```
//@line 2 "/builds/slave/rel-m-beta-and_bld-00000000000/build/mobile/android/chrome/content/browser.js"
// -*- Mode: js2; tab-width: 2; indent-tabs-mode: nil; js2-basic-offset: 2; js2-skip-preprocessor-directives: t; -*-
/* This Source Code Form is subject to the terms of the Mozilla Public
 * License, v. 2.0. If a copy of the MPL was not distributed with this
 * file, You can obtain one at http://mozilla.org/MPL/2.0/. */
"use strict";

let Cc = Components.classes;
let Ci = Components.interfaces;
let Cu = Components.utils;
let Cr = Components.results;

Cu.import("resource://gre/modules/XPCOMUtils.jsm");
Cu.import("resource://gre/modules/Services.jsm");
Cu.import("resource://gre/modules/AddonManager.jsm");
Cu.import("resource://gre/modules/FileUtils.jsm");
Cu.import("resource://gre/modules/JNI.jsm");
Cu.import('resource://gre/modules/Payment.jsm');
Cu.import("resource://gre/modules/NotificationDB.jsm");
Cu.import("resource://gre/modules/SpatialNavigation.jsm");
Cu.import("resource://gre/modules/UITelemetry.jsm");

//@line 24 "/builds/slave/rel-m-beta-and_bld-00000000000/build/mobile/android/chrome/content/browser.js"
Cu.import("resource://gre/modules/accessibility/AccessFu.jsm");
//@line 26 "/builds/slave/rel-m-beta-and_bld-00000000000/build/mobile/android/chrome/content/browser.js"

XPCOMUtils.defineLazyModuleGetter(this, "PluralForm",
                                  "resource://gre/modules/PluralForm.jsm");

XPCOMUtils.defineLazyModuleGetter(this, "sendMessageToJava",
                                  "resource://gre/modules/Messaging.jsm");

XPCOMUtils.defineLazyModuleGetter(this, "DebuggerServer",
                                  "resource://gre/modules/devtools/dbg-server.jsm");

XPCOMUtils.defineLazyModuleGetter(this, "UserAgentOverrides",
                                  "resource://gre/modules/UserAgentOverrides.jsm");

XPCOMUtils.defineLazyModuleGetter(this, "LoginManagerContent",
                                  "resource://gre/modules/LoginManagerContent.jsm");

XPCOMUtils.defineLazyModuleGetter(this, "LoginManagerParent",
                                  "resource://gre/modules/LoginManagerParent.jsm");

XPCOMUtils.defineLazyModuleGetter(this, "Task", "resource://gre/modules/Task.jsm");
XPCOMUtils.defineLazyModuleGetter(this, "OS", "resource://gre/modules/osfile.jsm");

//@line 49 "/builds/slave/rel-m-beta-and_bld-00000000000/build/mobile/android/chrome/content/browser.js"
XPCOMUtils.defineLazyModuleGetter(this, "SafeBrowsing",
                                  "resource://gre/modules/SafeBrowsing.jsm");
//@line 52 "/builds/slave/rel-m-beta-and_bld-00000000000/build/mobile/android/chrome/content/browser.js"

XPCOMUtils.defineLazyModuleGetter(this, "PrivateBrowsingUtils",
                                  "resource://gre/modules/PrivateBrowsingUtils.jsm");

XPCOMUtils.defineLazyModuleGetter(this, "Sanitizer",
                                  "resource://gre/modules/Sanitizer.jsm");

XPCOMUtils.defineLazyModuleGetter(this, "Prompt",
                                  "resource://gre/modules/Prompt.jsm");

XPCOMUtils.defineLazyModuleGetter(this, "HelperApps",
                                  "resource://gre/modules/HelperApps.jsm");

XPCOMUtils.defineLazyModuleGetter(this, "SSLExceptions",
                                  "resource://gre/modules/SSLExceptions.jsm");

XPCOMUtils.defineLazyModuleGetter(this, "FormHistory",
                                  "resource://gre/modules/FormHistory.jsm");

XPCOMUtils.defineLazyServiceGetter(this, "uuidgen",
                                   "@mozilla.org/uuid-generator;1",
                                   "nsIUUIDGenerator");

XPCOMUtils.defineLazyModuleGetter(this, "SimpleServiceDiscovery",
                                  "resource://gre/modules/SimpleServiceDiscovery.jsm");

//@line 82 "/builds/slave/rel-m-beta-and_bld-00000000000/build/mobile/android/chrome/content/browser.js"

XPCOMUtils.defineLazyModuleGetter(this, "WebappManager",
                                  "resource://gre/modules/WebappManager.jsm");

XPCOMUtils.defineLazyModuleGetter(this, "CharsetMenu",
                                  "resource://gre/modules/CharsetMenu.jsm");

XPCOMUtils.defineLazyModuleGetter(this, "PermissionsUtils",
                                  "resource://gre/modules/PermissionsUtils.jsm");

// Lazily-loaded browser scripts:
[
  ["SelectHelper", "chrome://browser/content/SelectHelper.js"],
  ["InputWidgetHelper", "chrome://browser/content/InputWidgetHelper.js"],
  ["AboutReader", "chrome://browser/content/aboutReader.js"],
  ["MasterPassword", "chrome://browser/content/MasterPassword.js"],
  ["PluginHelper", "chrome://browser/content/PluginHelper.js"],
  ["OfflineApps", "chrome://browser/content/OfflineApps.js"],
  ["Linkifier", "chrome://browser/content/Linkify.js"],
  ["ZoomHelper", "chrome://browser/content/ZoomHelper.js"],
  ["CastingApps", "chrome://browser/content/CastingApps.js"],
].forEach(function (aScript) {
  let [name, script] = aScript;
  XPCOMUtils.defineLazyGetter(window, name, function() {
    let sandbox = {};
    Services.scriptloader.loadSubScript(script, sandbox);
    return sandbox[name];
  });
});

[
//@line 114 "/builds/slave/rel-m-beta-and_bld-00000000000/build/mobile/android/chrome/content/browser.js"
  ["WebrtcUI", ["getUserMedia:request", "recording-device-events"], "chrome://browser/content/WebrtcUI.js"],
//@line 116 "/builds/slave/rel-m-beta-and_bld-00000000000/build/mobile/android/chrome/content/browser.js"
  ["MemoryObserver", ["memory-pressure", "Memory:Dump"], "chrome://browser/content/MemoryObserver.js"],
  ["ConsoleAPI", ["console-api-log-event"], "chrome://browser/content/ConsoleAPI.js"],
  ["FindHelper", ["FindInPage:Find", "FindInPage:Prev", "FindInPage:Next", "FindInPage:Closed", "Tab:Selected"], "chrome://browser/content/FindHelper.js"],
  ["PermissionsHelper", ["Permissions:Get", "Permissions:Clear"], "chrome://browser/content/PermissionsHelper.js"],
  ["FeedHandler", ["Feeds:Subscribe"], "chrome://browser/content/FeedHandler.js"],
  ["Feedback", ["Feedback:Show"], "chrome://browser/content/Feedback.js"],
  ["SelectionHandler", ["TextSelection:Get"], "chrome://browser/content/SelectionHandler.js"],
  ["Notifications", ["Notification:Event"], "chrome://browser/content/Notifications.jsm"],
].forEach(function (aScript) {
  let [name, notifications, script] = aScript;
  XPCOMUtils.defineLazyGetter(window, name, function() {
    let sandbox = {};
    Services.scriptloader.loadSubScript(script, sandbox);
    return sandbox[name];
  });
  let observer = (s, t, d) => {
    Services.obs.removeObserver(observer, t);
    Services.obs.addObserver(window[name], t, false);
    window[name].observe(s, t, d); // Explicitly notify new observer
  };
  notifications.forEach((notification) => {
    Services.obs.addObserver(observer, notification, false);
  });
});

// Lazily-loaded JS modules that use observer notifications
[
  ["Home", ["HomeBanner:Get", "HomePanels:Get", "HomePanels:Authenticate", "HomePanels:RefreshView",
            "HomePanels:Installed", "HomePanels:Uninstalled"], "resource://gre/modules/Home.jsm"],
].forEach(module => {
  let [name, notifications, resource] = module;
  XPCOMUtils.defineLazyModuleGetter(this, name, resource);
  let observer = (s, t, d) => {
    Services.obs.removeObserver(observer, t);
    Services.obs.addObserver(this[name], t, false);
    this[name].observe(s, t, d); // Explicitly notify new observer
  };
  notifications.forEach(notification => {
    Services.obs.addObserver(observer, notification, false);
  });
});

XPCOMUtils.defineLazyServiceGetter(this, "Haptic",
  "@mozilla.org/widget/hapticfeedback;1", "nsIHapticFeedback");

XPCOMUtils.defineLazyServiceGetter(this, "DOMUtils",
  "@mozilla.org/inspector/dom-utils;1", "inIDOMUtils");

XPCOMUtils.defineLazyServiceGetter(window, "URIFixup",
  "@mozilla.org/docshell/urifixup;1", "nsIURIFixup");

//@line 168 "/builds/slave/rel-m-beta-and_bld-00000000000/build/mobile/android/chrome/content/browser.js"
XPCOMUtils.defineLazyServiceGetter(this, "MediaManagerService",
  "@mozilla.org/mediaManagerService;1", "nsIMediaManagerService");
//@line 171 "/builds/slave/rel-m-beta-and_bld-00000000000/build/mobile/android/chrome/content/browser.js"

const kStateActive = 0x00000001; // :active pseudoclass for elements

const kXLinkNamespace = "http://www.w3.org/1999/xlink";

const kDefaultCSSViewportWidth = 980;
const kDefaultCSSViewportHeight = 480;

const kViewportRemeasureThrottle = 500;
const kDoNotTrackPrefState = Object.freeze({
  NO_PREF: "0",
  DISALLOW_TRACKING: "1",
  ALLOW_TRACKING: "2",
});

let Log = Cu.import("resource://gre/modules/AndroidLog.jsm", {}).AndroidLog;

// Define the "dump" function as a binding of the Log.d function so it specifies
// the "debug" priority and a log tag.
let dump = Log.d.bind(null, "Browser");

function doChangeMaxLineBoxWidth(aWidth) {
  gReflowPending = null;
  let webNav = BrowserApp.selectedTab.window.QueryInterface(Ci.nsIInterfaceRequestor).getInterface(Ci.nsIWebNavigation);
  let docShell = webNav.QueryInterface(Ci.nsIDocShell);
  let docViewer = docShell.contentViewer.QueryInterface(Ci.nsIMarkupDocumentViewer);

  let range = null;
  if (BrowserApp.selectedTab._mReflozPoint) {
    range = BrowserApp.selectedTab._mReflozPoint.range;
  }

  try {
    docViewer.pausePainting();
    docViewer.changeMaxLineBoxWidth(aWidth);

    if (range) {
      ZoomHelper.zoomInAndSnapToRange(range);
    } else {
      // In this case, we actually didn't zoom into a specific range. It
      // probably happened from a page load reflow-on-zoom event, so we
      // need to make sure painting is re-enabled.
      BrowserApp.selectedTab.clearReflowOnZoomPendingActions();
    }
  } finally {
    docViewer.resumePainting();
  }
}

function fuzzyEquals(a, b) {
  return (Math.abs(a - b) < 1e-6);
}

/**
 * Convert a font size to CSS pixels (px) from twentieiths-of-a-point
 * (twips).
 */
function convertFromTwipsToPx(aSize) {
  return aSize/240 * 16.0;
}

//@line 234 "/builds/slave/rel-m-beta-and_bld-00000000000/build/mobile/android/chrome/content/browser.js"
Cu.import("resource://gre/modules/XPCOMUtils.jsm");
XPCOMUtils.defineLazyServiceGetter(this, "CrashReporter",
  "@mozilla.org/xre/app-info;1", "nsICrashReporter");
//@line 238 "/builds/slave/rel-m-beta-and_bld-00000000000/build/mobile/android/chrome/content/browser.js"

XPCOMUtils.defineLazyGetter(this, "ContentAreaUtils", function() {
  let ContentAreaUtils = {};
  Services.scriptloader.loadSubScript("chrome://global/content/contentAreaUtils.js", ContentAreaUtils);
  return ContentAreaUtils;
});

XPCOMUtils.defineLazyModuleGetter(this, "Rect",
                                  "resource://gre/modules/Geometry.jsm");

function resolveGeckoURI(aURI) {
  if (!aURI)
    throw "Can't resolve an empty uri";

  if (aURI.startsWith("chrome://")) {
    let registry = Cc['@mozilla.org/chrome/chrome-registry;1'].getService(Ci["nsIChromeRegistry"]);
    return registry.convertChromeURL(Services.io.newURI(aURI, null, null)).spec;
  } else if (aURI.startsWith("resource://")) {
    let handler = Services.io.getProtocolHandler("resource").QueryInterface(Ci.nsIResProtocolHandler);
    return handler.resolveURI(Services.io.newURI(aURI, null, null));
  }
  return aURI;
}

/**
 * Cache of commonly used string bundles.
 */
var Strings = {};
[
  ["brand",      "chrome://branding/locale/brand.properties"],
  ["browser",    "chrome://browser/locale/browser.properties"]
].forEach(function (aStringBundle) {
  let [name, bundle] = aStringBundle;
  XPCOMUtils.defineLazyGetter(Strings, name, function() {
    return Services.strings.createBundle(bundle);
  });
});

const kFormHelperModeDisabled = 0;
const kFormHelperModeEnabled = 1;
const kFormHelperModeDynamic = 2;   // disabled on tablets

var BrowserApp = {
  _tabs: [],
  _selectedTab: null,
  _prefObservers: [],
  isGuest: false,

  get isTablet() {
    let sysInfo = Cc["@mozilla.org/system-info;1"].getService(Ci.nsIPropertyBag2);
    delete this.isTablet;
    return this.isTablet = sysInfo.get("tablet");
  },

  get isOnLowMemoryPlatform() {
    let memory = Cc["@mozilla.org/xpcom/memory-service;1"].getService(Ci.nsIMemory);
    delete this.isOnLowMemoryPlatform;
    return this.isOnLowMemoryPlatform = memory.isLowMemoryPlatform();
  },

  deck: null,

  startup: function startup() {
    window.QueryInterface(Ci.nsIDOMChromeWindow).browserDOMWindow = new nsBrowserAccess();
    dump("zerdatime " + Date.now() + " - browser chrome startup finished.");

    this.deck = document.getElementById("browsers");
    this.deck.addEventListener("DOMContentLoaded", function BrowserApp_delayedStartup() {
      try {
        BrowserApp.deck.removeEventListener("DOMContentLoaded", BrowserApp_delayedStartup, false);
        Services.obs.notifyObservers(window, "browser-delayed-startup-finished", "");
        sendMessageToJava({ type: "Gecko:DelayedStartup" });

        // Queue up some other performance-impacting initializations
        Services.tm.mainThread.dispatch(function() {
          // Init LoginManager
          Cc["@mozilla.org/login-manager;1"].getService(Ci.nsILoginManager);
        }, Ci.nsIThread.DISPATCH_NORMAL);

//@line 318 "/builds/slave/rel-m-beta-and_bld-00000000000/build/mobile/android/chrome/content/browser.js"
        Services.tm.mainThread.dispatch(function() {
          // Bug 778855 - Perf regression if we do this here. To be addressed in bug 779008.
          SafeBrowsing.init();
        }, Ci.nsIThread.DISPATCH_NORMAL);
//@line 323 "/builds/slave/rel-m-beta-and_bld-00000000000/build/mobile/android/chrome/content/browser.js"
      } catch(ex) { console.log(ex); }
    }, false);

    BrowserEventHandler.init();
    ViewportHandler.init();

    Services.androidBridge.browserApp = this;

    Services.obs.addObserver(this, "Locale:Changed", false);
    Services.obs.addObserver(this, "Tab:Load", false);
    Services.obs.addObserver(this, "Tab:Selected", false);
    Services.obs.addObserver(this, "Tab:Closed", false);
    Services.obs.addObserver(this, "Session:Back", false);
    Services.obs.addObserver(this, "Session:ShowHistory", false);
    Services.obs.addObserver(this, "Session:Forward", false);
    Services.obs.addObserver(this, "Session:Reload", false);
    Services.obs.addObserver(this, "Session:Stop", false);
    Services.obs.addObserver(this, "SaveAs:PDF", false);
    Services.obs.addObserver(this, "Browser:Quit", false);
    Services.obs.addObserver(this, "Preferences:Set", false);
    Services.obs.addObserver(this, "ScrollTo:FocusedInput", false);
    Services.obs.addObserver(this, "Sanitize:ClearData", false);
    Services.obs.addObserver(this, "FullScreen:Exit", false);
    Services.obs.addObserver(this, "Viewport:Change", false);
    Services.obs.addObserver(this, "Viewport:Flush", false);
    Services.obs.addObserver(this, "Viewport:FixedMarginsChanged", false);
    Services.obs.addObserver(this, "Passwords:Init", false);
    Services.obs.addObserver(this, "FormHistory:Init", false);
    Services.obs.addObserver(this, "gather-telemetry", false);
    Services.obs.addObserver(this, "keyword-search", false);
    Services.obs.addObserver(this, "webapps-runtime-install", false);
    Services.obs.addObserver(this, "webapps-runtime-install-package", false);
    Services.obs.addObserver(this, "webapps-ask-install", false);
    Services.obs.addObserver(this, "webapps-launch", false);
    Services.obs.addObserver(this, "webapps-runtime-uninstall", false);
    Services.obs.addObserver(this, "Webapps:AutoInstall", false);
    Services.obs.addObserver(this, "Webapps:Load", false);
    Services.obs.addObserver(this, "Webapps:AutoUninstall", false);
    Services.obs.addObserver(this, "sessionstore-state-purge-complete", false);

    function showFullScreenWarning() {
      NativeWindow.toast.show(Strings.browser.GetStringFromName("alertFullScreenToast"), "short");
    }

    window.addEventListener("fullscreen", function() {
      sendMessageToJava({
        type: window.fullScreen ? "ToggleChrome:Show" : "ToggleChrome:Hide"
      });
    }, false);

    window.addEventListener("mozfullscreenchange", function(e) {
      // This event gets fired on the document and its entire ancestor chain
      // of documents. When enabling fullscreen, it is fired on the top-level
      // document first and goes down; when disabling the order is reversed
      // (per spec). This means the last event on enabling will be for the innermost
      // document, which will have mozFullScreenElement set correctly.
      let doc = e.target;
      sendMessageToJava({
        type: doc.mozFullScreen ? "DOMFullScreen:Start" : "DOMFullScreen:Stop",
        rootElement: (doc.mozFullScreen && doc.mozFullScreenElement == doc.documentElement)
      });

      if (doc.mozFullScreen)
        showFullScreenWarning();
    }, false);

    // When a restricted key is pressed in DOM full-screen mode, we should display
    // the "Press ESC to exit" warning message.
    window.addEventListener("MozShowFullScreenWarning", showFullScreenWarning, true);

    NativeWindow.init();
    LightWeightThemeWebInstaller.init();
    Downloads.init();
    FormAssistant.init();
    IndexedDB.init();
    HealthReportStatusListener.init();
    XPInstallObserver.init();
    CharacterEncoding.init();
    ActivityObserver.init();
    // TODO: replace with Android implementation of WebappOSUtils.isLaunchable.
    Cu.import("resource://gre/modules/Webapps.jsm");
    DOMApplicationRegistry.allAppsLaunchable = true;
    RemoteDebugger.init();
    Reader.init();
    UserAgentOverrides.init();
    DesktopUserAgent.init();
    CastingApps.init();
    Distribution.init();
    Tabs.init();
//@line 413 "/builds/slave/rel-m-beta-and_bld-00000000000/build/mobile/android/chrome/content/browser.js"
    AccessFu.attach(window);
//@line 418 "/builds/slave/rel-m-beta-and_bld-00000000000/build/mobile/android/chrome/content/browser.js"

    let url = null;
    let pinned = false;
    if ("arguments" in window) {
      if (window.arguments[0])
        url = window.arguments[0];
      if (window.arguments[1])
        gScreenWidth = window.arguments[1];
      if (window.arguments[2])
        gScreenHeight = window.arguments[2];
      if (window.arguments[3])
        pinned = window.arguments[3];
      if (window.arguments[4])
        this.isGuest = window.arguments[4];
    }

    if (pinned) {
      this._initRuntime(this._startupStatus, url, aUrl => this.addTab(aUrl));
    } else {
      SearchEngines.init();
      this.initContextMenu();
    }
    // The order that context menu items are added is important
    // Make sure the "Open in App" context menu item appears at the bottom of the list
    ExternalApps.init();

    // XXX maybe we don't do this if the launch was kicked off from external
    Services.io.offline = false;

    // Broadcast a UIReady message so add-ons know we are finished with startup
    let event = document.createEvent("Events");
    event.initEvent("UIReady", true, false);
    window.dispatchEvent(event);

    if (this._startupStatus)
      this.onAppUpdated();

    // notify java that gecko has loaded
    sendMessageToJava({ type: "Gecko:Ready" });
  },

  get _startupStatus() {
    delete this._startupStatus;

    let savedMilestone = null;
    try {
      savedMilestone = Services.prefs.getCharPref("browser.startup.homepage_override.mstone");
    } catch (e) {
    }
   let ourMilestone = "33.0";
    this._startupStatus = "";
    if (ourMilestone != savedMilestone) {
      Services.prefs.setCharPref("browser.startup.homepage_override.mstone", ourMilestone);
      this._startupStatus = savedMilestone ? "upgrade" : "new";
    }

    return this._startupStatus;
  },

  /**
   * Pass this a locale string, such as "fr" or "es_ES".
   */
  setLocale: function (locale) {
    console.log("browser.js: requesting locale set: " + locale);
    sendMessageToJava({ type: "Locale:Set", locale: locale });
  },

  _initRuntime: function(status, url, callback) {
    let sandbox = {};
    Services.scriptloader.loadSubScript("chrome://browser/content/WebappRT.js", sandbox);
    window.WebappRT = sandbox.WebappRT;
    WebappRT.init(status, url, callback);
  },

  initContextMenu: function ba_initContextMenu() {
    // TODO: These should eventually move into more appropriate classes
    NativeWindow.contextmenus.add(Strings.browser.GetStringFromName("contextmenu.openInNewTab"),
      NativeWindow.contextmenus.linkOpenableNonPrivateContext,
      function(aTarget) {
        UITelemetry.addEvent("action.1", "contextmenu", null, "web_open_new_tab");
        UITelemetry.addEvent("loadurl.1", "contextmenu", null);

        let url = NativeWindow.contextmenus._getLinkURL(aTarget);
        ContentAreaUtils.urlSecurityCheck(url, aTarget.ownerDocument.nodePrincipal);
        let tab = BrowserApp.addTab(url, { selected: false, parentId: BrowserApp.selectedTab.id });

        let newtabStrings = Strings.browser.GetStringFromName("newtabpopup.opened");
        let label = PluralForm.get(1, newtabStrings).replace("#1", 1);
        let buttonLabel = Strings.browser.GetStringFromName("newtabpopup.switch");
        NativeWindow.toast.show(label, "long", {
          button: {
            icon: "drawable://switch_button_icon",
            label: buttonLabel,
            callback: () => { BrowserApp.selectTab(tab); },
          }
        });
      });

    NativeWindow.contextmenus.add(Strings.browser.GetStringFromName("contextmenu.openInPrivateTab"),
      NativeWindow.contextmenus.linkOpenableContext,
      function(aTarget) {
        UITelemetry.addEvent("action.1", "contextmenu", null, "web_open_private_tab");
        UITelemetry.addEvent("loadurl.1", "contextmenu", null);

        let url = NativeWindow.contextmenus._getLinkURL(aTarget);
        ContentAreaUtils.urlSecurityCheck(url, aTarget.ownerDocument.nodePrincipal);
        let tab = BrowserApp.addTab(url, { selected: false, parentId: BrowserApp.selectedTab.id, isPrivate: true });

        let newtabStrings = Strings.browser.GetStringFromName("newprivatetabpopup.opened");
        let label = PluralForm.get(1, newtabStrings).replace("#1", 1);
        let buttonLabel = Strings.browser.GetStringFromName("newtabpopup.switch");
        NativeWindow.toast.show(label, "long", {
          button: {
            icon: "drawable://switch_button_icon",
            label: buttonLabel,
            callback: () => { BrowserApp.selectTab(tab); },
          }
        });
      });

    NativeWindow.contextmenus.add(Strings.browser.GetStringFromName("contextmenu.copyLink"),
      NativeWindow.contextmenus.linkCopyableContext,
      function(aTarget) {
        UITelemetry.addEvent("action.1", "contextmenu", null, "web_copy_link");

        let url = NativeWindow.contextmenus._getLinkURL(aTarget);
        NativeWindow.contextmenus._copyStringToDefaultClipboard(url);
      });

    NativeWindow.contextmenus.add(Strings.browser.GetStringFromName("contextmenu.copyEmailAddress"),
      NativeWindow.contextmenus.emailLinkContext,
      function(aTarget) {
        UITelemetry.addEvent("action.1", "contextmenu", null, "web_copy_email");

        let url = NativeWindow.contextmenus._getLinkURL(aTarget);
        let emailAddr = NativeWindow.contextmenus._stripScheme(url);
        NativeWindow.contextmenus._copyStringToDefaultClipboard(emailAddr);
      });

    NativeWindow.contextmenus.add(Strings.browser.GetStringFromName("contextmenu.copyPhoneNumber"),
      NativeWindow.contextmenus.phoneNumberLinkContext,
      function(aTarget) {
        UITelemetry.addEvent("action.1", "contextmenu", null, "web_copy_phone");

        let url = NativeWindow.contextmenus._getLinkURL(aTarget);
        let phoneNumber = NativeWindow.contextmenus._stripScheme(url);
        NativeWindow.contextmenus._copyStringToDefaultClipboard(phoneNumber);
      });

    NativeWindow.contextmenus.add({
      label: Strings.browser.GetStringFromName("contextmenu.shareLink"),
      order: NativeWindow.contextmenus.DEFAULT_HTML5_ORDER - 1, // Show above HTML5 menu items
      selector: NativeWindow.contextmenus._disableInGuest(NativeWindow.contextmenus.linkShareableContext),
      showAsActions: function(aElement) {
        return {
          title: aElement.textContent.trim() || aElement.title.trim(),
          uri: NativeWindow.contextmenus._getLinkURL(aElement),
        };
      },
      icon: "drawable://ic_menu_share",
      callback: function(aTarget) {
        UITelemetry.addEvent("action.1", "contextmenu", null, "web_share_link");
      }
    });

    NativeWindow.contextmenus.add({
      label: Strings.browser.GetStringFromName("contextmenu.shareEmailAddress"),
      order: NativeWindow.contextmenus.DEFAULT_HTML5_ORDER - 1,
      selector: NativeWindow.contextmenus._disableInGuest(NativeWindow.contextmenus.emailLinkContext),
      showAsActions: function(aElement) {
        let url = NativeWindow.contextmenus._getLinkURL(aElement);
        let emailAddr = NativeWindow.contextmenus._stripScheme(url);
        let title = aElement.textContent || aElement.title;
        return {
          title: title,
          uri: emailAddr,
        };
      },
      icon: "drawable://ic_menu_share",
      callback: function(aTarget) {
        UITelemetry.addEvent("action.1", "contextmenu", null, "web_share_email");
      }
    });

    NativeWindow.contextmenus.add({
      label: Strings.browser.GetStringFromName("contextmenu.sharePhoneNumber"),
      order: NativeWindow.contextmenus.DEFAULT_HTML5_ORDER - 1,
      selector: NativeWindow.contextmenus._disableInGuest(NativeWindow.contextmenus.phoneNumberLinkContext),
      showAsActions: function(aElement) {
        let url = NativeWindow.contextmenus._getLinkURL(aElement);
        let phoneNumber = NativeWindow.contextmenus._stripScheme(url);
        let title = aElement.textContent || aElement.title;
        return {
          title: title,
          uri: phoneNumber,
        };
      },
      icon: "drawable://ic_menu_share",
      callback: function(aTarget) {
        UITelemetry.addEvent("action.1", "contextmenu", null, "web_share_phone");
      }
    });

    NativeWindow.contextmenus.add(Strings.browser.GetStringFromName("contextmenu.addToContacts"),
      NativeWindow.contextmenus._disableInGuest(NativeWindow.contextmenus.emailLinkContext),
      function(aTarget) {
        UITelemetry.addEvent("action.1", "contextmenu", null, "web_contact_email");

        let url = NativeWindow.contextmenus._getLinkURL(aTarget);
        sendMessageToJava({
          type: "Contact:Add",
          email: url
        });
      });

    NativeWindow.contextmenus.add(Strings.browser.GetStringFromName("contextmenu.addToContacts"),
      NativeWindow.contextmenus._disableInGuest(NativeWindow.contextmenus.phoneNumberLinkContext),
      function(aTarget) {
        UITelemetry.addEvent("action.1", "contextmenu", null, "web_contact_phone");

        let url = NativeWindow.contextmenus._getLinkURL(aTarget);
        sendMessageToJava({
          type: "Contact:Add",
          phone: url
        });
      });

    NativeWindow.contextmenus.add(Strings.browser.GetStringFromName("contextmenu.bookmarkLink"),
      NativeWindow.contextmenus._disableInGuest(NativeWindow.contextmenus.linkBookmarkableContext),
      function(aTarget) {
        UITelemetry.addEvent("action.1", "contextmenu", null, "web_bookmark");

        let url = NativeWindow.contextmenus._getLinkURL(aTarget);
        let title = aTarget.textContent || aTarget.title || url;
        sendMessageToJava({
          type: "Bookmark:Insert",
          url: url,
          title: title
        });
      });

    NativeWindow.contextmenus.add(Strings.browser.GetStringFromName("contextmenu.playMedia"),
      NativeWindow.contextmenus.mediaContext("media-paused"),
      function(aTarget) {
        UITelemetry.addEvent("action.1", "contextmenu", null, "web_play");
        aTarget.play();
      });

    NativeWindow.contextmenus.add(Strings.browser.GetStringFromName("contextmenu.pauseMedia"),
      NativeWindow.contextmenus.mediaContext("media-playing"),
      function(aTarget) {
        UITelemetry.addEvent("action.1", "contextmenu", null, "web_pause");
        aTarget.pause();
      });

    NativeWindow.contextmenus.add(Strings.browser.GetStringFromName("contextmenu.showControls2"),
      NativeWindow.contextmenus.mediaContext("media-hidingcontrols"),
      function(aTarget) {
        UITelemetry.addEvent("action.1", "contextmenu", null, "web_controls_media");
        aTarget.setAttribute("controls", true);
      });

    NativeWindow.contextmenus.add({
      label: Strings.browser.GetStringFromName("contextmenu.shareMedia"),
      order: NativeWindow.contextmenus.DEFAULT_HTML5_ORDER - 1,
      selector: NativeWindow.contextmenus._disableInGuest(NativeWindow.contextmenus.SelectorContext("video")),
      showAsActions: function(aElement) {
        let url = (aElement.currentSrc || aElement.src);
        let title = aElement.textContent || aElement.title;
        return {
          title: title,
          uri: url,
          type: "video/*",
        };
      },
      icon: "drawable://ic_menu_share",
      callback: function(aTarget) {
        UITelemetry.addEvent("action.1", "contextmenu", null, "web_share_media");
      }
    });

    NativeWindow.contextmenus.add(Strings.browser.GetStringFromName("contextmenu.fullScreen"),
      NativeWindow.contextmenus.SelectorContext("video:not(:-moz-full-screen)"),
      function(aTarget) {
        UITelemetry.addEvent("action.1", "contextmenu", null, "web_fullscreen");
        aTarget.mozRequestFullScreen();
      });

    NativeWindow.contextmenus.add(Strings.browser.GetStringFromName("contextmenu.mute"),
      NativeWindow.contextmenus.mediaContext("media-unmuted"),
      function(aTarget) {
        UITelemetry.addEvent("action.1", "contextmenu", null, "web_mute");
        aTarget.muted = true;
      });

    NativeWindow.contextmenus.add(Strings.browser.GetStringFromName("contextmenu.unmute"),
      NativeWindow.contextmenus.mediaContext("media-muted"),
      function(aTarget) {
        UITelemetry.addEvent("action.1", "contextmenu", null, "web_unmute");
        aTarget.muted = false;
      });

    NativeWindow.contextmenus.add(Strings.browser.GetStringFromName("contextmenu.copyImageLocation"),
      NativeWindow.contextmenus.imageLocationCopyableContext,
      function(aTarget) {
        UITelemetry.addEvent("action.1", "contextmenu", null, "web_copy_image");

        let url = aTarget.src;
        NativeWindow.contextmenus._copyStringToDefaultClipboard(url);
      });

    NativeWindow.contextmenus.add({
      label: Strings.browser.GetStringFromName("contextmenu.shareImage"),
      selector: NativeWindow.contextmenus._disableInGuest(NativeWindow.contextmenus.imageSaveableContext),
      order: NativeWindow.contextmenus.DEFAULT_HTML5_ORDER - 1, // Show above HTML5 menu items
      showAsActions: function(aTarget) {
        let doc = aTarget.ownerDocument;
        let imageCache = Cc["@mozilla.org/image/tools;1"].getService(Ci.imgITools)
                                                         .getImgCacheForDocument(doc);
        let props = imageCache.findEntryProperties(aTarget.currentURI, doc.characterSet);
        let src = aTarget.src;
        return {
          title: src,
          uri: src,
          type: "image/*",
        };
      },
      icon: "drawable://ic_menu_share",
      menu: true,
      callback: function(aTarget) {
        UITelemetry.addEvent("action.1", "contextmenu", null, "web_share_image");
      }
    });

    NativeWindow.contextmenus.add(Strings.browser.GetStringFromName("contextmenu.saveImage"),
      NativeWindow.contextmenus.imageSaveableContext,
      function(aTarget) {
        UITelemetry.addEvent("action.1", "contextmenu", null, "web_save_image");

        ContentAreaUtils.saveImageURL(aTarget.currentURI.spec, null, "SaveImageTitle",
                                      false, true, aTarget.ownerDocument.documentURIObject,
                                      aTarget.ownerDocument);
      });

    NativeWindow.contextmenus.add(Strings.browser.GetStringFromName("contextmenu.setImageAs"),
      NativeWindow.contextmenus._disableInGuest(NativeWindow.contextmenus.imageSaveableContext),
      function(aTarget) {
        UITelemetry.addEvent("action.1", "contextmenu", null, "web_background_image");

        let src = aTarget.src;
        sendMessageToJava({
          type: "Image:SetAs",
          url: src
        });
      });

    NativeWindow.contextmenus.add(
      function(aTarget) {
        if (aTarget instanceof HTMLVideoElement) {
          // If a video element is zero width or height, its essentially
          // an HTMLAudioElement.
          if (aTarget.videoWidth == 0 || aTarget.videoHeight == 0 )
            return Strings.browser.GetStringFromName("contextmenu.saveAudio");
          return Strings.browser.GetStringFromName("contextmenu.saveVideo");
        } else if (aTarget instanceof HTMLAudioElement) {
          return Strings.browser.GetStringFromName("contextmenu.saveAudio");
        }
        return Strings.browser.GetStringFromName("contextmenu.saveVideo");
      }, NativeWindow.contextmenus.mediaSaveableContext,
      function(aTarget) {
        UITelemetry.addEvent("action.1", "contextmenu", null, "web_save_media");

        let url = aTarget.currentSrc || aTarget.src;
        let filePickerTitleKey = (aTarget instanceof HTMLVideoElement &&
                                  (aTarget.videoWidth != 0 && aTarget.videoHeight != 0))
                                  ? "SaveVideoTitle" : "SaveAudioTitle";
        // Skipped trying to pull MIME type out of cache for now
        ContentAreaUtils.internalSave(url, null, null, null, null, false,
                                      filePickerTitleKey, null, aTarget.ownerDocument.documentURIObject,
                                      aTarget.ownerDocument, true, null);
      });
  },

  onAppUpdated: function() {
    // initialize the form history and passwords databases on upgrades
    Services.obs.notifyObservers(null, "FormHistory:Init", "");
    Services.obs.notifyObservers(null, "Passwords:Init", "");

    // Migrate user-set "plugins.click_to_play" pref. See bug 884694.
