21.2 Creating `Path` Objects

File operations that a Java program invokes are performed on a file system. Unless the program utilizes a specific file system constructed by the factory methods of the `java.nio.file.FileSystems` utility class, file operations are performed on a file system called the *default file system* that is accessible to the JVM. We can think of the default file system as the *local* file system. The default file system is platform specific and can be obtained as an instance of the `java.nio.file.FileSystem` abstract class by invoking the `getDefault()` factory method of the `FileSystems` utility class.

```
FileSystem dfs = FileSystems.getDefault();     // The default file system
```

File operations can access the default file system directly to perform their operations. The default file system can be queried for various file system properties.

`static FileSystem getDefault()` Declared in `java.nio.file.FileSystems`

Returns the platform-specific default file system. If the method is called several times, it returns the same `FileSystem` instance for the default file system.

`abstract String getSeparator()` Declared in `java.nio.file.FileSystem`

Returns the platform-specific name separator for name elements in a path, represented as a string.

Paths in the file system are programmatically represented by objects that implement the `Path` interface. The `Path` interface and various other classes provide factory methods that create `Path` objects (see below). A `Path` object is also platform specific.

A `Path` object implements the `Iterable<Path>` interface, meaning it is possible to traverse over its name elements from the first name element to the last. It is also immutable and therefore thread-safe.

In this section, we look at how to create `Path` objects. A `Path` object can be queried and manipulated in various ways, and may not represent an existing directory entry in the file system (p. 1294). A `Path` object can be used in file operations to access and manipulate the directory entry it denotes in the file system (p. 1304).

The methods below can be used to create `Path` objects. The methods are also shown in Figure 21.1, together with the relationship between the different classes of these methods. Note in particular the interoperability between the `java.nio.file.Path` interface that represents a path and the two classes `java.io.File` and `java.net.URI`. The class `java.io.File` represents a pathname in the standard I/O API and the class `java.net.URI` represents a *Uniform Resource Identifier* (*URI*) that identifies a resource (e.g., a file or a website).

```
static Path of(String first, String... more)
static Path of(URI uri)
```

These methods are declared in the `java.nio.file.Path` interface.

```
abstract Path getPath(String first, String... more)
```

This method is declared in the `java.nio.file.FileSystem` class.

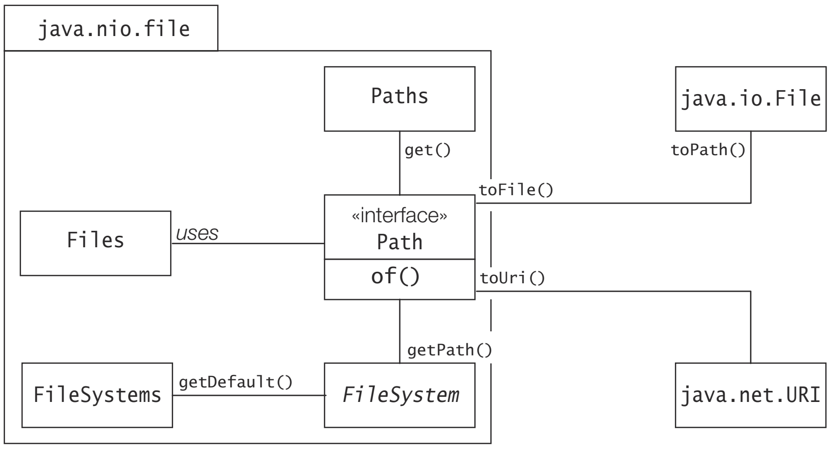
```
static Path get(String first, String... more)
static Path get(URI uri)
```

These static methods are declared in the `java.nio.file.Paths` class.

```
Path toPath()
```

This method is declared in the `java.io.File` class.

Figure 21.1 *Creating* Path *Objects*



Any parameters for a method are not shown.

Creating `Path` Objects with the `Path.of()` Method

The simplest way to create a `Path` is to use the static factory method `Path.of(String first, String... more)`. It joins the `first` string with any strings in the variable arity parameter `more` to create a path string that is the basis of constructing a `Path` object. This method creates a `Path` object in accordance with the default file system.

For instance, the default file system can be queried for the platform-specific name separator for name elements in a path.

The three absolute paths below are equivalent on a Unix platform, as they create a `Path` object based on the same path string. The `nameSeparator` string is the platform-specific name separator obtained from the default file system. At (1) and (2) below, only the `first` string parameter is specified in the `Path.of()` method, and it is the basis for constructing a `Path` object. However, specifying the variable arity parameter where possible is recommended when a `Path` object is constructed as shown at (3), as joining of the name elements is implicitly done using the platform-specific name separator. The `equals()` methods simply checks for equality on the path string, not on any directory entry denoted by the `Path` objects.

```
FileSystem dfs = FileSystems.getDefault();     // Obtain the default file system.
String nameSeparator = dfs.getSeparator();     // The name separator for a path.

Path absPath1 = Path.of("/a/b/c");                         // (1) /a/b/c
Path absPath2 = Path.of(nameSeparator + "a" +              // (2) /a/b/c
                        nameSeparator + "b" +
                        nameSeparator + "c");
Path absPath3 = Path.of(nameSeparator, "a", "b", "c");     // (3) /a/b/c
System.out.println(absPath1.equals(absPath2) &&
                   absPath2.equals(absPath3));             // true
```

