**Java**

**Topics**

1. Programming basics
2. Why Java?
3. Setting up the Java environment.
4. Introduction to Java Platform.
5. Object oriented programming concepts
6. Creating a Java project using Eclipse IDE

=====================================================================================

**Software basics**

Assembler language

Programming lanugauges like C

Operating system

Compiler

Programming languages

<http://en.wikipedia.org/wiki/List_of_programming_languages>

Why Java? (Requirements of a programming language.

The Java programming language is a high-level language that can be characterized by all of the following buzzwords:

|  |  |
| --- | --- |
| * Simple * Object oriented * Distributed * Multithreaded * Dynamic | * Architecture neutral * Portable * High performance * Robust * Secure |

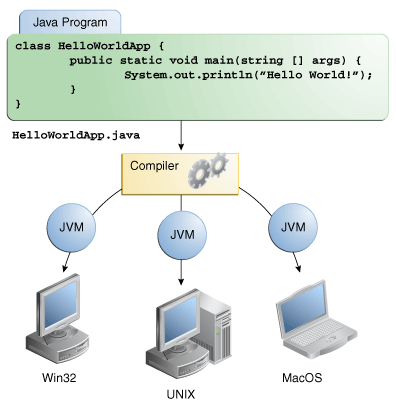
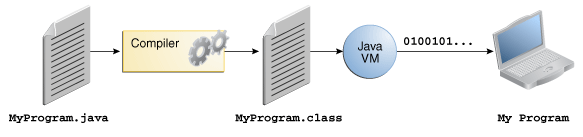
Start with a hello world programme using simple notepad

1. Download the java sdk from website.
2. Execute the programme

Start with a hello world programme using eclipse IDE

<http://docs.oracle.com/javase/tutorial/getStarted/cupojava/netbeans.html>

In the Java programming language, all source code is first written in plain text files ending with the .java extension. Those source files are then compiled into .class files by the javaccompiler. A .class file does not contain code that is native to your processor; it instead contains bytecodes — the machine language of the Java Virtual Machine[1](http://docs.oracle.com/javase/tutorial/getStarted/intro/definition.html#FOOT) (Java VM). The javalauncher tool then runs your application with an instance of the Java Virtual Machine.



## The Java Platform

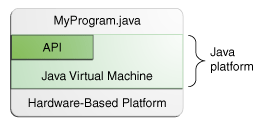
A platform is the hardware or software environment in which a program runs. We've already mentioned some of the most popular platforms like Microsoft Windows, Linux, Solaris OS, and Mac OS. Most platforms can be described as a combination of the operating system and underlying hardware. The Java platform differs from most other platforms in that it's a software-only platform that runs on top of other hardware-based platforms.

The Java platform has two components:

* The Java Virtual Machine
* The Java Application Programming Interface (API)

You've already been introduced to the Java Virtual Machine; it's the base for the Java platform and is ported onto various hardware-based platforms.

The API is a large collection of ready-made software components that provide many useful capabilities. It is grouped into libraries of related classes and interfaces; these libraries are known aspackages. The next section, [What Can Java Technology Do?](http://docs.oracle.com/javase/tutorial/getStarted/intro/cando.html) highlights some of the functionality provided by the API.



The API and Java Virtual Machine insulate the program from the underlying hardware.

As a platform-independent environment, the Java platform can be a bit slower than native code. However, advances in compiler and virtual machine technologies are bringing performance close to that of native code without threatening portability.

# What Can Java Technology Provide?

The general-purpose, high-level Java programming language is a powerful software platform. Every full implementation of the Java platform gives you the following features:

* **Development Tools**: The development tools provide everything you'll need for compiling, running, monitoring, debugging, and documenting your applications. As a new developer, the main tools you'll be using are the javac compiler, the java launcher, and the javadoc documentation tool.
* **Application Programming Interface (API)**: The API provides the core functionality of the Java programming language. It offers a wide array of useful classes ready for use in your own applications. It spans everything from basic objects, to networking and security, to XML generation and database access, and more. The core API is very large; to get an overview of what it contains, consult the [Java Platform Standard Edition 8 Documentation](http://docs.oracle.com/javase/8/docs/index.html).
* **Deployment Technologies**: The JDK software provides standard mechanisms such as the Java Web Start software and Java Plug-In software for deploying your applications to end users.
* **User Interface Toolkits**: The JavaFX, Swing, and Java 2D toolkits make it possible to create sophisticated Graphical User Interfaces (GUIs).
* **Integration Libraries**: Integration libraries such as the Java IDL API, JDBC API, Java Naming and Directory Interface (JNDI) API, Java RMI, and Java Remote Method Invocation over Internet Inter-ORB Protocol Technology (Java RMI-IIOP Technology) enable database access and manipulation of remote objects.