    // Because the default value is true, a user-set pref means that the pref was set to false.
    if (Services.prefs.prefHasUserValue("plugins.click_to_play")) {
      Services.prefs.setIntPref("plugin.default.state", Ci.nsIPluginTag.STATE_ENABLED);
      Services.prefs.clearUserPref("plugins.click_to_play");
    }
  },

  shutdown: function shutdown() {
    NativeWindow.uninit();
    LightWeightThemeWebInstaller.uninit();
    FormAssistant.uninit();
    IndexedDB.uninit();
    ViewportHandler.uninit();
    XPInstallObserver.uninit();
    HealthReportStatusListener.uninit();
    CharacterEncoding.uninit();
    SearchEngines.uninit();
    RemoteDebugger.uninit();
    Reader.uninit();
    UserAgentOverrides.uninit();
    DesktopUserAgent.uninit();
    ExternalApps.uninit();
    CastingApps.uninit();
    Distribution.uninit();
    Tabs.uninit();
  },

  // This function returns false during periods where the browser displayed document is
  // different from the browser content document, so user actions and some kinds of viewport
  // updates should be ignored. This period starts when we start loading a new page or
  // switch tabs, and ends when the new browser content document has been drawn and handed
  // off to the compositor.
  isBrowserContentDocumentDisplayed: function() {
    try {
      if (!Services.androidBridge.isContentDocumentDisplayed())
        return false;
    } catch (e) {
      return false;
    }

    let tab = this.selectedTab;
    if (!tab)
      return false;
    return tab.contentDocumentIsDisplayed;
  },

  contentDocumentChanged: function() {
    window.top.QueryInterface(Ci.nsIInterfaceRequestor).getInterface(Ci.nsIDOMWindowUtils).isFirstPaint = true;
    Services.androidBridge.contentDocumentChanged();
  },

  get tabs() {
    return this._tabs;
  },

  get selectedTab() {
    return this._selectedTab;
  },

  set selectedTab(aTab) {
    if (this._selectedTab == aTab)
      return;

    if (this._selectedTab) {
      this._selectedTab.setActive(false);
    }

    this._selectedTab = aTab;
    if (!aTab)
      return;

    aTab.setActive(true);
    aTab.setResolution(aTab._zoom, true);
    this.contentDocumentChanged();
    this.deck.selectedPanel = aTab.browser;
    // Focus the browser so that things like selection will be styled correctly.
    aTab.browser.focus();
  },

  get selectedBrowser() {
    if (this._selectedTab)
      return this._selectedTab.browser;
    return null;
  },

  getTabForId: function getTabForId(aId) {
    let tabs = this._tabs;
    for (let i=0; i < tabs.length; i++) {
       if (tabs[i].id == aId)
         return tabs[i];
    }
    return null;
  },

  getTabForBrowser: function getTabForBrowser(aBrowser) {
    let tabs = this._tabs;
    for (let i = 0; i < tabs.length; i++) {
      if (tabs[i].browser == aBrowser)
        return tabs[i];
    }
    return null;
  },

  getTabForWindow: function getTabForWindow(aWindow) {
    let tabs = this._tabs;
    for (let i = 0; i < tabs.length; i++) {
      if (tabs[i].browser.contentWindow == aWindow)
        return tabs[i];
    }
    return null;
  },

  getBrowserForWindow: function getBrowserForWindow(aWindow) {
    let tabs = this._tabs;
    for (let i = 0; i < tabs.length; i++) {
      if (tabs[i].browser.contentWindow == aWindow)
        return tabs[i].browser;
    }
    return null;
  },

  getBrowserForDocument: function getBrowserForDocument(aDocument) {
    let tabs = this._tabs;
    for (let i = 0; i < tabs.length; i++) {
      if (tabs[i].browser.contentDocument == aDocument)
        return tabs[i].browser;
    }
    return null;
  },

  loadURI: function loadURI(aURI, aBrowser, aParams) {
    aBrowser = aBrowser || this.selectedBrowser;
    if (!aBrowser)
      return;

    aParams = aParams || {};

    let flags = "flags" in aParams ? aParams.flags : Ci.nsIWebNavigation.LOAD_FLAGS_NONE;
    let postData = ("postData" in aParams && aParams.postData) ? aParams.postData : null;
    let referrerURI = "referrerURI" in aParams ? aParams.referrerURI : null;
    let charset = "charset" in aParams ? aParams.charset : null;

    let tab = this.getTabForBrowser(aBrowser);
    if (tab) {
      if ("userSearch" in aParams) tab.userSearch = aParams.userSearch;
    }

    try {
      aBrowser.loadURIWithFlags(aURI, flags, referrerURI, charset, postData);
    } catch(e) {
      if (tab) {
        let message = {
          type: "Content:LoadError",
          tabID: tab.id
        };
        sendMessageToJava(message);
        dump("Handled load error: " + e)
      }
    }
  },

  addTab: function addTab(aURI, aParams) {
    aParams = aParams || {};

    let newTab = new Tab(aURI, aParams);

    if (typeof aParams.tabIndex == "number") {
      this._tabs.splice(aParams.tabIndex, 0, newTab);
    } else {
      this._tabs.push(newTab);
    }

    let selected = "selected" in aParams ? aParams.selected : true;
    if (selected)
      this.selectedTab = newTab;

    let pinned = "pinned" in aParams ? aParams.pinned : false;
    if (pinned) {
      let ss = Cc["@mozilla.org/browser/sessionstore;1"].getService(Ci.nsISessionStore);
      ss.setTabValue(newTab, "appOrigin", aURI);
    }

    let evt = document.createEvent("UIEvents");
    evt.initUIEvent("TabOpen", true, false, window, null);
    newTab.browser.dispatchEvent(evt);

    return newTab;
  },

  // Use this method to close a tab from JS. This method sends a message
  // to Java to close the tab in the Java UI (we'll get a Tab:Closed message
  // back from Java when that happens).
  closeTab: function closeTab(aTab) {
    if (!aTab) {
      Cu.reportError("Error trying to close tab (tab doesn't exist)");
      return;
    }

    let message = {
      type: "Tab:Close",
      tabID: aTab.id
    };
    sendMessageToJava(message);
  },

  _loadWebapp: function(aMessage) {

    this._initRuntime(this._startupStatus, aMessage.url, aUrl => {
      this.manifestUrl = aMessage.url;
      this.addTab(aUrl, { title: aMessage.name });
    });
  },

  // Calling this will update the state in BrowserApp after a tab has been
  // closed in the Java UI.
  _handleTabClosed: function _handleTabClosed(aTab, aShowUndoToast) {
    if (aTab == this.selectedTab)
      this.selectedTab = null;

    let tabIndex = this._tabs.indexOf(aTab);

    let evt = document.createEvent("UIEvents");
    evt.initUIEvent("TabClose", true, false, window, tabIndex);
    aTab.browser.dispatchEvent(evt);

    if (aShowUndoToast) {
      // Get a title for the undo close toast. Fall back to the URL if there is no title.
      let ss = Cc["@mozilla.org/browser/sessionstore;1"].getService(Ci.nsISessionStore);
      let closedTabData = ss.getClosedTabs(window)[0];

      let message;
      let title = closedTabData.entries[closedTabData.index - 1].title;

      if (title) {
        message = Strings.browser.formatStringFromName("undoCloseToast.message", [title], 1);
      } else {
        message = Strings.browser.GetStringFromName("undoCloseToast.messageDefault");
      }

      NativeWindow.toast.show(message, "short", {
        button: {
          icon: "drawable://undo_button_icon",
          label: Strings.browser.GetStringFromName("undoCloseToast.action2"),
          callback: function() {
            UITelemetry.addEvent("undo.1", "toast", null, "closetab");
            ss.undoCloseTab(window, closedTabData);
          }
        }
      });
    }

    aTab.destroy();
    this._tabs.splice(tabIndex, 1);
  },

  // Use this method to select a tab from JS. This method sends a message
```

```
// to Java to select the tab in the Java UI (we'll get a Tab:Selected message
// back from Java when that happens).
selectTab: function selectTab(aTab) {
  if (!aTab) {
    Cu.reportError("Error trying to select tab (tab doesn't exist)");
    return;
  }

  // There's nothing to do if the tab is already selected
  if (aTab == this.selectedTab)
    return;

  let message = {
    type: "Tab:Select",
    tabID: aTab.id
  };
  sendMessageToJava(message);
},

/**
 * Gets an open tab with the given URL.
 *
 * @param  aURL URL to look for
 * @return the tab with the given URL, or null if no such tab exists
 */
getTabWithURL: function getTabWithURL(aURL) {
  let uri = Services.io.newURI(aURL, null, null);
  for (let i = 0; i < this._tabs.length; ++i) {
    let tab = this._tabs[i];
    if (tab.browser.currentURI.equals(uri)) {
      return tab;
    }
  }
  return null;
},

/**
 * If a tab with the given URL already exists, that tab is selected.
 * Otherwise, a new tab is opened with the given URL.
 *
 * @param aURL URL to open
 */
selectOrOpenTab: function selectOrOpenTab(aURL) {
  let tab = this.getTabWithURL(aURL);
  if (tab == null) {
    this.addTab(aURL);
  } else {
    this.selectTab(tab);
  }
},

// This method updates the state in BrowserApp after a tab has been selected
// in the Java UI.
_handleTabSelected: function _handleTabSelected(aTab) {
  this.selectedTab = aTab;

  let evt = document.createEvent("UIEvents");
  evt.initUIEvent("TabSelect", true, false, window, null);
  aTab.browser.dispatchEvent(evt);
},

quit: function quit(aClear = { sanitize: {}, dontSaveSession: false }) {
  // Figure out if there's at least one other browser window around.
  let lastBrowser = true;
  let e = Services.wm.getEnumerator("navigator:browser");
  while (e.hasMoreElements() && lastBrowser) {
    let win = e.getNext();
    if (!win.closed && win != window)
      lastBrowser = false;
  }

  if (lastBrowser) {
    // Let everyone know we are closing the last browser window
    let closingCanceled = Cc["@mozilla.org/supports-PRBool;1"].createInstance(Ci.nsISupportsPRBool);
    Services.obs.notifyObservers(closingCanceled, "browser-lastwindow-close-requested", null);
    if (closingCanceled.data)
      return;

    Services.obs.notifyObservers(null, "browser-lastwindow-close-granted", null);
  }

  // Tell session store to forget about this window
  if (aClear.dontSaveSession) {
    let ss = Cc["@mozilla.org/browser/sessionstore;1"].getService(Ci.nsISessionStore);
    ss.removeWindow(window);
  }

  BrowserApp.sanitize(aClear.sanitize, function() {
    window.QueryInterface(Ci.nsIDOMChromeWindow).minimize();
    window.close();
  });
},

saveAsPDF: function saveAsPDF(aBrowser) {
  // Create the final destination file location
  let fileName = ContentAreaUtils.getDefaultFileName(aBrowser.contentTitle, aBrowser.currentURI, null, null);
  fileName = fileName.trim() + ".pdf";

  let dm = Cc["@mozilla.org/download-manager;1"].getService(Ci.nsIDownloadManager);
  let downloadsDir = dm.defaultDownloadsDirectory;

  let file = downloadsDir.clone();
  file.append(fileName);
  file.createUnique(file.NORMAL_FILE_TYPE, parseInt("666", 8));

  let printSettings = Cc["@mozilla.org/gfx/printsettings-service;1"].getService(Ci.nsIPrintSettingsService).newPrintSettings;
  printSettings.printSilent = true;
  printSettings.showPrintProgress = false;
  printSettings.printBGImages = true;
  printSettings.printBGColors = true;
  printSettings.printToFile = true;
  printSettings.toFileName = file.path;
  printSettings.printFrameType = Ci.nsIPrintSettings.kFramesAsIs;
  printSettings.outputFormat = Ci.nsIPrintSettings.kOutputFormatPDF;

  //XXX we probably need a preference here, the header can be useful
  printSettings.footerStrCenter = "";
  printSettings.footerStrLeft   = "";
  printSettings.footerStrRight  = "";
  printSettings.headerStrCenter = "";
  printSettings.headerStrLeft   = "";
  printSettings.headerStrRight  = "";

  // Create a valid mimeInfo for the PDF
  let ms = Cc["@mozilla.org/mime;1"].getService(Ci.nsIMIMEService);
  let mimeInfo = ms.getFromTypeAndExtension("application/pdf", "pdf");

  let webBrowserPrint = aBrowser.contentWindow.QueryInterface(Ci.nsIInterfaceRequestor)
                                              .getInterface(Ci.nsIWebBrowserPrint);

  let cancelable = {
    cancel: function (aReason) {
      webBrowserPrint.cancel();
    }
  }
  let isPrivate = PrivateBrowsingUtils.isWindowPrivate(aBrowser.contentWindow);
  let download = dm.addDownload(Ci.nsIDownloadManager.DOWNLOAD_TYPE_DOWNLOAD,
                                aBrowser.currentURI,
                                Services.io.newFileURI(file), "", mimeInfo,
                                Date.now() * 1000, null, cancelable, isPrivate);

  webBrowserPrint.print(printSettings, download);
},

notifyPrefObservers: function(aPref) {
  this._prefObservers[aPref].forEach(function(aRequestId) {
    this.getPreferences(aRequestId, [aPref], 1);
  }, this);
},

handlePreferencesRequest: function handlePreferencesRequest(aRequestId,
                                                            aPrefNames,
                                                            aListen) {

  let prefs = [];

  for (let prefName of aPrefNames) {
    let pref = {
      name: prefName,
      type: "",
      value: null
    };

    if (aListen) {
      if (this._prefObservers[prefName])
        this._prefObservers[prefName].push(aRequestId);
      else
        this._prefObservers[prefName] = [ aRequestId ];
      Services.prefs.addObserver(prefName, this, false);
    }

    // These pref names are not "real" pref names.
    // They are used in the setting menu,
    // and these are passed when initializing the setting menu.
    switch (prefName) {
      // The plugin pref is actually two separate prefs, so
```

```
        // we need to handle it differently
        case "plugin.enable":
          pref.type = "string";// Use a string type for java's ListPreference
          pref.value = PluginHelper.getPluginPreference();
          prefs.push(pref);
          continue;
        // Handle master password
        case "privacy.masterpassword.enabled":
          pref.type = "bool";
          pref.value = MasterPassword.enabled;
          prefs.push(pref);
          continue;
        // Handle do-not-track preference
        case "privacy.donottrackheader":
          pref.type = "string";

          let enableDNT = Services.prefs.getBoolPref("privacy.donottrackheader.enabled");
          if (!enableDNT) {
            pref.value = kDoNotTrackPrefState.NO_PREF;
          } else {
            let dntState = Services.prefs.getIntPref("privacy.donottrackheader.value");
            pref.value = (dntState === 0) ? kDoNotTrackPrefState.ALLOW_TRACKING :
                                            kDoNotTrackPrefState.DISALLOW_TRACKING;
          }

          prefs.push(pref);
          continue;
//@line 1267 "/builds/slave/rel-m-beta-and_bld-00000000000/build/mobile/android/chrome/content/browser.js"
        // Crash reporter submit pref must be fetched from nsICrashReporter service.
        case "datareporting.crashreporter.submitEnabled":
          pref.type = "bool";
          pref.value = CrashReporter.submitReports;
          prefs.push(pref);
          continue;
//@line 1274 "/builds/slave/rel-m-beta-and_bld-00000000000/build/mobile/android/chrome/content/browser.js"
      }

      try {
        switch (Services.prefs.getPrefType(prefName)) {
          case Ci.nsIPrefBranch.PREF_BOOL:
            pref.type = "bool";
            pref.value = Services.prefs.getBoolPref(prefName);
            break;
          case Ci.nsIPrefBranch.PREF_INT:
            pref.type = "int";
            pref.value = Services.prefs.getIntPref(prefName);
            break;
          case Ci.nsIPrefBranch.PREF_STRING:
          default:
            pref.type = "string";
            try {
              // Try in case it's a localized string (will throw an exception if not)
              pref.value = Services.prefs.getComplexValue(prefName, Ci.nsIPrefLocalizedString).data;
            } catch (e) {
              pref.value = Services.prefs.getCharPref(prefName);
            }
            break;
        }
      } catch (e) {
        dump("Error reading pref [" + prefName + "]: " + e);
        // preference does not exist; do not send it
        continue;
      }

      // Some Gecko preferences use integers or strings to reference
      // state instead of directly representing the value.
      // Since the Java UI uses the type to determine which ui elements
      // to show and how to handle them, we need to normalize these
      // preferences to the correct type.
      switch (prefName) {
        // (string) index for determining which multiple choice value to display.
        case "browser.chrome.titlebarMode":
        case "network.cookie.cookieBehavior":
        case "font.size.inflation.minTwips":
        case "home.sync.updateMode":
          pref.type = "string";
          pref.value = pref.value.toString();
          break;
      }

      prefs.push(pref);
    }

    sendMessageToJava({
      type: "Preferences:Data",
      requestId: aRequestId,    // opaque request identifier, can be any string/int/whatever
      preferences: prefs
    });
  },

  setPreferences: function setPreferences(aPref) {
    let json = JSON.parse(aPref);

    switch (json.name) {
      // The plugin pref is actually two separate prefs, so
      // we need to handle it differently
      case "plugin.enable":
        PluginHelper.setPluginPreference(json.value);
        return;

      // MasterPassword pref is not real, we just need take action and leave
      case "privacy.masterpassword.enabled":
        if (MasterPassword.enabled)
          MasterPassword.removePassword(json.value);
        else
          MasterPassword.setPassword(json.value);
        return;

      // "privacy.donottrackheader" is not "real" pref name, it's used in the setting menu.
      case "privacy.donottrackheader":
        switch (json.value) {
          // Don't tell anything about tracking me
          case kDoNotTrackPrefState.NO_PREF:
            Services.prefs.setBoolPref("privacy.donottrackheader.enabled", false);
            Services.prefs.clearUserPref("privacy.donottrackheader.value");
            break;
          // Accept tracking me
          case kDoNotTrackPrefState.ALLOW_TRACKING:
            Services.prefs.setBoolPref("privacy.donottrackheader.enabled", true);
            Services.prefs.setIntPref("privacy.donottrackheader.value", 0);
            break;
          // Not accept tracking me
          case kDoNotTrackPrefState.DISALLOW_TRACKING:
            Services.prefs.setBoolPref("privacy.donottrackheader.enabled", true);
            Services.prefs.setIntPref("privacy.donottrackheader.value", 1);
            break;
        }
        return;

      // Enabling or disabling suggestions will prevent future prompts
      case SearchEngines.PREF_SUGGEST_ENABLED:
        Services.prefs.setBoolPref(SearchEngines.PREF_SUGGEST_PROMPTED, true);
        break;

//@line 1374 "/builds/slave/rel-m-beta-and_bld-00000000000/build/mobile/android/chrome/content/browser.js"
      // Crash reporter preference is in a service; set and return.
      case "datareporting.crashreporter.submitEnabled":
        CrashReporter.submitReports = json.value;
        return;
//@line 1379 "/builds/slave/rel-m-beta-and_bld-00000000000/build/mobile/android/chrome/content/browser.js"
      // When sending to Java, we normalized special preferences that use
      // integers and strings to represent booleans. Here, we convert them back
      // to their actual types so we can store them.
      case "browser.chrome.titlebarMode":
      case "network.cookie.cookieBehavior":
      case "font.size.inflation.minTwips":
      case "home.sync.updateMode":
        json.type = "int";
        json.value = parseInt(json.value);
        break;
    }

    switch (json.type) {
      case "bool":
        Services.prefs.setBoolPref(json.name, json.value);
        break;
      case "int":
        Services.prefs.setIntPref(json.name, json.value);
        break;
      default: {
        let pref = Cc["@mozilla.org/pref-localizedstring;1"].createInstance(Ci.nsIPrefLocalizedString);
        pref.data = json.value;
        Services.prefs.setComplexValue(json.name, Ci.nsISupportsString, pref);
        break;
      }
    }
  },

  sanitize: function (aItems, callback) {
    let success = true;

    for (let key in aItems) {
      if (!aItems[key])
        continue;

      key = key.replace("private.data.", "");
      var promises = [];
      switch (key) {
        case "cookies_sessions":
          promises.push(Sanitizer.clearItem("cookies"));
          promises.push(Sanitizer.clearItem("sessions"));
          break;
        default:
          promises.push(Sanitizer.clearItem(key));
      }
    }

    Promise.all(promises).then(function() {
      sendMessageToJava({
        type: "Sanitize:Finished",
        success: true
      });

      if (callback) {
        callback();
      }
    }).catch(function(err) {
      sendMessageToJava({
        type: "Sanitize:Finished",
        error: err,
        success: false
      });

      if (callback) {
        callback();
      }
    })
  },

  getFocusedInput: function(aBrowser, aOnlyInputElements = false) {
    if (!aBrowser)
      return null;

    let doc = aBrowser.contentDocument;
    if (!doc)
      return null;

    let focused = doc.activeElement;
    while (focused instanceof HTMLFrameElement || focused instanceof HTMLIFrameElement) {
      doc = focused.contentDocument;
      focused = doc.activeElement;
    }

    if (focused instanceof HTMLInputElement && focused.mozIsTextField(false))
      return focused;

    if (aOnlyInputElements)
      return null;

    if (focused && (focused instanceof HTMLTextAreaElement || focused.isContentEditable)) {

      if (focused instanceof HTMLBodyElement) {
        // we are putting focus into a contentEditable frame. scroll the frame into
        // view instead of the contentEditable document contained within, because that
        // results in a better user experience
        focused = focused.ownerDocument.defaultView.frameElement;
      }
      return focused;
    }
    return null;
  },

  scrollToFocusedInput: function(aBrowser, aAllowZoom = true) {
    let formHelperMode = Services.prefs.getIntPref("formhelper.mode");
    if (formHelperMode == kFormHelperModeDisabled)
      return;

    let focused = this.getFocusedInput(aBrowser);

    if (focused) {
      let shouldZoom = Services.prefs.getBoolPref("formhelper.autozoom");
      if (formHelperMode == kFormHelperModeDynamic && this.isTablet)
        shouldZoom = false;
      // ZoomHelper.zoomToElement will handle not sending any message if this input is already mostly filling the screen
      ZoomHelper.zoomToElement(focused, -1, false,
          aAllowZoom && shouldZoom && !ViewportHandler.getViewportMetadata(aBrowser.contentWindow).isSpecified);
    }
  },

  observe: function(aSubject, aTopic, aData) {
    let browser = this.selectedBrowser;

    switch (aTopic) {

      case "Session:Back":
        browser.goBack();
        break;

      case "Session:Forward":
        browser.goForward();
        break;

      case "Session:Reload": {
        let flags = Ci.nsIWebNavigation.LOAD_FLAGS_BYPASS_PROXY | Ci.nsIWebNavigation.LOAD_FLAGS_BYPASS_CACHE;

        // Check to see if this is a message to enable/disable mixed content blocking.
        if (aData) {
          let allowMixedContent = JSON.parse(aData).allowMixedContent;
          if (allowMixedContent) {
            // Set a flag to disable mixed content blocking.
            flags = Ci.nsIWebNavigation.LOAD_FLAGS_ALLOW_MIXED_CONTENT;
          } else {
            // Set mixedContentChannel to null to re-enable mixed content blocking.
            let docShell = browser.webNavigation.QueryInterface(Ci.nsIDocShell);
            docShell.mixedContentChannel = null;
          }
        }

        // Try to use the session history to reload so that framesets are
        // handled properly. If the window has no session history, fall back
        // to using the web navigation's reload method.
        let webNav = browser.webNavigation;
        try {
          let sh = webNav.sessionHistory;
          if (sh)
            webNav = sh.QueryInterface(Ci.nsIWebNavigation);
        } catch (e) {}
        webNav.reload(flags);
        break;
      }

      case "Session:Stop":
        browser.stop();
        break;

      case "Session:ShowHistory": {
        let data = JSON.parse(aData);
        this.showHistory(data.fromIndex, data.toIndex, data.selIndex);
        break;
      }

      case "Tab:Load": {
        let data = JSON.parse(aData);
        let url = data.url;
        let flags;

        if (!data.engine && /^\w+$/.test(url.trim())) {
          // If the query is a single word and we're not using a search engine,
          // force a search (see bug 993705; workaround for bug 693808).
          url = URIFixup.keywordToURI(url).spec;
        } else {
          flags |= Ci.nsIWebNavigation.LOAD_FLAGS_ALLOW_THIRD_PARTY_FIXUP |
                   Ci.nsIWebNavigation.LOAD_FLAGS_FIXUP_SCHEME_TYPOS;
        }

        // Pass LOAD_FLAGS_DISALLOW_INHERIT_OWNER to prevent any loads from
        // inheriting the currently loaded document's principal.
        if (data.userEntered) {
          flags |= Ci.nsIWebNavigation.LOAD_FLAGS_DISALLOW_INHERIT_OWNER;
        }

        let delayLoad = ("delayLoad" in data) ? data.delayLoad : false;
        let params = {
          selected: ("selected" in data) ? data.selected : !delayLoad,
          parentId: ("parentId" in data) ? data.parentId : -1,
          flags: flags,
          tabID: data.tabID,
          isPrivate: (data.isPrivate === true),
          pinned: (data.pinned === true),
          delayLoad: (delayLoad === true),
          desktopMode: (data.desktopMode === true)
        };

        if (data.engine) {
          let engine = Services.search.getEngineByName(data.engine);
          if (engine) {
            params.userSearch = url;
            let submission = engine.getSubmission(url);
            url = submission.uri.spec;
            params.postData = submission.postData;
          }
        }

        if (data.newTab) {
          this.addTab(url, params);
        } else {
          if (data.tabId) {
            // Use a specific browser instead of the selected browser, if it exists
            let specificBrowser = this.getTabForId(data.tabId).browser;
            if (specificBrowser)
              browser = specificBrowser;
          }
          this.loadURI(url, browser, params);
        }
        break;
      }

      case "Tab:Selected":
        this._handleTabSelected(this.getTabForId(parseInt(aData)));
        break;

      case "Tab:Closed": {
        let data = JSON.parse(aData);
        this._handleTabClosed(this.getTabForId(data.tabId), data.showUndoToast);
        break;
      }

      case "keyword-search":
        // This event refers to a search via the URL bar, not a bookmarks
        // keyword search. Note that this code assumes that the user can only
        // perform a keyword search on the selected tab.
        this.selectedTab.userSearch = aData;

        // Don't store queries in private browsing mode.
        let isPrivate = PrivateBrowsingUtils.isWindowPrivate(this.selectedTab.browser.contentWindow);
        let query = isPrivate ? "" : aData;

        let engine = aSubject.QueryInterface(Ci.nsISearchEngine);
        sendMessageToJava({
          type: "Search:Keyword",
          identifier: engine.identifier,
          name: engine.name,
          query: query
        });
        break;

      case "Browser:Quit":
        // Add-ons like QuitNow and CleanQuit provide aData as an empty-string ("").
        // Pass undefined to invoke the methods default parms.
        this.quit(aData ? JSON.parse(aData) : undefined);
        break;

      case "SaveAs:PDF":
        this.saveAsPDF(browser);
        break;

      case "Preferences:Set":
        this.setPreferences(aData);
        break;

      case "ScrollTo:FocusedInput":
        // these messages come from a change in the viewable area and not user interaction
        // we allow scrolling to the selected input, but not zooming the page
        this.scrollToFocusedInput(browser, false);
        break;

      case "Sanitize:ClearData":
        this.sanitize(JSON.parse(aData));
        break;

      case "FullScreen:Exit":
        browser.contentDocument.mozCancelFullScreen();
        break;

      case "Viewport:Change":
        if (this.isBrowserContentDocumentDisplayed())
          this.selectedTab.setViewport(JSON.parse(aData));
        break;

      case "Viewport:Flush":
        this.contentDocumentChanged();
        break;

      case "Passwords:Init": {
        let storage = Cc["@mozilla.org/login-manager/storage/mozStorage;1"].
                      getService(Ci.nsILoginManagerStorage);
        storage.initialize();
        Services.obs.removeObserver(this, "Passwords:Init");
        break;
      }

      case "FormHistory:Init": {
        // Force creation/upgrade of formhistory.sqlite
        FormHistory.count({});
        Services.obs.removeObserver(this, "FormHistory:Init");
        break;
      }

      case "sessionstore-state-purge-complete":
        sendMessageToJava({ type: "Session:StatePurged" });
        break;

      case "gather-telemetry":
        sendMessageToJava({ type: "Telemetry:Gather" });
        break;

      case "Viewport:FixedMarginsChanged":
        gViewportMargins = JSON.parse(aData);
        this.selectedTab.updateViewportSize(gScreenWidth);
        break;

      case "nsPref:changed":
        this.notifyPrefObservers(aData);
        break;

      case "webapps-runtime-install":
        WebappManager.install(JSON.parse(aData), aSubject);
        break;

      case "webapps-runtime-install-package":
        WebappManager.installPackage(JSON.parse(aData), aSubject);
        break;

      case "webapps-ask-install":
        WebappManager.askInstall(JSON.parse(aData));
        break;

      case "webapps-launch": {
        WebappManager.launch(JSON.parse(aData));
        break;
      }

      case "webapps-runtime-uninstall": {
        WebappManager.uninstall(JSON.parse(aData), aSubject);
        break;
      }

      case "Webapps:AutoInstall":
        WebappManager.autoInstall(JSON.parse(aData));
        break;

      case "Webapps:Load":
        this._loadWebapp(JSON.parse(aData));
        break;

      case "Webapps:AutoUninstall":
        WebappManager.autoUninstall(JSON.parse(aData));
        break;

      case "Locale:Changed":
        if (aData) {
          // The value provided to Locale:Changed should be a BCP47 language tag
          // understood by Gecko -- for example, "es-ES" or "de".
          console.log("Locale:Changed: " + aData);
          Services.prefs.setCharPref("general.useragent.locale", aData);
        } else {
          // Resetting.
          console.log("Switching to system locale.");
          Services.prefs.clearUserPref("general.useragent.locale");
        }

        Services.prefs.setBoolPref("intl.locale.matchOS", !aData);

        // Ensure that this choice is immediately persisted, because
        // Gecko won't be told again if it forgets.
        Services.prefs.savePrefFile(null);

        // Blow away the string cache so that future lookups get the
        // correct locale.
        Services.strings.flushBundles();
        break;

      default:
        dump('BrowserApp.observe: unexpected topic "' + aTopic + '"\n');
        break;

    }
  },

  get defaultBrowserWidth() {
    delete this.defaultBrowserWidth;
    let width = Services.prefs.getIntPref("browser.viewport.desktopWidth");
    return this.defaultBrowserWidth = width;
  },

  get layersTileWidth() {
    delete this.layersTileWidth;
    let width = Services.prefs.getIntPref("layers.tile-width");
    return this.layersTileWidth = width;
  },

  get layersTileHeight() {
    delete this.layersTileHeight;
    let height = Services.prefs.getIntPref("layers.tile-height");
    return this.layersTileHeight = height;
  },

  // nsIAndroidBrowserApp
  getBrowserTab: function(tabId) {
    return this.getTabForId(tabId);
  },

  getUITelemetryObserver: function() {
    return UITelemetry;
  },

  getPreferences: function getPreferences(requestId, prefNames, count) {
    this.handlePreferencesRequest(requestId, prefNames, false);
  },

  observePreferences: function observePreferences(requestId, prefNames, count) {
    this.handlePreferencesRequest(requestId, prefNames, true);
  },

  removePreferenceObservers: function removePreferenceObservers(aRequestId) {
    let newPrefObservers = [];
    for (let prefName in this._prefObservers) {
      let requestIds = this._prefObservers[prefName];
      // Remove the requestID from the preference handlers
      let i = requestIds.indexOf(aRequestId);
      if (i >= 0) {
        requestIds.splice(i, 1);
      }

      // If there are no more request IDs, remove the observer
      if (requestIds.length == 0) {
        Services.prefs.removeObserver(prefName, this);
      } else {
        newPrefObservers[prefName] = requestIds;
      }
    }
    this._prefObservers = newPrefObservers;
  },

  // This method will print a list from fromIndex to toIndex, optionally
  // selecting selIndex(if fromIndex<=selIndex<=toIndex)
  showHistory: function(fromIndex, toIndex, selIndex) {
    let browser = this.selectedBrowser;
    let hist = browser.sessionHistory;
    let listitems = [];
    for (let i = toIndex; i >= fromIndex; i--) {
      let entry = hist.getEntryAtIndex(i, false);
      let item = {
        label: entry.title || entry.URI.spec,
        selected: (i == selIndex)
      };
      listitems.push(item);
    }

    let p = new Prompt({
      window: browser.contentWindow
    }).setSingleChoiceItems(listitems).show(function(data) {
        let selected = data.button;
        if (selected == -1)
          return;

        browser.gotoIndex(toIndex-selected);
    });
  },
};

var NativeWindow = {
  init: function() {
    Services.obs.addObserver(this, "Menu:Clicked", false);
    Services.obs.addObserver(this, "PageActions:Clicked", false);
    Services.obs.addObserver(this, "PageActions:LongClicked", false);
    Services.obs.addObserver(this, "Doorhanger:Reply", false);
    Services.obs.addObserver(this, "Toast:Click", false);
    Services.obs.addObserver(this, "Toast:Hidden", false);
    this.contextmenus.init();
  },

  uninit: function() {
    Services.obs.removeObserver(this, "Menu:Clicked");
    Services.obs.removeObserver(this, "PageActions:Clicked");
    Services.obs.removeObserver(this, "PageActions:LongClicked");
    Services.obs.removeObserver(this, "Doorhanger:Reply");
    Services.obs.removeObserver(this, "Toast:Click", false);
    Services.obs.removeObserver(this, "Toast:Hidden", false);
    this.contextmenus.uninit();
  },

  loadDex: function(zipFile, implClass) {
    sendMessageToJava({
      type: "Dex:Load",
      zipfile: zipFile,
      impl: implClass || "Main"
    });
  },

  unloadDex: function(zipFile) {
    sendMessageToJava({
      type: "Dex:Unload",
      zipfile: zipFile
    });
  },

  toast: {
    _callbacks: {},
    show: function(aMessage, aDuration, aOptions) {
      let msg = {
        type: "Toast:Show",
        message: aMessage,
        duration: aDuration
      };

      if (aOptions && aOptions.button) {
        msg.button = {
          id: uuidgen.generateUUID().toString(),
        };

        // null is badly handled by the receiver, so try to avoid including nulls.
        if (aOptions.button.label) {
          msg.button.label = aOptions.button.label;
        }

        if (aOptions.button.icon) {
          // If the caller specified a button, make sure we convert any chrome urls
          // to jar:jar urls so that the frontend can show them
          msg.button.icon = resolveGeckoURI(aOptions.button.icon);
        };

        this._callbacks[msg.button.id] = aOptions.button.callback;
      }

      sendMessageToJava(msg);
    }
  },

  pageactions: {
    _items: { },
    add: function(aOptions) {
      let id = uuidgen.generateUUID().toString();
      sendMessageToJava({
        type: "PageActions:Add",
        id: id,
        title: aOptions.title,
        icon: resolveGeckoURI(aOptions.icon),
        important: "important" in aOptions ? aOptions.important : false
      });
      this._items[id] = {
        clickCallback: aOptions.clickCallback,
        longClickCallback: aOptions.longClickCallback
      };
      return id;
    },
    remove: function(id) {
      sendMessageToJava({
        type: "PageActions:Remove",
        id: id
      });
      delete this._items[id];
    }
```

```
},

menu: {
  _callbacks: [],
  _menuId: 1,
  toolsMenuID: -1,
  add: function() {
    let options;
    if (arguments.length == 1) {
      options = arguments[0];
    } else if (arguments.length == 3) {
        options = {
          name: arguments[0],
          icon: arguments[1],
          callback: arguments[2]
        };
    } else {
       throw "Incorrect number of parameters";
    }

    options.type = "Menu:Add";
    options.id = this._menuId;

    sendMessageToJava(options);
    this._callbacks[this._menuId] = options.callback;
    this._menuId++;
    return this._menuId - 1;
  },

  remove: function(aId) {
    sendMessageToJava({ type: "Menu:Remove", id: aId });
  },

  update: function(aId, aOptions) {
    if (!aOptions)
      return;

    sendMessageToJava({
      type: "Menu:Update",
      id: aId,
      options: aOptions
    });
  }
},

doorhanger: {
  _callbacks: {},
  _callbacksId: 0,
  _promptId: 0,

/**
 * @param aOptions
 *        An options JavaScript object holding additional properties for the
 *        notification. The following properties are currently supported:
 *        persistence: An integer. The notification will not automatically
 *                     dismiss for this many page loads. If persistence is set
 *                     to -1, the doorhanger will never automatically dismiss.
