KAPITOLA 9

Implementace vlastních widgetů

V této kapitole:
- Implementace vlastního widgetu
- Události v Qt a jejich obsluha
- Časovače a časování v Qt
- Práce se soubory v Qt

V šesté kapitole jsme předvedli, jak lze v Qt vytvořit vlastní složený widget. Tento postup využijete v mnoha případech, kdy budete skládat jednoduché uživatelské rozhraní své aplikace, kde bude třeba pouze několik tlačítek, vstupních polí a nějaká prezentační či interaktivní plocha pro uživatelská data. Existují však i specifické případy, u kterých se základními komponentami nabízenými Qt Creatorem či samotným Qt nevystačíte. V těchto případech budete muset implementovat, či spíše reimplementovat, vlastní widget, splňující vaše nadstandardní požadavky. Na to, jak to provést, se zaměříme v této kapitole.

Při re-implementaci vlastního widgetu přijdete často do styku s tzv. událostmi, se kterými jste se už jemně setkali v předchozí kapitole. V této kapitole pak podrobněji probereme, co to události vlastně jsou a jak je lze jednoduše využívat a obsluhovat.

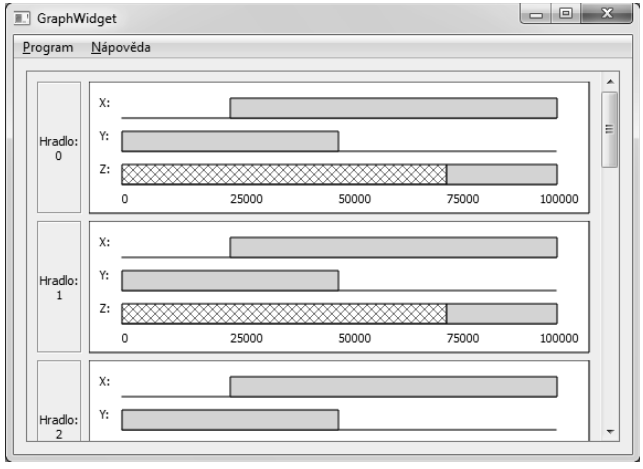
S událostmi nepřímo souvisí také časovače a jejich obsluha ve vašich aplikacích. Protože potřeba časování je v aplikacích poměrně častá při provádění jednoduchých animací či časově závislých operacích, podíváme se zde i na toto téma. Nakonec ještě popíšeme základní práci se soubory, která je díky Qt velmi snadná a navíc zajišťuje skutečně multiplatformní přístup.

Implementace vlastního widgetu

Vlastní widgety lze odvozovat přímo z bázové třídy všech widgetů – `QWidget` nebo z jakékoli jiné třídy specifického widgetu, například z třídy číselníku (`QSpinBox`). V případě, že odvodíte vlastní widget ze specifičtější třídy, jakou je třeba zmíněná třída `QSpinBox`, můžete stavět na jejím základě a pouze rozšiřovat či modifikovat její funkčnost. Toho dosáhnete reimplementací jejích virtuálních funkcí a doplněním vlastních a potenciálně nových atributů, což způsobí požadovanou změnu chování, či dokonce vzhledu daného prvku.

My se zde však zaměříme na tvorbu vlastního widgetu od začátku (anglicky *from scratch*), naším widgetem bude jednoduchý graf logického hradla se třemi horizontálními osami X, Y a Z

znázorňujícími průběh logické hodnoty na jednotlivých portech hradla v určitém čase. Tento widget odvodíme přímo ze třídy `QWidget` a podrobně popíšeme postup jeho tvorby.



Obrázek 9.1: Výsledná podoba ukázkové aplikace GraphWidget

Výpis 9.1: Vlastní widget – GraphWidget

```
#include <QtWidgets>

class GraphWidget: public QWidget
{ Q_OBJECT

  public: GraphWidget(QWidget *rodic = 0);

    enum RozmeryAPozice { Okraj = 30, OdsazeniGrafu = 30,
      VyskaGrafu = 120, PoziceGrafuX = 33, PoziceGrafuY = 63,
      PoziceGrafuZ = 93, PoziceYTextu = 110 };

    // gettery a settery (přístupové metody)

    // maximální čas simulace
    int maximalniCas() { return mMaximalniCas; }
    void nastavitMaximalniCas(int maximalniCas)
    { mMaximalniCas = maximalniCas; }

    // počáteční úroveň podgrafů X, Y a Z
    int pocatecniUrovenX() const { return mPocatecniUroven[0]; }
    void nastavitPocatecniUrovenX(int puX) { mPocatecniUroven[0] = puX; }

    // další gettery a settery atributů..

    QSize sizeHint() const { return QSize(300, 100); }
    QSize minimumSizeHint() const { return QSize(150, 70); }
```

```
  protected:
    void paintEvent(QPaintEvent *udalost); // vykreslovací událost

  private:
    int mMaximalniCas; // délka doby simulace

    int mPocatecniUroven[3]; // počáteční úroveň: 0 | 1 | X
    int mZmenenaUroven[3];  // úroveň po změně: 0 | 1 | X
    int mCasZmeny[3];      // čas změny úrovně
};
```

Nejprve se podíváme na hlavičkový soubor tvořeného widgetu, který tvoří úplný základ tohoto pracovního postupu. Po zpřístupnění nejrůznějších Qt deklarací inkludováním hlavičkového souboru `<QtWidgets>` zde nejprve veřejně odvodíme náš widget ze třídy `QWidget`. Na tomto místě by se však mohla nacházet jakákoli jiná Qt třída nebo i vlastní třída widgetu, na základě které bychom chtěli náš nový widget postavit. My však chceme kompletně vlastní prvek, proto využijeme jen bázovou funkcionalitu, kterou nabízí základní třída obecného widgetu.

Po tomto kroku je třeba, jako u jakýchkoli jiných tříd, deklarovat vlastní konstruktor, veřejné konstanty třídy, přístupové metody, atributy a ostatní datové členy. Na tom by nebylo nic zvláštního, protože widget je třída jako kterákoli jiná, pouze obsahuje několik specialit, které z ní tvoří zobrazitelný interaktivní prvek ve vaší aplikaci. První z těchto specialit je událost `paintEvent()`, která umožňuje vykreslení widgetu. Co jsou to události se podrobněji dozvíte níže, nyní stačí vědět, že v těle metody `paintEvent()` budeme náš widget s pomocí nám už známé třídy `QPainter` vykreslovat.

Další specialitou jsou pak metody `sizeHint()` a `minimumSizeHint()`, které se starají o vrácení preferovaných rozměrů našeho widgetu, když je umístěn do správce rozvržení. Jak jistě sami odhadnete, metoda `sizeHint()` vrací preferovanou velikost v podobě struktury `QSize`, zatímco metoda `minimumSizeHint()` vrací ve stejné podobě minimální požadovanou velikost tohoto prvku. My zde v obou metodách pro potřeby ilustrace vracíme pevně přednastavené hodnoty, vy zde v praktických aplikacích jistě obsáhnete potřebný výpočet požadovaných rozměrů dle vašich potřeb.

