```
/* 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/. */

import { XPCOMUtils } from "resource://gre/modules/XPCOMUtils.sys.mjs";

const INTERNAL_FIELDS = new Set(["_level", "_message", "_time", "_namespace"]);

/*
 * Dump a message everywhere we can if we have a failure.
 */
function dumpError(text) {
  dump(text + "¥n");
  // TODO: Bug 1801091 - Figure out how to replace this.
  // eslint-disable-next-line mozilla/no-cu-reportError
  Cu.reportError(text);
}

export var Log = {
  Level: {
    Fatal: 70,
    Error: 60,
    Warn: 50,
    Info: 40,
    Config: 30,
    Debug: 20,
    Trace: 10,
    All: -1, // We don't want All to be falsy.
    Desc: {
      70: "FATAL",
      60: "ERROR",
      50: "WARN",
      40: "INFO",
      30: "CONFIG",
      20: "DEBUG",
      10: "TRACE",
      "-1": "ALL",
    },
    Numbers: {
      FATAL: 70,
      ERROR: 60,
      WARN: 50,
      INFO: 40,
      CONFIG: 30,
      DEBUG: 20,
      TRACE: 10,
      ALL: -1,
    },
  },

  get repository() {
    delete Log.repository;
    Log.repository = new LoggerRepository();
    return Log.repository;
  },
  set repository(value) {
    delete Log.repository;
    Log.repository = value;
  },

  _formatError(e) {
    let result = String(e);
    if (e.fileName) {
      let loc = [e.fileName];
      if (e.lineNumber) {
        loc.push(e.lineNumber);
      }
      if (e.columnNumber) {
        loc.push(e.columnNumber);
      }
      result += `(${loc.join(":")})`;
    }
    return `${result} ${Log.stackTrace(e)}`;
  },
```

```
  // This is for back compatibility with services/common/utils.js; we duplicate
  // some of the logic in ParameterFormatter
  exceptionStr(e) {
    if (!e) {
      return String(e);
    }
    if (e instanceof Ci.nsIException) {
      return `${e} ${Log.stackTrace(e)}`;
    } else if (isError(e)) {
      return Log._formatError(e);
    }
    // else
    let message = e.message || e;
    return `${message} ${Log.stackTrace(e)}`;
  },

  stackTrace(e) {
    if (!e) {
      return Components.stack.caller.formattedStack.trim();
    }
    // Wrapped nsIException
    if (e.location) {
      let frame = e.location;
      let output = [];
      while (frame) {
        // Works on frames or exceptions, munges file:// URIs to shorten the paths
        // FIXME: filename munging is sort of hackish.
        let str = "<file:unknown>";

        let file = frame.filename || frame.fileName;
        if (file) {
          str = file.replace(/^(?:chrome|file):.*?([^¥/¥.]+(¥.¥w+)+)$/, "$1");
        }

        if (frame.lineNumber) {
          str += ":" + frame.lineNumber;
        }

        if (frame.name) {
          str = frame.name + "()@" + str;
        }

        if (str) {
          output.push(str);
        }
        frame = frame.caller;
      }
      return `Stack trace: ${output.join("¥n")}`;
    }
    // Standard JS exception
    if (e.stack) {
      let stack = e.stack;
      return (
        "JS Stack trace: " +
        stack.trim().replace(/@[^@]*?([^¥/¥.]+(¥.¥w+)+:)/g, "@$1")
      );
    }

    if (e instanceof Ci.nsIStackFrame) {
      return e.formattedStack.trim();
    }
    return "No traceback available";
  },
};

/*
 * LogMessage
 * Encapsulates a single log event's data
 */
class LogMessage {
  constructor(loggerName, level, message, params) {
    this.loggerName = loggerName;
    this.level = level;
    /*
     * Special case to handle "log./level/(object)", for example logging a caught exception
```

```
     * without providing text or params like: catch(e) { logger.warn(e) }
     * Treating this as an empty text with the object in the 'params' field causes the
     * object to be formatted properly by BasicFormatter.