 *        persistWhileVisible:
 *                     A boolean. If true, a visible notification will always
 *                     persist across location changes.
 *        timeout:     A time in milliseconds. The notification will not
 *                     automatically dismiss before this time.
 *        checkbox:    A string to appear next to a checkbox under the notification
 *                     message. The button callback functions will be called with
 *                     the checked state as an argument.
 */
  show: function(aMessage, aValue, aButtons, aTabID, aOptions) {
    if (aButtons == null) {
      aButtons = [];
    }

    aButtons.forEach((function(aButton) {
      this._callbacks[this._callbacksId] = { cb: aButton.callback, prompt: this._promptId };
      aButton.callback = this._callbacksId;
      this._callbacksId++;
    }).bind(this));

    this._promptId++;
    let json = {
      type: "Doorhanger:Add",
      message: aMessage,
      value: aValue,
      buttons: aButtons,
      // use the current tab if none is provided
      tabID: aTabID || BrowserApp.selectedTab.id,
      options: aOptions || {}
    };
    sendMessageToJava(json);
  },

  hide: function(aValue, aTabID) {
    sendMessageToJava({
      type: "Doorhanger:Remove",
      value: aValue,
      tabID: aTabID
    });
  }
},

observe: function(aSubject, aTopic, aData) {
  if (aTopic == "Menu:Clicked") {
    if (this.menu._callbacks[aData])
      this.menu._callbacks[aData]();
  } else if (aTopic == "PageActions:Clicked") {
      if (this.pageactions._items[aData].clickCallback)
        this.pageactions._items[aData].clickCallback();
  } else if (aTopic == "PageActions:LongClicked") {
      if (this.pageactions._items[aData].longClickCallback)
        this.pageactions._items[aData].longClickCallback();
  } else if (aTopic == "Toast:Click") {
    if (this.toast._callbacks[aData]) {
      this.toast._callbacks[aData]();
      delete this.toast._callbacks[aData];
    }
  } else if (aTopic == "Toast:Hidden") {
    if (this.toast._callbacks[aData])
      delete this.toast._callbacks[aData];
  } else if (aTopic == "Doorhanger:Reply") {
    let data = JSON.parse(aData);
    let reply_id = data["callback"];

    if (this.doorhanger._callbacks[reply_id]) {
      // Pass the value of the optional checkbox to the callback
      let checked = data["checked"];
      this.doorhanger._callbacks[reply_id].cb(checked, data.inputs);

      let prompt = this.doorhanger._callbacks[reply_id].prompt;
      for (let id in this.doorhanger._callbacks) {
        if (this.doorhanger._callbacks[id].prompt == prompt) {
          delete this.doorhanger._callbacks[id];
        }
      }
    }
  }
},
contextmenus: {
  items: {}, //  a list of context menu items that we may show
  DEFAULT_HTML5_ORDER: -1, // Sort order for HTML5 context menu items

  init: function() {
    Services.obs.addObserver(this, "Gesture:LongPress", false);
  },

  uninit: function() {
    Services.obs.removeObserver(this, "Gesture:LongPress");
  },

  add: function() {
    let args;
    if (arguments.length == 1) {
      args = arguments[0];
    } else if (arguments.length == 3) {
      args = {
        label : arguments[0],
        selector: arguments[1],
        callback: arguments[2]
      };
    } else {
      throw "Incorrect number of parameters";
    }

    if (!args.label)
      throw "Menu items must have a name";

    let cmItem = new ContextMenuItem(args);
    this.items[cmItem.id] = cmItem;
    return cmItem.id;
  },

  remove: function(aId) {
    delete this.items[aId];
  },

  SelectorContext: function(aSelector) {
    return {
      matches: function(aElt) {
```

```
      if (aElt.mozMatchesSelector)
        return aElt.mozMatchesSelector(aSelector);
      return false;
    }
  };
},

linkOpenableNonPrivateContext: {
  matches: function linkOpenableNonPrivateContextMatches(aElement) {
    let doc = aElement.ownerDocument;
    if (!doc || PrivateBrowsingUtils.isWindowPrivate(doc.defaultView)) {
      return false;
    }

    return NativeWindow.contextmenus.linkOpenableContext.matches(aElement);
  }
},

linkOpenableContext: {
  matches: function linkOpenableContextMatches(aElement) {
    let uri = NativeWindow.contextmenus._getLink(aElement);
    if (uri) {
      let scheme = uri.scheme;
      let dontOpen = /^(javascript|mailto|news|snews|tel)$/;
      return (scheme && !dontOpen.test(scheme));
    }
    return false;
  }
},

linkCopyableContext: {
  matches: function linkCopyableContextMatches(aElement) {
    let uri = NativeWindow.contextmenus._getLink(aElement);
    if (uri) {
      let scheme = uri.scheme;
      let dontCopy = /^(mailto|tel)$/;
      return (scheme && !dontCopy.test(scheme));
    }
    return false;
  }
},

linkShareableContext: {
  matches: function linkShareableContextMatches(aElement) {
    let uri = NativeWindow.contextmenus._getLink(aElement);
    if (uri) {
      let scheme = uri.scheme;
      let dontShare = /^(about|chrome|file|javascript|mailto|resource|tel)$/;
      return (scheme && !dontShare.test(scheme));
    }
    return false;
  }
},

linkBookmarkableContext: {
  matches: function linkBookmarkableContextMatches(aElement) {
    let uri = NativeWindow.contextmenus._getLink(aElement);
    if (uri) {
      let scheme = uri.scheme;
      let dontBookmark = /^(mailto|tel)$/;
      return (scheme && !dontBookmark.test(scheme));
    }
    return false;
  }
},

emailLinkContext: {
  matches: function emailLinkContextMatches(aElement) {
    let uri = NativeWindow.contextmenus._getLink(aElement);
    if (uri)
      return uri.schemeIs("mailto");
    return false;
  }
},

phoneNumberLinkContext: {
  matches: function phoneNumberLinkContextMatches(aElement) {
    let uri = NativeWindow.contextmenus._getLink(aElement);
    if (uri)
      return uri.schemeIs("tel");
    return false;
  }
},

imageLocationCopyableContext: {
  matches: function imageLinkCopyableContextMatches(aElement) {
    return (aElement instanceof Ci.nsIImageLoadingContent && aElement.currentURI);
  }
},

imageSaveableContext: {
  matches: function imageSaveableContextMatches(aElement) {
    if (aElement instanceof Ci.nsIImageLoadingContent && aElement.currentURI) {
      // The image must be loaded to allow saving
      let request = aElement.getRequest(Ci.nsIImageLoadingContent.CURRENT_REQUEST);
      return (request && (request.imageStatus & request.STATUS_SIZE_AVAILABLE));
    }
    return false;
  }
},

mediaSaveableContext: {
  matches: function mediaSaveableContextMatches(aElement) {
    return (aElement instanceof HTMLVideoElement ||
            aElement instanceof HTMLAudioElement);
  }
},

mediaContext: function(aMode) {
  return {
    matches: function(aElt) {
      if (aElt instanceof Ci.nsIDOMHTMLMediaElement) {
        let hasError = aElt.error != null || aElt.networkState == aElt.NETWORK_NO_SOURCE;
        if (hasError)
          return false;

        let paused = aElt.paused || aElt.ended;
        if (paused && aMode == "media-paused")
          return true;
        if (!paused && aMode == "media-playing")
          return true;
        let controls = aElt.controls;
        if (!controls && aMode == "media-hidingcontrols")
          return true;

        let muted = aElt.muted;
        if (muted && aMode == "media-muted")
          return true;
        else if (!muted && aMode == "media-unmuted")
          return true;
      }
      return false;
    }
  };
},

/* Holds a WeakRef to the original target element this context menu was shown for.
 * Most API's will have to walk up the tree from this node to find the correct element
 * to act on
 */
get _target() {
  if (this._targetRef)
    return this._targetRef.get();
  return null;
},

set _target(aTarget) {
  if (aTarget)
    this._targetRef = Cu.getWeakReference(aTarget);
  else this._targetRef = null;
},

get defaultContext() {
  delete this.defaultContext;
  return this.defaultContext = Strings.browser.GetStringFromName("browser.menu.context.default");
},

/* Gets menuitems for an arbitrary node
 * Parameters:
 *   element - The element to look at. If this element has a contextmenu attribute, the
 *             corresponding contextmenu will be used.
 */
_getHTMLContextMenuItemsForElement: function(element) {
  let htmlMenu = element.contextMenu;
  if (!htmlMenu) {
    return [];
  }

  htmlMenu.QueryInterface(Components.interfaces.nsIHTMLMenu);
  htmlMenu.sendShowEvent();

  return this._getHTMLContextMenuItemsForMenu(htmlMenu, element);
},

/* Add a menuitem for an HTML <menu> node
 * Parameters:
```

```
 *   menu - The <menu> element to iterate through for menuitems
 *   target - The target element these context menu items are attached to
 */
_getHTMLContextMenuItemsForMenu: function(menu, target) {
  let items = [];
  for (let i = 0; i < menu.childNodes.length; i++) {
    let elt = menu.childNodes[i];
    if (!elt.label)
      continue;

    items.push(new HTMLContextMenuItem(elt, target));
  }

  return items;
},

// Searches the current list of menuitems to show for any that match this id
_findMenuItem: function(aId) {
  if (!this.menus) {
    return null;
  }

  for (let context in this.menus) {
    let menu = this.menus[context];
    for (let i = 0; i < menu.length; i++) {
      if (menu[i].id === aId) {
        return menu[i];
      }
    }
  }
  return null;
},

// Returns true if there are any context menu items to show
shouldShow: function() {
  for (let context in this.menus) {
    let menu = this.menus[context];
    if (menu.length > 0) {
      return true;
    }
  }
  return false;
},

/* Returns a label to be shown in a tabbed ui if there are multiple "contexts". For instance, if this
 * is an image inside an <a> tag, we may have a "link" context and an "image" one.
 */
_getContextType: function(element) {
  // For anchor nodes, we try to use the scheme to pick a string
  if (element instanceof Ci.nsIDOMHTMLAnchorElement) {
    let uri = this.makeURI(this._getLinkURL(element));
    try {
      return Strings.browser.GetStringFromName("browser.menu.context." + uri.scheme);
    } catch(ex) { }
  }

  // Otherwise we try the nodeName
  try {
    return Strings.browser.GetStringFromName("browser.menu.context." + element.nodeName.toLowerCase());
  } catch(ex) { }

  // Fallback to the default
  return this.defaultContext;
},

// Adds context menu items added through the add-on api
_getNativeContextMenuItems: function(element, x, y) {
  let res = [];
  for (let itemId of Object.keys(this.items)) {
    let item = this.items[itemId];

    if (!this._findMenuItem(item.id) && item.matches(element, x, y)) {
      res.push(item);
    }
  }

  return res;
},

_findTarget: function(x, y) {
  let isDescendant = function(parent, child) {
    let node = child;
    while (node) {
      if (node === parent) {
        return true;
      }

      node = node.parentNode;
    }

    return false;
  };

  let target = BrowserEventHandler._highlightElement;
  let touchTarget = ElementTouchHelper.anyElementFromPoint(x, y);

  // If we have a highlighted element that has a click handler, we want to ensure our target is inside it
  if (isDescendant(target, touchTarget)) {
    target = touchTarget;
  } else if (!target) {
    // Otherwise, let's try to find something clickable
    target = ElementTouchHelper.elementFromPoint(x, y);

    // If that failed, we'll just fall back to anything under the user's finger
    if (!target) {
      target = touchTarget;
    }
  }

  return target;
},

/* Checks if there are context menu items to show, and if it finds them
 * sends a contextmenu event to content. We also send showing events to
 * any html5 context menus we are about to show, and fire some local notifications
 * for chrome consumers to do lazy menuitem construction
 */
_sendToContent: function(x, y) {
  let target = this._findTarget(x, y);
  if (!target)
    return;

  this._target = target;

  Services.obs.notifyObservers(null, "before-build-contextmenu", "");
  this._buildMenu(x, y);

  // only send the contextmenu event to content if we are planning to show a context menu (i.e. not on every long tap)
  if (this.shouldShow()) {
    let event = target.ownerDocument.createEvent("MouseEvent");
    event.initMouseEvent("contextmenu", true, true, target.defaultView,
                         0, x, y, x, y, false, false, false, false,
                         0, null);
    target.ownerDocument.defaultView.addEventListener("contextmenu", this, false);
    target.dispatchEvent(event);
  } else {
    this.menus = null;
    Services.obs.notifyObservers({target: target, x: x, y: y}, "context-menu-not-shown", "");

    if (SelectionHandler.canSelect(target)) {
      if (!SelectionHandler.startSelection(target, {
        mode: SelectionHandler.SELECT_AT_POINT,
        x: x,
        y: y
      })) {
        SelectionHandler.attachCaret(target);
      }
    }
  }
},

// Returns a title for a context menu. If no title attribute exists, will fall back to looking for a url
_getTitle: function(node) {
  if (node.hasAttribute && node.hasAttribute("title")) {
    return node.getAttribute("title");
  }
  return this._getUrl(node);
},

// Returns a url associated with a node
_getUrl: function(node) {
  if ((node instanceof Ci.nsIDOMHTMLAnchorElement && node.href) ||
      (node instanceof Ci.nsIDOMHTMLAreaElement && node.href)) {
    return this._getLinkURL(node);
  } else if (node instanceof Ci.nsIImageLoadingContent && node.currentURI) {
    return node.currentURI.spec;
  } else if (node instanceof Ci.nsIDOMHTMLMediaElement) {
    return (node.currentSrc || node.src);
  }

  return "";
},

// Adds an array of menuitems to the current list of items to show, in the correct context
_addMenuItems: function(items, context) {
    if (!this.menus[context]) {
      this.menus[context] = [];
    }
    this.menus[context] = this.menus[context].concat(items);
},

/* Does the basic work of building a context menu to show. Will combine HTML and Native
 * context menus items, as well as sorting menuitems into different menus based on context.
 */
_buildMenu: function(x, y) {
  // now walk up the tree and for each node look for any context menu items that apply
  let element = this._target;

  // this.menus holds a hashmap of "contexts" to menuitems associated with that context
  // For instance, if the user taps an image inside a link, we'll have something like:
  // {
  //   link:  [ ContextMenuItem, ContextMenuItem ]
  //   image: [ ContextMenuItem, ContextMenuItem ]
  // }
  this.menus = {};

  while (element) {
    let context = this._getContextType(element);

    // First check for any html5 context menus that might exist...
    var items = this._getHTMLContextMenuItemsForElement(element);
    if (items.length > 0) {
      this._addMenuItems(items, context);
    }

    // then check for any context menu items registered in the ui.
    items = this._getNativeContextMenuItems(element, x, y);
    if (items.length > 0) {
      this._addMenuItems(items, context);
    }

    // walk up the tree and find more items to show
    element = element.parentNode;
  }
},

// Actually shows the native context menu by passing a list of context menu items to
// show to the Java.
_show: function(aEvent) {
  let popupNode = this._target;
  this._target = null;
  if (aEvent.defaultPrevented || !popupNode) {
    return;
  }
  this._innerShow(popupNode, aEvent.clientX, aEvent.clientY);
},

// Walks the DOM tree to find a title from a node
_findTitle: function(node) {
  let title = "";
  while(node && !title) {
    title = this._getTitle(node);
    node = node.parentNode;
  }
  return title;
},

/* Reformats the list of menus to show into an object that can be sent to Prompt.jsm
 * If there is one menu, will return a flat array of menuitems. If there are multiple
 * menus, will return an array with appropriate tabs/items inside it. i.e. :
 * [
 *    { label: "link", items: [...] },
 *    { label: "image", items: [...] }
 * ]
 */
_reformatList: function(target) {
  let contexts = Object.keys(this.menus);

  if (contexts.length === 1) {
    // If there's only one context, we'll only show a single flat single select list
    return this._reformatMenuItems(target, this.menus[contexts[0]]);
  }

  // If there are multiple contexts, we'll only show a tabbed ui with multiple lists
  return this._reformatListAsTabs(target, this.menus);
},

/* Reformats the list of menus to show into an object that can be sent to Prompt.jsm's
 * addTabs method. i.e. :
 * { link: [...], image: [...] } becomes
 * [ { label: "link", items: [...] } ]
 *
 * Also reformats items and resolves any parmaeters that aren't known until display time
 * (for instance Helper app menu items adjust their title to reflect what Helper App can be used for this link).
 */
_reformatListAsTabs: function(target, menus) {
  let itemArray = [];

  // Sort the keys so that "link" is always first
  let contexts = Object.keys(this.menus);
  contexts.sort((context1, context2) => {
    if (context1 === this.defaultContext) {
      return -1;
    } else if (context2 === this.defaultContext) {
      return 1;
    }
    return 0;
  });

  contexts.forEach(context => {
    itemArray.push({
      label: context,
      items: this._reformatMenuItems(target, menus[context])
    });
  });

  return itemArray;
},

/* Reformats an array of ContextMenuItems into an array that can be handled by Prompt.jsm. Also reformats items
 * and resolves any parmaeters that aren't known until display time
 * (for instance Helper app menu items adjust their title to reflect what Helper App can be used for this link).
 */
_reformatMenuItems: function(target, menuitems) {
  let itemArray = [];

  for (let i = 0; i < menuitems.length; i++) {
    let t = target;
    while(t) {
      if (menuitems[i].matches(t)) {
        let val = menuitems[i].getValue(t);

        // hidden menu items will return null from getValue
        if (val) {
          itemArray.push(val);
          break;
        }
      }

      t = t.parentNode;
    }
  }

  return itemArray;
},

// Called where we're finally ready to actually show the contextmenu. Sorts the items and shows a prompt.
_innerShow: function(target, x, y) {
  Haptic.performSimpleAction(Haptic.LongPress);

  // spin through the tree looking for a title for this context menu
  let title = this._findTitle(target);

  for (let context in this.menus) {
    let menu = this.menus[context];
    menu.sort((a,b) => {
      if (a.order === b.order) {
        return 0;
      }
      return (a.order > b.order) ? 1 : -1;
    });
  }

  let useTabs = Object.keys(this.menus).length > 1;
  let prompt = new Prompt({
    window: target.ownerDocument.defaultView,
    title: useTabs ? undefined : title
  });

  let items = this._reformatList(target);
  if (useTabs) {
    prompt.addTabs({
      id: "tabs",
      items: items
    });
  } else {
    prompt.setSingleChoiceItems(items);
  }

  prompt.show(this._promptDone.bind(this, target, x, y, items));
},
  // Called when the contextmenu prompt is closed
  _promptDone: function(target, x, y, items, data) {
    if (data.button == -1) {
      // Prompt was cancelled, or an ActionView was used.
      return;
    }

    let selectedItemId;
    if (data.tabs) {
      let menu = items[data.tabs.tab];
      selectedItemId = menu.items[data.tabs.item].id;
    } else {
      selectedItemId = items[data.list[0]].id
    }

    let selectedItem = this._findMenuItem(selectedItemId);
    this.menus = null;

    if (!selectedItem || !selectedItem.matches || !selectedItem.callback) {
      return;
    }

    // for menuitems added using the native UI, pass the dom element that matched that item to the callback
    while (target) {
      if (selectedItem.matches(target, x, y)) {
        selectedItem.callback(target, x, y);
        break;
      }
      target = target.parentNode;
    }
  },

  // Called when the contextmenu is done propagating to content. If the event wasn't cancelled, will show a contextmenu.
  handleEvent: function(aEvent) {
    BrowserEventHandler._cancelTapHighlight();
    aEvent.target.ownerDocument.defaultView.removeEventListener("contextmenu", this, false);
    this._show(aEvent);
  },

  // Called when a long press is observed in the native Java frontend. Will start the process of generating/showing a contextmenu.
  observe: function(aSubject, aTopic, aData) {
    let data = JSON.parse(aData);
    // content gets first crack at cancelling context menus
    this._sendToContent(data.x, data.y);
  },

  // XXX - These are stolen from Util.js, we should remove them if we bring it back
  makeURLAbsolute: function makeURLAbsolute(base, url) {
    // Note:  makeURI() will throw if url is not a valid URI
    return this.makeURI(url, null, this.makeURI(base)).spec;
  },

  makeURI: function makeURI(aURL, aOriginCharset, aBaseURI) {
    return Services.io.newURI(aURL, aOriginCharset, aBaseURI);
  },

  _getLink: function(aElement) {
    if (aElement.nodeType == Ci.nsIDOMNode.ELEMENT_NODE &&
        ((aElement instanceof Ci.nsIDOMHTMLAnchorElement && aElement.href) ||
        (aElement instanceof Ci.nsIDOMHTMLAreaElement && aElement.href) ||
        aElement instanceof Ci.nsIDOMHTMLLinkElement ||
        aElement.getAttributeNS(kXLinkNamespace, "type") == "simple")) {
      try {
        let url = this._getLinkURL(aElement);
        return Services.io.newURI(url, null, null);
      } catch (e) {}
    }
    return null;
  },

  _disableInGuest: function _disableInGuest(selector) {
    return {
      matches: function _disableInGuestMatches(aElement, aX, aY) {
        if (BrowserApp.isGuest)
          return false;
        return selector.matches(aElement, aX, aY);
      }
    };
  },

  _getLinkURL: function ch_getLinkURL(aLink) {
    let href = aLink.href;
    if (href)
      return href;

    href = aLink.getAttributeNS(kXLinkNamespace, "href");
    if (!href || !href.match(/\S/)) {
      // Without this we try to save as the current doc,
      // for example, HTML case also throws if empty
      throw "Empty href";
    }

    return this.makeURLAbsolute(aLink.baseURI, href);
  },

  _copyStringToDefaultClipboard: function(aString) {
    let clipboard = Cc["@mozilla.org/widget/clipboardhelper;1"].getService(Ci.nsIClipboardHelper);
    clipboard.copyString(aString);
  },

  _shareStringWithDefault: function(aSharedString, aTitle) {
    let sharing = Cc["@mozilla.org/uriloader/external-sharing-app-service;1"].getService(Ci.nsIExternalSharingAppService);
    sharing.shareWithDefault(aSharedString, "text/plain", aTitle);
  },

  _stripScheme: function(aString) {
    let index = aString.indexOf(":");
    return aString.slice(index + 1);
  }
}
};

var LightWeightThemeWebInstaller = {
init: function sh_init() {
  let temp = {};
  Cu.import("resource://gre/modules/LightweightThemeConsumer.jsm", temp);
  let theme = new temp.LightweightThemeConsumer(document);
  BrowserApp.deck.addEventListener("InstallBrowserTheme", this, false, true);
  BrowserApp.deck.addEventListener("PreviewBrowserTheme", this, false, true);
  BrowserApp.deck.addEventListener("ResetBrowserThemePreview", this, false, true);
},

uninit: function() {
  BrowserApp.deck.addEventListener("InstallBrowserTheme", this, false, true);
  BrowserApp.deck.addEventListener("PreviewBrowserTheme", this, false, true);
  BrowserApp.deck.addEventListener("ResetBrowserThemePreview", this, false, true);
},

handleEvent: function (event) {
  switch (event.type) {
    case "InstallBrowserTheme":
    case "PreviewBrowserTheme":
    case "ResetBrowserThemePreview":
      // ignore requests from background tabs
      if (event.target.ownerDocument.defaultView.top != content)
        return;
  }

  switch (event.type) {
    case "InstallBrowserTheme":
      this._installRequest(event);
      break;
    case "PreviewBrowserTheme":
      this._preview(event);
      break;
    case "ResetBrowserThemePreview":
      this._resetPreview(event);
      break;
    case "pagehide":
    case "TabSelect":
      this._resetPreview();
      break;
  }
},

get _manager () {
  let temp = {};
  Cu.import("resource://gre/modules/LightweightThemeManager.jsm", temp);
  delete this._manager;
  return this._manager = temp.LightweightThemeManager;
},

_installRequest: function (event) {
  let node = event.target;
  let data = this._getThemeFromNode(node);
  if (!data)
    return;

  if (this._isAllowed(node)) {
    this._install(data);
    return;
  }

  let allowButtonText = Strings.browser.GetStringFromName("lwthemeInstallRequest.allowButton");
  let message = Strings.browser.formatStringFromName("lwthemeInstallRequest.message", [node.ownerDocument.location.hostname], 1);
  let buttons = [{
    label: allowButtonText,
    callback: function () {
      LightWeightThemeWebInstaller._install(data);
    }
  }];

  NativeWindow.doorhanger.show(message, "Personas", buttons, BrowserApp.selectedTab.id);
},

_install: function (newLWTheme) {
  this._manager.currentTheme = newLWTheme;
},

_previewWindow: null,
_preview: function (event) {
  if (!this._isAllowed(event.target))
    return;
  let data = this._getThemeFromNode(event.target);
  if (!data)
    return;
  this._resetPreview();

  this._previewWindow = event.target.ownerDocument.defaultView;
  this._previewWindow.addEventListener("pagehide", this, true);
  BrowserApp.deck.addEventListener("TabSelect", this, false);
  this._manager.previewTheme(data);
},

_resetPreview: function (event) {
  if (!this._previewWindow ||
      event && !this._isAllowed(event.target))
    return;

  this._previewWindow.removeEventListener("pagehide", this, true);
  this._previewWindow = null;
  BrowserApp.deck.removeEventListener("TabSelect", this, false);

  this._manager.resetPreview();
},

_isAllowed: function (node) {
  // Make sure the whitelist has been imported to permissions
  PermissionsUtils.importFromPrefs("xpinstall.", "install");

  let pm = Services.perms;

  let uri = node.ownerDocument.documentURIObject;
  return pm.testPermission(uri, "install") == pm.ALLOW_ACTION;
},

_getThemeFromNode: function (node) {
  return this._manager.parseTheme(node.getAttribute("data-browsertheme"), node.baseURI);
}
};

var DesktopUserAgent = {
DESKTOP_UA: null,

init: function ua_init() {
  Services.obs.addObserver(this, "DesktopMode:Change", false);
  UserAgentOverrides.addComplexOverride(this.onRequest.bind(this));

  // See https://developer.mozilla.org/en/Gecko_user_agent_string_reference
  this.DESKTOP_UA = Cc["@mozilla.org/network/protocol;1?name=http"]
                      .getService(Ci.nsIHttpProtocolHandler).userAgent
                      .replace(/Android; [a-zA-Z]+/, "X11; Linux x86_64")
                      .replace(/Gecko\/[0-9\.]+/, "Gecko/20100101");
},

uninit: function ua_uninit() {
  Services.obs.removeObserver(this, "DesktopMode:Change");
},

onRequest: function(channel, defaultUA) {
  let channelWindow = this._getWindowForRequest(channel);
  let tab = BrowserApp.getTabForWindow(channelWindow);
  if (tab == null)
    return null;

  return this.getUserAgentForTab(tab);
},

getUserAgentForWindow: function ua_getUserAgentForWindow(aWindow) {
  let tab = BrowserApp.getTabForWindow(aWindow.top);
  if (tab)
    return this.getUserAgentForTab(tab);

  return null;
},

getUserAgentForTab: function ua_getUserAgentForTab(aTab) {
  // Send desktop UA if "Request Desktop Site" is enabled.
  if (aTab.desktopMode)
    return this.DESKTOP_UA;

  return null;
},

_getRequestLoadContext: function ua_getRequestLoadContext(aRequest) {
  if (aRequest && aRequest.notificationCallbacks) {
    try {
      return aRequest.notificationCallbacks.getInterface(Ci.nsILoadContext);
    } catch (ex) { }
  }

  if (aRequest && aRequest.loadGroup && aRequest.loadGroup.notificationCallbacks) {
    try {
      return aRequest.loadGroup.notificationCallbacks.getInterface(Ci.nsILoadContext);
    } catch (ex) { }
  }

  return null;
},

_getWindowForRequest: function ua_getWindowForRequest(aRequest) {
  let loadContext = this._getRequestLoadContext(aRequest);
  if (loadContext) {
    try {
      return loadContext.associatedWindow;
    } catch (e) {
      // loadContext.associatedWindow can throw when there's no window
    }
  }
  return null;
},

observe: function ua_observe(aSubject, aTopic, aData) {
  if (aTopic === "DesktopMode:Change") {
    let args = JSON.parse(aData);
    let tab = BrowserApp.getTabForId(args.tabId);
    if (tab != null)
      tab.reloadWithMode(args.desktopMode);
  }
}
};


function nsBrowserAccess() {
}

nsBrowserAccess.prototype = {
QueryInterface: XPCOMUtils.generateQI([Ci.nsIBrowserDOMWindow]),

_getBrowser: function _getBrowser(aURI, aOpener, aWhere, aContext) {
  let isExternal = (aContext == Ci.nsIBrowserDOMWindow.OPEN_EXTERNAL);
  if (isExternal && aURI && aURI.schemeIs("chrome"))
    return null;

  let loadflags = isExternal ?
                    Ci.nsIWebNavigation.LOAD_FLAGS_FROM_EXTERNAL :
                    Ci.nsIWebNavigation.LOAD_FLAGS_NONE;
  if (aWhere == Ci.nsIBrowserDOMWindow.OPEN_DEFAULTWINDOW) {
    switch (aContext) {
      case Ci.nsIBrowserDOMWindow.OPEN_EXTERNAL:
        aWhere = Services.prefs.getIntPref("browser.link.open_external");
        break;
      default: // OPEN_NEW or an illegal value
        aWhere = Services.prefs.getIntPref("browser.link.open_newwindow");
    }
  }

  Services.io.offline = false;

  let referrer;
  if (aOpener) {
    try {
      let location = aOpener.location;
      referrer = Services.io.newURI(location, null, null);
    } catch(e) { }
  }

  let ss = Cc["@mozilla.org/browser/sessionstore;1"].getService(Ci.nsISessionStore);
  let pinned = false;

  if (aURI && aWhere == Ci.nsIBrowserDOMWindow.OPEN_SWITCHTAB) {
    pinned = true;
    let spec = aURI.spec;
    let tabs = BrowserApp.tabs;
    for (let i = 0; i < tabs.length; i++) {
      let appOrigin = ss.getTabValue(tabs[i], "appOrigin");
      if (appOrigin == spec) {
        let tab = tabs[i];
        BrowserApp.selectTab(tab);
        return tab.browser;
      }
    }
  }

  let newTab = (aWhere == Ci.nsIBrowserDOMWindow.OPEN_NEWWINDOW ||
                aWhere == Ci.nsIBrowserDOMWindow.OPEN_NEWTAB ||
                aWhere == Ci.nsIBrowserDOMWindow.OPEN_SWITCHTAB);
  let isPrivate = false;

  if (newTab) {
    let parentId = -1;
    if (!isExternal && aOpener) {
      let parent = BrowserApp.getTabForWindow(aOpener.top);
      if (parent) {
        parentId = parent.id;
        isPrivate = PrivateBrowsingUtils.isWindowPrivate(parent.browser.contentWindow);
      }
    }

    // BrowserApp.addTab calls loadURIWithFlags with the appropriate params
    let tab = BrowserApp.addTab(aURI ? aURI.spec : "about:blank", { flags: loadflags,
                                                                    referrerURI: referrer,
                                                                    external: isExternal,
                                                                    parentId: parentId,
                                                                    selected: true,
                                                                    isPrivate: isPrivate,
                                                                    pinned: pinned });

    return tab.browser;
  }

  // OPEN_CURRENTWINDOW and illegal values
  let browser = BrowserApp.selectedBrowser;
  if (aURI && browser)
    browser.loadURIWithFlags(aURI.spec, loadflags, referrer, null, null);

  return browser;
},

openURI: function browser_openURI(aURI, aOpener, aWhere, aContext) {
  let browser = this._getBrowser(aURI, aOpener, aWhere, aContext);
  return browser ? browser.contentWindow : null;
},

openURIInFrame: function browser_openURIInFrame(aURI, aOpener, aWhere, aContext) {
  let browser = this._getBrowser(aURI, aOpener, aWhere, aContext);
  return browser ? browser.QueryInterface(Ci.nsIFrameLoaderOwner) : null;
},

isTabContentWindow: function(aWindow) {
  return BrowserApp.getBrowserForWindow(aWindow) != null;
},

get contentWindow() {
  return BrowserApp.selectedBrowser.contentWindow;
}
};


// track the last known screen size so that new tabs
// get created with the right size rather than being 1x1
let gScreenWidth = 1;
let gScreenHeight = 1;
let gReflowPending = null;

// The margins that should be applied to the viewport for fixed position
// children. This is used to avoid browser chrome permanently obscuring
// fixed position content, and also to make sure window-sized pages take
// into account said browser chrome.
let gViewportMargins = { top: 0, right: 0, bottom: 0, left: 0};

function Tab(aURL, aParams) {
this.browser = null;
this.id = 0;
this.lastTouchedAt = Date.now();
this._zoom = 1.0;
this._drawZoom = 1.0;
this._restoreZoom = false;
this._fixedMarginLeft = 0;
this._fixedMarginTop = 0;
this._fixedMarginRight = 0;
this._fixedMarginBottom = 0;
this._readerEnabled = false;
this._readerActive = false;
this.userScrollPos = { x: 0, y: 0 };
this.viewportExcludesHorizontalMargins = true;
this.viewportExcludesVerticalMargins = true;
this.viewportMeasureCallback = null;
this.lastPageSizeAfterViewportRemeasure = { width: 0, height: 0 };
this.contentDocumentIsDisplayed = true;
this.pluginDoorhangerTimeout = null;
this.shouldShowPluginDoorhanger = true;
this.clickToPlayPluginsActivated = false;
this.desktopMode = false;
this.originalURI = null;
this.savedArticle = null;
this.hasTouchListener = false;
this.browserWidth = 0;
this.browserHeight = 0;

this.create(aURL, aParams);
}

Tab.prototype = {
create: function(aURL, aParams) {
  if (this.browser)
    return;

  aParams = aParams || {};

  this.browser = document.createElement("browser");
  this.browser.setAttribute("type", "content-targetable");
  this.setBrowserSize(kDefaultCSSViewportWidth, kDefaultCSSViewportHeight);

  // Make sure the previously selected panel remains selected. The selected panel of a deck is
  // not stable when panels are added.
  let selectedPanel = BrowserApp.deck.selectedPanel;
  BrowserApp.deck.insertBefore(this.browser, aParams.sibling || null);
  BrowserApp.deck.selectedPanel = selectedPanel;

  if (BrowserApp.manifestUrl) {
    let appsService = Cc["@mozilla.org/AppsService;1"].getService(Ci.nsIAppsService);
    let manifest = appsService.getAppByManifestURL(BrowserApp.manifestUrl);
    if (manifest) {
      let app = manifest.QueryInterface(Ci.mozIApplication);
      this.browser.docShell.setIsApp(app.localId);
    }
  }

  // Must be called after appendChild so the docshell has been created.
  this.setActive(false);

  let isPrivate = ("isPrivate" in aParams) && aParams.isPrivate;
  if (isPrivate) {
    this.browser.docShell.QueryInterface(Ci.nsILoadContext).usePrivateBrowsing = true;
  }

  this.browser.stop();

  let frameLoader = this.browser.QueryInterface(Ci.nsIFrameLoaderOwner).frameLoader;
  frameLoader.renderMode = Ci.nsIFrameLoader.RENDER_MODE_ASYNC_SCROLL;

  // only set tab uri if uri is valid
  let uri = null;
  let title = aParams.title || aURL;
  try {
    uri = Services.io.newURI(aURL, null, null).spec;
  } catch (e) {}

  // When the tab is stubbed from Java, there's a window between the stub
  // creation and the tab creation in Gecko where the stub could be removed
  // or the selected tab can change (which is easiest to hit during startup).