A to je v základu vše. Pokud bychom potřebovali dále ovlivňovat chování widgetu, mohli bychom reimplementovat jeho další metody a události (viz referenční dokumentace třídy `QWidget`) či si přidat ještě několik vlastních. Dle potřeby by samozřejmě šlo jednoduše doplnit i vlastní signály a sloty reagující na změny v grafu apod., viz makro `Q_OBJECT`. My se však budeme držet pro ilustraci toho nejjednoduššího scénáře a případná vylepšení zatím ponecháme na vás.

Tip: V případě, že odvodíte vlastní třídu přímo z bázové třídy všech Qt tříd – `QObject`, lze přidáním makra `Q_OBJECT` aktivovat podporu mechanismu Signálu & Slotu i u takovéto vlastní třídy. Tato vlastní třída pak samozřejmě nemusí mít nic společného s widgety ani s ostatními Qt třídami. Jedná se čistě o možnost využití mechanismu Signálu & Slotu a tím i pokročilejší modularizace vašeho projektu. Toto platí i v případě využití vícenásobné dědičnosti, v tomto případě je však nutno uvádět třídu `QObject` jako první v pořadí tříd, jinak mohou vznikat nelogicky se jevící chyby při sestavení výsledné aplikace, způsobené takovouto nekonzistencí introspekčního systému Qt.

```
GraphWidget::GraphWidget(QWidget *rodic): QWidget(rodic)
{
  // inicializace atributů
  mMaximalniCas = 100000; mPocatecniUroven[0] = 0,
    mPocatecniUroven[1] = 1, mPocatecniUroven[2] = 2;
  mZmenenaUroven[0] = 1, mZmenenaUroven[1] = 0, mZmenenaUroven[2] = 1;
  mCasZmeny[0] = 25000, mCasZmeny[1] = 50000, mCasZmeny[2] = 75000;

  // nastavení rozměrů widgetů: expandující šířka a pevná výška
  setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Fixed);
  setMinimumHeight(VyskaGrafu); setMaximumHeight(VyskaGrafu);
}
```

Konstruktor našeho widgetu se příliš neliší od ostatních implementací, které jste doposud mohli vidět v této publikaci. V jeho inicializační části se nejprve vytvoří bázový widget, kterému se předá případný rodič. Poté se už klasicky inicializují atributy a nastaví se parametry vlastního prvku – metody `set*()`. My zde pouze nastavíme politiku velikosti v horizontálním a vertikálním směru. Konkrétně zařídíme, aby měl náš widget pevnou výšku a flexibilní šířku.

```
void GraphWidget::paintEvent(QPaintEvent*)
{
  QPainter painter(this); // bude se kreslit na tento widget
  painter.setRenderHint(QPainter::Antialiasing); // a vyhlazovaně

  // bílé a plně-vyplněné pozadí
  painter.setBrush(QBrush(Qt::white, Qt::SolidPattern));
  // okrajový obdélník grafu
  painter.drawRect(0, 0, (width() - 1), (VyskaGrafu - 1));

  // popisky grafu
  painter.drawText((Okraj/3), (PoziceGrafuX-10), "X:");
  painter.drawText((Okraj/3), (PoziceGrafuY-10), "Y:");
  painter.drawText((Okraj/3), (PoziceGrafuZ-10), "Z:");

  // vodicí čáry grafu
  painter.drawLine(Okraj, PoziceGrafuX, (width() - Okraj-1), PoziceGrafuX);
  painter.drawLine(Okraj, PoziceGrafuY, (width() - Okraj-1), PoziceGrafuY);
  painter.drawLine(Okraj, PoziceGrafuZ, (width() - Okraj-1), PoziceGrafuZ);

  // počáteční a koncové časy na časové ose
  painter.drawText(Okraj, PoziceYTextu, "0");
  painter.drawText((width() - Okraj-1-15),
    PoziceYTextu, QString("%1").arg(mMaximalniCas));

  int zmenaGrafu[3];
  // výpočet pozic změn v grafu 0 = X, 1 = Y a 2 = Z
  for (int i = 0; i < 3; i++)
  { zmenaGrafu[i] = int(double(mCasZmeny[i]) /
      mMaximalniCas * (width() - 2*Okraj - 1)); }

  // vykreslení grafu vstupu X, vstupu Y a výstupu Z
  for (int i = 0, obdelnikYPozice = PoziceGrafuX, obdelnikY = 15; i < 3;
```

```
    i++, obdelnikYPozice += OdsazeniGrafu, obdelnikY += OdsazeniGrafu)
  {
    // vypsání časů změn
    painter.drawText((Okraj + zmenaGrafu[i]),
      PoziceYTextu, QString("%1").arg(mCasZmeny[i]));

    if (mPocatecniUroven[i] != 0)
    {
      if (mPocatecniUroven[i] == 1) // počáteční úroveň je 1
      { painter.setBrush(QBrush(Qt::green, Qt::SolidPattern));
      }
      else if (mPocatecniUroven[i] == 2) // počáteční úroveň je X
      { painter.setBrush(QBrush(Qt::darkYellow, Qt::DiagCrossPattern));
      }
      else { ; } // počáteční úroveň je 0 == vykreslena pouze vodicí čára..

      painter.drawRect(Okraj, obdelnikY, zmenaGrafu[i],
       (obdelnikYPozice - obdelnikY));
    }

    if (mZmenenaUroven[i] != 0)
    {
      if (mZmenenaUroven[i] == 1) // počáteční úroveň změny je 1
      { painter.setBrush(QBrush(Qt::green, Qt::SolidPattern));
      }
      else if (mZmenenaUroven[i] == 2) // počáteční úroveň změny je X
      { painter.setBrush(QBrush(Qt::darkYellow, Qt::DiagCrossPattern));
      }
      else { ; } // počáteční úroveň je 0 == vykreslena pouze vodicí čára..

      painter.drawRect((Okraj + zmenaGrafu[i]), obdelnikY,
        (width() - 2*Okraj - zmenaGrafu[i]),
          (obdelnikYPozice - obdelnikY));
    }
  }
}
```

Kompletní vykreslení našeho widgetu probíhá v metodě `paintEvent()` pomocí třídy `QPainter`, jejíž instance je inicializována ukazatelem `this`, což zajistí, že se bude kreslit na kreslicí zařízení reprezentující vizuální plochu tohoto widgetu. Po tomto kroku se už klasicky vykreslí vlastní grafický obsah pomocí painteru. Pro připomenutí významu jednotlivých operací se podívejte do předchozí kapitoly.