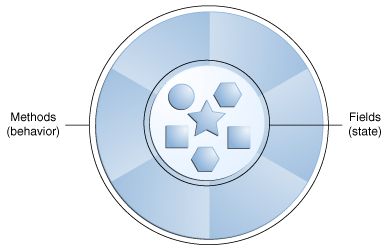
# Object-Oriented Programming Concepts

Object

Real-world objects share two characteristics: They all have *state* and *behavior*. Dogs have state (name, color, breed, hungry) and behavior (barking, fetching, wagging tail). Bicycles also have state (current gear, current pedal cadence, current speed) and behavior (changing gear, changing pedal cadence, applying brakes). Identifying the state and behavior for real-world objects is a great way to begin thinking in terms of object-oriented programming.

Ex: Automobiles

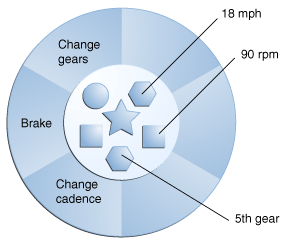
Take a minute right now to observe the real-world objects that are in your immediate area. For each object that you see, ask yourself two questions: "What possible states can this object be in?" and "What possible behavior can this object perform?". Make sure to write down your observations. As you do, you'll notice that real-world objects vary in complexity; your desktop lamp may have only two possible states (on and off) and two possible behaviors (turn on, turn off), but your desktop radio might have additional states (on, off, current volume, current station) and behavior (turn on, turn off, increase volume, decrease volume, seek, scan, and tune). You may also notice that some objects, in turn, will also contain other objects. These real-world observations all translate into the world of object-oriented programming.



A software object.

Software objects are conceptually similar to real-world objects: they too consist of state and related behavior. An object stores its state in *fields* (variables in some programming languages) and exposes its behavior through *methods* (functions in some programming languages). Methods operate on an object's internal state and serve as the primary mechanism for object-to-object communication. Hiding internal state and requiring all interaction to be performed through an object's methods is known as *data encapsulation* — a fundamental principle of object-oriented programming.

Consider a bicycle, for example:



A bicycle modeled as a software object.

By attributing state (current speed, current pedal cadence, and current gear) and providing methods for changing that state, the object remains in control of how the outside world is allowed to use it. For example, if the bicycle only has 6 gears, a method to change gears could reject any value that is less than 1 or greater than 6.

Bundling code into individual software objects provides a number of benefits, including:

1. Modularity: The source code for an object can be written and maintained independently of the source code for other objects. Once created, an object can be easily passed around inside the system.
2. Information-hiding: By interacting only with an object's methods, the details of its internal implementation remain hidden from the outside world.
3. Code re-use: If an object already exists (perhaps written by another software developer), you can use that object in your program. This allows specialists to implement/test/debug complex, task-specific objects, which you can then trust to run in your own code.
4. Pluggability and debugging ease: If a particular object turns out to be problematic, you can simply remove it from your application and plug in a different object as its replacement. This is analogous to fixing mechanical problems in the real world. If a bolt breaks, you replace *it*, not the entire machine.

Class

n the real world, you'll often find many individual objects all of the same kind. There may be thousands of other bicycles in existence, all of the same make and model. Each bicycle was built from the same set of blueprints and therefore contains the same components. In object-oriented terms, we say that your bicycle is an *instance* of the *class of objects* known as bicycles. A *class* is the blueprint from which individual objects are created.

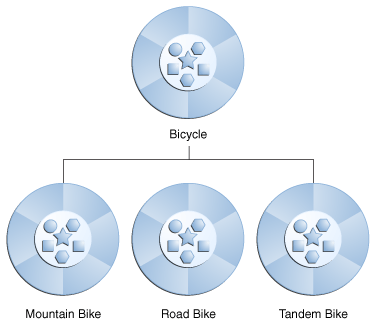
The following [Bicycle](http://docs.oracle.com/javase/tutorial/java/concepts/examples/Bicycle.java) class is one possible implementation of a bicycle:

# What Is Inheritance?

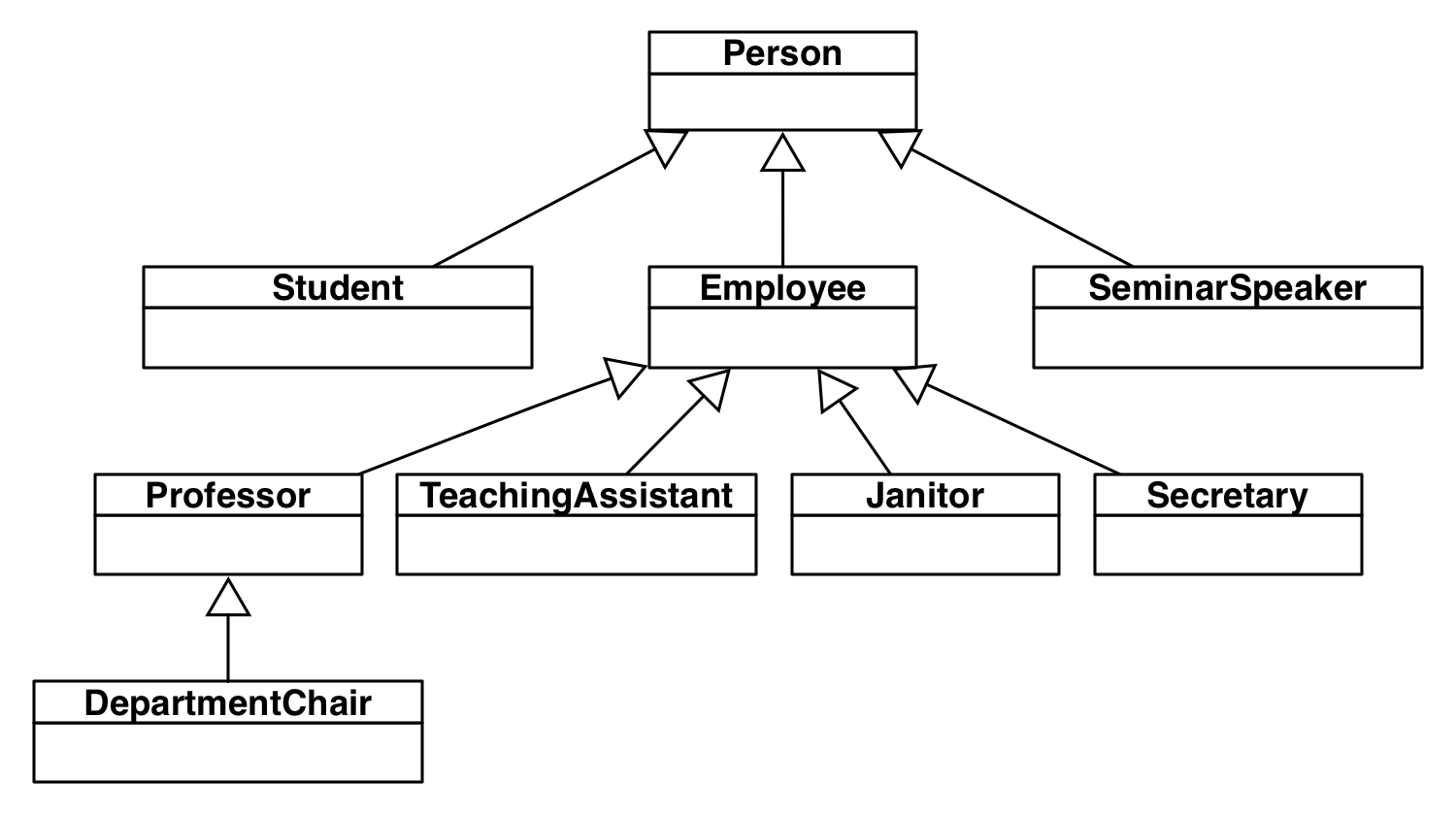
**Inheritance** can be defined as the process whereby an object of a class acquires characteristics from the object of the other class. All the objects of a similar kind are grouped together to form a class. However, sometimes a situation arises when different objects cannot be combined together under a single group as they share only some common characteristics. In this situation, the classes are defined in such a way that the common features are combined to form a generalized class and the specific features are combined to form a specialized class. The specialized class is defined in such a way that in addition to the individual characteristics and functions, it also inherits all the properties and the functions of its generalized class.

Different kinds of objects often have a certain amount in common with each other. Mountain bikes, road bikes, and tandem bikes, for example, all share the characteristics of bicycles (current speed, current pedal cadence, current gear). Yet each also defines additional features that make them different: tandem bicycles have two seats and two sets of handlebars; road bikes have drop handlebars; some mountain bikes have an additional chain ring, giving them a lower gear ratio.