  // To prevent these races, we need to differentiate between tab stubs from
  // Java and new tabs from Gecko.
  let stub = false;

  if (!aParams.zombifying) {
    if ("tabID" in aParams) {
      this.id = aParams.tabID;
      stub = true;
    } else {
      let jni = new JNI();
      let cls = jni.findClass("org/mozilla/gecko/Tabs");
      let method = jni.getStaticMethodID(cls, "getNextTabId", "()I");
      this.id = jni.callStaticIntMethod(cls, method);
      jni.close();
    }
    this.desktopMode = ("desktopMode" in aParams) ? aParams.desktopMode : false;

    let message = {
      type: "Tab:Added",
      tabID: this.id,
      uri: uri,
      parentId: ("parentId" in aParams) ? aParams.parentId : -1,
      tabIndex: ("tabIndex" in aParams) ? aParams.tabIndex : -1,
      external: ("external" in aParams) ? aParams.external : false,
      selected: ("selected" in aParams) ? aParams.selected : true,
      title: title,
      delayLoad: aParams.delayLoad || false,
      desktopMode: this.desktopMode,
      isPrivate: isPrivate,
      stub: stub
    };
    sendMessageToJava(message);

    this.overscrollController = new OverscrollController(this);
  }

  this.browser.contentWindow.controllers.insertControllerAt(0, this.overscrollController);

  let flags = Ci.nsIWebProgress.NOTIFY_STATE_ALL |
              Ci.nsIWebProgress.NOTIFY_LOCATION |
              Ci.nsIWebProgress.NOTIFY_SECURITY;
  this.browser.addProgressListener(this, flags);
  this.browser.sessionHistory.addSHistoryListener(this);

  this.browser.addEventListener("DOMContentLoaded", this, true);
  this.browser.addEventListener("DOMFormHasPassword", this, true);
  this.browser.addEventListener("DOMLinkAdded", this, true);
  this.browser.addEventListener("DOMLinkChanged", this, true);
  this.browser.addEventListener("DOMMetaAdded", this, false);
  this.browser.addEventListener("DOMTitleChanged", this, true);
  this.browser.addEventListener("DOMWindowClose", this, true);
  this.browser.addEventListener("DOMWillOpenModalDialog", this, true);
  this.browser.addEventListener("DOMAutoComplete", this, true);
  this.browser.addEventListener("blur", this, true);
  this.browser.addEventListener("scroll", this, true);
  this.browser.addEventListener("MozScrolledAreaChanged", this, true);
  this.browser.addEventListener("pageshow", this, true);
  this.browser.addEventListener("MozApplicationManifest", this, true);

  // Note that the XBL binding is untrusted
  this.browser.addEventListener("PluginBindingAttached", this, true, true);
  this.browser.addEventListener("VideoBindingAttached", this, true, true);
  this.browser.addEventListener("VideoBindingCast", this, true, true);

  Services.obs.addObserver(this, "before-first-paint", false);
  Services.obs.addObserver(this, "after-viewport-change", false);
  Services.prefs.addObserver("browser.ui.zoom.force-user-scalable", this, false);

  if (aParams.delayLoad) {
    // If this is a zombie tab, attach restore data so the tab will be
    // restored when selected
    this.browser.__SS_data = {
      entries: [{
        url: aURL,
        title: title
      }],
      index: 1
    };
    this.browser.__SS_restore = true;
  } else {
    let flags = "flags" in aParams ? aParams.flags : Ci.nsIWebNavigation.LOAD_FLAGS_NONE;
    let postData = ("postData" in aParams && aParams.postData) ? aParams.postData.value : null;
    let referrerURI = "referrerURI" in aParams ? aParams.referrerURI : null;
    let charset = "charset" in aParams ? aParams.charset : null;

    // The search term the user entered to load the current URL
    this.userSearch = "userSearch" in aParams ? aParams.userSearch : "";

    try {
      this.browser.loadURIWithFlags(aURL, flags, referrerURI, charset, postData);
    } catch(e) {
      let message = {
        type: "Content:LoadError",
        tabID: this.id
      };
      sendMessageToJava(message);
      dump("Handled load error: " + e);
    }
  }
},

/**
 * Retrieves the font size in twips for a given element.
 */
getInflatedFontSizeFor: function(aElement) {
  // GetComputedStyle should always give us CSS pixels for a font size.
  let fontSizeStr = this.window.getComputedStyle(aElement)['fontSize'];
  let fontSize = fontSizeStr.slice(0, -2);
  return aElement.fontSizeInflation * fontSize;
},

/**
 * This returns the zoom necessary to match the font size of an element to
 * the minimum font size specified by the browser.zoom.reflowOnZoom.minFontSizeTwips
 * preference.
 */
getZoomToMinFontSize: function(aElement) {
  // We only use the font.size.inflation.minTwips preference because this is
  // the only one that is controlled by the user-interface in the 'Settings'
  // menu. Thus, if font.size.inflation.emPerLine is changed, this does not
  // effect reflow-on-zoom.
  let minFontSize = convertFromTwipsToPx(Services.prefs.getIntPref("font.size.inflation.minTwips"));
  return minFontSize / this.getInflatedFontSizeFor(aElement);
},

clearReflowOnZoomPendingActions: function() {
  // Reflow was completed, so now re-enable painting.
  let webNav = BrowserApp.selectedTab.window.QueryInterface(Ci.nsIInterfaceRequestor).getInterface(Ci.nsIWebNavigation);
  let docShell = webNav.QueryInterface(Ci.nsIDocShell);
  let docViewer = docShell.contentViewer.QueryInterface(Ci.nsIMarkupDocumentViewer);
  docViewer.resumePainting();

  BrowserApp.selectedTab._mReflozPositioned = false;
},

/**
 * Reflow on zoom consists of a few different sub-operations:
 *
 * 1. When a double-tap event is seen, we verify that the correct preferences
 *    are enabled and perform the pre-position handling calculation. We also
 *    signal that reflow-on-zoom should be performed at this time, and pause
 *    painting.
 * 2. During the next call to setViewport(), which is in the Tab prototype,
 *    we detect that a call to changeMaxLineBoxWidth should be performed. If
 *    we're zooming out, then the max line box width should be reset at this
 *    time. Otherwise, we call performReflowOnZoom.
 *   2a. PerformReflowOnZoom() and resetMaxLineBoxWidth() schedule a call to
 *       doChangeMaxLineBoxWidth, based on a timeout specified in preferences.
 * 3. doChangeMaxLineBoxWidth changes the line box width (which also
 *    schedules a reflow event), and then calls ZoomHelper.zoomInAndSnapToRange.
 * 4. ZoomHelper.zoomInAndSnapToRange performs the positioning of reflow-on-zoom
 *    and then re-enables painting.
 *
 * Some of the events happen synchronously, while others happen asynchronously.
 * The following is a rough sketch of the progression of events:
 *
 * double tap event seen -> onDoubleTap() -> ... asynchronous ...
 *   -> setViewport() -> performReflowOnZoom() -> ... asynchronous ...
 *   -> doChangeMaxLineBoxWidth() -> ZoomHelper.zoomInAndSnapToRange()
 *   -> ... asynchronous ... -> setViewport() -> Observe('after-viewport-change')
 *   -> resumePainting()
 */
performReflowOnZoom: function(aViewport) {
  let zoom = this._drawZoom ? this._drawZoom : aViewport.zoom;

  let viewportWidth = gScreenWidth / zoom;
  let reflozTimeout = Services.prefs.getIntPref("browser.zoom.reflowZoom.reflowTimeout");

  if (gReflowPending) {
    clearTimeout(gReflowPending);
  }

  // We add in a bit of fudge just so that the end characters
  // don't accidentally get clipped. 15px is an arbitrary choice.
  gReflowPending = setTimeout(doChangeMaxLineBoxWidth,
                              reflozTimeout,
                              viewportWidth - 15);
},

/**
 * Reloads the tab with the desktop mode setting.
 */
reloadWithMode: function (aDesktopMode) {
  // Set desktop mode for tab and send change to Java
  if (this.desktopMode != aDesktopMode) {
    this.desktopMode = aDesktopMode;
    sendMessageToJava({
      type: "DesktopMode:Changed",
      desktopMode: aDesktopMode,
      tabID: this.id
    });
  }

  // Only reload the page for http/https schemes
  let currentURI = this.browser.currentURI;
  if (!currentURI.schemeIs("http") && !currentURI.schemeIs("https"))
    return;

  let url = currentURI.spec;
  let flags = Ci.nsIWebNavigation.LOAD_FLAGS_BYPASS_CACHE |
              Ci.nsIWebNavigation.LOAD_FLAGS_REPLACE_HISTORY;
  if (this.originalURI && !this.originalURI.equals(currentURI)) {
    // We were redirected; reload the original URL
    url = this.originalURI.spec;
  }

  this.browser.docShell.loadURI(url, flags, null, null, null);
},

destroy: function() {
  if (!this.browser)
    return;

  this.browser.contentWindow.controllers.removeController(this.overscrollController);

  this.browser.removeProgressListener(this);
  this.browser.sessionHistory.removeSHistoryListener(this);

  this.browser.removeEventListener("DOMContentLoaded", this, true);
  this.browser.removeEventListener("DOMFormHasPassword", this, true);
  this.browser.removeEventListener("DOMLinkAdded", this, true);
  this.browser.removeEventListener("DOMLinkChanged", this, true);
  this.browser.removeEventListener("DOMMetaAdded", this, false);
  this.browser.removeEventListener("DOMTitleChanged", this, true);
  this.browser.removeEventListener("DOMWindowClose", this, true);
  this.browser.removeEventListener("DOMWillOpenModalDialog", this, true);
  this.browser.removeEventListener("DOMAutoComplete", this, true);
  this.browser.removeEventListener("blur", this, true);
  this.browser.removeEventListener("scroll", this, true);
  this.browser.removeEventListener("MozScrolledAreaChanged", this, true);
  this.browser.removeEventListener("pageshow", this, true);
  this.browser.removeEventListener("MozApplicationManifest", this, true);

  this.browser.removeEventListener("PluginBindingAttached", this, true, true);
  this.browser.removeEventListener("VideoBindingAttached", this, true, true);
  this.browser.removeEventListener("VideoBindingCast", this, true, true);

  Services.obs.removeObserver(this, "before-first-paint");
  Services.obs.removeObserver(this, "after-viewport-change");
  Services.prefs.removeObserver("browser.ui.zoom.force-user-scalable", this);

  // Make sure the previously selected panel remains selected. The selected panel of a deck is
  // not stable when panels are removed.
  let selectedPanel = BrowserApp.deck.selectedPanel;
  BrowserApp.deck.removeChild(this.browser);
  BrowserApp.deck.selectedPanel = selectedPanel;

  this.browser = null;
  this.savedArticle = null;
},

// This should be called to update the browser when the tab gets selected/unselected
setActive: function setActive(aActive) {
  if (!this.browser || !this.browser.docShell)
    return;

  this.lastTouchedAt = Date.now();

  if (aActive) {
    this.browser.setAttribute("type", "content-primary");
    this.browser.focus();
    this.browser.docShellIsActive = true;
    Reader.updatePageAction(this);
    ExternalApps.updatePageAction(this.browser.currentURI);
  } else {
    this.browser.setAttribute("type", "content-targetable");
    this.browser.docShellIsActive = false;
  }
},

getActive: function getActive() {
  return this.browser.docShellIsActive;
},

setDisplayPort: function(aDisplayPort) {
  let zoom = this._zoom;
  let resolution = aDisplayPort.resolution;
  if (zoom <= 0 || resolution <= 0)
    return;

  // "zoom" is the user-visible zoom of the "this" tab
  // "resolution" is the zoom at which we wish gecko to render "this" tab at
  // these two may be different if we are, for example, trying to render a
  // large area of the page at low resolution because the user is panning real
  // fast.
  // The gecko scroll position is in CSS pixels. The display port rect
  // values (aDisplayPort), however, are in CSS pixels multiplied by the desired
  // rendering resolution. Therefore care must be taken when doing math with
  // these sets of values, to ensure that they are normalized to the same coordinate
  // space first.

  let element = this.browser.contentDocument.documentElement;
  if (!element)
    return;

  // we should never be drawing background tabs at resolutions other than the user-
  // visible zoom. for foreground tabs, however, if we are drawing at some other
  // resolution, we need to set the resolution as specified.
  let cwu = this.browser.contentWindow.QueryInterface(Ci.nsIInterfaceRequestor).getInterface(Ci.nsIDOMWindowUtils);
  if (BrowserApp.selectedTab == this) {
    if (resolution != this._drawZoom) {
      this._drawZoom = resolution;
      cwu.setResolution(resolution / window.devicePixelRatio, resolution / window.devicePixelRatio);
    }
  } else if (!fuzzyEquals(resolution, zoom)) {
    dump("Warning: setDisplayPort resolution did not match zoom for background tab! (" + resolution + " != " + zoom + ")");
  }

  // Finally, we set the display port as a set of margins around the visible viewport.

  let scrollx = this.browser.contentWindow.scrollX * zoom;
  let scrolly = this.browser.contentWindow.scrollY * zoom;
  let screenWidth = gScreenWidth - gViewportMargins.left - gViewportMargins.right;
  let screenHeight = gScreenHeight - gViewportMargins.top - gViewportMargins.bottom;
  let displayPortMargins = {
    left: scrollx - aDisplayPort.left,
    top: scrolly - aDisplayPort.top,
    right: aDisplayPort.right - (scrollx + screenWidth),
    bottom: aDisplayPort.bottom - (scrolly + screenHeight)
  };

  if (this._oldDisplayPortMargins == null ||
      !fuzzyEquals(displayPortMargins.left, this._oldDisplayPortMargins.left) ||
      !fuzzyEquals(displayPortMargins.top, this._oldDisplayPortMargins.top) ||
      !fuzzyEquals(displayPortMargins.right, this._oldDisplayPortMargins.right) ||
      !fuzzyEquals(displayPortMargins.bottom, this._oldDisplayPortMargins.bottom)) {
    cwu.setDisplayPortMarginsForElement(displayPortMargins.left,
                                        displayPortMargins.top,
                                        displayPortMargins.right,
                                        displayPortMargins.bottom,
                                        BrowserApp.layersTileWidth, BrowserApp.layersTileHeight,
                                        element, 0);
  }
  this._oldDisplayPortMargins = displayPortMargins;
},

setScrollClampingSize: function(zoom) {
  let viewportWidth = gScreenWidth / zoom;
  let viewportHeight = gScreenHeight / zoom;
  let screenWidth = gScreenWidth;
  let screenHeight = gScreenHeight;

  // Shrink the viewport appropriately if the margins are excluded
  if (this.viewportExcludesVerticalMargins) {
    screenHeight = gScreenHeight - gViewportMargins.top - gViewportMargins.bottom;
    viewportHeight = screenHeight / zoom;
  }
  if (this.viewportExcludesHorizontalMargins) {
    screenWidth = gScreenWidth - gViewportMargins.left - gViewportMargins.right;
    viewportWidth = screenWidth / zoom;
  }

  // Make sure the aspect ratio of the screen is maintained when setting
  // the clamping scroll-port size.
  let factor = Math.min(viewportWidth / screenWidth,
                        viewportHeight / screenHeight);
  let scrollPortWidth = screenWidth * factor;
  let scrollPortHeight = screenHeight * factor;

  let win = this.browser.contentWindow;
  win.QueryInterface(Ci.nsIInterfaceRequestor).getInterface(Ci.nsIDOMWindowUtils).
      setScrollPositionClampingScrollPortSize(scrollPortWidth, scrollPortHeight);
},

setViewport: function(aViewport) {
  // Transform coordinates based on zoom
  let x = aViewport.x / aViewport.zoom;
  let y = aViewport.y / aViewport.zoom;

  this.setScrollClampingSize(aViewport.zoom);

  // Adjust the max line box width to be no more than the viewport width, but
  // only if the reflow-on-zoom preference is enabled.
  let isZooming = !fuzzyEquals(aViewport.zoom, this._zoom);

  let docViewer = null;

  if (isZooming &&
      BrowserEventHandler.mReflozPref &&
      BrowserApp.selectedTab._mReflozPoint &&
      BrowserApp.selectedTab.probablyNeedRefloz) {
    let webNav = BrowserApp.selectedTab.window.QueryInterface(Ci.nsIInterfaceRequestor).getInterface(Ci.nsIWebNavigation);
    let docShell = webNav.QueryInterface(Ci.nsIDocShell);
    docViewer = docShell.contentViewer.QueryInterface(Ci.nsIMarkupDocumentViewer);
    docViewer.pausePainting();

    BrowserApp.selectedTab.performReflowOnZoom(aViewport);
    BrowserApp.selectedTab.probablyNeedRefloz = false;
  }

  let win = this.browser.contentWindow;
  win.scrollTo(x, y);
  this.saveSessionZoom(aViewport.zoom);

  this.userScrollPos.x = win.scrollX;
  this.userScrollPos.y = win.scrollY;
  this.setResolution(aViewport.zoom, false);

  if (aViewport.displayPort)
    this.setDisplayPort(aViewport.displayPort);

  // Store fixed margins for later retrieval in getViewport.
  this._fixedMarginLeft = aViewport.fixedMarginLeft;
  this._fixedMarginTop = aViewport.fixedMarginTop;
  this._fixedMarginRight = aViewport.fixedMarginRight;
  this._fixedMarginBottom = aViewport.fixedMarginBottom;

  let dwi = win.QueryInterface(Ci.nsIInterfaceRequestor).getInterface(Ci.nsIDOMWindowUtils);
  dwi.setContentDocumentFixedPositionMargins(
    aViewport.fixedMarginTop / aViewport.zoom,
    aViewport.fixedMarginRight / aViewport.zoom,
    aViewport.fixedMarginBottom / aViewport.zoom,
    aViewport.fixedMarginLeft / aViewport.zoom);

  Services.obs.notifyObservers(null, "after-viewport-change", "");
  if (docViewer) {
      docViewer.resumePainting();
  }
},

setResolution: function(aZoom, aForce) {
  // Set zoom level
  if (aForce || !fuzzyEquals(aZoom, this._zoom)) {
    this._zoom = aZoom;
    if (BrowserApp.selectedTab == this) {
      let cwu = this.browser.contentWindow.QueryInterface(Ci.nsIInterfaceRequestor).getInterface(Ci.nsIDOMWindowUtils);
      this._drawZoom = aZoom;
      cwu.setResolution(aZoom / window.devicePixelRatio, aZoom / window.devicePixelRatio);
    }
  }
},

getViewport: function() {
  let screenW = gScreenWidth - gViewportMargins.left - gViewportMargins.right;
  let screenH = gScreenHeight - gViewportMargins.top - gViewportMargins.bottom;
  let zoom = this.restoredSessionZoom() || this._zoom;

  let viewport = {
    width: screenW,
    height: screenH,
    cssWidth: screenW / zoom,
    cssHeight: screenH / zoom,
    pageLeft: 0,
    pageTop: 0,
    pageRight: screenW,
    pageBottom: screenH,
    // We make up matching css page dimensions
    cssPageLeft: 0,
    cssPageTop: 0,
    cssPageRight: screenW / zoom,
    cssPageBottom: screenH / zoom,
    fixedMarginLeft: this._fixedMarginLeft,
    fixedMarginTop: this._fixedMarginTop,
    fixedMarginRight: this._fixedMarginRight,
    fixedMarginBottom: this._fixedMarginBottom,
    zoom: zoom,
  };

  // Set the viewport offset to current scroll offset
  viewport.cssX = this.browser.contentWindow.scrollX || 0;
  viewport.cssY = this.browser.contentWindow.scrollY || 0;

  // Transform coordinates based on zoom
  viewport.x = Math.round(viewport.cssX * viewport.zoom);
  viewport.y = Math.round(viewport.cssY * viewport.zoom);

  let doc = this.browser.contentDocument;
  if (doc != null) {
    let cwu = this.browser.contentWindow.QueryInterface(Ci.nsIInterfaceRequestor).getInterface(Ci.nsIDOMWindowUtils);
    let cssPageRect = cwu.getRootBounds();

    /*
     * Avoid sending page sizes of less than screen size before we hit DOMContentLoaded, because
     * this causes the page size to jump around wildly during page load. After the page is loaded,
     * send updates regardless of page size; we'll zoom to fit the content as needed.
     *
     * In the check below, we floor the viewport size because there might be slight rounding errors
     * introduced in the CSS page size due to the conversion to and from app units in Gecko. The
     * error should be no more than one app unit so doing the floor is overkill, but safe in the
     * sense that the extra page size updates that get sent as a result will be mostly harmless.
     */
    let pageLargerThanScreen = (cssPageRect.width >= Math.floor(viewport.cssWidth))
                            && (cssPageRect.height >= Math.floor(viewport.cssHeight));
    if (doc.readyState === 'complete' || pageLargerThanScreen) {
      viewport.cssPageLeft = cssPageRect.left;
      viewport.cssPageTop = cssPageRect.top;
      viewport.cssPageRight = cssPageRect.right;
      viewport.cssPageBottom = cssPageRect.bottom;
      /* Transform the page width and height based on the zoom factor. */
      viewport.pageLeft = (viewport.cssPageLeft * viewport.zoom);
      viewport.pageTop = (viewport.cssPageTop * viewport.zoom);
      viewport.pageRight = (viewport.cssPageRight * viewport.zoom);
      viewport.pageBottom = (viewport.cssPageBottom * viewport.zoom);
    }
  }

  return viewport;
},

sendViewportUpdate: function(aPageSizeUpdate) {
  let viewport = this.getViewport();
  let displayPort = Services.androidBridge.getDisplayPort(aPageSizeUpdate, BrowserApp.isBrowserContentDocumentDisplayed(), this.id, viewport);
  if (displayPort != null)
    this.setDisplayPort(displayPort);
},

updateViewportForPageSize: function() {
  let hasHorizontalMargins = gViewportMargins.left != 0 || gViewportMargins.right != 0;
  let hasVerticalMargins = gViewportMargins.top != 0 || gViewportMargins.bottom != 0;

  if (!hasHorizontalMargins && !hasVerticalMargins) {
    // If there are no margins, then we don't need to do any remeasuring
    return;
  }

  // If the page size has changed so that it might or might not fit on the
  // screen with the margins included, run updateViewportSize to resize the
  // browser accordingly.
  // A page will receive the smaller viewport when its page size fits
  // within the screen size, so remeasure when the page size remains within
  // the threshold of screen + margins, in case it's sizing itself relative
  // to the viewport.
  let viewport = this.getViewport();
  let pageWidth = viewport.pageRight - viewport.pageLeft;
  let pageHeight = viewport.pageBottom - viewport.pageTop;
  let remeasureNeeded = false;

  if (hasHorizontalMargins) {
    let viewportShouldExcludeHorizontalMargins = (pageWidth <= gScreenWidth - 0.5);
    if (viewportShouldExcludeHorizontalMargins != this.viewportExcludesHorizontalMargins) {
      remeasureNeeded = true;
    }
  }
  if (hasVerticalMargins) {
    let viewportShouldExcludeVerticalMargins = (pageHeight <= gScreenHeight - 0.5);
    if (viewportShouldExcludeVerticalMargins != this.viewportExcludesVerticalMargins) {
      remeasureNeeded = true;
    }
  }

  if (remeasureNeeded) {
    if (!this.viewportMeasureCallback) {
      this.viewportMeasureCallback = setTimeout(function() {
```

```
        this.viewportMeasureCallback = null;

        // Re-fetch the viewport as it may have changed between setting the timeout
        // and running this callback
        let viewport = this.getViewport();
        let pageWidth = viewport.pageRight - viewport.pageLeft;
        let pageHeight = viewport.pageBottom - viewport.pageTop;

        if (Math.abs(pageWidth - this.lastPageSizeAfterViewportRemeasure.width) >= 0.5 ||
            Math.abs(pageHeight - this.lastPageSizeAfterViewportRemeasure.height) >= 0.5) {
          this.updateViewportSize(gScreenWidth);
        }
      }.bind(this), kViewportRemeasureThrottle);
    }
  } else if (this.viewportMeasureCallback) {
    // If the page changed size twice since we last measured the viewport and
    // the latest size change reveals we don't need to remeasure, cancel any
    // pending remeasure.
    clearTimeout(this.viewportMeasureCallback);
    this.viewportMeasureCallback = null;
  }
},

// These constants are used to prioritize high quality metadata over low quality data, so that
// we can collect data as we find meta tags, and replace low quality metadata with higher quality
// matches. For instance a msApplicationTile icon is a better tile image than an og:image tag.
METADATA_GOOD_MATCH: 10,
METADATA_NORMAL_MATCH: 1,

addMetadata: function(type, value, quality = 1) {
  if (!this.metatags) {
    this.metatags = {
      url: this.browser.currentURI.spec
    };
  }

  if (!this.metatags[type] || this.metatags[type + "_quality"] < quality) {
    this.metatags[type] = value;
    this.metatags[type + "_quality"] = quality;
  }
},

handleEvent: function(aEvent) {
  switch (aEvent.type) {
    case "DOMContentLoaded": {
      let target = aEvent.originalTarget;

      // ignore on frames and other documents
      if (target != this.browser.contentDocument)
        return;

      // Sample the background color of the page and pass it along. (This is used to draw the
      // checkerboard.) Right now we don't detect changes in the background color after this
      // event fires; it's not clear that doing so is worth the effort.
      var backgroundColor = null;
      try {
        let { contentDocument, contentWindow } = this.browser;
        let computedStyle = contentWindow.getComputedStyle(contentDocument.body);
        backgroundColor = computedStyle.backgroundColor;
      } catch (e) {
        // Ignore. Catching and ignoring exceptions here ensures that Talos succeeds.
      }

      let docURI = target.documentURI;
      let errorType = "";
      if (docURI.startsWith("about:certerror"))
        errorType = "certerror";
      else if (docURI.startsWith("about:blocked"))
        errorType = "blocked"
      else if (docURI.startsWith("about:neterror"))
        errorType = "neterror";

      sendMessageToJava({
        type: "DOMContentLoaded",
        tabID: this.id,
        bgColor: backgroundColor,
        errorType: errorType,
        metadata: this.metatags
      });

      this.metatags = null;

      // Attach a listener to watch for "click" events bubbling up from error
      // pages and other similar page. This lets us fix bugs like 401575 which
      // require error page UI to do privileged things, without letting error
      // pages have any privilege themselves.
      if (docURI.startsWith("about:certerror") || docURI.startsWith("about:blocked")) {
        this.browser.addEventListener("click", ErrorPageEventHandler, true);
        let listener = function() {
          this.browser.removeEventListener("click", ErrorPageEventHandler, true);
          this.browser.removeEventListener("pagehide", listener, true);
        }.bind(this);

        this.browser.addEventListener("pagehide", listener, true);
      }

      if (docURI.startsWith("about:reader")) {
        // During browser restart / recovery, duplicate "DOMContentLoaded" messages are received here
        // For the visible tab ... where more than one tab is being reloaded, the inital "DOMContentLoaded"
        // Message can be received before the document body is available ... so we avoid instantiating an
        // AboutReader object, expecting that an eventual valid message will follow.
        let contentDocument = this.browser.contentDocument;
        if (contentDocument.body) {
          new AboutReader(contentDocument, this.browser.contentWindow);
        }
      }

      break;
    }

    case "DOMFormHasPassword": {
      LoginManagerContent.onFormPassword(aEvent);
      break;
    }

    case "DOMMetaAdded":
      let target = aEvent.originalTarget;
      let browser = BrowserApp.getBrowserForDocument(target.ownerDocument);

      switch (target.name) {
        case "msapplication-TileImage":
          this.addMetadata("tileImage", browser.currentURI.resolve(target.content), this.METADATA_GOOD_MATCH);
          break;
        case "msapplication-TileColor":
          this.addMetadata("tileColor", target.content, this.METADATA_GOOD_MATCH);
          break;
      }

      break;

    case "DOMLinkAdded":
    case "DOMLinkChanged": {
      let target = aEvent.originalTarget;
      if (!target.href || target.disabled)
        return;

      // Ignore on frames and other documents
      if (target.ownerDocument != this.browser.contentDocument)
        return;

      // Sanitize the rel string
      let list = [];
      if (target.rel) {
        list = target.rel.toLowerCase().split(/\s+/);
        let hash = {};
        list.forEach(function(value) { hash[value] = true; });
        list = [];
        for (let rel in hash)
          list.push("[" + rel + "]");
      }

      if (list.indexOf("[icon]") != -1) {
        // We want to get the largest icon size possible for our UI.
        let maxSize = 0;

        // We use the sizes attribute if available
        // see http://www.whatwg.org/specs/web-apps/current-work/multipage/links.html#rel-icon
        if (target.hasAttribute("sizes")) {
          let sizes = target.getAttribute("sizes").toLowerCase();

          if (sizes == "any") {
            // Since Java expects an integer, use -1 to represent icons with sizes="any"
            maxSize = -1;
          } else {
            let tokens = sizes.split(" ");
            tokens.forEach(function(token) {
              // TODO: check for invalid tokens
              let [w, h] = token.split("x");
              maxSize = Math.max(maxSize, Math.max(w, h));
            });
          }
        }

        let json = {
          type: "Link:Favicon",
          tabID: this.id,
          href: resolveGeckoURI(target.href),
          size: maxSize
        };
        sendMessageToJava(json);
      } else if (list.indexOf("[alternate]") != -1 && aEvent.type == "DOMLinkAdded") {
        let type = target.type.toLowerCase().replace(/^\s+|\s*(?:;.*)?$/g, "");
        let isFeed = (type == "application/rss+xml" || type == "application/atom+xml");

        if (!isFeed)
          return;

        try {
          // urlSecurityCeck will throw if things are not OK
          ContentAreaUtils.urlSecurityCheck(target.href, target.ownerDocument.nodePrincipal, Ci.nsIScriptSecurityManager.DISALLOW_INHERIT_PRINCIPAL);

          if (!this.browser.feeds)
            this.browser.feeds = [];
          this.browser.feeds.push({ href: target.href, title: target.title, type: type });

          let json = {
            type: "Link:Feed",
            tabID: this.id
          };
          sendMessageToJava(json);
        } catch (e) {}
      } else if (list.indexOf("[search]" != -1) && aEvent.type == "DOMLinkAdded") {
        let type = target.type && target.type.toLowerCase();

        // Replace all starting or trailing spaces or spaces before "*;" globally w/ "".
        type = type.replace(/^\s+|\s*(?:;.*)?$/g, "");

        // Check that type matches opensearch.
        let isOpenSearch = (type == "application/opensearchdescription+xml");
        if (isOpenSearch && target.title && /^(?:https?|ftp):/i.test(target.href)) {
          let visibleEngines = Services.search.getVisibleEngines();
          // NOTE: Engines are currently identified by name, but this can be changed
          // when Engines are identified by URL (see bug 335102).
          if (visibleEngines.some(function(e) {
            return e.name == target.title;
          })) {
            // This engine is already present, do nothing.
            return;
          }

          if (this.browser.engines) {
            // This engine has already been handled, do nothing.
            if (this.browser.engines.some(function(e) {
              return e.url == target.href;
            })) {
                return;
            }
          } else {
            this.browser.engines = [];
          }

          // Get favicon.
          let iconURL = target.ownerDocument.documentURIObject.prePath + "/favicon.ico";

          let newEngine = {
            title: target.title,
            url: target.href,
            iconURL: iconURL
          };

          this.browser.engines.push(newEngine);

          // Don't send a message to display engines if we've already handled an engine.
          if (this.browser.engines.length > 1)
            return;

          // Broadcast message that this tab contains search engines that should be visible.
          let newEngineMessage = {
            type: "Link:OpenSearch",
            tabID: this.id,
            visible: true
          };

          sendMessageToJava(newEngineMessage);
        }
      }
      break;
    }

    case "DOMTitleChanged": {
      if (!aEvent.isTrusted)
        return;

      // ignore on frames and other documents
      if (aEvent.originalTarget != this.browser.contentDocument)
        return;

      sendMessageToJava({
        type: "DOMTitleChanged",
        tabID: this.id,
        title: aEvent.target.title.substring(0, 255)
      });
      break;
    }

    case "DOMWindowClose": {
      if (!aEvent.isTrusted)
        return;

      // Find the relevant tab, and close it from Java
      if (this.browser.contentWindow == aEvent.target) {
        aEvent.preventDefault();

        sendMessageToJava({
          type: "Tab:Close",
          tabID: this.id
        });
      }
      break;
    }

    case "DOMWillOpenModalDialog": {
      if (!aEvent.isTrusted)
        return;

      // We're about to open a modal dialog, make sure the opening
      // tab is brought to the front.
      let tab = BrowserApp.getTabForWindow(aEvent.target.top);
      BrowserApp.selectTab(tab);
      break;
    }

    case "DOMAutoComplete":
    case "blur": {
      LoginManagerContent.onUsernameInput(aEvent);
      break;
    }

    case "scroll": {
      let win = this.browser.contentWindow;
      if (this.userScrollPos.x != win.scrollX || this.userScrollPos.y != win.scrollY) {
        this.sendViewportUpdate();
      }
      break;
    }

    case "MozScrolledAreaChanged": {
      // This event is only fired for root scroll frames, and only when the
      // scrolled area has actually changed, so no need to check for that.
      // Just make sure it's the event for the correct root scroll frame.
      if (aEvent.originalTarget != this.browser.contentDocument)
        return;

      this.sendViewportUpdate(true);
      this.updateViewportForPageSize();
      break;
    }

    case "PluginBindingAttached": {
      PluginHelper.handlePluginBindingAttached(this, aEvent);
      break;
    }

    case "VideoBindingAttached": {
      CastingApps.handleVideoBindingAttached(this, aEvent);
      break;
    }

    case "VideoBindingCast": {
      CastingApps.handleVideoBindingCast(this, aEvent);
      break;
    }

    case "MozApplicationManifest": {
      OfflineApps.offlineAppRequested(aEvent.originalTarget.defaultView);
      break;
    }

    case "pageshow": {
      // only send pageshow for the top-level document
      if (aEvent.originalTarget.defaultView != this.browser.contentWindow)
        return;
      sendMessageToJava({
        type: "Content:PageShow",
        tabID: this.id
      });

      if (!aEvent.persisted && Services.prefs.getBoolPref("browser.ui.linkify.phone")) {
        if (!this._linkifier)
          this._linkifier = new Linkifier();
        this._linkifier.linkifyNumbers(this.browser.contentWindow.document);
      }

      // Update page actions for helper apps.
      let uri = this.browser.currentURI;
      if (BrowserApp.selectedTab == this) {
        if (ExternalApps.shouldCheckUri(uri)) {
          ExternalApps.updatePageAction(uri);
        } else {
          ExternalApps.clearPageAction();
        }
      }

      if (!Reader.isEnabledForParseOnLoad)
        return;

      // Once document is fully loaded, parse it
      Reader.parseDocumentFromTab(this.id, function (article) {
        // The loaded page may have changed while we were parsing the document.