     */
    if (
      !params &&
      message &&
      typeof message == "object" &&
      typeof message.valueOf() != "string"
    ) {
      this.message = null;
      this.params = message;
    } else {
      // If the message text is empty, or a string, or a String object, normal handling
      this.message = message;
      this.params = params;
    }

    // The _structured field will correspond to whether this message is to
    // be interpreted as a structured message.
    this._structured = this.params && this.params.action;
    this.time = Date.now();
  }

  get levelDesc() {
    if (this.level in Log.Level.Desc) {
      return Log.Level.Desc[this.level];
    }
    return "UNKNOWN";
  }

  toString() {
    let msg = `${this.time} ${this.level} ${this.message}`;
    if (this.params) {
      msg += ` ${JSON.stringify(this.params)}`;
    }
    return `LogMessage [${msg}]`;
  }
}

/*
 * Logger
 * Hierarchical version.  Logs to all appenders, assigned or inherited
 */

class Logger {
  constructor(name, repository) {
    if (!repository) {
      repository = Log.repository;
    }
    this._name = name;
    this.children = [];
    this.ownAppenders = [];
    this.appenders = [];
    this._repository = repository;

    this._levelPrefName = null;
    this._levelPrefValue = null;
    this._level = null;
    this._parent = null;
  }

  get name() {
    return this._name;
  }

  get level() {
    if (this._levelPrefName) {
      // We've been asked to use a preference to configure the logs. If the
      // pref has a value we use it, otherwise we continue to use the parent.
      const lpv = this._levelPrefValue;
      if (lpv) {
        const levelValue = Log.Level[lpv];
        if (levelValue) {
          // stash it in _level just in case a future value of the pref is
```

```
        // invalid, in which case we end up continuing to use this value.
        this._level = levelValue;
        return levelValue;
      }
    } else {
      // in case the pref has transitioned from a value to no value, we reset
      // this._level and fall through to using the parent.
      this._level = null;
    }
  }
  if (this._level != null) {
    return this._level;
  }
  if (this.parent) {
    return this.parent.level;
  }
  dumpError("Log warning: root logger configuration error: no level defined");
  return Log.Level.All;
}
set level(level) {
  if (this._levelPrefName) {
    // I guess we could honor this by nuking this._levelPrefValue, but it
    // almost certainly implies confusion, so we'll warn and ignore.
    dumpError(
      `Log warning: The log '${this.name}' is configured to use ` +
        `the preference '${this._levelPrefName}' - you must adjust ` +
        `the level by setting this preference, not by using the ` +
        `level setter`
    );
    return;
  }
  this._level = level;
}

get parent() {
  return this._parent;
}
set parent(parent) {
  if (this._parent == parent) {
    return;
  }
  // Remove ourselves from parent's children
  if (this._parent) {
    let index = this._parent.children.indexOf(this);
    if (index != -1) {
      this._parent.children.splice(index, 1);
    }
  }
  this._parent = parent;
  parent.children.push(this);
  this.updateAppenders();
}

manageLevelFromPref(prefName) {
  if (prefName == this._levelPrefName) {
    // We've already configured this log with an observer for that pref.
    return;
  }
  if (this._levelPrefName) {
    dumpError(
      `The log '${this.name}' is already configured with the ` +
        `preference '${this._levelPrefName}' - ignoring request to ` +
        `also use the preference '${prefName}'`
    );
    return;
  }
  this._levelPrefName = prefName;
  XPCOMUtils.defineLazyPreferenceGetter(this, "_levelPrefValue", prefName);
}

updateAppenders() {
  if (this._parent) {
    let notOwnAppenders = this._parent.appenders.filter(function (appender) {
      return !this.ownAppenders.includes(appender);
    }, this);
```

```
    this.appenders = notOwnAppenders.concat(this.ownAppenders);
  } else {
    this.appenders = this.ownAppenders.slice();
  }

  // Update children's appenders.
  for (let i = 0; i < this.children.length; i++) {
    this.children[i].updateAppenders();
  }
}

addAppender(appender) {
  if (this.ownAppenders.includes(appender)) {
    return;
  }
  this.ownAppenders.push(appender);
  this.updateAppenders();
}

removeAppender(appender) {
  let index = this.ownAppenders.indexOf(appender);
  if (index == -1) {
    return;
  }
  this.ownAppenders.splice(index, 1);
  this.updateAppenders();
}

_unpackTemplateLiteral(string, params) {
  if (!Array.isArray(params)) {
    // Regular log() call.
    return [string, params];
  }

  if (!Array.isArray(string)) {
    // Not using template literal. However params was packed into an array by
    // the this.[level] call, so we need to unpack it here.
    return [string, params[0]];
  }

  // We're using template literal format (logger.warn `foo ${bar}`). Turn the
  // template strings into one string containing "${0}"..."${n}" tokens, and
  // feed it to the basic formatter. The formatter will treat the numbers as
  // indices into the params array, and convert the tokens to the params.

