter{0};  
};
```

## ESPORTAZIONE DI UN OGGETTO AL QML

Per rendere visibile un oggetto C++ al QML si usa il metodo `setContextProperty()`, con i seguenti parametri:

1. nome (id) che vogliamo usare in QML
2. puntatore all'oggetto

```
// In C++  
MyClass myclass;  
engine.rootContext()->setContextProperty("myclass", &myclass);
```

```
// In QML  
text: myclass.date  
text: myclass.counter
```

## ESERCIZIO

Aggiungere una property `currentDateTime` alla classe `Controller`, esportarla al QML e usare quella property per aggiornare la data e ora sul cruscotto

Passi? :)

## ESERCIZIO

- Aggiungere la property di tipo QDateTime (read-only) chiamata "currentDateTime"
- Aggiungere un membro privato di tipo QDateTime m\_currentDateTime
- Aggiungere il segnale currentDateTimeChanged()

## ESERCIZIO

- Aggiungere e implementare il metodo `getter` `"currentDateTime()"`
- Ogni secondo assegnare il nuovo valore a `m_currentDateTime` e emettere il segnale `currentDateTimeChanged()`
- Esportare l'oggetto `Controller` al QML

## ESERCIZIO

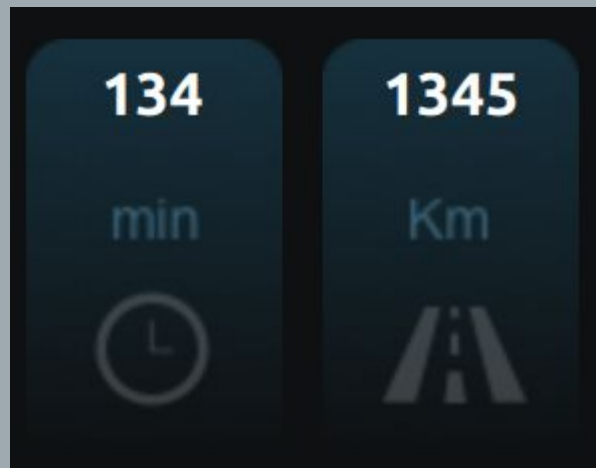
- Usare la property nel QML al posto di “new Date()” che avete nelle Qt.formatDateTime()
- Rimuovere eventuali Timer{} che avete nel QML

## ESERCIZIO FINALE

Aggiungere alla classe Controller una property “trip” di tipo double che indica i **km** percorsi.

La property viene incrementata di 10 **metri** ogni secondo

Usare la property in QML

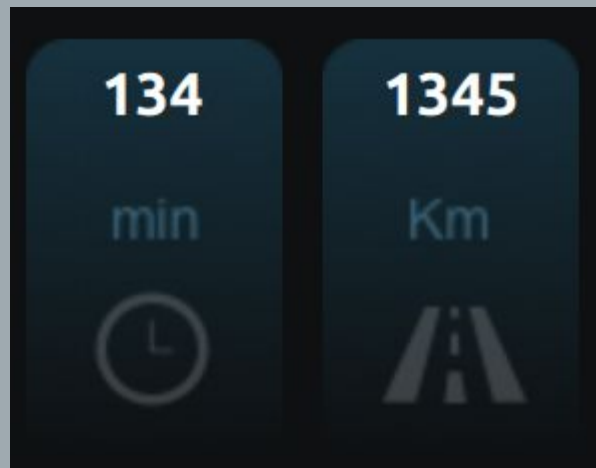




## ESERCIZIO FINALE

### Passi:

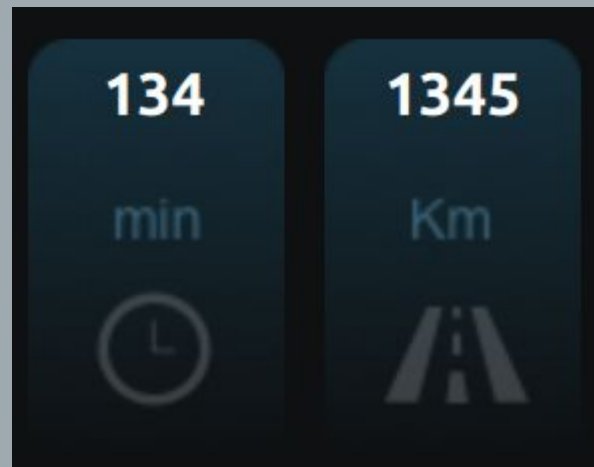
- Aggiungere la property read-only alla classe Controller
- Aggiungere il metodo getter, il segnale e il membro privato
- Ogni secondo incrementare il membro di **10 metri** (cioé, 0.01)
- Emettere il segnale



## ESERCIZIO FINALE

Passi (cont):

- In QML, usare “controller.trip” nel campo di testo opportuno



## COMPONENTI

I componenti servono per comporre insieme oggetti più semplici

Sono il modo principale con cui si riusa il codice in QML

## COMPONENTI

Vediamo come possiamo rendere riusabile il componente batteria



0 %