        // Make sure we've got the current one.
        let uri = this.browser.currentURI;
        let tabURL = uri.specIgnoringRef;
        // Do nothing if there's no article or the page in this tab has
        // changed
        if (article == null || (article.url != tabURL)) {
          // Don't clear the article for about:reader pages since we want to
          // use the article from the previous page
          if (!tabURL.startsWith("about:reader")) {
            this.savedArticle = null;
            this.readerEnabled = false;
            this.readerActive = false;
          } else {
            this.readerActive = true;
          }
          return;
        }

        this.savedArticle = article;

        sendMessageToJava({
          type: "Content:ReaderEnabled",
          tabID: this.id
        });

        if(this.readerActive)
          this.readerActive = false;

        if(!this.readerEnabled)
          this.readerEnabled = true;
      }.bind(this));
    }
  }
},

onStateChange: function(aWebProgress, aRequest, aStateFlags, aStatus) {
  let contentWin = aWebProgress.DOMWindow;
  if (contentWin != contentWin.top)
      return;

  // Filter optimization: Only really send NETWORK state changes to Java listener
  if (aStateFlags & Ci.nsIWebProgressListener.STATE_IS_NETWORK) {
    if ((aStateFlags & Ci.nsIWebProgressListener.STATE_STOP) && aWebProgress.isLoadingDocument) {
      // We may receive a document stop event while a document is still loading
      // (such as when doing URI fixup). Don't notify Java UI in these cases.
      return;
    }

    // Clear page-specific opensearch engines and feeds for a new request.
    if (aStateFlags & Ci.nsIWebProgressListener.STATE_START && aRequest && aWebProgress.isTopLevel) {
      this.browser.engines = null;
      this.browser.feeds = null;
    }

    // true if the page loaded successfully (i.e., no 404s or other errors)
    let success = false;
    let uri = "";
    try {
      // Remember original URI for UA changes on redirected pages
      this.originalURI = aRequest.QueryInterface(Components.interfaces.nsIChannel).originalURI;

      if (this.originalURI != null)
        uri = this.originalURI.spec;
    } catch (e) { }
    try {
      success = aRequest.QueryInterface(Components.interfaces.nsIHttpChannel).requestSucceeded;
    } catch (e) {
      // If the request does not handle the nsIHttpChannel interface, use nsIRequest's success
      // status. Used for local files. See bug 948849.
      success = aRequest.status == 0;
    }

    // Check to see if we restoring the content from a previous presentation (session)
    // since there should be no real network activity
    let restoring = (aStateFlags & Ci.nsIWebProgressListener.STATE_RESTORING) > 0;

    let message = {
      type: "Content:StateChange",
      tabID: this.id,
      uri: uri,
      state: aStateFlags,
      restoring: restoring,
      success: success
    };
    sendMessageToJava(message);
  }
},

onLocationChange: function(aWebProgress, aRequest, aLocationURI, aFlags) {
  let contentWin = aWebProgress.DOMWindow;

  // Browser webapps may load content inside iframes that can not reach across the app/frame boundary
  // i.e. even though the page is loaded in an iframe window.top != webapp
  // Make cure this window is a top level tab before moving on.
  if (BrowserApp.getBrowserForWindow(contentWin) == null)
    return;

  this._hostChanged = true;

  let fixedURI = aLocationURI;
  try {
    fixedURI = URIFixup.createExposableURI(aLocationURI);
  } catch (ex) { }

  let contentType = contentWin.document.contentType;

  // If fixedURI matches browser.lastURI, we assume this isn't a real location
  // change but rather a spurious addition like a wyciwyg URI prefix. See Bug 747883.
  // Note that we have to ensure fixedURI is not the same as aLocationURI so we
  // don't false-positive page reloads as spurious additions.
  let sameDocument = (aFlags & Ci.nsIWebProgressListener.LOCATION_CHANGE_SAME_DOCUMENT) != 0 ||
                     ((this.browser.lastURI != null) && fixedURI.equals(this.browser.lastURI) && !fixedURI.equals(aLocationURI));
  this.browser.lastURI = fixedURI;

  // Reset state of click-to-play plugin notifications.
  clearTimeout(this.pluginDoorhangerTimeout);
  this.pluginDoorhangerTimeout = null;
  this.shouldShowPluginDoorhanger = true;
  this.clickToPlayPluginsActivated = false;
  // Borrowed from desktop Firefox: http://mxr.mozilla.org/mozilla-central/source/browser/base/content/urlbarBindings.xml#174
  let documentURI = contentWin.document.documentURIObject.spec
  let matchedURL = documentURI.match(/^((?:[a-z]+:\/\/)?(?:[^\/]+@)?)(.+?)(?::\d+)?(?:\/|$)/);
  let baseDomain = "";
  if (matchedURL) {
    var domain = "";
    [, , domain] = matchedURL;

    try {
      baseDomain = Services.eTLD.getBaseDomainFromHost(domain);
      if (!domain.endsWith(baseDomain)) {
        // getBaseDomainFromHost converts its resultant to ACE.
        let IDNService = Cc["@mozilla.org/network/idn-service;1"].getService(Ci.nsIIDNService);
        baseDomain = IDNService.convertACEtoUTF8(baseDomain);
      }
    } catch (e) {}
  }

  // Update the page actions URI for helper apps.
  if (BrowserApp.selectedTab == this) {
    ExternalApps.updatePageActionUri(fixedURI);
  }

  let message = {
    type: "Content:LocationChange",
    tabID: this.id,
    uri: fixedURI.spec,
    userSearch: this.userSearch || "",
    baseDomain: baseDomain,
    contentType: (contentType ? contentType : ""),
    sameDocument: sameDocument
  };

  sendMessageToJava(message);

  // The search term is only valid for this location change event, so reset it here.
  this.userSearch = "";

  if (!sameDocument) {
    // XXX This code assumes that this is the earliest hook we have at which
    // browser.contentDocument is changed to the new document we're loading

    // We have a new browser and a new window, so the old browserWidth and
    // browserHeight are no longer valid.  We need to force-set the browser
    // size to ensure it sets the CSS viewport size before the document
    // has a chance to check it.
    this.setBrowserSize(kDefaultCSSViewportWidth, kDefaultCSSViewportHeight, true);

    this.contentDocumentIsDisplayed = false;
    this.hasTouchListener = false;
  } else {
    this.sendViewportUpdate();
  }
},

// Properties used to cache security state used to update the UI
_state: null,
_hostChanged: false, // onLocationChange will flip this bit

onSecurityChange: function(aWebProgress, aRequest, aState) {
  // Don't need to do anything if the data we use to update the UI hasn't changed
  if (this._state == aState && !this._hostChanged)
    return;

  this._state = aState;
  this._hostChanged = false;

  let identity = IdentityHandler.checkIdentity(aState, this.browser);

  let message = {
    type: "Content:SecurityChange",
    tabID: this.id,
    identity: identity
  };

  sendMessageToJava(message);
},

onProgressChange: function(aWebProgress, aRequest, aCurSelfProgress, aMaxSelfProgress, aCurTotalProgress, aMaxTotalProgress) {
},

onStatusChange: function(aBrowser, aWebProgress, aRequest, aStatus, aMessage) {
},

_sendHistoryEvent: function(aMessage, aParams) {
  let message = {
    type: "SessionHistory:" + aMessage,
    tabID: this.id,
  };

  // Restore zoom only when moving in session history, not for new page loads.
  this._restoreZoom = aMessage != "New";

  if (aParams) {
    if ("url" in aParams)
      message.url = aParams.url;
    if ("index" in aParams)
      message.index = aParams.index;
    if ("numEntries" in aParams)
      message.numEntries = aParams.numEntries;
  }

  sendMessageToJava(message);
},

_getGeckoZoom: function() {
  let res = {x: {}, y: {}};
  let cwu = this.browser.contentWindow.QueryInterface(Ci.nsIInterfaceRequestor).getInterface(Ci.nsIDOMWindowUtils);
  cwu.getResolution(res.x, res.y);
  let zoom = res.x.value * window.devicePixelRatio;
  return zoom;
},

saveSessionZoom: function(aZoom) {
  let cwu = this.browser.contentWindow.QueryInterface(Ci.nsIInterfaceRequestor).getInterface(Ci.nsIDOMWindowUtils);
  cwu.setResolution(aZoom / window.devicePixelRatio, aZoom / window.devicePixelRatio);
},

restoredSessionZoom: function() {
  let cwu = this.browser.contentWindow.QueryInterface(Ci.nsIInterfaceRequestor).getInterface(Ci.nsIDOMWindowUtils);

  if (this._restoreZoom && cwu.isResolutionSet) {
    return this._getGeckoZoom();
  }
  return null;
},

OnHistoryNewEntry: function(aUri) {
  this._sendHistoryEvent("New", { url: aUri.spec });
},

OnHistoryGoBack: function(aUri) {
  this._sendHistoryEvent("Back");
  return true;
},

OnHistoryGoForward: function(aUri) {
  this._sendHistoryEvent("Forward");
  return true;
},

OnHistoryReload: function(aUri, aFlags) {
  // we don't do anything with this, so don't propagate it
  // for now anyway
  return true;
},

OnHistoryGotoIndex: function(aIndex, aUri) {
  this._sendHistoryEvent("Goto", { index: aIndex });
  return true;
},

OnHistoryPurge: function(aNumEntries) {
  this._sendHistoryEvent("Purge", { numEntries: aNumEntries });
  return true;
},

OnHistoryReplaceEntry: function(aIndex) {
  // we don't do anything with this, so don't propogate it
  // for now anyway.
},

get metadata() {
  return ViewportHandler.getMetadataForDocument(this.browser.contentDocument);
},

/** Update viewport when the metadata changes. */
updateViewportMetadata: function updateViewportMetadata(aMetadata, aInitialLoad) {
  if (Services.prefs.getBoolPref("browser.ui.zoom.force-user-scalable")) {
    aMetadata.allowZoom = true;
    aMetadata.allowDoubleTapZoom = true;
    aMetadata.minZoom = aMetadata.maxZoom = NaN;
  }

  let scaleRatio = window.devicePixelRatio;

  if (aMetadata.defaultZoom > 0)
    aMetadata.defaultZoom *= scaleRatio;
  if (aMetadata.minZoom > 0)
    aMetadata.minZoom *= scaleRatio;
  if (aMetadata.maxZoom > 0)
    aMetadata.maxZoom *= scaleRatio;

  aMetadata.isRTL = this.browser.contentDocument.documentElement.dir == "rtl";

  ViewportHandler.setMetadataForDocument(this.browser.contentDocument, aMetadata);
  this.sendViewportMetadata();

  this.updateViewportSize(gScreenWidth, aInitialLoad);
},

/** Update viewport when the metadata or the window size changes. */
updateViewportSize: function updateViewportSize(aOldScreenWidth, aInitialLoad) {
  // When this function gets called on window resize, we must execute
  // this.sendViewportUpdate() so that refreshDisplayPort is called.
  // Ensure that when making changes to this function that code path
  // is not accidentally removed (the call to sendViewportUpdate() is
  // at the very end).

  if (this.viewportMeasureCallback) {
    clearTimeout(this.viewportMeasureCallback);
    this.viewportMeasureCallback = null;
  }

  let browser = this.browser;
  if (!browser)
    return;

  let screenW = gScreenWidth - gViewportMargins.left - gViewportMargins.right;
  let screenH = gScreenHeight - gViewportMargins.top - gViewportMargins.bottom;
  let viewportW, viewportH;

  let metadata = this.metadata;
  if (metadata.autoSize) {
    viewportW = screenW / window.devicePixelRatio;
    viewportH = screenH / window.devicePixelRatio;
  } else {
    viewportW = metadata.width;
    viewportH = metadata.height;

    // If (scale * width) < device-width, increase the width (bug 561413).
    let maxInitialZoom = metadata.defaultZoom || metadata.maxZoom;
    if (maxInitialZoom && viewportW) {
      viewportW = Math.max(viewportW, screenW / maxInitialZoom);
    }

    let validW = viewportW > 0;
    let validH = viewportH > 0;

    if (!validW)
      viewportW = validH ? (viewportH * (screenW / screenH)) : BrowserApp.defaultBrowserWidth;
    if (!validH)
      viewportH = viewportW * (screenH / screenW);
  }

  // Make sure the viewport height is not shorter than the window when
  // the page is zoomed out to show its full width. Note that before
  // we set the viewport width, the "full width" of the page isn't properly
  // defined, so that's why we have to call setBrowserSize twice - once
  // to set the width, and the second time to figure out the height based
  // on the layout at that width.
  let oldBrowserWidth = this.browserWidth;
  this.setBrowserSize(viewportW, viewportH);

  // This change to the zoom accounts for all types of changes I can conceive:
  // 1. screen size changes, CSS viewport does not (pages with no meta viewport
  //    or a fixed size viewport)
  // 2. screen size changes, CSS viewport also does (pages with a device-width
  //    viewport)
  // 3. screen size remains constant, but CSS viewport changes (meta viewport
  //    tag is added or removed)
  // 4. neither screen size nor CSS viewport changes
  //
  // In all of these cases, we maintain how much actual content is visible
  // within the screen width. Note that "actual content" may be different
  // with respect to CSS pixels because of the CSS viewport size changing.
  let zoom = this.restoredSessionZoom() || metadata.defaultZoom;
  if (!zoom || !aInitialLoad) {
    let zoomScale = (screenW * oldBrowserWidth) / (aOldScreenWidth * viewportW);
    zoom = this.clampZoom(this._zoom * zoomScale);
  }
  this.setResolution(zoom, false);
  this.setScrollClampingSize(zoom);

  // if this page has not been painted yet, then this must be getting run
  // because a meta-viewport element was added (via the DOMMetaAdded handler).
  // in this case, we should not do anything that forces a reflow (see bug 759678)
  // such as requesting the page size or sending a viewport update. this code
  // will get run again in the before-first-paint handler and that point we
  // will run though all of it. the reason we even bother executing up to this
  // point on the DOMMetaAdded handler is so that scripts that use window.innerWidth
  // before they are painted have a correct value (bug 771575).
  if (!this.contentDocumentIsDisplayed) {
    return;
  }

  this.viewportExcludesHorizontalMargins = true;
  this.viewportExcludesVerticalMargins = true;
  let minScale = 1.0;
  if (this.browser.contentDocument) {
    // this may get run during a Viewport:Change message while the document
    // has not yet loaded, so need to guard against a null document.
    let cwu = this.browser.contentWindow.QueryInterface(Ci.nsIInterfaceRequestor).getInterface(Ci.nsIDOMWindowUtils);
    let cssPageRect = cwu.getRootBounds();

    // In the situation the page size equals or exceeds the screen size,
    // lengthen the viewport on the corresponding axis to include the margins.
    // The '- 0.5' is to account for rounding errors.
    if (cssPageRect.width * this._zoom > gScreenWidth - 0.5) {
      screenW = gScreenWidth;
      this.viewportExcludesHorizontalMargins = false;
    }
    if (cssPageRect.height * this._zoom > gScreenHeight - 0.5) {
      screenH = gScreenHeight;
      this.viewportExcludesVerticalMargins = false;
    }

    minScale = screenW / cssPageRect.width;
  }
  minScale = this.clampZoom(minScale);
  viewportH = Math.max(viewportH, screenH / minScale);

  // In general we want to keep calls to setBrowserSize and setScrollClampingSize
  // together because setBrowserSize could mark the viewport size as dirty, creating
  // a pending resize event for content. If that resize gets dispatched (which happens
  // on the next reflow) without setScrollClampingSize having being called, then
  // content might be exposed to incorrect innerWidth/innerHeight values.
  this.setBrowserSize(viewportW, viewportH);
  this.setScrollClampingSize(zoom);

  // Avoid having the scroll position jump around after device rotation.
  let win = this.browser.contentWindow;
  this.userScrollPos.x = win.scrollX;
  this.userScrollPos.y = win.scrollY;

  this.sendViewportUpdate();

  if (metadata.allowZoom && !Services.prefs.getBoolPref("browser.ui.zoom.force-user-scalable")) {
    // If the CSS viewport is narrower than the screen (i.e. width <= device-width)
    // then we disable double-tap-to-zoom behaviour.
    var oldAllowDoubleTapZoom = metadata.allowDoubleTapZoom;
    var newAllowDoubleTapZoom = (!metadata.isSpecified) || (viewportW > screenW / window.devicePixelRatio);
    if (oldAllowDoubleTapZoom !== newAllowDoubleTapZoom) {
      metadata.allowDoubleTapZoom = newAllowDoubleTapZoom;
      this.sendViewportMetadata();
    }
  }

  // Store the page size that was used to calculate the viewport so that we
  // can verify it's changed when we consider remeasuring in updateViewportForPageSize
  let viewport = this.getViewport();
  this.lastPageSizeAfterViewportRemeasure = {
    width: viewport.pageRight - viewport.pageLeft,
    height: viewport.pageBottom - viewport.pageTop
  };
},

sendViewportMetadata: function sendViewportMetadata() {
  let metadata = this.metadata;
  sendMessageToJava({
    type: "Tab:ViewportMetadata",
    allowZoom: metadata.allowZoom,
    allowDoubleTapZoom: metadata.allowDoubleTapZoom,
    defaultZoom: metadata.defaultZoom || window.devicePixelRatio,
    minZoom: metadata.minZoom || 0,
    maxZoom: metadata.maxZoom || 0,
    isRTL: metadata.isRTL,
    tabID: this.id
  });
},

setBrowserSize: function(aWidth, aHeight, aForce) {
  if (!aForce) {
    if (fuzzyEquals(this.browserWidth, aWidth) && fuzzyEquals(this.browserHeight, aHeight)) {
      return;
    }
  }

  this.browserWidth = aWidth;
  this.browserHeight = aHeight;

  if (!this.browser.contentWindow)
    return;
  let cwu = this.browser.contentWindow.QueryInterface(Ci.nsIInterfaceRequestor).getInterface(Ci.nsIDOMWindowUtils);
  cwu.setCSSViewport(aWidth, aHeight);
},

/** Takes a scale and restricts it based on this tab's zoom limits. */
clampZoom: function clampZoom(aZoom) {
  let zoom = ViewportHandler.clamp(aZoom, kViewportMinScale, kViewportMaxScale);

  let md = this.metadata;
  if (!md.allowZoom)
    return md.defaultZoom || zoom;

  if (md && md.minZoom)
    zoom = Math.max(zoom, md.minZoom);
  if (md && md.maxZoom)
    zoom = Math.min(zoom, md.maxZoom);
  return zoom;
},
observe: function(aSubject, aTopic, aData) {
  switch (aTopic) {
    case "before-first-paint":
      // Is it on the top level?
      let contentDocument = aSubject;
      if (contentDocument == this.browser.contentDocument) {
        if (BrowserApp.selectedTab == this) {
          BrowserApp.contentDocumentChanged();
        }
        this.contentDocumentIsDisplayed = true;

        // reset CSS viewport and zoom to default on new page, and then calculate
        // them properly using the actual metadata from the page. note that the
        // updateMetadata call takes into account the existing CSS viewport size
        // and zoom when calculating the new ones, so we need to reset these
        // things here before calling updateMetadata.
        this.setBrowserSize(kDefaultCSSViewportWidth, kDefaultCSSViewportHeight);
        let zoom = this.restoredSessionZoom() || gScreenWidth / this.browserWidth;
        this.setResolution(zoom, true);
        ViewportHandler.updateMetadata(this, true);

        // Note that if we draw without a display-port, things can go wrong. By the
        // time we execute this, it's almost certain a display-port has been set via
        // the MozScrolledAreaChanged event. If that didn't happen, the updateMetadata
        // call above does so at the end of the updateViewportSize function. As long
        // as that is happening, we don't need to do it again here.

        if (!this.restoredSessionZoom() && contentDocument.mozSyntheticDocument) {
          // for images, scale to fit width. this needs to happen *after* the call
          // to updateMetadata above, because that call sets the CSS viewport which
          // will affect the page size (i.e. contentDocument.body.scroll*) that we
          // use in this calculation. also we call sendViewportUpdate after changing
          // the resolution so that the display port gets recalculated appropriately.
          let fitZoom = Math.min(gScreenWidth / contentDocument.body.scrollWidth,
                                 gScreenHeight / contentDocument.body.scrollHeight);
          this.setResolution(fitZoom, false);
          this.sendViewportUpdate();
        }
      }

      // If the reflow-text-on-page-load pref is enabled, and reflow-on-zoom
      // is enabled, and our defaultZoom level is set, then we need to get
      // the default zoom and reflow the text according to the defaultZoom
      // level.
      let rzEnabled = BrowserEventHandler.mReflozPref;
      let rzPl = Services.prefs.getBoolPref("browser.zoom.reflowZoom.reflowTextOnPageLoad");

      if (rzEnabled && rzPl) {
        // Retrieve the viewport width and adjust the max line box width
        // accordingly.
        let vp = BrowserApp.selectedTab.getViewport();
        BrowserApp.selectedTab.performReflowOnZoom(vp);
      }
      break;
    case "after-viewport-change":
      if (BrowserApp.selectedTab._mReflozPositioned) {
        BrowserApp.selectedTab.clearReflowOnZoomPendingActions();
      }
      break;
    case "nsPref:changed":
      if (aData == "browser.ui.zoom.force-user-scalable")
        ViewportHandler.updateMetadata(this, false);
      break;
  }
},

set readerEnabled(isReaderEnabled) {
  this._readerEnabled = isReaderEnabled;
  if (this.getActive())
    Reader.updatePageAction(this);
},

get readerEnabled() {
  return this._readerEnabled;
},

set readerActive(isReaderActive) {
  this._readerActive = isReaderActive;
  if (this.getActive())
    Reader.updatePageAction(this);
},

get readerActive() {
  return this._readerActive;
},

// nsIBrowserTab
get window() {
  if (!this.browser)
    return null;
  return this.browser.contentWindow;
},

get scale() {
  return this._zoom;
},

QueryInterface: XPCOMUtils.generateQI([
  Ci.nsIWebProgressListener,
  Ci.nsISHistoryListener,
  Ci.nsIObserver,
  Ci.nsISupportsWeakReference,
  Ci.nsIBrowserTab
])
};

var BrowserEventHandler = {
init: function init() {
  Services.obs.addObserver(this, "Gesture:SingleTap", false);
  Services.obs.addObserver(this, "Gesture:CancelTouch", false);
  Services.obs.addObserver(this, "Gesture:DoubleTap", false);
  Services.obs.addObserver(this, "Gesture:Scroll", false);
  Services.obs.addObserver(this, "dom-touch-listener-added", false);

  BrowserApp.deck.addEventListener("DOMUpdatePageReport", PopupBlockerObserver.onUpdatePageReport, false);
  BrowserApp.deck.addEventListener("touchstart", this, true);
  BrowserApp.deck.addEventListener("click", InputWidgetHelper, true);
  BrowserApp.deck.addEventListener("click", SelectHelper, true);

  SpatialNavigation.init(BrowserApp.deck, null);

  document.addEventListener("MozMagnifyGesture", this, true);

  Services.prefs.addObserver("browser.zoom.reflowOnZoom", this, false);
  this.updateReflozPref();
},

resetMaxLineBoxWidth: function() {
  BrowserApp.selectedTab.probablyNeedRefloz = false;

  if (gReflowPending) {
    clearTimeout(gReflowPending);
  }

  let reflozTimeout = Services.prefs.getIntPref("browser.zoom.reflowZoom.reflowTimeout");
  gReflowPending = setTimeout(doChangeMaxLineBoxWidth,
                              reflozTimeout, 0);
},

updateReflozPref: function() {
   this.mReflozPref = Services.prefs.getBoolPref("browser.zoom.reflowOnZoom");
},

handleEvent: function(aEvent) {
  switch (aEvent.type) {
    case 'touchstart':
      this._handleTouchStart(aEvent);
      break;
    case 'MozMagnifyGesture':
      this.observe(this, aEvent.type,
                   JSON.stringify({x: aEvent.screenX, y: aEvent.screenY,
                                   zoomDelta: aEvent.delta}));
      break;
  }
},

_handleTouchStart: function(aEvent) {
  if (!BrowserApp.isBrowserContentDocumentDisplayed() || aEvent.touches.length > 1 || aEvent.defaultPrevented)
    return;

  let closest = aEvent.target;

  if (closest) {
    // If we've pressed a scrollable element, let Java know that we may
    // want to override the scroll behaviour (for document sub-frames)
    this._scrollableElement = this._findScrollableElement(closest, true);
    this._firstScrollEvent = true;

    if (this._scrollableElement != null) {
      // Discard if it's the top-level scrollable, we let Java handle this
      // The top-level scrollable is the body in quirks mode and the html element
      // in standards mode
      let doc = BrowserApp.selectedBrowser.contentDocument;
      let rootScrollable = (doc.compatMode === "BackCompat" ? doc.body : doc.documentElement);
      if (this._scrollableElement != rootScrollable) {
        sendMessageToJava({ type: "Panning:Override" });
      }
    }
  }

  if (!ElementTouchHelper.isElementClickable(closest, null, false))
    closest = ElementTouchHelper.elementFromPoint(aEvent.changedTouches[0].screenX,
                                                  aEvent.changedTouches[0].screenY);
  if (!closest)
    closest = aEvent.target;

  if (closest) {
    let uri = this._getLinkURI(closest);
    if (uri) {
      try {
        Services.io.QueryInterface(Ci.nsISpeculativeConnect).speculativeConnect(uri, null);
      } catch (e) {}
    }
    this._doTapHighlight(closest);
  }
},

_getLinkURI: function(aElement) {
  if (aElement.nodeType == Ci.nsIDOMNode.ELEMENT_NODE &&
      ((aElement instanceof Ci.nsIDOMHTMLAnchorElement && aElement.href) ||
      (aElement instanceof Ci.nsIDOMHTMLAreaElement && aElement.href))) {
    try {
      return Services.io.newURI(aElement.href, null, null);
    } catch (e) {}
  }
  return null;
},

observe: function(aSubject, aTopic, aData) {
  if (aTopic == "dom-touch-listener-added") {
    let tab = BrowserApp.getTabForWindow(aSubject.top);
    if (!tab || tab.hasTouchListener)
      return;

    tab.hasTouchListener = true;
    sendMessageToJava({
      type: "Tab:HasTouchListener",
      tabID: tab.id
    });
    return;
  } else if (aTopic == "nsPref:changed") {
    if (aData == "browser.zoom.reflowOnZoom") {
      this.updateReflozPref();
    }
    return;
  }

  // the remaining events are all dependent on the browser content document being the
  // same as the browser displayed document. if they are not the same, we should ignore
  // the event.
  if (BrowserApp.isBrowserContentDocumentDisplayed()) {
    this.handleUserEvent(aTopic, aData);
  }
},

handleUserEvent: function(aTopic, aData) {
  switch (aTopic) {

    case "Gesture:Scroll": {
      // If we've lost our scrollable element, return. Don't cancel the
      // override, as we probably don't want Java to handle panning until the
      // user releases their finger.
      if (this._scrollableElement == null)
        return;

      // If this is the first scroll event and we can't scroll in the direction
      // the user wanted, and neither can any non-root sub-frame, cancel the
      // override so that Java can handle panning the main document.
      let data = JSON.parse(aData);

      // round the scroll amounts because they come in as floats and might be
      // subject to minor rounding errors because of zoom values. I've seen values
      // like 0.99 come in here and get truncated to 0; this avoids that problem.
      let zoom = BrowserApp.selectedTab._zoom;
      let x = Math.round(data.x / zoom);
      let y = Math.round(data.y / zoom);

      if (this._firstScrollEvent) {
        while (this._scrollableElement != null &&
               !this._elementCanScroll(this._scrollableElement, x, y))
          this._scrollableElement = this._findScrollableElement(this._scrollableElement, false);

        let doc = BrowserApp.selectedBrowser.contentDocument;
        if (this._scrollableElement == null ||
            this._scrollableElement == doc.documentElement) {
          sendMessageToJava({ type: "Panning:CancelOverride" });
          return;
        }

        this._firstScrollEvent = false;
      }

      // Scroll the scrollable element
      if (this._elementCanScroll(this._scrollableElement, x, y)) {
        this._scrollElementBy(this._scrollableElement, x, y);
        sendMessageToJava({ type: "Gesture:ScrollAck", scrolled: true });
        SelectionHandler.subdocumentScrolled(this._scrollableElement);
      } else {
        sendMessageToJava({ type: "Gesture:ScrollAck", scrolled: false });
      }

      break;
    }

    case "Gesture:CancelTouch":
      this._cancelTapHighlight();
      break;

    case "Gesture:SingleTap": {
      let element = this._highlightElement;
      if (element) {
        try {
          let data = JSON.parse(aData);
          let [x, y] = [data.x, data.y];
          if (ElementTouchHelper.isElementClickable(element)) {
            [x, y] = this._moveClickPoint(element, x, y);
          }

          // Was the element already focused before it was clicked?
          let isFocused = (element == BrowserApp.getFocusedInput(BrowserApp.selectedBrowser));

          this._sendMouseEvent("mousemove", element, x, y);
          this._sendMouseEvent("mousedown", element, x, y);
          this._sendMouseEvent("mouseup",   element, x, y);

          // If the element was previously focused, show the caret attached to it.
          if (isFocused)
            SelectionHandler.attachCaret(element);

          // scrollToFocusedInput does its own checks to find out if an element should be zoomed into
          BrowserApp.scrollToFocusedInput(BrowserApp.selectedBrowser);
        } catch(e) {
          Cu.reportError(e);
        }
      }
      this._cancelTapHighlight();
      break;
    }

    case"Gesture:DoubleTap":
      this._cancelTapHighlight();
      this.onDoubleTap(aData);
      break;

    case "MozMagnifyGesture":
      this.onPinchFinish(aData);
      break;

    default:
      dump('BrowserEventHandler.handleUserEvent: unexpected topic "' + aTopic + '"');
      break;
  }
},

onDoubleTap: function(aData) {
  let metadata = BrowserApp.selectedTab.metadata;
  if (!metadata.allowDoubleTapZoom) {
    return;
  }

  let data = JSON.parse(aData);
  let element = ElementTouchHelper.anyElementFromPoint(data.x, data.y);

  // We only want to do this if reflow-on-zoom is enabled, we don't already
  // have a reflow-on-zoom event pending, and the element upon which the user
  // double-tapped isn't of a type we want to avoid reflow-on-zoom.
  if (BrowserEventHandler.mReflozPref &&
     !BrowserApp.selectedTab._mReflozPoint &&
     !this._shouldSuppressReflowOnZoom(element)) {

    // See comment above performReflowOnZoom() for a detailed description of
    // the events happening in the reflow-on-zoom operation.
    let data = JSON.parse(aData);
    let zoomPointX = data.x;
    let zoomPointY = data.y;

    BrowserApp.selectedTab._mReflozPoint = { x: zoomPointX, y: zoomPointY,
      range: BrowserApp.selectedBrowser.contentDocument.caretPositionFromPoint(zoomPointX, zoomPointY) };

    // Before we perform a reflow on zoom, let's disable painting.
    let webNav = BrowserApp.selectedTab.window.QueryInterface(Ci.nsIInterfaceRequestor).getInterface(Ci.nsIWebNavigation);
    let docShell = webNav.QueryInterface(Ci.nsIDocShell);
    let docViewer = docShell.contentViewer.QueryInterface(Ci.nsIMarkupDocumentViewer);
    docViewer.pausePainting();

    BrowserApp.selectedTab.probablyNeedRefloz = true;
  }

  if (!element) {
    ZoomHelper.zoomOut();
    return;
  }

  while (element && !this._shouldZoomToElement(element))
    element = element.parentNode;

  if (!element) {
    ZoomHelper.zoomOut();
  } else {
    ZoomHelper.zoomToElement(element, data.y);
  }
},

/**
 * Determine if reflow-on-zoom functionality should be suppressed, given a
 * particular element. Double-tapping on the following elements suppresses
 * reflow-on-zoom:
 *
 * <video>, <object>, <embed>, <applet>, <canvas>, <img>, <media>, <pre>
 */
_shouldSuppressReflowOnZoom: function(aElement) {
  if (aElement instanceof HTMLVideoElement ||
      aElement instanceof HTMLObjectElement ||
      aElement instanceof HTMLEmbedElement ||
      aElement instanceof HTMLAppletElement ||
      aElement instanceof HTMLCanvasElement ||
      aElement instanceof HTMLImageElement ||
      aElement instanceof HTMLMediaElement ||
      aElement instanceof HTMLPreElement) {
    return true;
  }

  return false;
},

onPinchFinish: function(aData) {
  let data = {};
  try {
    data = JSON.parse(aData);
  } catch(ex) {
    console.log(ex);
    return;
  }

  if (BrowserEventHandler.mReflozPref &&
      data.zoomDelta < 0.0) {
    BrowserEventHandler.resetMaxLineBoxWidth();
  }
},

_shouldZoomToElement: function(aElement) {
  let win = aElement.ownerDocument.defaultView;
  if (win.getComputedStyle(aElement, null).display == "inline")
    return false;
  if (aElement instanceof Ci.nsIDOMHTMLLIElement)
    return false;
  if (aElement instanceof Ci.nsIDOMHTMLQuoteElement)
    return false;
  return true;
},

_firstScrollEvent: false,

_scrollableElement: null,

_highlightElement: null,

_doTapHighlight: function _doTapHighlight(aElement) {
  DOMUtils.setContentState(aElement, kStateActive);
  this._highlightElement = aElement;
},

_cancelTapHighlight: function _cancelTapHighlight() {
  if (!this._highlightElement)
    return;

  // If the active element is in a sub-frame, we need to make that frame's document
  // active to remove the element's active state.
  if (this._highlightElement.ownerDocument != BrowserApp.selectedBrowser.contentWindow.document)
    DOMUtils.setContentState(this._highlightElement.ownerDocument.documentElement, kStateActive);

  DOMUtils.setContentState(BrowserApp.selectedBrowser.contentWindow.document.documentElement, kStateActive);
  this._highlightElement = null;
},

_updateLastPosition: function(x, y, dx, dy) {
  this.lastX = x;
  this.lastY = y;
  this.lastTime = Date.now();

  this.motionBuffer.push({ dx: dx, dy: dy, time: this.lastTime });
},

_moveClickPoint: function(aElement, aX, aY) {
  // the element can be out of the aX/aY point because of the touch radius
  // if outside, we gracefully move the touch point to the edge of the element
  if (!(aElement instanceof HTMLHtmlElement)) {
    let isTouchClick = true;
    let rects = ElementTouchHelper.getContentClientRects(aElement);
    for (let i = 0; i < rects.length; i++) {
      let rect = rects[i];
      let inBounds =
        (aX > rect.left && aX < (rect.left + rect.width)) &&
        (aY > rect.top && aY < (rect.top + rect.height));
      if (inBounds) {
        isTouchClick = false;
        break;
      }
    }

    if (isTouchClick) {
      let rect = rects[0];
      // if either width or height is zero, we don't want to move the click to the edge of the element. See bug 757208
      if (rect.width != 0 && rect.height != 0) {
        aX = Math.min(Math.ceil(rect.left + rect.width) - 1, Math.max(Math.ceil(rect.left), aX));
        aY = Math.min(Math.ceil(rect.top + rect.height) - 1, Math.max(Math.ceil(rect.top),  aY));
      }
    }
  }
  return [aX, aY];
},

_sendMouseEvent: function _sendMouseEvent(aName, aElement, aX, aY) {
  let window = aElement.ownerDocument.defaultView;
  try {
    let cwu = window.top.QueryInterface(Ci.nsIInterfaceRequestor).getInterface(Ci.nsIDOMWindowUtils);
    cwu.sendMouseEventToWindow(aName, aX, aY, 0, 1, 0, true);
  } catch(e) {
    Cu.reportError(e);
  }
},

_hasScrollableOverflow: function(elem) {
  var win = elem.ownerDocument.defaultView;
  if (!win)
    return false;
  var computedStyle = win.getComputedStyle(elem);
  if (!computedStyle)
    return false;
  // We check for overflow:hidden only because all the other cases are scrollable
  // under various conditions. See https://bugzilla.mozilla.org/show_bug.cgi?id=911574#c24
  // for some more details.
  return !(computedStyle.overflowX == 'hidden' && computedStyle.overflowY == 'hidden');
},

_findScrollableElement: function(elem, checkElem) {
  // Walk the DOM tree until we find a scrollable element
  let scrollable = false;
  while (elem) {
    /* Element is scrollable if its scroll-size exceeds its client size, and:
     * - It has overflow other than 'hidden', or
     * - It's a textarea node, or
     * - It's a text input, or
     * - It's a select element showing multiple rows
     */
    if (checkElem) {
      if ((elem.scrollTopMax > 0 || elem.scrollLeftMax > 0) &&
          (this._hasScrollableOverflow(elem) ||
           elem.mozMatchesSelector("textarea")) ||
          (elem instanceof HTMLInputElement && elem.mozIsTextField(false)) ||
          (elem instanceof HTMLSelectElement && (elem.size > 1 || elem.multiple))) {
        scrollable = true;
        break;
      }
    } else {
      checkElem = true;
    }

    // Propagate up iFrames
    if (!elem.parentNode && elem.documentElement && elem.documentElement.ownerDocument)
      elem = elem.documentElement.ownerDocument.defaultView.frameElement;
    else
      elem = elem.parentNode;
  }

  if (!scrollable)
    return null;

  return elem;
},

_scrollElementBy: function(elem, x, y) {
  elem.scrollTop = elem.scrollTop + y;
  elem.scrollLeft = elem.scrollLeft + x;
},

_elementCanScroll: function(elem, x, y) {
  let scrollX = (x < 0 && elem.scrollLeft > 0)
             || (x > 0 && elem.scrollLeft < elem.scrollLeftMax);

  let scrollY = (y < 0 && elem.scrollTop > 0)
             || (y > 0 && elem.scrollTop < elem.scrollTopMax);

  return scrollX || scrollY;
}
};

const kReferenceDpi = 240; // standard "pixel" size used in some preferences

const ElementTouchHelper = {
  /* Return the element at the given coordinates, starting from the given window and
     drilling down through frames. If no window is provided, the top-level window of
     the currently selected tab is used. The coordinates provided should be CSS pixels
     relative to the window's scroll position. */
  anyElementFromPoint: function(aX, aY, aWindow) {
    let win = (aWindow ? aWindow : BrowserApp.selectedBrowser.contentWindow);
    let cwu = win.QueryInterface(Ci.nsIInterfaceRequestor).getInterface(Ci.nsIDOMWindowUtils);
    let elem = cwu.elementFromPoint(aX, aY, true, true);

    while (elem && (elem instanceof HTMLIFrameElement || elem instanceof HTMLFrameElement)) {
      let rect = elem.getBoundingClientRect();
      aX -= rect.left;
      aY -= rect.top;
      cwu = elem.contentDocument.defaultView.QueryInterface(Ci.nsIInterfaceRequestor).getInterface(Ci.nsIDOMWindowUtils);
      elem = cwu.elementFromPoint(aX, aY, true, true);
    }

    return elem;
  },

  /* Return the most appropriate clickable element (if any), starting from the given window
     and drilling down through iframes as necessary. If no window is provided, the top-level
     window of the currently selected tab is used. The coordinates provided should be CSS
     pixels relative to the window's scroll position. The element returned may not actually
     contain the coordinates passed in because of touch radius and clickability heuristics. */
  elementFromPoint: function(aX, aY, aWindow) {
    // browser's elementFromPoint expect browser-relative client coordinates.