  if (!params.length) {
    // No params; we need to set params to undefined, so the formatter
    // doesn't try to output the params array.
    return [string[0], undefined];
  }

  let concat = string[0];
  for (let i = 0; i < params.length; i++) {
    concat += `¥${${i}}${string[i + 1]}`;
  }
  return [concat, params];
}

log(level, string, params) {
  if (this.level > level) {
    return;
  }

  // Hold off on creating the message object until we actually have
  // an appender that's responsible.
  let message;
  let appenders = this.appenders;
  for (let appender of appenders) {
    if (appender.level > level) {
      continue;
    }
    if (!message) {
      [string, params] = this._unpackTemplateLiteral(string, params);
      message = new LogMessage(this._name, level, string, params);
    }
```

```
      appender.append(message);
    }
  }

  fatal(string, ...params) {
    this.log(Log.Level.Fatal, string, params);
  }
  error(string, ...params) {
    this.log(Log.Level.Error, string, params);
  }
  warn(string, ...params) {
    this.log(Log.Level.Warn, string, params);
  }
  info(string, ...params) {
    this.log(Log.Level.Info, string, params);
  }
  config(string, ...params) {
    this.log(Log.Level.Config, string, params);
  }
  debug(string, ...params) {
    this.log(Log.Level.Debug, string, params);
  }
  trace(string, ...params) {
    this.log(Log.Level.Trace, string, params);
  }
}

/*
 * LoggerRepository
 * Implements a hierarchy of Loggers
 */

class LoggerRepository {
  constructor() {
    this._loggers = {};
    this._rootLogger = null;
  }

  get rootLogger() {
    if (!this._rootLogger) {
      this._rootLogger = new Logger("root", this);
      this._rootLogger.level = Log.Level.All;
    }
    return this._rootLogger;
  }
  set rootLogger(logger) {
    throw new Error("Cannot change the root logger");
  }

  _updateParents(name) {
    let pieces = name.split(".");
    let cur, parent;

    // find the closest parent
    // don't test for the logger name itself, as there's a chance it's already
    // there in this._loggers
    for (let i = 0; i < pieces.length - 1; i++) {
      if (cur) {
        cur += "." + pieces[i];
      } else {
        cur = pieces[i];
      }
      if (cur in this._loggers) {
        parent = cur;
      }
    }

    // if we didn't assign a parent above, there is no parent
    if (!parent) {
      this._loggers[name].parent = this.rootLogger;
    } else {
      this._loggers[name].parent = this._loggers[parent];
    }

    // trigger updates for any possible descendants of this logger
```

```
    for (let logger in this._loggers) {
      if (logger != name && logger.indexOf(name) == 0) {
        this._updateParents(logger);
      }
    }
  }

  /**
   * Obtain a named Logger.
   *
   * The returned Logger instance for a particular name is shared among
   * all callers. In other words, if two consumers call getLogger("foo"),
   * they will both have a reference to the same object.
   *
   * @return Logger
   */
  getLogger(name) {
    if (name in this._loggers) {
      return this._loggers[name];
    }
    this._loggers[name] = new Logger(name, this);
    this._updateParents(name);
    return this._loggers[name];
  }

  /**
   * Obtain a Logger that logs all string messages with a prefix.
   *
   * A common pattern is to have separate Logger instances for each instance
   * of an object. But, you still want to distinguish between each instance.
   * Since Log.repository.getLogger() returns shared Logger objects,
   * monkeypatching one Logger modifies them all.
   *
   * This function returns a new object with a prototype chain that chains
   * up to the original Logger instance. The new prototype has log functions
   * that prefix content to each message.
   *
   * @param name
   *        (string) The Logger to retrieve.
   * @param prefix
   *        (string) The string to prefix each logged message with.
   */
  getLoggerWithMessagePrefix(name, prefix) {
    let log = this.getLogger(name);

    let proxy = Object.create(log);
    proxy.log = (level, string, params) => {
      if (Array.isArray(string) && Array.isArray(params)) {
        // Template literal.
        // We cannot change the original array, so create a new one.
        string = [prefix + string[0]].concat(string.slice(1));
      } else {
        string = prefix + string; // Regular string.
      }
      return log.log(level, string, params);
    };
    return proxy;
  }
}

/*
 * Formatters
 * These massage a LogMessage into whatever output is desired.
 */

// Basic formatter that doesn't do anything fancy.
class BasicFormatter {
  constructor(dateFormat) {
    if (dateFormat) {
      this.dateFormat = dateFormat;
    }
    this.parameterFormatter = new ParameterFormatter();
  }

  /**
```

```
   * Format the text of a message with optional parameters.
   * If the text contains ${identifier}, replace that with
   * the value of params[identifier]; if ${}, replace that with
   * the entire params object. If no params have been substituted
   * into the text, format the entire object and append that
   * to the message.