```
import QtQuick 2.11

Item {
    id: batt
    width: 200
    height: 200
    Image {
        id: batt_img
        anchors.top: batt.top
        anchors.horizontalCenter: batt.horizontalCenter
        source: "assets/images/batteria_vuota.png"
    }
    Text {
        anchors.top: batt_img.bottom
        anchors.topMargin: 10
        anchors.horizontalCenter: batt.horizontalCenter
        text: "0 %"
    }
}
```

## COMPONENTI

Aggiungiamo una property all'elemento root

Le property dell'oggetto root fanno da interfaccia per il componente

Salviamo il file con iniziale maiuscola (Battery.qml)

```
import QtQuick 2.11

Item {
    id: batt

    property int battery: 0
    width: 200
    height: 200
    Image {
        id: batt_img
        anchors.top: batt.top
        anchors.horizontalCenter: batt.horizontalCenter
        source: "assets/images/batteria_vuota.png"
    }
    Text {
        anchors.top: batt_img.bottom
        anchors.topMargin: 10
        anchors.horizontalCenter: batt.horizontalCenter
        text: batt.battery + "%"
    }
}
```

## COMPONENTI

```
Item {  
    id: mainWindow  
    Battery {  
        battery: control_unit.battery  
    }  
}
```

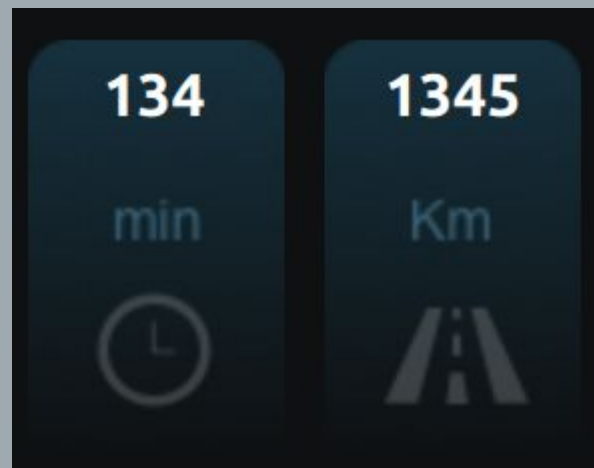
main.qml