    // subtract browser's scroll values to adjust
    let win = (aWindow ? aWindow : BrowserApp.selectedBrowser.contentWindow);
    let cwu = win.QueryInterface(Ci.nsIInterfaceRequestor).getInterface(Ci.nsIDOMWindowUtils);
    let elem = this.getClosest(cwu, aX, aY);

    // step through layers of IFRAMEs and FRAMES to find innermost element
    while (elem && (elem instanceof HTMLIFrameElement || elem instanceof HTMLFrameElement)) {
      // adjust client coordinates' origin to be top left of iframe viewport
      let rect = elem.getBoundingClientRect();
      aX -= rect.left;
      aY -= rect.top;
      cwu = elem.contentDocument.defaultView.QueryInterface(Ci.nsIInterfaceRequestor).getInterface(Ci.nsIDOMWindowUtils);
      elem = this.getClosest(cwu, aX, aY);
    }

    return elem;
  },

  /* Returns the touch radius in content px. */
  getTouchRadius: function getTouchRadius() {
    let dpiRatio = ViewportHandler.displayDPI / kReferenceDpi;
    let zoom = BrowserApp.selectedTab._zoom;
    return {
      top: this.radius.top * dpiRatio / zoom,
      right: this.radius.right * dpiRatio / zoom,
      bottom: this.radius.bottom * dpiRatio / zoom,
      left: this.radius.left * dpiRatio / zoom
    };
  },

  /* Returns the touch radius in reference pixels. */
  get radius() {
    let prefs = Services.prefs;
    delete this.radius;
    return this.radius = { "top": prefs.getIntPref("browser.ui.touch.top"),
                           "right": prefs.getIntPref("browser.ui.touch.right"),
                           "bottom": prefs.getIntPref("browser.ui.touch.bottom"),
                           "left": prefs.getIntPref("browser.ui.touch.left")
                         };
  },

  get weight() {
    delete this.weight;
    return this.weight = { "visited": Services.prefs.getIntPref("browser.ui.touch.weight.visited") };
  },

  /* Retrieve the closest element to a point by looking at borders position */
  getClosest: function getClosest(aWindowUtils, aX, aY) {
    let target = aWindowUtils.elementFromPoint(aX, aY,
                                               true,   /* ignore root scroll frame*/
                                               false); /* don't flush layout */

    // if this element is clickable we return quickly. also, if it isn't,
    // use a cache to speed up future calls to isElementClickable in the
    // loop below.
    let unclickableCache = new Array();
    if (this.isElementClickable(target, unclickableCache, false))
      return target;

    target = null;
    let radius = this.getTouchRadius();
    let nodes = aWindowUtils.nodesFromRect(aX, aY, radius.top, radius.right, radius.bottom, radius.left, true, false);

    let threshold = Number.POSITIVE_INFINITY;
    for (let i = 0; i < nodes.length; i++) {
      let current = nodes[i];
      if (!current.mozMatchesSelector || !this.isElementClickable(current, unclickableCache, true))
        continue;

      let rect = current.getBoundingClientRect();
      let distance = this._computeDistanceFromRect(aX, aY, rect);

      // increase a little bit the weight for already visited items
      if (current && current.mozMatchesSelector("*:visited"))
        distance *= (this.weight.visited / 100);

      if (distance < threshold) {
        target = current;
        threshold = distance;
      }
    }

    return target;
  },

  isElementClickable: function isElementClickable(aElement, aUnclickableCache, aAllowBodyListeners) {
    const selector = "a,:link,:visited,[role=button],button,input,select,textarea";

    let stopNode = null;
    if (!aAllowBodyListeners && aElement && aElement.ownerDocument)
      stopNode = aElement.ownerDocument.body;

    for (let elem = aElement; elem && elem != stopNode; elem = elem.parentNode) {
      if (aUnclickableCache && aUnclickableCache.indexOf(elem) != -1)
        continue;
      if (this._hasMouseListener(elem))
        return true;
      if (elem.mozMatchesSelector && elem.mozMatchesSelector(selector))
        return true;
      if (elem instanceof HTMLLabelElement && elem.control != null)
        return true;
      if (aUnclickableCache)
        aUnclickableCache.push(elem);
    }
    return false;
  },
  _computeDistanceFromRect: function _computeDistanceFromRect(aX, aY, aRect) {
    let x = 0, y = 0;
    let xmost = aRect.left + aRect.width;
    let ymost = aRect.top + aRect.height;

    // compute horizontal distance from left/right border depending if X is
    // before/inside/after the element's rectangle
    if (aRect.left < aX && aX < xmost)
      x = Math.min(xmost - aX, aX - aRect.left);
    else if (aX < aRect.left)
      x = aRect.left - aX;
    else if (aX > xmost)
      x = aX - xmost;

    // compute vertical distance from top/bottom border depending if Y is
    // above/inside/below the element's rectangle
    if (aRect.top < aY && aY < ymost)
      y = Math.min(ymost - aY, aY - aRect.top);
    else if (aY < aRect.top)
      y = aRect.top - aY;
    if (aY > ymost)
      y = aY - ymost;

    return Math.sqrt(Math.pow(x, 2) + Math.pow(y, 2));
  },

  _els: Cc["@mozilla.org/eventlistenerservice;1"].getService(Ci.nsIEventListenerService),
  _clickableEvents: ["mousedown", "mouseup", "click"],
  _hasMouseListener: function _hasMouseListener(aElement) {
    let els = this._els;
    let listeners = els.getListenerInfoFor(aElement, {});
    for (let i = 0; i < listeners.length; i++) {
      if (this._clickableEvents.indexOf(listeners[i].type) != -1)
        return true;
    }
    return false;
  },

  getContentClientRects: function(aElement) {
    let offset = { x: 0, y: 0 };

    let nativeRects = aElement.getClientRects();
    // step out of iframes and frames, offsetting scroll values
    for (let frame = aElement.ownerDocument.defaultView; frame.frameElement; frame = frame.parent) {
      // adjust client coordinates' origin to be top left of iframe viewport
      let rect = frame.frameElement.getBoundingClientRect();
      let left = frame.getComputedStyle(frame.frameElement, "").borderLeftWidth;
      let top = frame.getComputedStyle(frame.frameElement, "").borderTopWidth;
      offset.x += rect.left + parseInt(left);
      offset.y += rect.top + parseInt(top);
    }

    let result = [];
    for (let i = nativeRects.length - 1; i >= 0; i--) {
      let r = nativeRects[i];
      result.push({ left: r.left + offset.x,
                    top: r.top + offset.y,
                    width: r.width,
                    height: r.height
                  });
    }
    return result;
  },

  getBoundingContentRect: function(aElement) {
    if (!aElement)
      return {x: 0, y: 0, w: 0, h: 0};

    let document = aElement.ownerDocument;
    while (document.defaultView.frameElement)
      document = document.defaultView.frameElement.ownerDocument;

    let cwu = document.defaultView.QueryInterface(Ci.nsIInterfaceRequestor).getInterface(Ci.nsIDOMWindowUtils);
    let scrollX = {}, scrollY = {};
    cwu.getScrollXY(false, scrollX, scrollY);

    let r = aElement.getBoundingClientRect();

    // step out of iframes and frames, offsetting scroll values
    for (let frame = aElement.ownerDocument.defaultView; frame.frameElement && frame != content; frame = frame.parent) {
      // adjust client coordinates' origin to be top left of iframe viewport
      let rect = frame.frameElement.getBoundingClientRect();
      let left = frame.getComputedStyle(frame.frameElement, "").borderLeftWidth;
      let top = frame.getComputedStyle(frame.frameElement, "").borderTopWidth;
      scrollX.value += rect.left + parseInt(left);
      scrollY.value += rect.top + parseInt(top);
    }

    return {x: r.left + scrollX.value,
            y: r.top + scrollY.value,
            w: r.width,
            h: r.height };
  }
};

var ErrorPageEventHandler = {
  handleEvent: function(aEvent) {
    switch (aEvent.type) {
      case "click": {
        // Don't trust synthetic events
        if (!aEvent.isTrusted)
          return;

        let target = aEvent.originalTarget;
        let errorDoc = target.ownerDocument;

        // If the event came from an ssl error page, it is probably either the "Add
        // Exception…" or "Get me out of here!" button
        if (errorDoc.documentURI.startsWith("about:certerror?e=nssBadCert")) {
          let perm = errorDoc.getElementById("permanentExceptionButton");
          let temp = errorDoc.getElementById("temporaryExceptionButton");
          if (target == temp || target == perm) {
            // Handle setting an cert exception and reloading the page
            try {
              // Add a new SSL exception for this URL
              let uri = Services.io.newURI(errorDoc.location.href, null, null);
              let sslExceptions = new SSLExceptions();

              if (target == perm)
                sslExceptions.addPermanentException(uri, errorDoc.defaultView);
              else
                sslExceptions.addTemporaryException(uri, errorDoc.defaultView);
            } catch (e) {
              dump("Failed to set cert exception: " + e + "\n");
            }
            errorDoc.location.reload();
          } else if (target == errorDoc.getElementById("getMeOutOfHereButton")) {
            errorDoc.location = "about:home";
          }
        } else if (errorDoc.documentURI.startsWith("about:blocked")) {
          // The event came from a button on a malware/phishing block page
          // First check whether it's malware or phishing, so that we can
          // use the right strings/links
          let isMalware = errorDoc.documentURI.contains("e=malwareBlocked");
          let bucketName = isMalware ? "WARNING_MALWARE_PAGE_" : "WARNING_PHISHING_PAGE_";
          let nsISecTel = Ci.nsISecurityUITelemetry;
          let isIframe = (errorDoc.defaultView.parent === errorDoc.defaultView);
          bucketName += isIframe ? "TOP_" : "FRAME_";

          let formatter = Cc["@mozilla.org/toolkit/URLFormatterService;1"].getService(Ci.nsIURLFormatter);

          if (target == errorDoc.getElementById("getMeOutButton")) {
            Telemetry.addData("SECURITY_UI", nsISecTel[bucketName + "GET_ME_OUT_OF_HERE"]);
            errorDoc.location = "about:home";
          } else if (target == errorDoc.getElementById("reportButton")) {
            // We log even if malware/phishing info URL couldn't be found:
            // the measurement is for how many users clicked the WHY BLOCKED button
            Telemetry.addData("SECURITY_UI", nsISecTel[bucketName + "WHY_BLOCKED"]);

            // This is the "Why is this site blocked" button.  For malware,
            // we can fetch a site-specific report, for phishing, we redirect
            // to the generic page describing phishing protection.
            if (isMalware) {
              // Get the stop badware "why is this blocked" report url, append the current url, and go there.
              try {
                let reportURL = formatter.formatURLPref("browser.safebrowsing.malware.reportURL");
                reportURL += errorDoc.location.href;
                BrowserApp.selectedBrowser.loadURI(reportURL);
              } catch (e) {
                Cu.reportError("Couldn't get malware report URL: " + e);
              }
            } else {
              // It's a phishing site, just link to the generic information page
              let url = Services.urlFormatter.formatURLPref("app.support.baseURL");
              BrowserApp.selectedBrowser.loadURI(url + "phishing-malware");
            }
          } else if (target == errorDoc.getElementById("ignoreWarningButton")) {
            Telemetry.addData("SECURITY_UI", nsISecTel[bucketName + "IGNORE_WARNING"]);

            // Allow users to override and continue through to the site,
            let webNav = BrowserApp.selectedBrowser.docShell.QueryInterface(Ci.nsIWebNavigation);
            let location = BrowserApp.selectedBrowser.contentWindow.location;
            webNav.loadURI(location, Ci.nsIWebNavigation.LOAD_FLAGS_BYPASS_CLASSIFIER, null, null, null);

            // ....but add a notify bar as a reminder, so that they don't lose
          // track after, e.g., tab switching.
          NativeWindow.doorhanger.show(Strings.browser.GetStringFromName("safeBrowsingDoorhanger"), "safebrowsing-warning", [], BrowserApp.selectedTab.id);
        }
      }
      break;
    }
  }
  }
};

var FormAssistant = {
  QueryInterface: XPCOMUtils.generateQI([Ci.nsIFormSubmitObserver]),

  // Used to keep track of the element that corresponds to the current
  // autocomplete suggestions
  _currentInputElement: null,

  _isBlocklisted: false,

  // Keep track of whether or not an invalid form has been submitted
  _invalidSubmit: false,

  init: function() {
    Services.obs.addObserver(this, "FormAssist:AutoComplete", false);
    Services.obs.addObserver(this, "FormAssist:Blocklisted", false);
    Services.obs.addObserver(this, "FormAssist:Hidden", false);
    Services.obs.addObserver(this, "invalidformsubmit", false);
    Services.obs.addObserver(this, "PanZoom:StateChange", false);

    // We need to use a capturing listener for focus events
    BrowserApp.deck.addEventListener("focus", this, true);
    BrowserApp.deck.addEventListener("click", this, true);
    BrowserApp.deck.addEventListener("input", this, false);
    BrowserApp.deck.addEventListener("pageshow", this, false);

    LoginManagerParent.init();
  },

  uninit: function() {
    Services.obs.removeObserver(this, "FormAssist:AutoComplete");
    Services.obs.removeObserver(this, "FormAssist:Blocklisted");
    Services.obs.removeObserver(this, "FormAssist:Hidden");
    Services.obs.removeObserver(this, "invalidformsubmit");
    Services.obs.removeObserver(this, "PanZoom:StateChange");

    BrowserApp.deck.removeEventListener("focus", this);
    BrowserApp.deck.removeEventListener("click", this);
    BrowserApp.deck.removeEventListener("input", this);
    BrowserApp.deck.removeEventListener("pageshow", this);
  },

  observe: function(aSubject, aTopic, aData) {
    switch (aTopic) {
      case "PanZoom:StateChange":
        // If the user is just touching the screen and we haven't entered a pan or zoom state yet do nothing
        if (aData == "TOUCHING" || aData == "WAITING_LISTENERS")
          break;
        if (aData == "NOTHING") {
          // only look for input elements, not contentEditable or multiline text areas
          let focused = BrowserApp.getFocusedInput(BrowserApp.selectedBrowser, true);
          if (!focused)
            break;

          if (this._showValidationMessage(focused))
            break;
          this._showAutoCompleteSuggestions(focused, function () {});
        } else {
          // temporarily hide the form assist popup while we're panning or zooming the page
          this._hideFormAssistPopup();
        }
        break;
      case "FormAssist:AutoComplete":
        if (!this._currentInputElement)
          break;

        let editableElement = this._currentInputElement.QueryInterface(Ci.nsIDOMNSEditableElement);

        // If we have an active composition string, commit it before sending
        // the autocomplete event with the text that will replace it.
        try {
          let imeEditor = editableElement.editor.QueryInterface(Ci.nsIEditorIMESupport);
          if (imeEditor.composing)
            imeEditor.forceCompositionEnd();
        } catch (e) {}

        editableElement.setUserInput(aData);

        let event = this._currentInputElement.ownerDocument.createEvent("Events");
        event.initEvent("DOMAutoComplete", true, true);
        this._currentInputElement.dispatchEvent(event);
        break;

      case "FormAssist:Blocklisted":
        this._isBlocklisted = (aData == "true");
        break;

      case "FormAssist:Hidden":
        this._currentInputElement = null;
        break;
    }
  },

  notifyInvalidSubmit: function notifyInvalidSubmit(aFormElement, aInvalidElements) {
    if (!aInvalidElements.length)
      return;

    // Ignore this notificaiton if the current tab doesn't contain the invalid form
    if (BrowserApp.selectedBrowser.contentDocument !=
        aFormElement.ownerDocument.defaultView.top.document)
      return;

    this._invalidSubmit = true;

    // Our focus listener will show the element's validation message
    let currentElement = aInvalidElements.queryElementAt(0, Ci.nsISupports);
    currentElement.focus();
  },

  handleEvent: function(aEvent) {
    switch (aEvent.type) {
      case "focus":
        let currentElement = aEvent.target;

        // Only show a validation message on focus.
        this._showValidationMessage(currentElement);
        break;

      case "click":
        currentElement = aEvent.target;

        // Prioritize a form validation message over autocomplete suggestions
        // when the element is first focused (a form validation message will
        // only be available if an invalid form was submitted)
        if (this._showValidationMessage(currentElement))
          break;

        let checkResultsClick = hasResults => {
          if (!hasResults) {
            this._hideFormAssistPopup();
          }
        };

        this._showAutoCompleteSuggestions(currentElement, checkResultsClick);
        break;

      case "input":
        currentElement = aEvent.target;

        // Since we can only show one popup at a time, prioritze autocomplete
        // suggestions over a form validation message
        let checkResultsInput = hasResults => {
          if (hasResults)
            return;

          if (this._showValidationMessage(currentElement))
            return;

          // If we're not showing autocomplete suggestions, hide the form assist popup
          this._hideFormAssistPopup();
        };

        this._showAutoCompleteSuggestions(currentElement, checkResultsInput);
        break;

      // Reset invalid submit state on each pageshow
      case "pageshow":
        if (!this._invalidSubmit)
          return;

        let selectedBrowser = BrowserApp.selectedBrowser;
        if (selectedBrowser) {
          let selectedDocument = selectedBrowser.contentDocument;
          let target = aEvent.originalTarget;
          if (target == selectedDocument || target.ownerDocument == selectedDocument)
            this._invalidSubmit = false;
        }
    }
  },

  // We only want to show autocomplete suggestions for certain elements
  _isAutoComplete: function _isAutoComplete(aElement) {
    if (!(aElement instanceof HTMLInputElement) || aElement.readOnly ||
        (aElement.getAttribute("type") == "password") ||
        (aElement.hasAttribute("autocomplete") &&
         aElement.getAttribute("autocomplete").toLowerCase() == "off"))
      return false;

    return true;
  },

  // Retrieves autocomplete suggestions for an element from the form autocomplete service.
  // aCallback(array_of_suggestions) is called when results are available.
  _getAutoCompleteSuggestions: function _getAutoCompleteSuggestions(aSearchString, aElement, aCallback) {
    // Cache the form autocomplete service for future use
    if (!this._formAutoCompleteService)
      this._formAutoCompleteService = Cc["@mozilla.org/satchel/form-autocomplete;1"].
                                      getService(Ci.nsIFormAutoComplete);

    let resultsAvailable = function (results) {
      let suggestions = [];
      for (let i = 0; i < results.matchCount; i++) {
        let value = results.getValueAt(i);

        // Do not show the value if it is the current one in the input field
        if (value == aSearchString)
          continue;

        // Supply a label and value, since they can differ for datalist suggestions
        suggestions.push({ label: value, value: value });
      }
      aCallback(suggestions);
    };

    this._formAutoCompleteService.autoCompleteSearchAsync(aElement.name || aElement.id,
                                                          aSearchString, aElement, null,
                                                          resultsAvailable);
  },

  /**
   * (Copied from mobile/xul/chrome/content/forms.js)
   * This function is similar to getListSuggestions from
   * components/satchel/src/nsInputListAutoComplete.js but sadly this one is
   * used by the autocomplete.xml binding which is not in used in fennec
   */
  _getListSuggestions: function _getListSuggestions(aElement) {
    if (!(aElement instanceof HTMLInputElement) || !aElement.list)
      return [];

    let suggestions = [];
    let filter = !aElement.hasAttribute("mozNoFilter");
    let lowerFieldValue = aElement.value.toLowerCase();

    let options = aElement.list.options;
    let length = options.length;
    for (let i = 0; i < length; i++) {
      let item = options.item(i);

      let label = item.value;
      if (item.label)
        label = item.label;
      else if (item.text)
        label = item.text;

      if (filter && !(label.toLowerCase().contains(lowerFieldValue)) )
        continue;
      suggestions.push({ label: label, value: item.value });
    }

    return suggestions;
  },

  // Retrieves autocomplete suggestions for an element from the form autocomplete service
  // and sends the suggestions to the Java UI, along with element position data. As
  // autocomplete queries are asynchronous, calls aCallback when done with a true
  // argument if results were found and false if no results were found.
  _showAutoCompleteSuggestions: function _showAutoCompleteSuggestions(aElement, aCallback) {
    if (!this._isAutoComplete(aElement)) {
      aCallback(false);
      return;
    }

    // Don't display the form auto-complete popup after the user starts typing
    // to avoid confusing somes IME. See bug 758820 and bug 632744.
    if (this._isBlocklisted && aElement.value.length > 0) {
      aCallback(false);
      return;
    }

    let resultsAvailable = autoCompleteSuggestions => {
      // On desktop, we show datalist suggestions below autocomplete suggestions,
      // without duplicates removed.
      let listSuggestions = this._getListSuggestions(aElement);
      let suggestions = autoCompleteSuggestions.concat(listSuggestions);

      // Return false if there are no suggestions to show
      if (!suggestions.length) {
        aCallback(false);
        return;
      }

      sendMessageToJava({
        type:  "FormAssist:AutoComplete",
        suggestions: suggestions,
        rect: ElementTouchHelper.getBoundingContentRect(aElement)
      });

      // Keep track of input element so we can fill it in if the user
      // selects an autocomplete suggestion
      this._currentInputElement = aElement;
      aCallback(true);
    };

    this._getAutoCompleteSuggestions(aElement.value, aElement, resultsAvailable);
  },

  // Only show a validation message if the user submitted an invalid form,
  // there's a non-empty message string, and the element is the correct type
  _isValidateable: function _isValidateable(aElement) {
    if (!this._invalidSubmit ||
        !aElement.validationMessage ||
        !(aElement instanceof HTMLInputElement ||
          aElement instanceof HTMLTextAreaElement ||
          aElement instanceof HTMLSelectElement ||
          aElement instanceof HTMLButtonElement))
      return false;

    return true;
  },

  // Sends a validation message and position data for an element to the Java UI.
  // Returns true if there's a validation message to show, false otherwise.
  _showValidationMessage: function _sendValidationMessage(aElement) {
    if (!this._isValidateable(aElement))
      return false;

    sendMessageToJava({
      type: "FormAssist:ValidationMessage",
      validationMessage: aElement.validationMessage,
      rect: ElementTouchHelper.getBoundingContentRect(aElement)
    });

    return true;
  },

  _hideFormAssistPopup: function _hideFormAssistPopup() {
    sendMessageToJava({ type: "FormAssist:Hide" });
  }
};

/**
 * An object to watch for Gecko status changes -- add-on installs, pref changes
 * -- and reflect them back to Java.
 */
let HealthReportStatusListener = {
  PREF_ACCEPT_LANG: "intl.accept_languages",
  PREF_BLOCKLIST_ENABLED: "extensions.blocklist.enabled",

  PREF_TELEMETRY_ENABLED:
//@line 5802 "/builds/slave/rel-m-beta-and_bld-00000000000/build/mobile/android/chrome/content/browser.js"
    "toolkit.telemetry.enabled",
//@line 5806 "/builds/slave/rel-m-beta-and_bld-00000000000/build/mobile/android/chrome/content/browser.js"

  init: function () {
    try {
      AddonManager.addAddonListener(this);
    } catch (ex) {
      console.log("Failed to initialize add-on status listener. FHR cannot report add-on state. " + ex);
    }

    console.log("Adding HealthReport:RequestSnapshot observer.");
    Services.obs.addObserver(this, "HealthReport:RequestSnapshot", false);
    Services.prefs.addObserver(this.PREF_ACCEPT_LANG, this, false);
    Services.prefs.addObserver(this.PREF_BLOCKLIST_ENABLED, this, false);
    if (this.PREF_TELEMETRY_ENABLED) {
      Services.prefs.addObserver(this.PREF_TELEMETRY_ENABLED, this, false);
    }
  },

  uninit: function () {
    Services.obs.removeObserver(this, "HealthReport:RequestSnapshot");
    Services.prefs.removeObserver(this.PREF_ACCEPT_LANG, this);
    Services.prefs.removeObserver(this.PREF_BLOCKLIST_ENABLED, this);
    if (this.PREF_TELEMETRY_ENABLED) {
      Services.prefs.removeObserver(this.PREF_TELEMETRY_ENABLED, this);
    }

    AddonManager.removeAddonListener(this);
  },

  observe: function (aSubject, aTopic, aData) {
    switch (aTopic) {
      case "HealthReport:RequestSnapshot":
        HealthReportStatusListener.sendSnapshotToJava();
        break;
      case "nsPref:changed":
        let response = {
          type: "Pref:Change",
          pref: aData,
          isUserSet: Services.prefs.prefHasUserValue(aData),
        };

        switch (aData) {
          case this.PREF_ACCEPT_LANG:
            response.value = Services.prefs.getCharPref(aData);
            break;
          case this.PREF_TELEMETRY_ENABLED:
          case this.PREF_BLOCKLIST_ENABLED:
            response.value = Services.prefs.getBoolPref(aData);
            break;
          default:
            console.log("Unexpected pref in HealthReportStatusListener: " + aData);
            return;
        }

        sendMessageToJava(response);
        break;
    }
  },

  MILLISECONDS_PER_DAY: 24 * 60 * 60 * 1000,

  COPY_FIELDS: [
    "blocklistState",
    "userDisabled",
    "appDisabled",
    "version",
    "type",
    "scope",
    "foreignInstall",
    "hasBinaryComponents",
  ],

  // Add-on types for which full details are recorded in FHR.
  // All other types are ignored.
  FULL_DETAIL_TYPES: [
    "plugin",
    "extension",
    "service",
  ],

  /**
   * Return true if the add-on is not of a type for which we report full details.
   * These add-ons will still make it over to Java, but will be filtered out.
   */
  _shouldIgnore: function (aAddon) {
    return this.FULL_DETAIL_TYPES.indexOf(aAddon.type) == -1;
  },

  _dateToDays: function (aDate) {
    return Math.floor(aDate.getTime() / this.MILLISECONDS_PER_DAY);
  },

  jsonForAddon: function (aAddon) {
    let o = {};
    if (aAddon.installDate) {
      o.installDay = this._dateToDays(aAddon.installDate);
    }
    if (aAddon.updateDate) {
      o.updateDay = this._dateToDays(aAddon.updateDate);
    }

    for (let field of this.COPY_FIELDS) {
      o[field] = aAddon[field];
    }

    return o;
  },

  notifyJava: function (aAddon, aNeedsRestart, aAction="Addons:Change") {
    let json = this.jsonForAddon(aAddon);
    if (this._shouldIgnore(aAddon)) {
      json.ignore = true;
    }
    sendMessageToJava({ type: aAction, id: aAddon.id, json: json });
  },

  // Add-on listeners.
  onEnabling: function (aAddon, aNeedsRestart) {
    this.notifyJava(aAddon, aNeedsRestart);
  },
  onDisabling: function (aAddon, aNeedsRestart) {
    this.notifyJava(aAddon, aNeedsRestart);
  },
  onInstalling: function (aAddon, aNeedsRestart) {
    this.notifyJava(aAddon, aNeedsRestart);
  },
  onUninstalling: function (aAddon, aNeedsRestart) {
    this.notifyJava(aAddon, aNeedsRestart, "Addons:Uninstalling");
  },
  onPropertyChanged: function (aAddon, aProperties) {
    this.notifyJava(aAddon);
  },
  onOperationCancelled: function (aAddon) {
    this.notifyJava(aAddon);
  },

  sendSnapshotToJava: function () {
    AddonManager.getAllAddons(function (aAddons) {
        let jsonA = {};
        if (aAddons) {
          for (let i = 0; i < aAddons.length; ++i) {
            let addon = aAddons[i];
            try {
              let addonJSON = HealthReportStatusListener.jsonForAddon(addon);
              if (HealthReportStatusListener._shouldIgnore(addon)) {
                addonJSON.ignore = true;
              }
              jsonA[addon.id] = addonJSON;
            } catch (e) {
              // Just skip this add-on.
            }
          }
        }

        // Now add prefs.
        let jsonP = {};
        for (let pref of [this.PREF_BLOCKLIST_ENABLED, this.PREF_TELEMETRY_ENABLED]) {
          if (!pref) {
            // This will be the case for PREF_TELEMETRY_ENABLED in developer builds.
            continue;
          }
          jsonP[pref] = {
            pref: pref,
            value: Services.prefs.getBoolPref(pref),
            isUserSet: Services.prefs.prefHasUserValue(pref),
          };
        }
        for (let pref of [this.PREF_ACCEPT_LANG]) {
          jsonP[pref] = {
            pref: pref,
            value: Services.prefs.getCharPref(pref),
            isUserSet: Services.prefs.prefHasUserValue(pref),
          };
        }

        console.log("Sending snapshot message.");
        sendMessageToJava({
          type: "HealthReport:Snapshot",
          json: {
            addons: jsonA,
            prefs: jsonP,
          },
        });
      }.bind(this));
  },
};

var XPInstallObserver = {
  init: function xpi_init() {
    Services.obs.addObserver(XPInstallObserver, "addon-install-blocked", false);
    Services.obs.addObserver(XPInstallObserver, "addon-install-started", false);

    AddonManager.addInstallListener(XPInstallObserver);
  },

  uninit: function xpi_uninit() {
    Services.obs.removeObserver(XPInstallObserver, "addon-install-blocked");
    Services.obs.removeObserver(XPInstallObserver, "addon-install-started");

    AddonManager.removeInstallListener(XPInstallObserver);
  },

  observe: function xpi_observer(aSubject, aTopic, aData) {
    switch (aTopic) {
      case "addon-install-started":
        NativeWindow.toast.show(Strings.browser.GetStringFromName("alertAddonsDownloading"), "short");
        break;
      case "addon-install-blocked":
        let installInfo = aSubject.QueryInterface(Ci.amIWebInstallInfo);
        let win = installInfo.originator;
        let tab = BrowserApp.getTabForWindow(win.top);
        if (!tab)
          return;

        let host = null;
        if (installInfo.originatingURI) {
          host = installInfo.originatingURI.host;
        }

        let brandShortName = Strings.brand.GetStringFromName("brandShortName");
        let notificationName, buttons, message;
        let strings = Strings.browser;
        let enabled = true;
        try {
          enabled = Services.prefs.getBoolPref("xpinstall.enabled");
        }
        catch (e) {}

        if (!enabled) {
          notificationName = "xpinstall-disabled";
          if (Services.prefs.prefIsLocked("xpinstall.enabled")) {
            message = strings.GetStringFromName("xpinstallDisabledMessageLocked");
            buttons = [];
          } else {
            message = strings.formatStringFromName("xpinstallDisabledMessage2", [brandShortName, host], 2);
            buttons = [{
              label: strings.GetStringFromName("xpinstallDisabledButton"),
              callback: function editPrefs() {
                Services.prefs.setBoolPref("xpinstall.enabled", true);
                return false;
              }
            }];
          }
        } else {
          notificationName = "xpinstall";
          if (host) {
            // We have a host which asked for the install.
            message = strings.formatStringFromName("xpinstallPromptWarning2", [brandShortName, host], 2);
          } else {
            // Without a host we address the add-on as the initiator of the install.
            let addon = null;
            if (installInfo.installs.length > 0) {
              addon = installInfo.installs[0].name;
            }
            if (addon) {
              // We have an addon name, show the regular message.
              message = strings.formatStringFromName("xpinstallPromptWarningLocal", [brandShortName, addon], 2);
            } else {
              // We don't have an addon name, show an alternative message.
              message = strings.formatStringFromName("xpinstallPromptWarningDirect", [brandShortName], 1);
            }
          }

          buttons = [{
            label: strings.GetStringFromName("xpinstallPromptAllowButton"),
            callback: function() {
              // Kick off the install
              installInfo.install();
              return false;
            }
          }];
        }
        NativeWindow.doorhanger.show(message, aTopic, buttons, tab.id);
        break;
    }
  },

  onInstallEnded: function(aInstall, aAddon) {
    let needsRestart = false;
    if (aInstall.existingAddon && (aInstall.existingAddon.pendingOperations & AddonManager.PENDING_UPGRADE))
      needsRestart = true;
    else if (aAddon.pendingOperations & AddonManager.PENDING_INSTALL)
      needsRestart = true;

    if (needsRestart) {
      this.showRestartPrompt();
    } else {
      // Display completion message for new installs or updates not done Automatically
      if (!aInstall.existingAddon || !AddonManager.shouldAutoUpdate(aInstall.existingAddon)) {
        let message = Strings.browser.GetStringFromName("alertAddonsInstalledNoRestart");
        NativeWindow.toast.show(message, "short");
      }
    }
  },

  onInstallFailed: function(aInstall) {
    NativeWindow.toast.show(Strings.browser.GetStringFromName("alertAddonsFail"), "short");
  },

  onDownloadProgress: function xpidm_onDownloadProgress(aInstall) {},

  onDownloadFailed: function(aInstall) {
    this.onInstallFailed(aInstall);
  },

  onDownloadCancelled: function(aInstall) {
    let host = (aInstall.originatingURI instanceof Ci.nsIStandardURL) && aInstall.originatingURI.host;
    if (!host)
      host = (aInstall.sourceURI instanceof Ci.nsIStandardURL) && aInstall.sourceURI.host;

    let error = (host || aInstall.error == 0) ? "addonError" : "addonLocalError";
    if (aInstall.error != 0)
      error += aInstall.error;
    else if (aInstall.addon && aInstall.addon.blocklistState == Ci.nsIBlocklistService.STATE_BLOCKED)
      error += "Blocklisted";
    else if (aInstall.addon && (!aInstall.addon.isCompatible || !aInstall.addon.isPlatformCompatible))
      error += "Incompatible";
    else
      return; // No need to show anything in this case.