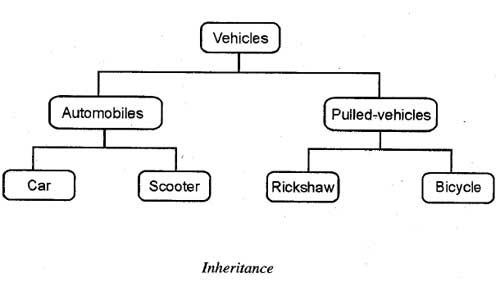
Object-oriented programming allows classes to *inherit* commonly used state and behavior from other classes. In this example, Bicycle now becomes the *superclass* of MountainBike,RoadBike, and TandemBike. In the Java programming language, each class is allowed to have one direct superclass, and each superclass has the potential for an unlimited number of*subclasses*:



Example:



Example 2:



Encapsulation

**Encapsulation** is the packing of data and functions into a single component. The features of encapsulation are supported using classes in most object-oriented programming languages, although other alternatives also exist. It allows selective hiding of properties and methods in an object by building an impenetrable wall to protect the code from accidental corruption.

Encapsulation can be used to hide data member and member function. Under this definition, encapsulation means that the internal representation of an [object](https://en.wikipedia.org/wiki/Object_(computer_science)) is generally hidden from view outside of the object's definition. Typically, only the object's own methods can directly inspect or manipulate its field.

Hiding the internals of the object protects its integrity by preventing users from setting the internal data of the component into an invalid or inconsistent state. A supposed benefit of encapsulation is that it can reduce system complexity, and thus increase [robustness](https://en.wikipedia.org/wiki/Robustness_(computer_science)).

# Polymorphism

The dictionary definition of polymorphism refers to a principle in biology in which an organism or species can have many different forms or stages. This principle can also be applied to object-oriented programming and languages like the Java language. Subclasses of a class can define their own unique behaviors and yet share some of the same functionality of the parent class.

**Abstraction**

**abstraction** is a technique for managing complexity of computer systems. It works by establishing a level of complexity on which a person interacts with the system, suppressing the more complex details below the current level. The programmer works with an idealized interface (usually well defined) and can add additional levels of functionality that would otherwise be too complex to handle. For example, a programmer writing code that involves numerical operations may not be interested in the way numbers are represented in the underlying hardware (e.g. whether they're *16 bit* or *32 bit*[*integers*](https://en.wikipedia.org/wiki/Integer_(computer_science))), and where those details have been suppressed it can be said that they were *abstracted away*, leaving simply *numbers* with which the programmer can work. In addition, a task of sending an email message across continents would be extremely complex if you start with a piece of optic cable and basic hardware components. By using layers of complexity that have been created to abstract away the physical cables, network layout and presenting the programmer with a virtual data channel, this task is manageable.

# What Is an Interface?

As you've already learned, objects define their interaction with the outside world through the methods that they expose. Methods form the object's *interface* with the outside world; the buttons on the front of your television set, for example, are the interface between you and the electrical wiring on the other side of its plastic casing. You press the "power" button to turn the television on and off.

In its most common form, an interface is a group of related methods with empty bodies. A bicycle's behavior, if specified as an interface, might appear as follows:

interface Bicycle {

// wheel revolutions per minute

void changeCadence(int newValue);

void changeGear(int newValue);

void speedUp(int increment);

void applyBrakes(int decrement);

}

To implement this interface, the name of your class would change (to a particular brand of bicycle, for example, such as ACMEBicycle), and you'd use the implements keyword in the class declaration:

class ACMEBicycle **implements** Bicycle {

int cadence = 0;

int speed = 0;

int gear = 1;

// The compiler will now require that methods

// changeCadence, changeGear, speedUp, and applyBrakes

// all be implemented. Compilation will fail if those

// methods are missing from this class.

void changeCadence(int newValue) {

cadence = newValue;

}

void changeGear(int newValue) {

gear = newValue;

}

void speedUp(int increment) {

speed = speed + increment;

}

void applyBrakes(int decrement) {

speed = speed - decrement;

}

void printStates() {

System.out.println("cadence:" +

cadence + " speed:" +

speed + " gear:" + gear);

}

}

# What Is a Package?

A package is a namespace that organizes a set of related classes and interfaces. Conceptually you can think of packages as being similar to different folders on your computer. You might keep HTML pages in one folder, images in another, and scripts or applications in yet another. Because software written in the Java programming language can be composed of hundreds or *thousands*of individual classes, it makes sense to keep things organized by placing related classes and interfaces into packages.