   */
  formatText(message) {
    let params = message.params;
    if (typeof params == "undefined") {
      return message.message || "";
    }
    // Defensive handling of non-object params
    // We could add a special case for NSRESULT values here...
    let pIsObject = typeof params == "object" || typeof params == "function";

    // if we have params, try and find substitutions.
    if (this.parameterFormatter) {
      // have we successfully substituted any parameters into the message?
      // in the log message
      let subDone = false;
      let regex = /¥$¥{(¥S*?)¥}/g;
      let textParts = [];
      if (message.message) {
        textParts.push(
          message.message.replace(regex, (_, sub) => {
            // ${foo} means use the params['foo']
            if (sub) {
              if (pIsObject && sub in message.params) {
                subDone = true;
                return this.parameterFormatter.format(message.params[sub]);
              }
              return "${" + sub + "}";
            }
            // ${} means use the entire params object.
            subDone = true;
            return this.parameterFormatter.format(message.params);
          })
        );
      }
      if (!subDone) {
        // There were no substitutions in the text, so format the entire params object
        let rest = this.parameterFormatter.format(message.params);
        if (rest !== null && rest != "{}") {
          textParts.push(rest);
        }
      }
      return textParts.join(": ");
    }
    return undefined;
  }

  format(message) {
    return (
      message.time +
      "¥t" +
      message.loggerName +
      "¥t" +
      message.levelDesc +
      "¥t" +
      this.formatText(message)
    );
  }
}

/**
 * Test an object to see if it is a Mozilla JS Error.
 */
function isError(aObj) {
  return (
    aObj &&
    typeof aObj == "object" &&
    "name" in aObj &&
    "message" in aObj &&
    "fileName" in aObj &&
```

```
    "lineNumber" in aObj &&
    "stack" in aObj
  );
}

/*
 * Parameter Formatters
 * These massage an object used as a parameter for a LogMessage into
 * a string representation of the object.
 */

class ParameterFormatter {
  constructor() {
    this._name = "ParameterFormatter";
  }

  format(ob) {
    try {
      if (ob === undefined) {
        return "undefined";
      }
      if (ob === null) {
        return "null";
      }
      // Pass through primitive types and objects that unbox to primitive types.
      if (
        (typeof ob != "object" || typeof ob.valueOf() != "object") &&
        typeof ob != "function"
      ) {
        return ob;
      }
      if (ob instanceof Ci.nsIException) {
        return `${ob} ${Log.stackTrace(ob)}`;
      } else if (isError(ob)) {
        return Log._formatError(ob);
      }
      // Just JSONify it. Filter out our internal fields and those the caller has
      // already handled.
      return JSON.stringify(ob, (key, val) => {
        if (INTERNAL_FIELDS.has(key)) {
          return undefined;
        }
        return val;
      });
    } catch (e) {
      dumpError(
        `Exception trying to format object for log message: ${Log.exceptionStr(
          e
        )}`
      );
    }
    // Fancy formatting failed. Just toSource() it - but even this may fail!
    try {
      return ob.toSource();
    } catch (_) {}
    try {
      return String(ob);
    } catch (_) {
      return "[object]";
    }
  }
}

/*
 * Appenders
 * These can be attached to Loggers to log to different places
 * Simply subclass and override doAppend to implement a new one
 */

class Appender {
  constructor(formatter) {
    this.level = Log.Level.All;
    this._name = "Appender";
    this._formatter = formatter || new BasicFormatter();
  }
```

```
  append(message) {
    if (message) {
      this.doAppend(this._formatter.format(message));
    }
  }

  toString() {
    return `${this._name} [level=${this.level}, formatter=${this._formatter}]`;
  }
}

/*
 * DumpAppender
 * Logs to standard out
 */

class DumpAppender extends Appender {
  constructor(formatter) {
    super(formatter);
    this._name = "DumpAppender";
  }

  doAppend(formatted) {
    dump(formatted + "¥n");
  }
}

/*
 * ConsoleAppender
 * Logs to the javascript console
 */

class ConsoleAppender extends Appender {
  constructor(formatter) {
    super(formatter);
    this._name = "ConsoleAppender";
  }

  // XXX this should be replaced with calls to the Browser Console
  append(message) {
    if (message) {
      let m = this._formatter.format(message);
      if (message.level > Log.Level.Warn) {
        // TODO: Bug 1801091 - Figure out how to replace this.
        // eslint-disable-next-line mozilla/no-cu-reportError
        Cu.reportError(m);
        return;
      }
      this.doAppend(m);
    }
  }

  doAppend(formatted) {
    Services.console.logStringMessage(formatted);
  }
}

Object.assign(Log, {
  LogMessage,
  Logger,
  LoggerRepository,

  BasicFormatter,

  Appender,
  DumpAppender,
  ConsoleAppender,

  ParameterFormatter,
});
```