```
import QtQuick 2.11  
  
Item {  
    id: batt  
  
    property int battery: 0  
    width: 200  
    height: 200  
    Image {  
        id: batt_img  
        anchors.top: batt.top  
        anchors.horizontalCenter: batt.horizontalCenter  
        source: "assets/images/batteria_vuota.png"  
    }  
    Text {  
        anchors.top: batt_img.bottom  
        anchors.topMargin: 10  
        anchors.horizontalCenter: batt.horizontalCenter  
        text: batt.battery + "%"   
    }  
}
```

Battery.qml

## ESERCIZIO

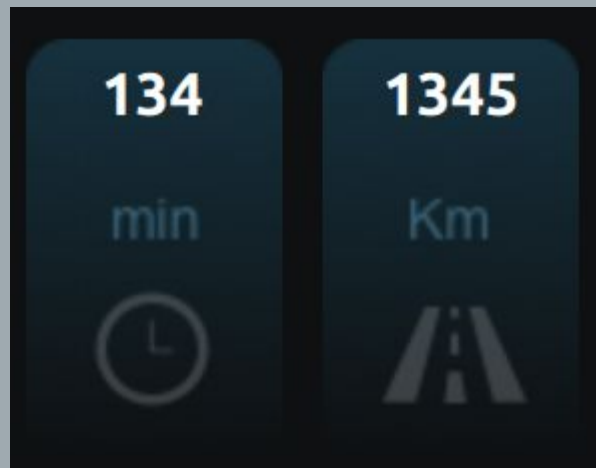
Creiamo un componente  
per gli indicatori a destra



## ESERCIZIO

### Passi:

- Creare un file Indicator.qml
- Mettere come elemento root una Image
- Aggiungere e posizionare un Text





## ESERCIZIO

Passi (cont):

- Aggiungere una property per personalizzare l'immagine

`property url icon:`

`"image.png"`

- Aggiungere una property per il testo

`property string text: "134"`

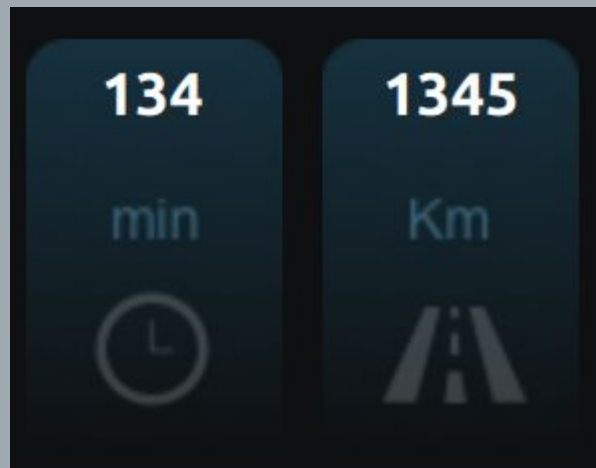
- Usare le property appena definite come "source" di Image e "text" di Text



## ESERCIZIO

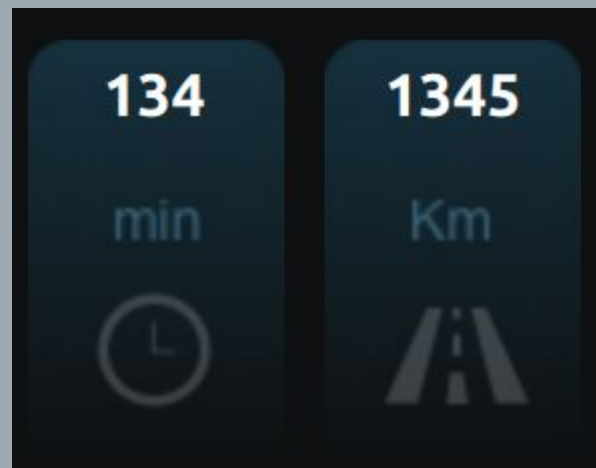
Passi (cont):

- Usare Indicator{} nel file main.qml, usando i valori di "icon" e "text" giusti



## ESERCIZIO BONUS

Aggiungere a Controller le property per gli elementi mancanti sulla destra



## CONTACTS

Luca Ottaviano

[luca.ottaviano@develer.com](mailto:luca.ottaviano@develer.com)

Twitter: @lucaotta



[www.develer.com](http://www.develer.com)