Na tomto místě ještě zmiňme, že vykreslení widgetu takovýmto způsobem vždy automaticky zajistí jeho požadované překreslení či aktualizaci požadované oblasti na výzvu hostitelského systému. V případě, že byste však radikálně změnili obsah widgetu programově a vyžadovali okamžité překreslení z některé jiné členské metody, lze využít metod `update()` či `repaint()`. Rozdíl je v tom, že `update()` naplánuje další překreslení do fronty se zamezením duplicitních překreslování, zatímco `repaint()` provede okamžité překreslení. Ve většině případů je tedy vhodnější používat metodu `update()`.

Integrace vlastního widgetu do aplikace

Widget vytvořený předchozím způsobem je velmi jednoduché využít ve své aplikaci – zachází se s ním totiž přesně stejně jako s jakýmkoli jiným integrovaným widgetem. Pouze k vložení jeho deklarací nemůžete využít globální inkluze pomocí `<QtWidgets>`, ale budete muset ručně vložit jeho hlavičkový soubor. Poté už přichází na řadu familiérní instanciace, parametrizace a utilizace, viz ukázka níže.

```
#include <QtWidgets>
#include "graph-widget.h"

class Okno: public QWidget
{ Q_OBJECT

  public:
    Okno(int pocetHradel); // konstruktor
    ~Okno(); // destruktor

    // nastaví text určitého popisku
    void nastavitTextPopisku(int popisek, QString text);

  public slots:
    void oProgramu();

  private:
    QLabel* vytvoritPopisek(const QString &text);

    QMenuBar *hlavniNabidka;
    QMenu *nabidkaProgram, *nabidkaNapoveda;
    QAction *akceKonec, *akceOProgramu;

    int mPocetHradel;
    QLabel **mPopiskyGrafu;
    GraphWidget **widgetyGrafu;
};
```

V hlavičkovém souboru třídy demonstračního okna pro vložení našeho widgetu si všimněte druhého a předposledního řádku. Na druhém řádku se nachází `#include "graph-widget.h"`, zpřístupňující deklarace našeho widgetu, a na předposledním řádku je pak ukazatel na ukazatel na náš widget. Ukazatel na ukazatel je užit proto, že nevíme, kolik „grafů“ budeme tvořit – to je parametr konstruktoru – budeme je tedy tvořit dynamicky a ukazatel na ukazatel je vlastně alias na jednorozměrné dynamické pole.

```
Okno::Okno(int pocetHradel): mPocetHradel(pocetHradel)
{
  // vytvoření hlavní nabídky aplikace
  hlavniNabidka = new QMenuBar(this);
  nabidkaProgram = hlavniNabidka->addMenu(QString::fromUtf8("&Program"));
  nabidkaNapoveda = hlavniNabidka->addMenu(QString::fromUtf8("&Nápověda"));

  // akce hlavní nabídky
```

```
akceKonec = nabidkaProgram->addAction(QString::fromUtf8("Konec"));
akceOProgramu = nabidkaNapoveda->addAction(
  QString::fromUtf8("O programu.."));

// propojení obslužných slotů
connect(akceKonec, SIGNAL(triggered()),
  this, SLOT(close())); // obsluha ukončení aplikace
connect(akceOProgramu, SIGNAL(triggered()),
  this, SLOT(oProgramu())); // obsluha krátké nápovědy

// tvorba UI aplikace
QScrollArea *posuvnaOblast = new QScrollArea;
QVBoxLayout *rozvrzeni = new QVBoxLayout;

rozvrzeni->setMenuBar(hlavniNabidka);
rozvrzeni->addWidget(posuvnaOblast, 100);
posuvnaOblast->setWidgetResizable(true);

mPopiskyGrafu = new QLabel*[mPocetHradel];
QGridLayout *rozvrzeniGrafu = new QGridLayout;

// tvorba a přidání popisku grafu do rozvržení grafu
for (int i = 0; i < mPocetHradel; i++)
{ mPopiskyGrafu[i] = vytvoritPopisek(
    tr("Hradlo:\n") + QString("%1").arg(i));
  rozvrzeniGrafu->addWidget(mPopiskyGrafu[i], i, 0);
  rozvrzeniGrafu->setRowMinimumHeight(i, 100); }

QWidget *grafy = new QWidget();
widgetyGrafu = new GraphWidget*[mPocetHradel]; // alokace widgetů grafů

qsrand(13); // deterministická inicializace generátoru náhodných čísel

// inicializace grafu náhodnými hodnotami
for (int i = 0; i < mPocetHradel; i++)
{ widgetyGrafu[i] = new GraphWidget(grafy);

  widgetyGrafu[i]->nastavitPocatecniUrovenX(qrand() % 3);
  widgetyGrafu[i]->nastavitPocatecniUrovenY(qrand() % 3);
  widgetyGrafu[i]->nastavitPocatecniUrovenZ(qrand() % 3);

  widgetyGrafu[i]->nastavitZmenenouUrovenX(
    (widgetyGrafu[i]->pocatecniUrovenX() + 1) % 3);
  widgetyGrafu[i]->nastavitZmenenouUrovenY(
    (widgetyGrafu[i]->pocatecniUrovenY() + 1) % 3);
  widgetyGrafu[i]->nastavitZmenenouUrovenZ(
    (widgetyGrafu[i]->pocatecniUrovenZ() + 1) % 3);

  widgetyGrafu[i]->nastavitCasZmenyX(qrand() % 100000);
  widgetyGrafu[i]->nastavitCasZmenyY(qrand() % 100000);
  widgetyGrafu[i]->nastavitCasZmenyZ(qrand() % 100000); }
```

```
    // přidání grafů do jejich rozvržení
    for (int i = 0; i < mPocetHradel; i++)
    { rozvrzeniGrafu->addWidget(widgetyGrafu[i], i , 1); }

    // nastavení rozvržení grafů a jejich předání do posuvné oblasti
    grafy->setLayout(rozvrzeniGrafu);
    posuvnaOblast->setWidget(grafy);

    setLayout(rozvrzeni); // aplikace vytvořeného rozvržení
    setMinimumSize(400, 200); // minimální rozměry okna
    resize(800, 600); // počáteční velikost okna
    setWindowTitle(QString::fromUtf8("GraphWidget")); // titulek okna
}
```