The two absolute paths below are equivalent on a Windows platform, as they create `Path` objects based on the same path string. Note that the backslash character must be escaped with a second backslash in a string. Otherwise, it will be interpreted as starting an escape sequence (§2.1, p. 38).

```
Path absPath4 = Path.of("C:\\a\\b\\c");                   // (4) C:\a\b\c
Path absPath5 = Path.of("C:", "a", "b", "c");             // (5) C:\a\b\c
```

The `Path` created below is a relative path, as no root component is specified in the arguments.

```
Path relPath1 = Path.of("c", "d");                        //  c/d
```

Often we need to create a `Path` object to denote the current directory. This can be done via a *system property* named `"user.dir"` that can be looked up, as shown at (1) below, and its value used to construct a `Path` object, as shown at (2). The path string of the current directory can be used to create paths relative to the current directory, as shown at (3).

```
String pathOfCurrDir = System.getProperty("user.dir");  // (1)
Path currDir = Path.of(pathOfCurrDir);                  // (2)
Path relPath = Path.of(pathOfCurrDir, "d");             // (3) <curr-dir-path>/d
```

Creating `Path` Objects with the `Paths.get()` Method

The `Paths` utility class provides the `get(String first, String... more)` static factory method to construct `Path` objects. In fact, this method invokes the `Path.of(String first, String... more)` convenience method to obtain a `Path`.

```
Path absPath7 = Paths.get(nameSeparator, "a", "b", "c");
Path relPath3 = Paths.get("c", "d");
```

Creating `Path` Objects Using the Default File System

We have seen how to obtain the default file system that is accessible to the JVM:

```
FileSystem dfs = FileSystems.getDefault();     // The default file system
```

The default file system provides the `getPath(String first, String... more)` method to construct `Path` objects. In fact, the `Path.of(String first, String... more)` method is a convenience method that invokes the `FileSystem.getPath()` method to obtain a `Path` object.

```
Path absPath6 = dfs.getPath(nameSeparator, "a", "b", "c");
Path relPath2 = dfs.getPath("c", "d");
```

Interoperability with the `java.io.File` Legacy Class

An object of the legacy class `java.io.File` can be used to query the file system for information about a file or a directory. The class also provides methods to create, rename, and delete directory entries in the file system. Although there is an overlap of functionality between the `Path` interface and the `File` class, the `Path` interface is recommended over the `File` class for new code. The interoperability between a `File` object and a `Path` object allows the limitations of the legacy class `java.io.File` to be addressed.

> `File(String pathname)`
> `Path toPath()`
>
> This constructor and this method of the `java.io.File` class can be used to create a `File` object from a pathname and to convert a `File` object to a `Path` object, respectively.
>
> `default File toFile()`
>
> This method of the `java.nio.file.Path` interface can be used to convert a `Path` object to a `File` object.

The code below illustrates the round trip between a `File` object and a `Path` object:

```
File file = new File(File.separator + "a" +
                     File.separator + "b" +
                     File.separator + "c");        // /a/b/c

// File --> Path, using the java.io.File.toPath() instance method
Path fileToPath = file.toPath();                   // /a/b/c

// Path --> File, using the java.nio.file.Path.toFile() default method.
File pathToFile = fileToPath.toFile();             // /a/b/c
```

Interoperability with the `java.net.URI` Class

A URI consists of a string that identifies a resource, which can be a local resource or a remote resource. Among other pertinent information, a URI specifies a *scheme* (e.g., `file`, `ftp`, `http`, and `https`) that indicates what protocol to use to handle the resource. The examples of URIs below show two *schema*: `file` and `http`. We will not elaborate on the syntax of URIs, or different schema, as it is beyond the scope of this book.

```
file:///a/b/c/d                  // Scheme: file, to access a local file.
http://www.whatever.com          // Scheme: http, to access a remote website.
```

> `URI(String str) throws URISyntaxException`
> `static URI create(String str)`
>
> This constructor and this static method in the `java.net.URI` class create a `URL` object based on the specified string. The second method is preferred when it is known that the URI string is well formed.

```
// Create a URI object, using the URL.create(String str) static factory method.
URI uri1 = URI.create("file:///a/b/c/d");   // Local file.
```

The following methods can be used to convert a `URI` object to a `Path` object. The `Paths.get(URI uri)` static factory method actually invokes the `Path.of(URI uri)` static factory method.

```
// URI --> Path, using the Path.of(URI uri) static factory method.
Path uriToPath1 = Path.of(uri1);       // /a/b/c/d

// URI --> Path, using the Paths.get(URI uri) static factory method.
Path uriToPath2 = Paths.get(uri1);     // /a/b/c/d
```

The following method in the `Path` interface can be used to convert a `Path` object to a `URI` object:

```
// Path --> URI, using the Path.toUri() instance method.
URI pathToUri = uriToPath1.toUri();    // file:///a/b/c/d
```

Interoperability between a `Path` object and a `URI` object allows an application to leverage both the NIO.2 API and the network API.

[redacted]