    let msg = Strings.browser.GetStringFromName(error);
    // TODO: formatStringFromName
    msg = msg.replace("#1", aInstall.name);
    if (host)
      msg = msg.replace("#2", host);
    msg = msg.replace("#3", Strings.brand.GetStringFromName("brandShortName"));
    msg = msg.replace("#4", Services.appinfo.version);

    NativeWindow.toast.show(msg, "short");
  },

  showRestartPrompt: function() {
    let buttons = [{
      label: Strings.browser.GetStringFromName("notificationRestart.button"),
      callback: function() {
        // Notify all windows that an application quit has been requested
        let cancelQuit = Cc["@mozilla.org/supports-PRBool;1"].createInstance(Ci.nsISupportsPRBool);
        Services.obs.notifyObservers(cancelQuit, "quit-application-requested", "restart");

        // If nothing aborted, quit the app
        if (cancelQuit.data == false) {
          let appStartup = Cc["@mozilla.org/toolkit/app-startup;1"].getService(Ci.nsIAppStartup);
          appStartup.quit(Ci.nsIAppStartup.eRestart | Ci.nsIAppStartup.eAttemptQuit);
        }
      }
    }];

    let message = Strings.browser.GetStringFromName("notificationRestart.normal");
    NativeWindow.doorhanger.show(message, "addon-app-restart", buttons, BrowserApp.selectedTab.id, { persistence: -1 });
  },

  hideRestartPrompt: function() {
    NativeWindow.doorhanger.hide("addon-app-restart", BrowserApp.selectedTab.id);
  }
};

// Blindly copied from Safari documentation for now.
const kViewportMinScale  = 0;
const kViewportMaxScale  = 10;
const kViewportMinWidth  = 200;
const kViewportMaxWidth  = 10000;
const kViewportMinHeight = 223;
const kViewportMaxHeight = 10000;
```

```
var ViewportHandler = {
  // The cached viewport metadata for each document. We tie viewport metadata to each document
  // instead of to each tab so that we don't have to update it when the document changes. Using an
  // ES6 weak map lets us avoid leaks.
  _metadata: new WeakMap(),

  init: function init() {
    addEventListener("DOMMetaAdded", this, false);
    Services.obs.addObserver(this, "Window:Resize", false);
  },

  uninit: function uninit() {
    removeEventListener("DOMMetaAdded", this, false);
    Services.obs.removeObserver(this, "Window:Resize");
  },

  handleEvent: function handleEvent(aEvent) {
    switch (aEvent.type) {
      case "DOMMetaAdded":
        let target = aEvent.originalTarget;
        if (target.name != "viewport")
          break;
        let document = target.ownerDocument;
        let browser = BrowserApp.getBrowserForDocument(document);
        let tab = BrowserApp.getTabForBrowser(browser);
        if (tab && tab.contentDocumentIsDisplayed)
          this.updateMetadata(tab, false);
        break;
    }
  },

  observe: function(aSubject, aTopic, aData) {
    switch (aTopic) {
      case "Window:Resize":
        if (window.outerWidth == gScreenWidth && window.outerHeight == gScreenHeight)
          break;
        if (window.outerWidth == 0 || window.outerHeight == 0)
          break;

        let oldScreenWidth = gScreenWidth;
        gScreenWidth = window.outerWidth * window.devicePixelRatio;
        gScreenHeight = window.outerHeight * window.devicePixelRatio;
        let tabs = BrowserApp.tabs;
        for (let i = 0; i < tabs.length; i++)
          tabs[i].updateViewportSize(oldScreenWidth);
        break;
    }
  },

  updateMetadata: function updateMetadata(tab, aInitialLoad) {
    let contentWindow = tab.browser.contentWindow;
    if (contentWindow.document.documentElement) {
      let metadata = this.getViewportMetadata(contentWindow);
      tab.updateViewportMetadata(metadata, aInitialLoad);
    }
  },

  /**
   * Returns the ViewportMetadata object.
   */
  getViewportMetadata: function getViewportMetadata(aWindow) {
    let windowUtils = aWindow.QueryInterface(Ci.nsIInterfaceRequestor).getInterface(Ci.nsIDOMWindowUtils);

    // viewport details found here
    // http://developer.apple.com/safari/library/documentation/AppleApplications/Reference/SafariHTMLRef/Articles/MetaTags.html
    // http://developer.apple.com/safari/library/documentation/AppleApplications/Reference/SafariWebContent/UsingtheViewport/UsingtheViewport.html

    // Note: These values will be NaN if parseFloat or parseInt doesn't find a number.
    // Remember that NaN is contagious: Math.max(1, NaN) == Math.min(1, NaN) == NaN.
    let hasMetaViewport = true;
    let scale = parseFloat(windowUtils.getDocumentMetadata("viewport-initial-scale"));
    let minScale = parseFloat(windowUtils.getDocumentMetadata("viewport-minimum-scale"));
    let maxScale = parseFloat(windowUtils.getDocumentMetadata("viewport-maximum-scale"));

    let widthStr = windowUtils.getDocumentMetadata("viewport-width");
    let heightStr = windowUtils.getDocumentMetadata("viewport-height");
    let width = this.clamp(parseInt(widthStr), kViewportMinWidth, kViewportMaxWidth) || 0;
    let height = this.clamp(parseInt(heightStr), kViewportMinHeight, kViewportMaxHeight) || 0;

    // Allow zoom unless explicity disabled or minScale and maxScale are equal.
    // WebKit allows 0, "no", or "false" for viewport-user-scalable.
    // Note: NaN != NaN. Therefore if minScale and maxScale are undefined the clause has no effect.
    let allowZoomStr = windowUtils.getDocumentMetadata("viewport-user-scalable");
    let allowZoom = !/^(0|no|false)$/.test(allowZoomStr) && (minScale != maxScale);

    // Double-tap should always be disabled if allowZoom is disabled. So we initialize
    // allowDoubleTapZoom to the same value as allowZoom and have additional conditions to
    // disable it in updateViewportSize.
    let allowDoubleTapZoom = allowZoom;

    let autoSize = true;

    if (isNaN(scale) && isNaN(minScale) && isNaN(maxScale) && allowZoomStr == "" && widthStr == "" && heightStr == "") {
      // Only check for HandheldFriendly if we don't have a viewport meta tag
      let handheldFriendly = windowUtils.getDocumentMetadata("HandheldFriendly");
      if (handheldFriendly == "true") {
        return new ViewportMetadata({
          defaultZoom: 1,
          autoSize: true,
          allowZoom: true,
          allowDoubleTapZoom: false
        });
      }

      let doctype = aWindow.document.doctype;
      if (doctype && /(WAP|WML|Mobile)/.test(doctype.publicId)) {
        return new ViewportMetadata({
          defaultZoom: 1,
          autoSize: true,
          allowZoom: true,
          allowDoubleTapZoom: false
        });
      }

      hasMetaViewport = false;
      let defaultZoom = Services.prefs.getIntPref("browser.viewport.defaultZoom");
      if (defaultZoom >= 0) {
        scale = defaultZoom / 1000;
        autoSize = false;
      }
    }

    scale = this.clamp(scale, kViewportMinScale, kViewportMaxScale);
    minScale = this.clamp(minScale, kViewportMinScale, kViewportMaxScale);
    maxScale = this.clamp(maxScale, minScale, kViewportMaxScale);

    if (autoSize) {
      // If initial scale is 1.0 and width is not set, assume width=device-width
      autoSize = (widthStr == "device-width" ||
                  (!widthStr && (heightStr == "device-height" || scale == 1.0)));
    }

    let isRTL = aWindow.document.documentElement.dir == "rtl";

    return new ViewportMetadata({
      defaultZoom: scale,
      minZoom: minScale,
      maxZoom: maxScale,
      width: width,
      height: height,
      autoSize: autoSize,
      allowZoom: allowZoom,
      allowDoubleTapZoom: allowDoubleTapZoom,
      isSpecified: hasMetaViewport,
      isRTL: isRTL
    });
  },

  clamp: function(num, min, max) {
    return Math.max(min, Math.min(max, num));
  },

  get displayDPI() {
    let utils = window.QueryInterface(Ci.nsIInterfaceRequestor).getInterface(Ci.nsIDOMWindowUtils);
    delete this.displayDPI;
    return this.displayDPI = utils.displayDPI;
  },

  /**
   * Returns the viewport metadata for the given document, or the default metrics if no viewport
   * metadata is available for that document.
   */
  getMetadataForDocument: function getMetadataForDocument(aDocument) {
    let metadata = this._metadata.get(aDocument, new ViewportMetadata());
    return metadata;
  },

  /** Updates the saved viewport metadata for the given content document. */
  setMetadataForDocument: function setMetadataForDocument(aDocument, aMetadata) {
    if (!aMetadata)
      this._metadata.delete(aDocument);
    else
      this._metadata.set(aDocument, aMetadata);
  }

};
```

```
/**
 * An object which represents the page's preferred viewport properties:
 *   width (int): The CSS viewport width in px.
 *   height (int): The CSS viewport height in px.
 *   defaultZoom (float): The initial scale when the page is loaded.
 *   minZoom (float): The minimum zoom level.
 *   maxZoom (float): The maximum zoom level.
 *   autoSize (boolean): Resize the CSS viewport when the window resizes.
 *   allowZoom (boolean): Let the user zoom in or out.
 *   allowDoubleTapZoom (boolean): Allow double-tap to zoom in.
 *   isSpecified (boolean): Whether the page viewport is specified or not.
 */
function ViewportMetadata(aMetadata = {}) {
  this.width = ("width" in aMetadata) ? aMetadata.width : 0;
  this.height = ("height" in aMetadata) ? aMetadata.height : 0;
  this.defaultZoom = ("defaultZoom" in aMetadata) ? aMetadata.defaultZoom : 0;
  this.minZoom = ("minZoom" in aMetadata) ? aMetadata.minZoom : 0;
  this.maxZoom = ("maxZoom" in aMetadata) ? aMetadata.maxZoom : 0;
  this.autoSize = ("autoSize" in aMetadata) ? aMetadata.autoSize : false;
  this.allowZoom = ("allowZoom" in aMetadata) ? aMetadata.allowZoom : true;
  this.allowDoubleTapZoom = ("allowDoubleTapZoom" in aMetadata) ? aMetadata.allowDoubleTapZoom : true;
  this.isSpecified = ("isSpecified" in aMetadata) ? aMetadata.isSpecified : false;
  this.isRTL = ("isRTL" in aMetadata) ? aMetadata.isRTL : false;
  Object.seal(this);
}

ViewportMetadata.prototype = {
  width: null,
  height: null,
  defaultZoom: null,
  minZoom: null,
  maxZoom: null,
  autoSize: null,
  allowZoom: null,
  allowDoubleTapZoom: null,
  isSpecified: null,
  isRTL: null,

  toString: function() {
    return "width=" + this.width
         + "; height=" + this.height
         + "; defaultZoom=" + this.defaultZoom
         + "; minZoom=" + this.minZoom
         + "; maxZoom=" + this.maxZoom
         + "; autoSize=" + this.autoSize
         + "; allowZoom=" + this.allowZoom
         + "; allowDoubleTapZoom=" + this.allowDoubleTapZoom
         + "; isSpecified=" + this.isSpecified
         + "; isRTL=" + this.isRTL;
  }
};


/**
 * Handler for blocked popups, triggered by DOMUpdatePageReport events in browser.xml
 */
var PopupBlockerObserver = {
  onUpdatePageReport: function onUpdatePageReport(aEvent) {
    let browser = BrowserApp.selectedBrowser;
    if (aEvent.originalTarget != browser)
      return;

    if (!browser.pageReport)
      return;

    let result = Services.perms.testExactPermission(BrowserApp.selectedBrowser.currentURI, "popup");
    if (result == Ci.nsIPermissionManager.DENY_ACTION)
      return;

    // Only show the notification again if we've not already shown it. Since
    // notifications are per-browser, we don't need to worry about re-adding
    // it.
    if (!browser.pageReport.reported) {
      if (Services.prefs.getBoolPref("privacy.popups.showBrowserMessage")) {
        let brandShortName = Strings.brand.GetStringFromName("brandShortName");
        let popupCount = browser.pageReport.length;

        let strings = Strings.browser;
        let message = PluralForm.get(popupCount, strings.GetStringFromName("popup.message"))
                                .replace("#1", brandShortName)
                                .replace("#2", popupCount);

        let buttons = [
          {
            label: strings.GetStringFromName("popup.show"),
            callback: function(aChecked) {
              // Set permission before opening popup windows
              if (aChecked)
                PopupBlockerObserver.allowPopupsForSite(true);

              PopupBlockerObserver.showPopupsForSite();
            }
          },
          {
            label: strings.GetStringFromName("popup.dontShow"),
            callback: function(aChecked) {
              if (aChecked)
                PopupBlockerObserver.allowPopupsForSite(false);
            }
          }
        ];

        let options = { checkbox: Strings.browser.GetStringFromName("popup.dontAskAgain") };
        NativeWindow.doorhanger.show(message, "popup-blocked", buttons, null, options);
      }
      // Record the fact that we've reported this blocked popup, so we don't
      // show it again.
      browser.pageReport.reported = true;
    }
  },

  allowPopupsForSite: function allowPopupsForSite(aAllow) {
    let currentURI = BrowserApp.selectedBrowser.currentURI;
    Services.perms.add(currentURI, "popup", aAllow
                       ?  Ci.nsIPermissionManager.ALLOW_ACTION
                       :  Ci.nsIPermissionManager.DENY_ACTION);
    dump("Allowing popups for: " + currentURI);
  },

  showPopupsForSite: function showPopupsForSite() {
    let uri = BrowserApp.selectedBrowser.currentURI;
    let pageReport = BrowserApp.selectedBrowser.pageReport;
    if (pageReport) {
      for (let i = 0; i < pageReport.length; ++i) {
        let popupURIspec = pageReport[i].popupWindowURI.spec;

        // Sometimes the popup URI that we get back from the pageReport
        // isn't useful (for instance, netscape.com's popup URI ends up
        // being "http://www.netscape.com", which isn't really the URI of
        // the popup they're trying to show).  This isn't going to be
        // useful to the user, so we won't create a menu item for it.
        if (popupURIspec == "" || popupURIspec == "about:blank" || popupURIspec == uri.spec)
          continue;

        let popupFeatures = pageReport[i].popupWindowFeatures;
        let popupName = pageReport[i].popupWindowName;

        let parent = BrowserApp.selectedTab;
        let isPrivate = PrivateBrowsingUtils.isWindowPrivate(parent.browser.contentWindow);
        BrowserApp.addTab(popupURIspec, { parentId: parent.id, isPrivate: isPrivate });
      }
    }
  }
};


var IndexedDB = {
  _permissionsPrompt: "indexedDB-permissions-prompt",
  _permissionsResponse: "indexedDB-permissions-response",

  _quotaPrompt: "indexedDB-quota-prompt",
  _quotaResponse: "indexedDB-quota-response",
  _quotaCancel: "indexedDB-quota-cancel",

  init: function IndexedDB_init() {
    Services.obs.addObserver(this, this._permissionsPrompt, false);
    Services.obs.addObserver(this, this._quotaPrompt, false);
    Services.obs.addObserver(this, this._quotaCancel, false);
  },

  uninit: function IndexedDB_uninit() {
    Services.obs.removeObserver(this, this._permissionsPrompt);
    Services.obs.removeObserver(this, this._quotaPrompt);
    Services.obs.removeObserver(this, this._quotaCancel);
  },

  observe: function IndexedDB_observe(subject, topic, data) {
    if (topic != this._permissionsPrompt &&
        topic != this._quotaPrompt &&
        topic != this._quotaCancel) {
      throw new Error("Unexpected topic!");
    }

    let requestor = subject.QueryInterface(Ci.nsIInterfaceRequestor);
    let contentWindow = requestor.getInterface(Ci.nsIDOMWindow);
    let contentDocument = contentWindow.document;
    let tab = BrowserApp.getTabForWindow(contentWindow);
    if (!tab)
      return;

    let host = contentDocument.documentURIObject.asciiHost;

    let strings = Strings.browser;

    let message, responseTopic;
    if (topic == this._permissionsPrompt) {
      message = strings.formatStringFromName("offlineApps.ask", [host], 1);
      responseTopic = this._permissionsResponse;
    } else if (topic == this._quotaPrompt) {
      message = strings.formatStringFromName("indexedDBQuota.wantsTo", [ host, data ], 2);
      responseTopic = this._quotaResponse;
    } else if (topic == this._quotaCancel) {
      responseTopic = this._quotaResponse;
    }

    const firstTimeoutDuration = 300000; // 5 minutes

    let timeoutId;

    let notificationID = responseTopic + host;
    let observer = requestor.getInterface(Ci.nsIObserver);

    // This will be set to the result of PopupNotifications.show() below, or to
    // the result of PopupNotifications.getNotification() if this is a
    // quotaCancel notification.
    let notification;

    function timeoutNotification() {
      // Remove the notification.
      NativeWindow.doorhanger.hide(notificationID, tab.id);

      // Clear all of our timeout stuff. We may be called directly, not just
      // when the timeout actually elapses.
      clearTimeout(timeoutId);

      // And tell the page that the popup timed out.
      observer.observe(null, responseTopic, Ci.nsIPermissionManager.UNKNOWN_ACTION);
    }

    if (topic == this._quotaCancel) {
      NativeWindow.doorhanger.hide(notificationID, tab.id);
      timeoutNotification();
      observer.observe(null, responseTopic, Ci.nsIPermissionManager.UNKNOWN_ACTION);
      return;
    }

    let buttons = [{
      label: strings.GetStringFromName("offlineApps.allow"),
      callback: function() {
        clearTimeout(timeoutId);
        observer.observe(null, responseTopic, Ci.nsIPermissionManager.ALLOW_ACTION);
      }
    },
    {
      label: strings.GetStringFromName("offlineApps.dontAllow2"),
      callback: function(aChecked) {
        clearTimeout(timeoutId);
        let action = aChecked ? Ci.nsIPermissionManager.DENY_ACTION : Ci.nsIPermissionManager.UNKNOWN_ACTION;
        observer.observe(null, responseTopic, action);
      }
    }];

    let options = { checkbox: Strings.browser.GetStringFromName("offlineApps.dontAskAgain") };
    NativeWindow.doorhanger.show(message, notificationID, buttons, tab.id, options);

    // Set the timeoutId after the popup has been created, and use the long
    // timeout value. If the user doesn't notice the popup after this amount of
    // time then it is most likely not visible and we want to alert the page.
    timeoutId = setTimeout(timeoutNotification, firstTimeoutDuration);
  }
};

var CharacterEncoding = {
  _charsets: [],

  init: function init() {
    Services.obs.addObserver(this, "CharEncoding:Get", false);
    Services.obs.addObserver(this, "CharEncoding:Set", false);
    this.sendState();
  },

  uninit: function uninit() {
    Services.obs.removeObserver(this, "CharEncoding:Get");
    Services.obs.removeObserver(this, "CharEncoding:Set");
  },

  observe: function observe(aSubject, aTopic, aData) {
    switch (aTopic) {
      case "CharEncoding:Get":
        this.getEncoding();
        break;
      case "CharEncoding:Set":
        this.setEncoding(aData);
        break;
    }
  },

  sendState: function sendState() {
    let showCharEncoding = "false";
    try {
      showCharEncoding = Services.prefs.getComplexValue("browser.menu.showCharacterEncoding", Ci.nsIPrefLocalizedString).data;
    } catch (e) { /* Optional */ }

    sendMessageToJava({
      type: "CharEncoding:State",
      visible: showCharEncoding
    });
  },

  getEncoding: function getEncoding() {
    function infoToCharset(info) {
      return { code: info.value, title: info.label };
    }

    if (!this._charsets.length) {
      let data = CharsetMenu.getData();

      // In the desktop UI, the pinned charsets are shown above the rest.
      let pinnedCharsets = data.pinnedCharsets.map(infoToCharset);
      let otherCharsets = data.otherCharsets.map(infoToCharset)

      this._charsets = pinnedCharsets.concat(otherCharsets);
    }

    // Look for the index of the selected charset. Default to -1 if the
    // doc charset isn't found in the list of available charsets.
    let docCharset = BrowserApp.selectedBrowser.contentDocument.characterSet;
    let selected = -1;
    let charsetCount = this._charsets.length;

    for (let i = 0; i < charsetCount; i++) {
      if (this._charsets[i].code === docCharset) {
        selected = i;
        break;
      }
    }

    sendMessageToJava({
      type: "CharEncoding:Data",
      charsets: this._charsets,
      selected: selected
    });
  },

  setEncoding: function setEncoding(aEncoding) {
    let browser = BrowserApp.selectedBrowser;
    browser.docShell.gatherCharsetMenuTelemetry();
    browser.docShell.charset = aEncoding;
    browser.reload(Ci.nsIWebNavigation.LOAD_FLAGS_CHARSET_CHANGE);
  }
};

var IdentityHandler = {
  // No trusted identity information. No site identity icon is shown.
  IDENTITY_MODE_UNKNOWN: "unknown",

  // Minimal SSL CA-signed domain verification. Blue lock icon is shown.
  IDENTITY_MODE_DOMAIN_VERIFIED: "verified",

  // High-quality identity information. Green lock icon is shown.
  IDENTITY_MODE_IDENTIFIED: "identified",

  // The following mixed content modes are only used if "security.mixed_content.block_active_content"
  // is enabled. Even though the mixed content state and identitity state are orthogonal,
  // our Java frontend coalesces them into one indicator.

  // Blocked active mixed content. Shield icon is shown, with a popup option to load content.
  IDENTITY_MODE_MIXED_CONTENT_BLOCKED: "mixed_content_blocked",

  // Loaded active mixed content. Yellow triangle icon is shown.
  IDENTITY_MODE_MIXED_CONTENT_LOADED: "mixed_content_loaded",

  // Cache the most recent SSLStatus and Location seen in getIdentityStrings
  _lastStatus : null,
  _lastLocation : null,

  /**
   * Helper to parse out the important parts of _lastStatus (of the SSL cert in
   * particular) for use in constructing identity UI strings
  */
  getIdentityData : function() {
    let result = {};
    let status = this._lastStatus.QueryInterface(Components.interfaces.nsISSLStatus);
    let cert = status.serverCert;

    // Human readable name of Subject
    result.subjectOrg = cert.organization;

    // SubjectName fields, broken up for individual access
    if (cert.subjectName) {
      result.subjectNameFields = {};
      cert.subjectName.split(",").forEach(function(v) {
        let field = v.split("=");
        this[field[0]] = field[1];
      }, result.subjectNameFields);

      // Call out city, state, and country specifically
      result.city = result.subjectNameFields.L;
      result.state = result.subjectNameFields.ST;
      result.country = result.subjectNameFields.C;
    }

    // Human readable name of Certificate Authority
    result.caOrg =  cert.issuerOrganization || cert.issuerCommonName;
    result.cert = cert;

    return result;
  },

  /**
   * Determines the identity mode corresponding to the icon we show in the urlbar.
   */
  getIdentityMode: function getIdentityMode(aState) {
    if (aState & Ci.nsIWebProgressListener.STATE_BLOCKED_MIXED_ACTIVE_CONTENT)
      return this.IDENTITY_MODE_MIXED_CONTENT_BLOCKED;

    // Only show an indicator for loaded mixed content if the pref to block it is enabled
    if ((aState & Ci.nsIWebProgressListener.STATE_LOADED_MIXED_ACTIVE_CONTENT) &&
         Services.prefs.getBoolPref("security.mixed_content.block_active_content"))
      return this.IDENTITY_MODE_MIXED_CONTENT_LOADED;

    if (aState & Ci.nsIWebProgressListener.STATE_IDENTITY_EV_TOPLEVEL)
      return this.IDENTITY_MODE_IDENTIFIED;

    if (aState & Ci.nsIWebProgressListener.STATE_IS_SECURE)
      return this.IDENTITY_MODE_DOMAIN_VERIFIED;

    return this.IDENTITY_MODE_UNKNOWN;
  },

  /**
   * Determine the identity of the page being displayed by examining its SSL cert
   * (if available). Return the data needed to update the UI.
   */
  checkIdentity: function checkIdentity(aState, aBrowser) {
    this._lastStatus = aBrowser.securityUI
                               .QueryInterface(Components.interfaces.nsISSLStatusProvider)
                               .SSLStatus;

    // Don't pass in the actual location object, since it can cause us to
    // hold on to the window object too long.  Just pass in the fields we
    // care about. (bug 424829)
    let locationObj = {};
    try {
      let location = aBrowser.contentWindow.location;
      locationObj.host = location.host;
      locationObj.hostname = location.hostname;
      locationObj.port = location.port;
    } catch (ex) {
      // Can sometimes throw if the URL being visited has no host/hostname,
      // e.g. about:blank. The _state for these pages means we won't need these
      // properties anyways, though.
    }
    this._lastLocation = locationObj;

    let mode = this.getIdentityMode(aState);
    let result = { mode: mode };

    // Don't show identity data for pages with an unknown identity or if any
    // mixed content is loaded (mixed display content is loaded by default).
    if (mode == this.IDENTITY_MODE_UNKNOWN ||
        aState & Ci.nsIWebProgressListener.STATE_IS_BROKEN)
      return result;

    // Ideally we'd just make this a Java string
    result.encrypted = Strings.browser.GetStringFromName("identity.encrypted2");
    result.host = this.getEffectiveHost();

    let iData = this.getIdentityData();
    result.verifier = Strings.browser.formatStringFromName("identity.identified.verifier", [iData.caOrg], 1);

    // If the cert is identified, then we can populate the results with credentials
    if (aState & Ci.nsIWebProgressListener.STATE_IDENTITY_EV_TOPLEVEL) {
      result.owner = iData.subjectOrg;

      // Build an appropriate supplemental block out of whatever location data we have
      let supplemental = "";
      if (iData.city)
        supplemental += iData.city + "\n";
      if (iData.state && iData.country)
        supplemental += Strings.browser.formatStringFromName("identity.identified.state_and_country", [iData.state, iData.country], 2);
      else if (iData.state) // State only
        supplemental += iData.state;
      else if (iData.country) // Country only
        supplemental += iData.country;
      result.supplemental = supplemental;

      return result;
    }

    // Otherwise, we don't know the cert owner
    result.owner = Strings.browser.GetStringFromName("identity.ownerUnknown3");

    // Cache the override service the first time we need to check it
    if (!this._overrideService)
      this._overrideService = Cc["@mozilla.org/security/certoverride;1"].getService(Ci.nsICertOverrideService);

    // Check whether this site is a security exception. XPConnect does the right
    // thing here in terms of converting _lastLocation.port from string to int, but
    // the overrideService doesn't like undefined ports, so make sure we have
    // something in the default case (bug 432241).
    // .hostname can return an empty string in some exceptional cases -
    // hasMatchingOverride does not handle that, so avoid calling it.
    // Updating the tooltip value in those cases isn't critical.
    // FIXME: Fixing bug 646690 would probably makes this check unnecessary
    if (this._lastLocation.hostname &&
        this._overrideService.hasMatchingOverride(this._lastLocation.hostname,
                                                  (this._lastLocation.port || 443),
                                                  iData.cert, {}, {}))
      result.verifier = Strings.browser.GetStringFromName("identity.identified.verified_by_you");

    return result;
  },

  /**
   * Return the eTLD+1 version of the current hostname
   */
  getEffectiveHost: function getEffectiveHost() {
    if (!this._IDNService)
      this._IDNService = Cc["@mozilla.org/network/idn-service;1"]
                         .getService(Ci.nsIIDNService);
    try {
      let baseDomain = Services.eTLD.getBaseDomainFromHost(this._lastLocation.hostname);
      return this._IDNService.convertToDisplayIDN(baseDomain, {});
    } catch (e) {
      // If something goes wrong (e.g. hostname is an IP address) just fail back
      // to the full domain.
      return this._lastLocation.hostname;
    }
  }
};

function OverscrollController(aTab) {
  this.tab = aTab;
}

OverscrollController.prototype = {
  supportsCommand : function supportsCommand(aCommand) {
    if (aCommand != "cmd_linePrevious" && aCommand != "cmd_scrollPageUp")
      return false;

    return (this.tab.getViewport().y == 0);
  },

  isCommandEnabled : function isCommandEnabled(aCommand) {
    return this.supportsCommand(aCommand);
  },

  doCommand : function doCommand(aCommand){
    sendMessageToJava({ type: "ToggleChrome:Focus" });
  },

  onEvent : function onEvent(aEvent) { }
};

var SearchEngines = {
  _contextMenuId: null,
  PREF_SUGGEST_ENABLED: "browser.search.suggest.enabled",
  PREF_SUGGEST_PROMPTED: "browser.search.suggest.prompted",

  init: function init() {
    Services.obs.addObserver(this, "SearchEngines:Add", false);
    Services.obs.addObserver(this, "SearchEngines:GetVisible", false);
    Services.obs.addObserver(this, "SearchEngines:Remove", false);
    Services.obs.addObserver(this, "SearchEngines:RestoreDefaults", false);
    Services.obs.addObserver(this, "SearchEngines:SetDefault", false);

    let filter = {
      matches: function (aElement) {
        // Copied from body of isTargetAKeywordField function in nsContextMenu.js
        if(!(aElement instanceof HTMLInputElement))
          return false;
        let form = aElement.form;
        if (!form || aElement.type == "password")
          return false;

        let method = form.method.toUpperCase();

        // These are the following types of forms we can create keywords for:
        //
        // method    encoding type        can create keyword
        // GET       *                                   YES
        //           *                                   YES
        // POST      *                                   YES
        // POST      application/x-www-form-urlencoded   YES
        // POST      text/plain                          NO ( a little tricky to do)
        // POST      multipart/form-data                 NO
        // POST      everything else                     YES
        return (method == "GET" || method == "") ||
               (form.enctype != "text/plain") && (form.enctype != "multipart/form-data");
      }
    };
    SelectionHandler.addAction({
      id: "search_add_action",
      label: Strings.browser.GetStringFromName("contextmenu.addSearchEngine2"),
      icon: "drawable://ab_add_search_engine",
      selector: filter,
      action: function(aElement) {
        UITelemetry.addEvent("action.1", "actionbar", null, "add_search_engine");
        SearchEngines.addEngine(aElement);
      }
    });
  },

  uninit: function uninit() {
    Services.obs.removeObserver(this, "SearchEngines:Add");
    Services.obs.removeObserver(this, "SearchEngines:GetVisible");
    Services.obs.removeObserver(this, "SearchEngines:Remove");
    Services.obs.removeObserver(this, "SearchEngines:RestoreDefaults");
    Services.obs.removeObserver(this, "SearchEngines:SetDefault");
    if (this._contextMenuId != null)
      NativeWindow.contextmenus.remove(this._contextMenuId);
  },

  // Fetch list of search engines. all ? All engines : Visible engines only.
  _handleSearchEnginesGetVisible: function _handleSearchEnginesGetVisible(rv, all) {
    if (!Components.isSuccessCode(rv)) {
      Cu.reportError("Could not initialize search service, bailing out.");
      return;
    }

    let engineData = Services.search.getVisibleEngines({});
    let searchEngines = engineData.map(function (engine) {
      return {
        name: engine.name,
        identifier: engine.identifier,
        iconURI: (engine.iconURI ? engine.iconURI.spec : null),
        hidden: engine.hidden
      };
    });

    let suggestTemplate = null;
    let suggestEngine = null;

    // Check to see if the default engine supports search suggestions. We only need to check
    // the default engine because we only show suggestions for the default engine in the UI.
    let engine = Services.search.defaultEngine;
    if (engine.supportsResponseType("application/x-suggestions+json")) {
      suggestEngine = engine.name;
      suggestTemplate = engine.getSubmission("__searchTerms__", "application/x-suggestions+json").uri.spec;
    }

    // By convention, the currently configured default engine is at position zero in searchEngines.
    sendMessageToJava({
      type: "SearchEngines:Data",
      searchEngines: searchEngines,
      suggest: {
        engine: suggestEngine,
        template: suggestTemplate,
        enabled: Services.prefs.getBoolPref(this.PREF_SUGGEST_ENABLED),
        prompted: Services.prefs.getBoolPref(this.PREF_SUGGEST_PROMPTED)
      }
    });

    // Send a speculative connection to the default engine.
    Services.search.defaultEngine.speculativeConnect({window: window});
  },

  // Helper method to extract the engine name from a JSON. Simplifies the observe function.
  _extractEngineFromJSON: function _extractEngineFromJSON(aData) {
    let data = JSON.parse(aData);
    return Services.search.getEngineByName(data.engine);
  },

  observe: function observe(aSubject, aTopic, aData) {
    let engine;
    switch(aTopic) {
      case "SearchEngines:Add":
        this.displaySearchEnginesList(aData);
        break;
      case "SearchEngines:GetVisible":
        Services.search.init(this._handleSearchEnginesGetVisible.bind(this));
        break;
      case "SearchEngines:Remove":
        // Make sure the engine isn't hidden before removing it, to make sure it's
        // visible if the user later re-adds it (works around bug 341833)
        engine = this._extractEngineFromJSON(aData);
        engine.hidden = false;
        Services.search.removeEngine(engine);
        break;
      case "SearchEngines:RestoreDefaults":
        // Un-hides all default engines.
        Services.search.restoreDefaultEngines();
        break;
      case "SearchEngines:SetDefault":
        engine = this._extractEngineFromJSON(aData);
        // Move the new default search engine to the top of the search engine list.
        Services.search.moveEngine(engine, 0);
        Services.search.defaultEngine = engine;
        break;

      default:
        dump("Unexpected message type observed: " + aTopic);
        break;
    }
  },

  // Display context menu listing names of the search engines available to be added.
  displaySearchEnginesList: function displaySearchEnginesList(aData) {
    let data = JSON.parse(aData);
    let tab = BrowserApp.getTabForId(data.tabId);

    if (!tab)
      return;

    let browser = tab.browser;
    let engines = browser.engines;

    let p = new Prompt({
      window: browser.contentWindow
    }).setSingleChoiceItems(engines.map(function(e) {
      return { label: e.title };
    })).show((function(data) {
      if (data.button == -1)
        return;

      this.addOpenSearchEngine(engines[data.button]);
      engines.splice(data.button, 1);

      if (engines.length < 1) {
        // Broadcast message that there are no more add-able search engines.
        let newEngineMessage = {
          type: "Link:OpenSearch",
          tabID: tab.id,
          visible: false
        };

        sendMessageToJava(newEngineMessage);
      }
    }).bind(this));
  },

  addOpenSearchEngine: function addOpenSearchEngine(engine) {
    Services.search.addEngine(engine.url, Ci.nsISearchEngine.DATA_XML, engine.iconURL, false, {
      onSuccess: function() {
        // Display a toast confirming addition of new search engine.