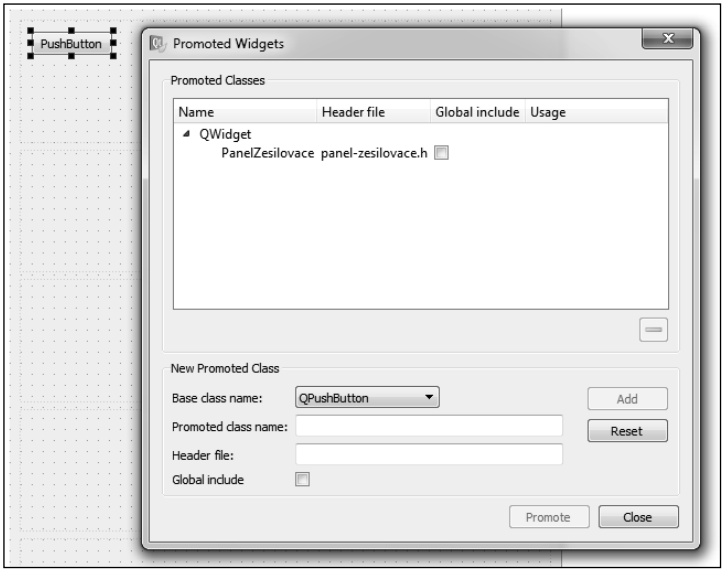
Z implementace třídy okna uvedeme pouze konstruktor, který je nejrelevantnější k použití našeho widgetu. Ostatní metody i soubory této ukázky samozřejmě naleznete v elektronické podobě. Na začátku těla konstruktoru opět vytvoříme, jak už je dobrým zvykem, uživatelské prostředí aplikace. Poté alokujeme potřebný počet ukazatelů na naše widgety, postupně je vytvoříme, zparametrizujeme pseudonáhodnými hodnotami a přidáme do celkového rozvržení okna. A to je v podstatě vše, nakonec je ještě třeba aplikovat layout a nastavit parametry výsledného okna.

Integrace vlastního widgetu do návrhového prostředí Qt Creatoru

V případě, že budete chtít využít svůj vlastní nově vytvořený widget v návrhovém prostředí Qt Creatoru, budete muset využít poněkud odlišný postup. Váš widget je totiž zatím pouze ve zdrojové podobě, a tak o něm Creator nemůže vědět a tím pádem logicky nenabídne žádnou možnost jeho vložení na návrhový formulář. Proto je pro vložení vlastního widgetu nutno zvolit následující postup.

Nejprve vložte na formulář widget, jehož bázová třída co nejvíce koresponduje s bázovou třídou vašeho widgetu. V podstatě však lze zvolit libovolný widget, jen pak budete muset přenastavit bázovou třídu. Na ten pak klepněte pravým tlačítkem myší a z kontextové nabídky zvolte volbu *Promote to..* (česky *Povýšit na..*). V následně zobrazeném dialogovém okně poté vyberte bázovou třídu vašeho widgetu (pokud se liší) a vyplňte cestu k hlavičkovému souboru vašeho widgetu. Po úspěšném odsouhlasení dialogu se vložený widget transformuje na váš vlastní, se kterým pak můžete pracovat jako s ostatními vestavěnými widgety.

Poznámka: Další možností je udělat z vašeho widgetu plugin pro Qt Creator. Jedná se však o pokročilejší postup, kterému se zde věnovat nebudeme. V případě zájmu vás tedy odkazujeme na patřičnou dokumentaci. Hledejte Qt Creator plug-ins apod. nebo nahlédněte do [1].



Obrázek 9.2: Dialog pro povýšení obecně libovolného widgetu na vlastní widget

Události v Qt a jejich obsluha

Při popisu tvorby vlastního widgetu a jeho integrace do uživatelské aplikace jsme se zatím zaměřili pouze na jeho statické vlastnosti, jako jsou doporučené rozměry, vizuální plocha atd. V typické aplikaci však probíhá mnoho dynamických dějů, jako jsou nejrůznější události, zaslané signály či dopočítané časovače, na které je třeba flexibilně reagovat a měnit tak její chování. Práci se signály a sloty jsme si už předvedli dříve, proto se nyní zaměříme na události a v další sekci na časovače, jejich užití a obsluhu.

S událostmi jste se v této publikaci už setkali na několika místech, naposledy s vykreslovací událostí `paintEvent()` výše v této kapitole, zatím však nebylo nikde podrobněji vysvětleno, co to vlastně událost je, proto to nyní napravíme. Událost je obecně změna stavu, která nastane buď vlivem uživatele, hostitelského sytému nebo samotné aplikace a obvykle s sebou nese datový objekt, který tuto změnu stavu podrobněji popisuje.

Například u události kliku myší – `mousePressEvent()` – je to objekt typu `QMouseEvent` nesoucí například pozici kurzoru při tomto klepnutí a další užitečné informace. V Qt se pak události obsluhují nejčastěji pomocí reimplementace chráněných metod dané třídy, reprezentující reakci na takovouto událost. Těmto metodám se také říká specializované *ovladače či obsluhy událostí* (anglicky *event handlers*). Tento přístup jste už mohli vidět na příkladu události `paintEvent()`, která je vyvolána v případě potřeby překreslení vizuální plochy widgetu.

Tip: Další možností obsluhy události je obecný přístup přes metodu event(). Když reimplementujete přímo metodu `event()`, která si bere obecný parametr typu `QEvent`, můžete v těle této metody obsloužit více typů událostí najednou. Nejedná se však o nejpřímočařejší techniku, a tak se doporučuje využívat spíše specializovaných chráněných metod jednotlivých událostí. Tento pokročilejší přístup se však může hodit při vytváření různých filtrů událostí, jejich zachytávání, monitoringu apod.

Z výše uvedeného lze tedy jednoduše odvodit, že existují v podstatě tři druhy či kategorie událostí, podle toho, kdo danou událost způsobil či vygeneroval. Jsou to tedy *události uživatelské*, *systémové* a *aplikační*. Mezi uživatelské události patří nejčastěji události myši (`mouse*Event()`) či klávesnice (`key*Event()`). Do kategorie systémových událostí by šlo zařadit všechny zprávy od hostitelského operačního systému – například `QCloseEvent`, reprezentující výzvu systému k uzavření okna či dialogu. A nakonec do aplikačních událostí řadíme třeba události od vlastních časovačů aplikace, popsané níže.

Poznámka: Jednotlivé události zasílá, respektive volá konkrétní obslužné metody, instance hlavního objektu aplikace – `QApplication`. A proto je lze pro jednotlivé widgety či celou aplikaci také filtrovat – k čemuž slouží tzv. *filtry událostí*. Jsou to objekty odvozené přímo z třídy `QObject`, které reimplementují chráněnou metodu `eventFilter(QObject *objekt, QEvent *udalost)`, jež tvoří vlastní filtr pro zachycení, respektive propagaci, daného typu události. Takovýto filtr se pak aplikuje na konkrétní prvek voláním metody `installEventFilter(QObject* filtr)`.

Poté co jsme si vyjasnili, co to události jsou a k čemu slouží, podíváme se nyní na jednotlivé události podrobněji v tabulce níže, kde uvedeme základní druhy událostí, jejich typ předávaného datového objektu a vlastní popis dané události. Na tomto místě také uveďme, že všechny typy datových objektů jsou odvozeny ze třídy `QEvent`, reprezentující obecnou událost, a v jejíž referenční dokumentaci lze dohledat odkazy na všechny ostatní typy datových objektů dostupných událostí, jako jsou například `QStatusTipEvent`, `QShowEvent` atp.

Tabulka 9.1: Základní typy událostí v Qt

Událost	Typ datového objektu	Popis
`keyPressEvent()`	`QKeyEvent`	Je vyvolána při stisku klávesy. Nese informaci o této klávese a případných modifikátorech.
`keyReleaseEvent()`	`QKeyEvent`	Je vyvolána při uvolnění klávesy. Nese informaci o této klávese a případných modifikátorech.
`mousePressEvent()`	`QMouseEvent`	Je vyvolána při stisku tlačítka myši a nese informaci o tomto tlačítku.
`mouseMoveEvent()`	`QMouseEvent`	Je vyvolána při pohybu myši při stisknutém tlačítku a nese informaci o tomto tlačítku. Pro permanentní sledování myši je třeba povolit vlastnost `mouseTracking`.
`mouseReleaseEvent()`	`QMouseEvent`	Je vyvolána při uvolnění tlačítka myši a nese informaci o tomto tlačítku.

Událost	Typ datového objektu	Popis
`mouseDoubleClickEvent()`	`QMouseEvent`	Je vyvolána při dvojkliku myši a nese dodatečné informace o této akci.
`wheelEvent()`	`QWheelEvent`	Je vyvolána při pohybu kolečkem myši a nese dodatečné informace o této akci.
`enterEvent()`	`QEvent`	Je vyvolána, při vstupu kurzoru nad plochu widgetu.
`leaveEvent()`	`QEvent`	Je vyvolána když kurzor opustí plochu widgetu.
`focusInEvent()`	`QFocusEvent`	Je vyvolána při předání focusu danému prvku. Permanentní fokusace lze docílit voláním `grabMouse()` či `grabKeybord()`.
`focusOutEvent()`	`QFocusEvent`	Je vyvolána při odebrání focusu danému prvku. Pro odstranění permanentní fokusace je nutno zavolat `releaseMouse()` či `releaseKeyboard()`. Toto je nutné, jinak se aplikace může permanentně „odmlčet“.
`showEvent()`	`QShowEvent`	Je vyvolána při zobrazení daného prvku a nese o tom dodatečné informace.
`hideEvent()`	`QHideEvent`	Je vyvolána při skrytí daného prvku a nese o tom dodatečné informace.
`paintEvent()`	`QPaintEvent`	Je vyvolána při žádosti o překreslení plochy widgetu. Jedná se o typický příklad systémové události.
`resizeEvent()`	`QResizeEvent`	Je vyvolána při změně rozměrů widgetu a může sloužit k novému výpočtu geometrie.
`closeEvent()`	`QCloseEvent`	Je vyvolána při žádosti o uzavření daného prvku. V jejím těle lze ještě danou žádost přezkoumat či potvrdit.
`contextMenuEvent()`	`QContextMenuEvent`	Je vyvolána při žádosti o kontextovou nabídku daného prvku. Obvykle je aktivována pravým tlačítkem myši. Pokud není patřičně obsloužena, transformuje se na klasické stisknutí tlačítka myši – `mousePressEvent()`.

Na konci této sekce se ještě podívejme na praktickou ukázku jednoduché obsluhy žádosti o kontextovou nabídku pomocí reimplementace metody `contextMenuEvent()`:

```
void VlastniWidget::contextMenuEvent(QContextMenuEvent *udalost)
{
  QMenu kontextovaNabidka;
  QAction *akcePridat = menu.addAction(QString::fromUtf8("Přidat.."));
  QAction *akceOdebrat = menu.addAction(QString::fromUtf8("Odebrat.."));
  kontextovaNabidka.addSeparator();
  QAction *akceOdebrat = menu.addAction(QString::fromUtf8("Nápověda"));

  // propojení akcí s obslužnými sloty..
```

```
    // vytvoření a spuštění nabídky na pozici kurzoru
    // kde byla událost vyvolána..
    QAction *vybranaAkce = kontextovaNabidka.exec(udalost->screenPos());

    // dodatečná reakce na vybranou akci...

    udalost->accept();
    // v případě udalost->ignore() by se propagovala dále..
}
```

Z ukázky je vidět, že tvorba kontextové nabídky není principiálně nijak složitá. Nejprve je třeba vytvořit vlastní nabídku s položkami typu `QAction`, obdobně jako byste je ručně tvořili v případě hlavní nabídky. Poté stačí tyto akce, respektive jejich patřičné signály, propojit s obslužnými sloty a vyvolat nabídku na místě stisku pravého tlačítka myši. Souřadnice tohoto místa v podobě struktury `QPoint` vrátí metoda `screenPos()` této události. Nakonec je ještě třeba událost přijmout neboli akceptovat metodou `accept()`, aby se nepropagovala dále a neobsloužila jako prosté klepnutí pravým tlačítkem myši na daný prvek.

V souvislosti s obsluhou události je také nutné představit dvojici komplementárních metod `accept()` a `ignore()`, které slouží k potvrzení, respektive propagaci, obsluhy dané události. Jak už jste viděli v ukázce výše, metoda `accept()` slouží k potvrzení obsluhy a tím de facto k zahození dané události pro další zpracování. Naopak metoda `ignore()` říká, že danou událost na programové úrovni naší aplikace ignorujeme a požadujeme, aby byla obsloužena výchozím způsobem. To obvykle znamená postupnou propagaci nahoru hierarchickou strukturou rodičů a následné zahození na nejvyšším místě hierarchie, tedy hlavním objektem aplikace typu `QApplication`.

V praxi může také nastat požadavek na vyvolání výchozí obsluhy dané události a pouze na jeho rozšíření. Toho lze dosáhnout zmíněnou metodou `ignore()` nebo i přímým voláním chráněné metody dané události ve vlastním kódu. Například `QWidget::mousePressEvent(udalost);` v případě, když chcete pouze danou obsluhu rozšířit vlastní implementací. Tuto techniku jsme často používali při implementaci grafické scény (`ScenaPlatna`) našeho ukázkového editoru, viz implementace metod `mouse*Event()` této třídy.

Poznámka: U popisu událostí ještě zmiňme, že existuje dokonce i programovací paradigma nazývané *událostmi řízené programování* (anglicky *event driven programming*), kde se, jednoduše řečeno, program chápe jako sada metod obsluhujících události jednotlivých programových komponent. Tento styl programování je typický zejména pro vývojová prostředí typu *C++ Builder* či *Delphi* firmy *Embarcadero*, dříve známé jako *Borland*.

Časovače a časování v Qt

Časování je jeden z velmi častých požadavků v mnoha aplikacích, které potřebují postupně vykreslit určitý grafický obsah a vytvořit tak jednoduchou animaci či sekvenci nebo jinak odstupňovat svou činnost. V Qt k tomuto účelu slouží dva přístupy nebo lépe řečeno třídy, po-

mocí kterých lze časování provádět a řídit. Jednodušší třída `QBasicTimer` poskytuje pro tento účel základní rozhraní a je i interně využívána samotným Qt frameworkem. I když se spíše doporučuje používat pokročilejší třídu `QTimer`, popsanou níže, podíváme se i na možnosti třídy `QBasicTimer`, která se díky své jednoduchosti a nenáročnosti může občas hodit.