        NativeWindow.toast.show(Strings.browser.formatStringFromName("alertSearchEngineAddedToast", [engine.title], 1), "long");
      },

      onError: function(aCode) {
        let errorMessage;
        if (aCode == 2) {
          // Engine is a duplicate.
          errorMessage = "alertSearchEngineDuplicateToast";

        } else {
          // Unknown failure. Display general error message.
          errorMessage = "alertSearchEngineErrorToast";
        }

        NativeWindow.toast.show(Strings.browser.formatStringFromName(errorMessage, [engine.title], 1), "long");
      }
    });
  },

  addEngine: function addEngine(aElement) {
    let form = aElement.form;
    let charset = aElement.ownerDocument.characterSet;
    let docURI = Services.io.newURI(aElement.ownerDocument.URL, charset, null);
    let formURL = Services.io.newURI(form.getAttribute("action"), charset, docURI).spec;
    let method = form.method.toUpperCase();
    let formData = [];

    for (let i = 0; i < form.elements.length; ++i) {
      let el = form.elements[i];
      if (!el.type)
        continue;

      // make this text field a generic search parameter
      if (aElement == el) {
        formData.push({ name: el.name, value: "{searchTerms}" });
        continue;
      }

      let type = el.type.toLowerCase();
      let escapedName = escape(el.name);
      let escapedValue = escape(el.value);

      // add other form elements as parameters
      switch (el.type) {
        case "checkbox":
        case "radio":
          if (!el.checked) break;
        case "text":
        case "hidden":
        case "textarea":
          formData.push({ name: escapedName, value: escapedValue });
          break;
        case "select-one":
          for (let option of el.options) {
            if (option.selected) {
              formData.push({ name: escapedName, value: escapedValue });
              break;
            }
          }
      }
    }

    // prompt user for name of search engine
    let promptTitle = Strings.browser.GetStringFromName("contextmenu.addSearchEngine2");
    let title = { value: (aElement.ownerDocument.title || docURI.host) };
    if (!Services.prompt.prompt(null, promptTitle, null, title, null, {}))
      return;

    // fetch the favicon for this page
    let dbFile = FileUtils.getFile("ProfD", ["browser.db"]);
    let mDBConn = Services.storage.openDatabase(dbFile);
    let stmts = [];
    stmts[0] = mDBConn.createStatement("SELECT favicon FROM history_with_favicons WHERE url = ?");
    stmts[0].bindByIndex(0, docURI.spec);
    let favicon = null;
    Services.search.init(function addEngine_cb(rv) {
      if (!Components.isSuccessCode(rv)) {
        Cu.reportError("Could not initialize search service, bailing out.");
        return;
      }
      mDBConn.executeAsync(stmts, stmts.length, {
        handleResult: function (results) {
          let bytes = results.getNextRow().getResultByName("favicon");
          if (bytes && bytes.length) {
            favicon = "data:image/x-icon;base64," + btoa(String.fromCharCode.apply(null, bytes));
          }
        },
        handleCompletion: function (reason) {
          // if there's already an engine with this name, add a number to
          // make the name unique (e.g., "Google" becomes "Google 2")
          let name = title.value;
          for (let i = 2; Services.search.getEngineByName(name); i++)
            name = title.value + " " + i;

          Services.search.addEngineWithDetails(name, favicon, null, null, method, formURL);
          let engine = Services.search.getEngineByName(name);
          engine.wrappedJSObject._queryCharset = charset;
          for (let i = 0; i < formData.length; ++i) {
            let param = formData[i];
            if (param.name && param.value)
              engine.addParam(param.name, param.value, null);
          }
        }
      });
    });
  }
};

var ActivityObserver = {
  init: function ao_init() {
    Services.obs.addObserver(this, "application-background", false);
    Services.obs.addObserver(this, "application-foreground", false);
  },

  observe: function ao_observe(aSubject, aTopic, aData) {
    let isForeground = false;
    let tab = BrowserApp.selectedTab;

    switch (aTopic) {
      case "application-background" :
        let doc = (tab ? tab.browser.contentDocument : null);
        if (doc && doc.mozFullScreen) {
          doc.mozCancelFullScreen();
        }
        isForeground = false;
        break;
      case "application-foreground" :
        isForeground = true;
        break;
    }

    if (tab && tab.getActive() != isForeground) {
      tab.setActive(isForeground);
    }
  }
};

var RemoteDebugger = {
  init: function rd_init() {
    Services.prefs.addObserver("devtools.debugger.", this, false);

    if (this._isEnabled())
      this._start();
  },

  observe: function rd_observe(aSubject, aTopic, aData) {
    if (aTopic != "nsPref:changed")
      return;

    switch (aData) {
      case "devtools.debugger.remote-enabled":
        if (this._isEnabled())
          this._start();
        else
          this._stop();
        break;

      case "devtools.debugger.remote-port":
        if (this._isEnabled())
          this._restart();
        break;
    }
  },

  uninit: function rd_uninit() {
    Services.prefs.removeObserver("devtools.debugger.", this);
    this._stop();
  },

  _getPort: function _rd_getPort() {
    return Services.prefs.getIntPref("devtools.debugger.remote-port");
  },

  _isEnabled: function rd_isEnabled() {
    return Services.prefs.getBoolPref("devtools.debugger.remote-enabled");
  },

  /**
   * Prompt the user to accept or decline the incoming connection.
   * This is passed to DebuggerService.init as a callback.
   *
   * @return true if the connection should be permitted, false otherwise
   */
  _showConnectionPrompt: function rd_showConnectionPrompt() {
    let title = Strings.browser.GetStringFromName("remoteIncomingPromptTitle");
    let msg = Strings.browser.GetStringFromName("remoteIncomingPromptMessage");
    let disable = Strings.browser.GetStringFromName("remoteIncomingPromptDisable");
    let cancel = Strings.browser.GetStringFromName("remoteIncomingPromptCancel");
    let agree = Strings.browser.GetStringFromName("remoteIncomingPromptAccept");

    // Make prompt. Note: button order is in reverse.
    let prompt = new Prompt({
      window: null,
      hint: "remotedebug",
      title: title,
      message: msg,
      buttons: [ agree, cancel, disable ],
      priority: 1
    });

    // The debugger server expects a synchronous response, so spin on result since Prompt is async.
    let result = null;

    prompt.show(function(data) {
      result = data.button;
    });

    // Spin this thread while we wait for a result.
    let thread = Services.tm.currentThread;
    while (result == null)
      thread.processNextEvent(true);

    if (result === 0)
      return true;
    if (result === 2) {
      Services.prefs.setBoolPref("devtools.debugger.remote-enabled", false);
      this._stop();
    }
    return false;
  },

  _restart: function rd_restart() {
    this._stop();
    this._start();
  },

  _start: function rd_start() {
    try {
      if (!DebuggerServer.initialized) {
        DebuggerServer.init(this._showConnectionPrompt.bind(this));
        DebuggerServer.addBrowserActors();
        DebuggerServer.registerModule("resource://gre/modules/dbg-browser-actors.js");
      }

      let port = this._getPort();
      DebuggerServer.openListener(port);
      dump("Remote debugger listening on port " + port);
    } catch(e) {
      dump("Remote debugger didn't start: " + e);
    }
  },

  _stop: function rd_start() {
    DebuggerServer.closeAllListeners();
    dump("Remote debugger stopped");
  }
};

var Telemetry = {
  addData: function addData(aHistogramId, aValue) {
    let histogram = Services.telemetry.getHistogramById(aHistogramId);
    histogram.add(aValue);
  },
};

let Reader = {
  // Version of the cache database schema
  DB_VERSION: 1,

  DEBUG: 0,

  READER_ADD_SUCCESS: 0,
  READER_ADD_FAILED: 1,
  READER_ADD_DUPLICATE: 2,

  // Don't try to parse the page if it has too many elements (for memory and
  // performance reasons)
  MAX_ELEMS_TO_PARSE: 3000,

  isEnabledForParseOnLoad: false,

  init: function Reader_init() {
    this.log("Init()");
    this._requests = {};

    this.isEnabledForParseOnLoad = this.getStateForParseOnLoad();

    Services.obs.addObserver(this, "Reader:Add", false);
    Services.obs.addObserver(this, "Reader:Remove", false);

    Services.prefs.addObserver("reader.parse-on-load.", this, false);
  },

  pageAction: {
    readerModeCallback: function(){
      sendMessageToJava({
        type: "Reader:Click",
      });
    },

    readerModeActiveCallback: function(){
      sendMessageToJava({
        type: "Reader:LongClick",
      });

      UITelemetry.addEvent("save.1", "pageaction", null, "reader");
    },
  },

  updatePageAction: function(tab) {
    if (this.pageAction.id) {
      NativeWindow.pageactions.remove(this.pageAction.id);
      delete this.pageAction.id;
    }

    if (tab.readerActive) {
      this.pageAction.id = NativeWindow.pageactions.add({
        title: Strings.browser.GetStringFromName("readerMode.exit"),
        icon: "drawable://reader_active",
        clickCallback: this.pageAction.readerModeCallback,
        important: true
      });

      // Only start a reader session if the viewer is in the foreground. We do
      // not track background reader viewers.
      UITelemetry.startSession("reader.1", null);
      return;
    }

    // Only stop a reader session if the foreground viewer is not visible.
    UITelemetry.stopSession("reader.1", "", null);

    if (tab.readerEnabled) {
      this.pageAction.id = NativeWindow.pageactions.add({
        title: Strings.browser.GetStringFromName("readerMode.enter"),
        icon: "drawable://reader",
        clickCallback:this.pageAction.readerModeCallback,
        longClickCallback: this.pageAction.readerModeActiveCallback,
        important: true
      });
    }
  },

  observe: function(aMessage, aTopic, aData) {
    switch(aTopic) {
      case "Reader:Add": {
        let args = JSON.parse(aData);
        if ('fromAboutReader' in args) {
          // Ignore adds initiated from aboutReader menu banner
          break;
        }

        let tabID = null;
        let url, urlWithoutRef;

        if ('tabID' in args) {
          tabID = args.tabID;

          let tab = BrowserApp.getTabForId(tabID);
          let currentURI = tab.browser.currentURI;

          url = currentURI.spec;
          urlWithoutRef = currentURI.specIgnoringRef;
        } else if ('url' in args) {
          let uri = Services.io.newURI(args.url, null, null);
          url = uri.spec;
          urlWithoutRef = uri.specIgnoringRef;
        } else {
          throw new Error("Reader:Add requires a tabID or an URL as argument");
        }

        let sendResult = function(result, article) {
          article = article || {};
          this.log("Reader:Add success=" + result + ", url=" + url + ", title=" + article.title + ", excerpt=" + article.excerpt);

          sendMessageToJava({
            type: "Reader:Added",
            result: result,
            title: article.title,
            url: url,
            length: article.length,
            excerpt: article.excerpt
          });
        }.bind(this);

        let handleArticle = function(article) {
          if (!article) {
            sendResult(this.READER_ADD_FAILED, null);
            return;
          }

          this.storeArticleInCache(article, function(success) {
            let result = (success ? this.READER_ADD_SUCCESS : this.READER_ADD_FAILED);
            sendResult(result, article);
          }.bind(this));
        }.bind(this);

        this.getArticleFromCache(urlWithoutRef, function (article) {
          // If the article is already in reading list, bail
          if (article) {
            sendResult(this.READER_ADD_DUPLICATE, null);
            return;
          }

          if (tabID != null) {
            this.getArticleForTab(tabID, urlWithoutRef, handleArticle);
          } else {
            this.parseDocumentFromURL(urlWithoutRef, handleArticle);
          }
        }.bind(this));
        break;
      }

      case "Reader:Remove": {
        let args = JSON.parse(aData);

        if (!("url" in args)) {
          throw new Error("Reader:Remove requires URL as an argument");
        }

        this.removeArticleFromCache(args.url, function(success) {
          this.log("Reader:Remove success=" + success + ", url=" + args.url);
          if (success && args.notify) {
            sendMessageToJava({
              type: "Reader:Removed",
              url: args.url
            });
          }
        }.bind(this));
        break;
      }

      case "nsPref:changed": {
        if (aData.startsWith("reader.parse-on-load.")) {
          this.isEnabledForParseOnLoad = this.getStateForParseOnLoad();
        }
        break;
      }
    }
  },

  getStateForParseOnLoad: function Reader_getStateForParseOnLoad() {
    let isEnabled = Services.prefs.getBoolPref("reader.parse-on-load.enabled");
    let isForceEnabled = Services.prefs.getBoolPref("reader.parse-on-load.force-enabled");
    // For low-memory devices, don't allow reader mode since it takes up a lot of memory.
    // See https://bugzilla.mozilla.org/show_bug.cgi?id=792603 for details.
    return isForceEnabled || (isEnabled && !BrowserApp.isOnLowMemoryPlatform);
  },

  parseDocumentFromURL: function Reader_parseDocumentFromURL(url, callback) {
    // If there's an on-going request for the same URL, simply append one
    // more callback to it to be called when the request is done.
    if (url in this._requests) {
      let request = this._requests[url];
      request.callbacks.push(callback);
      return;
    }

    let request = { url: url, callbacks: [callback] };
    this._requests[url] = request;

    try {
      this.log("parseDocumentFromURL: " + url);

      // First, try to find a cached parsed article in the DB
      this.getArticleFromCache(url, function(article) {
        if (article) {
          this.log("Page found in cache, return article immediately");
          this._runCallbacksAndFinish(request, article);
          return;
        }

        if (!this._requests) {
          this.log("Reader has been destroyed, abort");
          return;
        }

        // Article hasn't been found in the cache DB, we need to
        // download the page and parse the article out of it.
        this._downloadAndParseDocument(url, request);
      }.bind(this));
    } catch (e) {
      this.log("Error parsing document from URL: " + e);
      this._runCallbacksAndFinish(request, null);
    }
  },

  getArticleForTab: function Reader_getArticleForTab(tabId, url, callback) {
    let tab = BrowserApp.getTabForId(tabId);
    if (tab) {
      let article = tab.savedArticle;
      if (article && article.url == url) {
        this.log("Saved article found in tab");
        callback(article);
        return;
      }
    }

    this.parseDocumentFromURL(url, callback);
  },

  parseDocumentFromTab: function(tabId, callback) {
    try {
      this.log("parseDocumentFromTab: " + tabId);

      let tab = BrowserApp.getTabForId(tabId);
      let url = tab.browser.contentWindow.location.href;
      let uri = Services.io.newURI(url, null, null);

      if (!this._shouldCheckUri(uri)) {
        callback(null);
        return;
      }

      // First, try to find a cached parsed article in the DB
      this.getArticleFromCache(url, function(article) {
        if (article) {
          this.log("Page found in cache, return article immediately");
          callback(article);
          return;
        }

        let doc = tab.browser.contentWindow.document;
        this._readerParse(uri, doc, function (article) {
```

```
        if (!article) {
          this.log("Failed to parse page");
          callback(null);
          return;
        }

        callback(article);
      }.bind(this));
    }.bind(this));
  } catch (e) {
    this.log("Error parsing document from tab: " + e);
    callback(null);
  }
},

getArticleFromCache: function Reader_getArticleFromCache(url, callback) {
  this._getCacheDB(function(cacheDB) {
    if (!cacheDB) {
      callback(false);
      return;
    }

    let transaction = cacheDB.transaction(cacheDB.objectStoreNames);
    let articles = transaction.objectStore(cacheDB.objectStoreNames[0]);

    let request = articles.get(url);

    request.onerror = function(event) {
      this.log("Error getting article from the cache DB: " + url);
      callback(null);
    }.bind(this);

    request.onsuccess = function(event) {
      this.log("Got article from the cache DB: " + event.target.result);
      callback(event.target.result);
    }.bind(this);
  }.bind(this));
},

storeArticleInCache: function Reader_storeArticleInCache(article, callback) {
  this._getCacheDB(function(cacheDB) {
    if (!cacheDB) {
      callback(false);
      return;
    }

    let transaction = cacheDB.transaction(cacheDB.objectStoreNames, "readwrite");
    let articles = transaction.objectStore(cacheDB.objectStoreNames[0]);

    let request = articles.add(article);

    request.onerror = function(event) {
      this.log("Error storing article in the cache DB: " + article.url);
      callback(false);
    }.bind(this);

    request.onsuccess = function(event) {
      this.log("Stored article in the cache DB: " + article.url);
      callback(true);
    }.bind(this);
  }.bind(this));
},

removeArticleFromCache: function Reader_removeArticleFromCache(url, callback) {
  this._getCacheDB(function(cacheDB) {
    if (!cacheDB) {
      callback(false);
      return;
    }

    let transaction = cacheDB.transaction(cacheDB.objectStoreNames, "readwrite");
    let articles = transaction.objectStore(cacheDB.objectStoreNames[0]);

    let request = articles.delete(url);

    request.onerror = function(event) {
      this.log("Error removing article from the cache DB: " + url);
      callback(false);
    }.bind(this);

    request.onsuccess = function(event) {
      this.log("Removed article from the cache DB: " + url);
      callback(true);
    }.bind(this);
  }.bind(this));
},

uninit: function Reader_uninit() {
  Services.prefs.removeObserver("reader.parse-on-load.", this);

  Services.obs.removeObserver(this, "Reader:Add");
  Services.obs.removeObserver(this, "Reader:Remove");

  let requests = this._requests;
  for (let url in requests) {
    let request = requests[url];
    if (request.browser) {
      let browser = request.browser;
      browser.parentNode.removeChild(browser);
    }
  }
  delete this._requests;

  if (this._cacheDB) {
    this._cacheDB.close();
    delete this._cacheDB;
  }
},

log: function(msg) {
  if (this.DEBUG)
    dump("Reader: " + msg);
},

_shouldCheckUri: function Reader_shouldCheckUri(uri) {
  if ((uri.prePath + "/") === uri.spec) {
    this.log("Not parsing home page: " + uri.spec);
    return false;
  }

  if (!(uri.schemeIs("http") || uri.schemeIs("https") || uri.schemeIs("file"))) {
    this.log("Not parsing URI scheme: " + uri.scheme);
    return false;
  }

  return true;
},

_readerParse: function Reader_readerParse(uri, doc, callback) {
  let numTags = doc.getElementsByTagName("*").length;
  if (numTags > this.MAX_ELEMS_TO_PARSE) {
    this.log("Aborting parse for " + uri.spec + "; " + numTags + " elements found");
    callback(null);
    return;
  }

  let worker = new ChromeWorker("readerWorker.js");
  worker.onmessage = function (evt) {
    let article = evt.data;

    // Append URL to the article data. specIgnoringRef will ignore any hash
    // in the URL.
    if (article) {
      article.url = uri.specIgnoringRef;
      let flags = Ci.nsIDocumentEncoder.OutputSelectionOnly | Ci.nsIDocumentEncoder.OutputAbsoluteLinks;
      article.title = Cc["@mozilla.org/parserutils;1"].getService(Ci.nsIParserUtils)
                                                      .convertToPlainText(article.title, flags, 0);
    }

    callback(article);
  };

  try {
    worker.postMessage({
      uri: {
        spec: uri.spec,
        host: uri.host,
        prePath: uri.prePath,
        scheme: uri.scheme,
        pathBase: Services.io.newURI(".", null, uri).spec
      },
      doc: new XMLSerializer().serializeToString(doc)
    });
  } catch (e) {
    dump("Reader: could not build Readability arguments: " + e);
    callback(null);
  }
},

_runCallbacksAndFinish: function Reader_runCallbacksAndFinish(request, result) {
  delete this._requests[request.url];

  request.callbacks.forEach(function(callback) {
    callback(result);
  });
},
  _downloadDocument: function Reader_downloadDocument(url, callback) {
    // We want to parse those arbitrary pages safely, outside the privileged
    // context of chrome. We create a hidden browser element to fetch the
    // loaded page's document object then discard the browser element.

    let browser = document.createElement("browser");
    browser.setAttribute("type", "content");
    browser.setAttribute("collapsed", "true");
    browser.setAttribute("disablehistory", "true");

    document.documentElement.appendChild(browser);
    browser.stop();

    browser.webNavigation.allowAuth = false;
    browser.webNavigation.allowImages = false;
    browser.webNavigation.allowJavascript = false;
    browser.webNavigation.allowMetaRedirects = true;
    browser.webNavigation.allowPlugins = false;

    browser.addEventListener("DOMContentLoaded", function (event) {
      let doc = event.originalTarget;

      // ignore on frames and other documents
      if (doc != browser.contentDocument)
        return;

      this.log("Done loading: " + doc);
      if (doc.location.href == "about:blank") {
        callback(null);

        // Request has finished with error, remove browser element
        browser.parentNode.removeChild(browser);
        return;
      }

      callback(doc);
    }.bind(this));

    browser.loadURIWithFlags(url, Ci.nsIWebNavigation.LOAD_FLAGS_NONE,
                             null, null, null);

    return browser;
  },

  _downloadAndParseDocument: function Reader_downloadAndParseDocument(url, request) {
    try {
      this.log("Needs to fetch page, creating request: " + url);

      request.browser = this._downloadDocument(url, function(doc) {
        this.log("Finished loading page: " + doc);

        if (!doc) {
          this.log("Error loading page");
          this._runCallbacksAndFinish(request, null);
          return;
        }

        this.log("Parsing response with Readability");

        let uri = Services.io.newURI(url, null, null);
        this._readerParse(uri, doc, function (article) {
          // Delete reference to the browser element as we've finished parsing.
          let browser = request.browser;
          if (browser) {
            browser.parentNode.removeChild(browser);
            delete request.browser;
          }

          if (!article) {
            this.log("Failed to parse page");
            this._runCallbacksAndFinish(request, null);
            return;
          }

          this.log("Parsing has been successful");

          this._runCallbacksAndFinish(request, article);
        }.bind(this));
      }.bind(this));
    } catch (e) {
      this.log("Error downloading and parsing document: " + e);
      this._runCallbacksAndFinish(request, null);
    }
  },

  _getCacheDB: function Reader_getCacheDB(callback) {
    if (this._cacheDB) {
      callback(this._cacheDB);
      return;
    }

    let request = window.indexedDB.open("about:reader", this.DB_VERSION);

    request.onerror = function(event) {
      this.log("Error connecting to the cache DB");
      this._cacheDB = null;
      callback(null);
    }.bind(this);

    request.onsuccess = function(event) {
      this.log("Successfully connected to the cache DB");
      this._cacheDB = event.target.result;
      callback(this._cacheDB);
    }.bind(this);

    request.onupgradeneeded = function(event) {
      this.log("Database schema upgrade from " +
           event.oldVersion + " to " + event.newVersion);

      let cacheDB = event.target.result;

      // Create the articles object store
      this.log("Creating articles object store");
      cacheDB.createObjectStore("articles", { keyPath: "url" });

      this.log("Database upgrade done: " + this.DB_VERSION);
    }.bind(this);
  }
};

var ExternalApps = {
  _contextMenuId: null,

  // extend _getLink to pickup html5 media links.
  _getMediaLink: function(aElement) {
    let uri = NativeWindow.contextmenus._getLink(aElement);
    if (uri == null && aElement.nodeType == Ci.nsIDOMNode.ELEMENT_NODE && (aElement instanceof Ci.nsIDOMHTMLMediaElement)) {
      try {
        let mediaSrc = aElement.currentSrc || aElement.src;
        uri = ContentAreaUtils.makeURI(mediaSrc, null, null);
      } catch (e) {}
    }
    return uri;
  },

  init: function helper_init() {
    this._contextMenuId = NativeWindow.contextmenus.add(function(aElement) {
      let uri = null;
      var node = aElement;
      while (node && !uri) {
        uri = ExternalApps._getMediaLink(node);
        node = node.parentNode;
      }
      let apps = [];
      if (uri)
        apps = HelperApps.getAppsForUri(uri);

      return apps.length == 1 ? Strings.browser.formatStringFromName("helperapps.openWithApp2", [apps[0].name], 1) :
                                Strings.browser.GetStringFromName("helperapps.openWithList2");
    }, this.filter, this.openExternal);
  },

  uninit: function helper_uninit() {
    if (this._contextMenuId !== null) {
      NativeWindow.contextmenus.remove(this._contextMenuId);
    }
    this._contextMenuId = null;
  },

  filter: {
    matches: function(aElement) {
      let uri = ExternalApps._getMediaLink(aElement);
      let apps = [];
      if (uri) {
        apps = HelperApps.getAppsForUri(uri);
      }
      return apps.length > 0;
    }
  },

  openExternal: function(aElement) {
    let uri = ExternalApps._getMediaLink(aElement);
    HelperApps.launchUri(uri);
  },
  shouldCheckUri: function(uri) {
    if (!(uri.schemeIs("http") || uri.schemeIs("https") || uri.schemeIs("file"))) {
      return false;
    }

    return true;
  },

  updatePageAction: function updatePageAction(uri) {
    HelperApps.getAppsForUri(uri, { filterHttp: true }, (apps) => {
      this.clearPageAction();
      if (apps.length > 0)
        this._setUriForPageAction(uri, apps);
    });
  },

  updatePageActionUri: function updatePageActionUri(uri) {
    this._pageActionUri = uri;
  },

  _setUriForPageAction: function setUriForPageAction(uri, apps) {
    this.updatePageActionUri(uri);

    // If the pageaction is already added, simply update the URI to be launched when 'onclick' is triggered.
    if (this._pageActionId != undefined)
      return;

    this._pageActionId = NativeWindow.pageactions.add({
      title: Strings.browser.GetStringFromName("openInApp.pageAction"),
      icon: "drawable://icon_openinapp",

      clickCallback: () => {
        UITelemetry.addEvent("launch.1", "pageaction", null, "helper");

        if (apps.length > 1) {
          // Use the HelperApps prompt here to filter out any Http handlers
          HelperApps.prompt(apps, {
            title: Strings.browser.GetStringFromName("openInApp.pageAction"),
            buttons: [
              Strings.browser.GetStringFromName("openInApp.ok"),
              Strings.browser.GetStringFromName("openInApp.cancel")
            ]
          }, (result) => {
            if (result.button != 0) {
              return;
            }
            apps[result.icongrid0].launch(this._pageActionUri);
          });
        } else {
          apps[0].launch(this._pageActionUri);
        }
      }
    });
  },

  clearPageAction: function clearPageAction() {
    if(!this._pageActionId)
      return;

    NativeWindow.pageactions.remove(this._pageActionId);
    delete this._pageActionId;
  },
};

var Distribution = {
  // File used to store campaign data
  _file: null,

  init: function dc_init() {
    Services.obs.addObserver(this, "Distribution:Set", false);
    Services.obs.addObserver(this, "prefservice:after-app-defaults", false);
    Services.obs.addObserver(this, "Campaign:Set", false);

    // Look for file outside the APK:
    // /data/data/org.mozilla.xxx/distribution.json
    this._file = Services.dirsvc.get("XCurProcD", Ci.nsIFile);
    this._file.append("distribution.json");
    this.readJSON(this._file, this.update);
  },

  uninit: function dc_uninit() {
    Services.obs.removeObserver(this, "Distribution:Set");
    Services.obs.removeObserver(this, "prefservice:after-app-defaults");
    Services.obs.removeObserver(this, "Campaign:Set");
  },

  observe: function dc_observe(aSubject, aTopic, aData) {
    switch (aTopic) {
      case "Distribution:Set":
        // Reload the default prefs so we can observe "prefservice:after-app-defaults"
        Services.prefs.QueryInterface(Ci.nsIObserver).observe(null, "reload-default-prefs", null);
        break;

      case "prefservice:after-app-defaults":
        this.getPrefs();
        break;

      case "Campaign:Set": {
        // Update the prefs for this session
        try {
          this.update(JSON.parse(aData));
        } catch (ex) {
          Cu.reportError("Distribution: Could not parse JSON: " + ex);
          return;
        }

        // Asynchronously copy the data to the file.
        let array = new TextEncoder().encode(aData);
        OS.File.writeAtomic(this._file.path, array, { tmpPath: this._file.path + ".tmp" });
        break;
      }
    }
  },

  update: function dc_update(aData) {
    // Force the distribution preferences on the default branch
    let defaults = Services.prefs.getDefaultBranch(null);
    defaults.setCharPref("distribution.id", aData.id);
    defaults.setCharPref("distribution.version", aData.version);
  },

  getPrefs: function dc_getPrefs() {
    // Get the distribution directory, and bail if it doesn't exist.
    let file = FileUtils.getDir("XREAppDist", [], false);
    if (!file.exists())
      return;

    file.append("preferences.json");
    this.readJSON(file, this.applyPrefs);
  },

  applyPrefs: function dc_applyPrefs(aData) {
    // Check for required Global preferences
    let global = aData["Global"];
    if (!(global && global["id"] && global["version"] && global["about"])) {
      Cu.reportError("Distribution: missing or incomplete Global preferences");
      return;
    }

    // Force the distribution preferences on the default branch
    let defaults = Services.prefs.getDefaultBranch(null);
    defaults.setCharPref("distribution.id", global["id"]);
    defaults.setCharPref("distribution.version", global["version"]);

    let locale = Services.prefs.getCharPref("general.useragent.locale");
    let aboutString = Cc["@mozilla.org/supports-string;1"].createInstance(Ci.nsISupportsString);
    aboutString.data = global["about." + locale] || global["about"];
    defaults.setComplexValue("distribution.about", Ci.nsISupportsString, aboutString);

    let prefs = aData["Preferences"];
    for (let key in prefs) {
      try {
        let value = prefs[key];
        switch (typeof value) {
          case "boolean":
            defaults.setBoolPref(key, value);
            break;
          case "number":
            defaults.setIntPref(key, value);
            break;
          case "string":
          case "undefined":
            defaults.setCharPref(key, value);
            break;
        }
      } catch (e) { /* ignore bad prefs and move on */ }
    }

    // Apply a lightweight theme if necessary
    if (prefs["lightweightThemes.isThemeSelected"])
      Services.obs.notifyObservers(null, "lightweight-theme-apply", "");

    let localizedString = Cc["@mozilla.org/pref-localizedstring;1"].createInstance(Ci.nsIPrefLocalizedString);
    let localizeablePrefs = aData["LocalizablePreferences"];
    for (let key in localizeablePrefs) {
      try {
        let value = localizeablePrefs[key];
        value = value.replace("%LOCALE%", locale, "g");
        localizedString.data = "data:text/plain," + key + "=" + value;
        defaults.setComplexValue(key, Ci.nsIPrefLocalizedString, localizedString);
      } catch (e) { /* ignore bad prefs and move on */ }
    }

    let localizeablePrefsOverrides = aData["LocalizablePreferences." + locale];
    for (let key in localizeablePrefsOverrides) {
      try {
        let value = localizeablePrefsOverrides[key];
        localizedString.data = "data:text/plain," + key + "=" + value;
        defaults.setComplexValue(key, Ci.nsIPrefLocalizedString, localizedString);
      } catch (e) { /* ignore bad prefs and move on */ }
    }

    sendMessageToJava({ type: "Distribution:Set:OK" });
  },

  // aFile is an nsIFile
  // aCallback takes the parsed JSON object as a parameter
  readJSON: function dc_readJSON(aFile, aCallback) {
    Task.spawn(function() {
      let bytes = yield OS.File.read(aFile.path);
      let raw = new TextDecoder().decode(bytes) || "";

      try {
        aCallback(JSON.parse(raw));
      } catch (e) {
        Cu.reportError("Distribution: Could not parse JSON: " + e);
      }
    }).then(null, function onError(reason) {
      if (!(reason instanceof OS.File.Error && reason.becauseNoSuchFile)) {
        Cu.reportError("Distribution: Could not read from " + aFile.leafName + " file");
      }
    });
  }
};

var Tabs = {
  _enableTabExpiration: false,
  _domains: new Set(),

  init: function() {
    // On low-memory platforms, always allow tab expiration. On high-mem
    // platforms, allow it to be turned on once we hit a low-mem situation.
    if (BrowserApp.isOnLowMemoryPlatform) {
      this._enableTabExpiration = true;
    } else {
      Services.obs.addObserver(this, "memory-pressure", false);
    }

    Services.obs.addObserver(this, "Session:Prefetch", false);

    BrowserApp.deck.addEventListener("pageshow", this, false);
    BrowserApp.deck.addEventListener("TabOpen", this, false);
  },

  uninit: function() {
    if (!this._enableTabExpiration) {
      // If _enableTabExpiration is true then we won't have this
      // observer registered any more.
      Services.obs.removeObserver(this, "memory-pressure");
    }

    Services.obs.removeObserver(this, "Session:Prefetch");

    BrowserApp.deck.removeEventListener("pageshow", this);
    BrowserApp.deck.removeEventListener("TabOpen", this);
  },

  observe: function(aSubject, aTopic, aData) {
    switch (aTopic) {
      case "memory-pressure":
        if (aData != "heap-minimize") {
          // We received a low-memory related notification. This will enable
          // expirations.
          this._enableTabExpiration = true;
          Services.obs.removeObserver(this, "memory-pressure");
        } else {
          // Use "heap-minimize" as a trigger to expire the most stale tab.
          this.expireLruTab();
        }
        break;
      case "Session:Prefetch":
        if (aData) {
          let uri = Services.io.newURI(aData, null, null);
          try {
            if (uri && !this._domains.has(uri.host)) {
              Services.io.QueryInterface(Ci.nsISpeculativeConnect).speculativeConnect(uri, null);
              this._domains.add(uri.host);
            }
          } catch (e) {}
        }
        break;
    }
  },

  handleEvent: function(aEvent) {
    switch (aEvent.type) {
      case "pageshow":
        // Clear the domain cache whenever a page get loaded into any browser.
        this._domains.clear();
        break;
      case "TabOpen":
        // Use opening a new tab as a trigger to expire the most stale tab.
        this.expireLruTab();
        break;
    }
  },

  // Manage the most-recently-used list of tabs. Each tab has a timestamp
  // associated with it that indicates when it was last touched.
  expireLruTab: function() {
    if (!this._enableTabExpiration) {
      return false;
    }
    let expireTimeMs = Services.prefs.getIntPref("browser.tabs.expireTime") * 1000;
    if (expireTimeMs < 0) {
      // This behaviour is disabled.
      return false;
    }
    let tabs = BrowserApp.tabs;
    let selected = BrowserApp.selectedTab;
    let lruTab = null;
    // Find the least recently used non-zombie tab.
    for (let i = 0; i < tabs.length; i++) {
      if (tabs[i] == selected || tabs[i].browser.__SS_restore) {
        // This tab is selected or already a zombie, skip it.
        continue;
      }
      if (lruTab == null || tabs[i].lastTouchedAt < lruTab.lastTouchedAt) {
        lruTab = tabs[i];
      }
    }
    // If the tab was last touched more than browser.tabs.expireTime seconds ago,
    // zombify it.
    if (lruTab) {
      let tabAgeMs = Date.now() - lruTab.lastTouchedAt;
      if (tabAgeMs > expireTimeMs) {
        MemoryObserver.zombify(lruTab);
        Telemetry.addData("FENNEC_TAB_EXPIRED", tabAgeMs / 1000);
        return true;
      }
    }
    return false;
  },

  // For debugging
  dump: function(aPrefix) {
    let tabs = BrowserApp.tabs;
    for (let i = 0; i < tabs.length; i++) {
      dump(aPrefix + " | " + "Tab [" + tabs[i].browser.contentWindow.location.href + "]: lastTouchedAt:" + tabs[i].lastTouchedAt + ", zombie:" + tabs[i].browser.__SS_restore);
    }
  },
};

function ContextMenuItem(args) {
  this.id = uuidgen.generateUUID().toString();
  this.args = args;
}

ContextMenuItem.prototype = {
  get order() {
    return this.args.order || 0;
  },

  matches: function(elt, x, y) {
    return this.args.selector.matches(elt, x, y);
  },

  callback: function(elt) {
    this.args.callback(elt);
  },

  addVal: function(name, elt, defaultValue) {
    if (!(name in this.args))
      return defaultValue;

    if (typeof this.args[name] == "function")
      return this.args[name](elt);

    return this.args[name];
  },

  getValue: function(elt) {
    return {
      id: this.id,
      label: this.addVal("label", elt),
      showAsActions: this.addVal("showAsActions", elt),
      icon: this.addVal("icon", elt),
      isGroup: this.addVal("isGroup", elt, false),
      inGroup: this.addVal("inGroup", elt, false),
      disabled: this.addVal("disabled", elt, false),
      selected: this.addVal("selected", elt, false),
      isParent: this.addVal("isParent", elt, false),
    };
  }
}

function HTMLContextMenuItem(elt, target) {
  ContextMenuItem.call(this, { });

  this.menuElementRef = Cu.getWeakReference(elt);
  this.targetElementRef = Cu.getWeakReference(target);
}

HTMLContextMenuItem.prototype = Object.create(ContextMenuItem.prototype, {
  order: {
    value: NativeWindow.contextmenus.DEFAULT_HTML5_ORDER
  },

  matches: {
    value: function(target) {
      let t = this.targetElementRef.get();
      return t === target;
    },
  },

  callback: {
    value: function(target) {
      let elt = this.menuElementRef.get();
      if (!elt) {
        return;
      }

      // If this is a menu item, show a new context menu with the submenu in it
      if (elt instanceof Ci.nsIDOMHTMLMenuElement) {
        try {
          NativeWindow.contextmenus.menus = {};

          let elt = this.menuElementRef.get();
          let target = this.targetElementRef.get();
          if (!elt) {
            return;
          }

          var items = NativeWindow.contextmenus._getHTMLContextMenuItemsForMenu(elt, target);
          // This menu will always only have one context, but we still make sure its the "right" one.
          var context = NativeWindow.contextmenus._getContextType(target);
          if (items.length > 0) {
            NativeWindow.contextmenus._addMenuItems(items, context);
          }

        } catch(ex) {
          Cu.reportError(ex);
        }
      } else {
        // otherwise just click the menu item
        elt.click();
      }
    },
  },

  getValue: {
    value: function(target) {
      let elt = this.menuElementRef.get();
      if (!elt) {
        return null;
      }

      if (elt.hasAttribute("hidden")) {
        return null;
      }

      return {
        id: this.id,
        icon: elt.icon,
        label: elt.label,
        disabled: elt.disabled,
        menu: elt instanceof Ci.nsIDOMHTMLMenuElement
      };
    }
  },
});
```