Tip: Pro pokročilejší animace je vhodnější využít možnosti animačního frameworku Qt využívajícího specializované kolekce tříd, zahrnující třídy, jako je **`QTimeLine`**, **`QEasingCurve`** a další. Jejich stručný popis naleznete v předchozí kapitole v sekci *Animace entit*. Tyto třídy lze samozřejmě využít i mimo grafický framework.

Základní časování pomocí třídy QBasicTimer

Práce s třídou `QBasicTimer` je, jak už její název napovídá, velmi jednoduchá, protože její veřejné rozhraní neobsahuje mnoho metod ani atributů. Základem je konstrukce tohoto časovače pomocí konstruktoru nebo jeho vytvoření jako atributu nějaké třídy vlastního widgetu. Poté už stačí časovač spustit metodou `start()` a v kódu zvoleného prvku obsluhovat jeho událost. Metodě `start()` totiž musíte předat časový interval v milisekundách a ukazatel na Qt objekt, kterému má zasílat notifikaci svého vypršení v daných časových intervalech generováním události `timerEvent(QTimerEvent *udalost)`.

Pro úplnost ještě doplňme, že tento časovač lze také dočasně či permanentně zastavit metodou `stop()` a restartovat pomocí již zmíněné metody `start()`. V případě více aktivních časovačů, což je možné, se lze daného časovače dotázat na jeho identifikační číslo metodou `timerId()` nebo si ověřit jeho aktivitu metodou `isActive()`. Nyní níže nabízíme jednoduchou abstraktní ukázku použití tohoto časovače.

```
class MujWidget: public QWidget
{ Q_OBJECT

  public:
    explicit MujWiget(QWidget *rodic = 0);
    ~MujWidget();

    // ...

  protected:
    void timerEvent(QTimerEvent *udalost);

  private:
    QBasicTimer casovac1, casovac2, casovac3;

   // ...
};

MujWidget::MujWidget(QWidget *rodic): QWidget(rodic)
{
  casovac1.start(100, this);
```

```
  casovac2.start(500, this);
  casovac3.start(1000, this);

  // ...
}

void MujWidget::timerEvent(QTimerEvent *udalost)
{
   if (udalost->timerId() == casovac1.timerId())
   {
     // nějaký kód..
     casovac1.stop();
   }
   else if (udalost->timerId() == casovac2.timerId())
   {
     if (casovac3.isActive())
     { // nějaký kód..

       casovac3.stop();
       casovac3.start(700, this);
     }
   }
   else if (udalost->timerId() == casovac3.timerId())
   { // nějaký kód..
   }
   else { QWidget::timerEvent(udalost); } // výchozí obsluha..
}
```

Z ukázky je vidět, že použití je velmi přímočaré, jedinou prací navíc je de facto sledování identifikačního čísla daného časovače, které je nutné ověřit při použití více takovýchto časovačů ve stejné třídě. Všechny totiž generují stejnou událost (`timerEvent()`), a tak by je jinak nešlo rozlišit a provádět požadované akce odpovídající příslušným časovým intervalům.

Tip: K základním časovačům lze také přistupovat přes metody `startTimer()` a `killTimer()`. Tyto metody jsou dostupné každému prvku odvozenému ze třídy `QObject` a poskytují přístup k vestavěnému časovacímu mechanismu těchto typů objektů. Metoda `startTimer()` aktivuje časovač s nastaveným časovým intervalem v milisekundách a vrátí jeho identifikační číslo. Naopak metoda `killTimer()` umožňuje ukončit časovač s daným identifikačním číslem. I tyto časovače generují událost `timerEvent()` a pracuje se s nimi kromě jejich instanciace a inicializace naprosto stejně jako s časovači třídy `QBasicTimer`.

Komplexnější časování pomocí třídy QTimer

Třída `QTimer` nabízí pokročilejší alternativu ke třídě `QBasicTimer` a dodává navíc několik zajímavých vlastností. Prvním doplňkem je, že přímo podporuje mechanismus Signálu & Slotu a přidává kromě repetitivních (opakujících se) časovačů i možnost *jediného odpočítání časového intervalu* (anglicky *single-shot timer*).

I když je třída `QTimer` mírně komplexnější, neobsahuje nijak složité rozhraní. Základem jsou tři veřejně přístupné vlastnosti: `active`, `interval` a `singleShot`. Jak jistě sami odhadnete, atribut `active` uchovává boolovskou hodnotu aktivity časovače, `interval` pak jeho časový interval v milisekundách a nakonec atribut `singleShot` říká, jestli se časovač odpočítá pouze jednou (`true`) nebo je repetitivní (`false`).

```
void MujWidget::spustitCasovac(int interval)
{
  QTimer *casovac = new QTimer(this);
  connect(casovac, SIGNAL(timeout()),
    this, SLOT(dalsiKrokSekvence()));

  casovac->start(interval);
}
```

Samotné použití časovače je opět jednoduché. Po jeho vytvoření spojíte signál `timeOut()`, indikující dopočítání jednoho intervalu časovače s obslužným slotem nebo i více sloty, a v nich provedete požadované akce opakující se v daných intervalech. Samozřejmě je pak ještě nutné časovač spustit se zadaným intervalem metodou `start()`. Lze také využít její přetíženou alternativu bez parametru, která časovač pouze restartuje s přednastaveným intervalem z atributu `interval`. Ve výchozím stavu je interval nulový a takovéto nastavení slouží k často se opakujícím rutinám obsluhy GUI. Nakonec i u tohoto typu časovače je dostupná metoda `stop()` pro jeho zastavení.

```
QTimer::singleShot(3000, ukazatel_na_prijemce,
  SLOT(obsluznySlotPrijemce()));
```

Jak už bylo zmíněno výše, typ časovače *single-shot* podporuje možnost pouze jednoho odpočítání. K jejímu využití však nemusíte tvořit vlastní časovač (to je vhodné snad jen v případě mnohonásobného opakovaného využití s různými časovými intervaly), ale stačí zavolat statickou metodu `QTimer::singleShot()`. Té předáte časový interval v milisekundách k odpočítání spolu s ukazatelem na příjemce a jeho obslužný slot, který se automaticky zavolá po odpočítání tohoto intervalu začínajícího zavoláním této metody.

Upozornění: I když se jedná o pokročilejší časovač, ani `QTimer` nezajistí neomezeně přesné časové intervaly. To ani na klasickém PC není díky jeho architektuře možné – většinou je garantován pouze přibližný nejmenší rozlišitelný interval v délce 1 milisekundy. Ve specifických případech však lze použít určité platformně závislé postupy, které zajistí přesnější časování než obyčejné časovače. Avšak ani tyto postupy nejsou vhodné pro *real-timeové aplikace*, kde je ve většině případů třeba použít speciální hardware i software. Naopak je ale garantováno, že časovač nevyprší dříve, než je jeho stanovený interval.

Práce se soubory v Qt

I když je práce se soubory standardní součástí jazyka C/C++, ne vždy je tento zabudovaný nízkoúrovňový přístup vhodný nebo, lépe řečeno, výhodný. Přestože jsou v jazyce C++ k dispozici třídy, jako je `iostream`, `ifstream` či `ofstream`, které práci se soubory značně usnadňují – v porovnání se základním přístupem v jazyce C – struktura `FILE` ve spolupráci s funkcemi, jako je `fopen()`, `fprintf()`, `fclose()` a dalšími – ani tyto třídy nenabízejí bezproblémový multiplatformní přístup k práci se soubory.

Mezi největší problémy obou standardních zabudovaných přístupů patří jejich nízkoúrovňovost, kterou by však ještě šlo tolerovat, ale hlavně problematická *portabilita* (*přenostitelnost*) takto uložených dat. Ve standardu jazyka C ani C++ totiž není zaručena přesná velikost datových typů – k dispozici jsou pouze definice jistých hranic a vztahů mezi jednotlivými datovými typy – a tak při uložení například typu `int` na jedné platformě může tento typ zabírat například 2 bajty, zatímco na jiné bajty 4. A určitě si dovedete živě představit, k čemu by čtení takto zapsaných dat na takto odlišných platformách vedlo.

Poznámka: Někteří vnímají nepřesnost definic datových typů jazyků C/C++ jako jejich nevýhodu. Je to však daň za přenositelnost celého jazyka jako takového na množství podporovaných platforem. Proto je tento atribut jazyka třeba vnímat spíše jako vlastnost než chybu, počítat s ní a využívat některé z alternativních řešení tohoto problému – například níže zmíněný přístup v Qt.

Navíc je třeba myslet na hostitelské operační systémy i při práci s textovými soubory, u kterých se na jednotlivých platformách liší *terminátory* (*ukončovače*; anglicky *terminating characters*) jednotlivých řádků. Na Linuxu je to prosté `\n`, na Mac OS je to `\r` a na Windows „pragmaticky" zvolené `\r\n`. Proto Qt framework přichází se svým vlastním přístupem k manipulaci se soubory, který je plně multiplatformní a nabízí kolekci tříd pro jednotnou práci s binárními i textovými soubory a několik pomocných tříd pro procházení adresářové struktury a jednoduchou manipulaci s dočasnými soubory (`QTemporaryFile`).

Tip: Pro jednoduchou manipulaci s adresářovou strukturou lze využít třídu `QDir` a pro poloautomatické rekurzivní procházení adresářů pak lze použít specializovanou třídu `QDirIterator`. V případě potřeby podrobnějších informací o jednotlivých souborech či celém souborovém systému je k dispozici kolekce tříd `QFileInfo`, `QFileSystemModel` a `QFileSystemWatcher` ke sledování změn v daném adresáři či v konkrétním souboru.

Základem tohoto systému je třída `QFile`, odvozená z bázové třídy `QIODevice`, která zapouzdřuje v Qt všechna bloková zařízení, jako je `QFile`, `QBuffer` či `QAbstractSocket`, ze kterého jsou pak odvozeny třídy `QTcpSocket` a `QUdpSocket` pro práci se síťově přenášenými daty. Třída `QFile` umožňuje nízkoúrovňové multiplatformní čtení i zápis do binárních i textových souborů. Tuto třídu je však výhodné používat spolu se třídami `QDataStream` či `QTextStream`, které jsou specializované pro práci s binárními a textovými soubory.

`QTextStream` například umožňuje přímočaré čtení řádků (`readLine()`) textového souboru, aniž byste se explicitně starali o použité terminátory atd. `QDataStream` (česky *datový proud* či *proud dat*) pak slouží k jednoduché serializaci a deserializaci (čtení) binárních dat a ve spolupráci s přetíženým operátorem << pro vaše vlastní třídy a typy tvoří velmi robustní nástroj s jednoduchým rozhraním. Textovým souborům se zde věnovat nebudeme, naopak se podíváme podrobněji na třídu `QDataStream` a její použití pro serializaci a deserializaci vlastních dat. Ukázka níže pochází z jednoduchého grafického editoru představeného v předchozí kapitole o grafice.

```
void HlavniOkno::provestAkciUlozitJako()
{
  mAktualniNazevSouboruSKresbou = QFileDialog::getSaveFileName(this,
    QString::fromUtf8("Uložit kresbu jako.."),
    "./kresba.sgf", tr("kresba SimpleGraphic (*.sgf)"));

  mAktualniTitulekAplikace = mTitulekAplikace +
    QString(" - [ ").append(mAktualniNazevSouboruSKresbou).append(" ]");
  setWindowTitle(mAktualniTitulekAplikace);

  ulozitSouborSKresbou();
}
```

Při ukládání souboru s kresbou nejprve získáme pomocí souborového dialogu (`QFileDialog`) název ukládaného souboru s kresbou, poté ještě přenastavíme titulek aplikace a zavoláme vlastní metodu pro uložení souboru – `ulozitSouborSKresbou()`.

```
void HlavniOkno::ulozitSouborSKresbou()
{
  if (!mAktualniNazevSouboruSKresbou.isEmpty())
  { QFile soubor(mAktualniNazevSouboruSKresbou);
    soubor.open(QIODevice::WriteOnly); // otevření souboru pro zápis

    QDataStream vystupniStream(&soubor); // serializace dat do souboru
    vystupniStream.setVersion(QDataStream::Qt_5_0); // verze QDataStreamu

    // hlavička souboru: "magické číslo" a verze
    vystupniStream << quint32(0xAABBCCDD) << qint32(1);

    // zápis dat kresby do souboru
    scenaPlatna->ulozitKresbuDoSouboru(vystupniStream);
    soubor.close(); // uzavření souboru
  }
}
```

Před vlastním uložením souboru nejprve ověříme platnost jeho názvu, v případě, že je platný, vytvoříme soubor s daným názvem pomocí třídy `QFile`, který poté otevřeme pro zápis. Po tomto kroku inicializujeme instanci třídy `QDataStream` tímto souborem a můžeme začít zapisovat. Předtím však ještě nastavíme verzi datového proudu pro jeho jednoznačnou identifikaci a zapíšeme iniciální tzv. magické číslo, které bude identifikovat náš vlastní typ binárního souboru spolu s jeho verzí.

Vlastní zápis dat provedeme v těle metody `ulozitKresbuDoSouboru()`, popsané níže. Nakonec je nutné daný soubor uzavřít metodou `close()`, aby se úspěšně dokončil zápis, který je obvykle bufferovaný a došlo k uvolnění souboru pro další zájemce. Pro okamžitý zápis dat bez uzavření souboru lze také využít metodu `flush()`. Pro ukázku přidáváme už na tomto místě i kód pro načtení či deserializaci, chcete-li. Jak je z ukázky vidět, jedná se většinou o reverzní pořadí operací spolu s ověřením platnosti zapsaných identifikátorů.

```
void HlavniOkno::provestAkciOtevrit()
{
  mAktualniNazevSouboruSKresbou = QFileDialog::getOpenFileName(this,
    QString::fromUtf8("Otevřít kresbu.."), ".",
    tr("kresba SimpleGraphic (*.sgf)"));

  if (!mAktualniNazevSouboruSKresbou.isEmpty())
  { QFile soubor(mAktualniNazevSouboruSKresbou);
    soubor.open(QIODevice::ReadOnly); // otevření souboru pro čtení

    QDataStream vstupniStream(&soubor); // serializovaná data ze souboru
    vstupniStream.setVersion(QDataStream::Qt_5_0); // verze QDataStreamu

    // hlavička souboru: "magické číslo" a verze
    quint32 magickeCislo = 0, verze = 0;
    vstupniStream >> magickeCislo >> verze;

    if ((magickeCislo == 0xAABBCCDD) && (verze == 1))
    { // načtení dat kresby z binárního souboru
      scenaPlatna->nacistKresbuZeSouboru(vstupniStream); }

    soubor.close(); // uzavření souboru

    mAktualniTitulekAplikace = mTitulekAplikace +
      QString(" - [ ").append(mAktualniNazevSouboruSKresbou).append(" ]");
    setWindowTitle(mAktualniTitulekAplikace); }
}
```

Ukázky kódu výše vypadají na první pohled dosti jednoduše. Nyní se tedy podíváme na to, co za touto „jednoduchostí“ stojí. Nejprve uvádíme útržek kódu scény plátna, ve kterém jsou deklarace použitých metod.

```
class ScenaPlatna: public QGraphicsScene
{ Q_OBJECT

  public: ScenaPlatna(QObject *rodic = 0); // konstruktor
    // ...
    void nacistKresbuZeSouboru(QDataStream& stream);
    void ulozitKresbuDoSouboru(QDataStream& stream);
    // ...
};
```

Jak je z kódu vidět, obě metody přebírají jako svůj jediný parametr odkaz či referenci na instanci objektu typu `QDataStream`, který byl v předchozím kódu inicializován vytvořeným souborem a který bude sloužit pro vlastní zápis.

```
void ScenaPlatna::ulozitKresbuDoSouboru(QDataStream& vystupniStream)
{
  foreach (QGraphicsItem *entita, mGrafickeEntity)
  {
    if (entita->type() == LinearniEntita::Type)
    { LinearniEntita *cara = qgraphicsitem_cast<LinearniEntita*>(entita);
      if (cara) { vystupniStream << (*cara); }
    }
    else if (entita->type() == PolygonalniEntita::Type)
    { ... }
  }
}
```

Metoda pro zápis prochází všechny grafické entity a pomocí přetíženého operátoru << je serializuje do datového streamu. Toto se provádí pro všechny typy grafických entit. Proč tento „trik" funguje i pro naše vlastní typy, kterými samozřejmě grafické entity jsou, se dozvíte vzápětí. Předtím ještě doplníme komplementární ukázku kódu pro načtení jednoho typu grafické entity.

```
void ScenaPlatna::nacistKresbuZeSouboru(QDataStream& vstupniStream)
{
  clear(); // smazání entit scény
  mGrafickeEntity.clear(); // smazání kontejneru entit
  mVybraneEntity.clear(); // smazání seznamu vybraných entit

  // lokální proměnné pro dočasné uchování čtených atributů
  quint32 typEntity = 0, podtypEntity = 0;
  quint32 vrstvaEntity = 0; qreal rotaceEntity = 0.0;

  QPen obrysEntity; QBrush vyplnEntity; QPainterPath cestaEntity;
  QFont pismoEntity; QColor barvaPismaEntity; QString textEntity;
  QPointF poziceEntity;

  while (!vstupniStream.atEnd())
  { vstupniStream >> typEntity;

    // cesta se musí přemazat,
    // protože jinak si "pamatuje" předchozí body..
    cestaEntity = QPainterPath();

    if (typEntity == LinearniEntita::Type)
    {
      vstupniStream >> podtypEntity;
      vstupniStream >> vrstvaEntity >> rotaceEntity >> poziceEntity;
      vstupniStream >> obrysEntity >> vyplnEntity >> cestaEntity;

      LinearniEntita *cara = new LinearniEntita(
        cestaEntity, LinearniEntita::TypLinearniEntity(podtypEntity));
```

```
      cara->nastavitVrstvu(int(vrstvaEntity));
      cara->setRotation(rotaceEntity);
      cara->setPos(poziceEntity);

      cara->setPen(obrysEntity);
      cara->setBrush(vyplnEntity);
      cara->setPath(cestaEntity);

      cara->vytvoritRidiciBody();
      connect(cara, SIGNAL(linearniEntitaZmenena(QGraphicsItem*)),
        this, SLOT(provestPriZmeneEntity(QGraphicsItem*)));

      mGrafickeEntity.append(cara);
      addItem(cara);
    }
    else if (typEntity == PolygonalniEntita::Type)
    { ... }
  }
}
```

Celý trik jednoduchosti kódu pro přímý zápis vlastního datového typu do datového streamu, v tomto případě naší lineární grafické entity, je v reimplementaci a přetížení operátoru <<. Tomuto operátoru předáme odkaz na datový proud a naši entitu, v jeho těle zapíšeme standardní typy atributů, se kterými tento operátor umí už ve výchozím stavu pracovat, a vrátíme na něho referenci pro možnost řetězené syntaxe zápisu více položek. Dodejme, že operátor je deklarován jako přátelský pro možnost přímého přístupu k položkám třídy a tím i k mírnému urychlení vlastního zápisu.

```
class LinearniEntita: public GrafickaEntita
{ Q_OBJECT

  public:
    enum TypLinearniEntity
    { JednoduchaCara = 11, LomenaCara, CaraOdRuky };

    LinearniEntita(const QPainterPath &cesta,
    TypLinearniEntity typ = JednoduchaCara); // konstruktor
    ~LinearniEntita(); // destruktor

    // ...

    // přetížený operátor bitového posunu pro
    // zápis entity do binárního souboru
    friend QDataStream& operator<<(QDataStream& stream,
      const LinearniEntita& linearniEntita);
};
```

```
QDataStream& operator<<(QDataStream& stream,
  const LinearniEntita& linearniEntita)
{
  stream << qint32(LinearniEntita::Type)
    << qint32(linearniEntita.mTypLinearniEntity)
    << qint32(linearniEntita.mVrstva)
    << qreal(linearniEntita.rotation())
    << linearniEntita.scenePos() << linearniEntita.pen()
    << linearniEntita.brush() << linearniEntita.path();

  return stream;
}
```