[Language Basics](http://docs.oracle.com/javase/tutorial/java/nutsandbolts/index.html)

# Variables

As you learned in the previous lesson, an object stores its state in *fields*.

int **cadence** = 0;

int **speed** = 0;

int **gear** = 1;

The Java programming language defines the following kinds of variables:

* **Instance Variables (Non-Static Fields)** Technically speaking, objects store their individual states in "non-static fields", that is, fields declared without the static keyword. Non-static fields are also known as *instance variables* because their values are unique to each *instance* of a class (to each object, in other words); the currentSpeed of one bicycle is independent from the currentSpeed of another.
* **Class Variables (Static Fields)** A *class variable* is any field declared with the static modifier; this tells the compiler that there is exactly one copy of this variable in existence, regardless of how many times the class has been instantiated. A field defining the number of gears for a particular kind of bicycle could be marked as static since conceptually the same number of gears will apply to all instances. The code static int numGears = 6; would create such a static field. Additionally, the keyword final could be added to indicate that the number of gears will never change.
* **Local Variables** Similar to how an object stores its state in fields, a method will often store its temporary state in *local variables*. The syntax for declaring a local variable is similar to declaring a field (for example, int count = 0;). There is no special keyword designating a variable as local; that determination comes entirely from the location in which the variable is declared — which is between the opening and closing braces of a method. As such, local variables are only visible to the methods in which they are declared; they are not accessible from the rest of the class.
* **Parameters** You've already seen examples of parameters, both in the Bicycle class and in the main method of the "Hello World!" application. Recall that the signature for the mainmethod is public static void main(String[] args). Here, the args variable is the parameter to this method. The important thing to remember is that parameters are always classified as "variables" not "fields". This applies to other parameter-accepting constructs as well (such as constructors and exception handlers) that you'll learn about later in the tutorial.

## Naming

* Variable names are case-sensitive. A variable's name can be any legal identifier — an unlimited-length sequence of Unicode letters and digits, beginning with a letter, the dollar sign "$", or the underscore character "\_". The convention, however, is to always begin your variable names with a letter, not "$" or "\_". Additionally, the dollar sign character, by convention, is never used at all. You may find some situations where auto-generated names will contain the dollar sign, but your variable names should always avoid using it. A similar convention exists for the underscore character; while it's technically legal to begin your variable's name with "\_", this practice is discouraged. White space is not permitted.
* Subsequent characters may be letters, digits, dollar signs, or underscore characters. Conventions (and common sense) apply to this rule as well. When choosing a name for your variables, use full words instead of cryptic abbreviations. Doing so will make your code easier to read and understand. In many cases it will also make your code self-documenting; fields namedcadence, speed, and gear, for example, are much more intuitive than abbreviated versions, such as s, c, and g. Also keep in mind that the name you choose must not be a [keyword or reserved word](http://docs.oracle.com/javase/tutorial/java/nutsandbolts/_keywords.html).
* If the name you choose consists of only one word, spell that word in all lowercase letters. If it consists of more than one word, capitalize the first letter of each subsequent word. The names gearRatio and currentGear are prime examples of this convention. If your variable stores a constant value, such as static final int NUM\_GEARS = 6, the convention changes slightly, capitalizing every letter and separating subsequent words with the underscore character. By convention, the underscore character is never used elsewhere.

# Primitive Data Types

The Java programming language is statically-typed, which means that all variables must first be declared before they can be used. This involves stating the variable's type and name, as you've already seen:

int gear = 1;

Doing so tells your program that a field named "gear" exists, holds numerical data, and has an initial value of "1". A variable's data type determines the values it may contain, plus the operations that may be performed on it. In addition to int, the Java programming language supports seven other *primitive data types*. A primitive type is predefined by the language and is named by a reserved keyword. Primitive values do not share state with other primitive values. The eight primitive data types supported by the Java programming language are:

* **byte**: The byte data type is an 8-bit signed two's complement integer. It has a minimum value of -128 and a maximum value of 127 (inclusive). The byte data type can be useful for saving memory in large [arrays](http://docs.oracle.com/javase/tutorial/java/nutsandbolts/arrays.html), where the memory savings actually matters. They can also be used in place of int where their limits help to clarify your code; the fact that a variable's range is limited can serve as a form of documentation.
* **short**: The short data type is a 16-bit signed two's complement integer. It has a minimum value of -32,768 and a maximum value of 32,767 (inclusive). As with byte, the same guidelines apply: you can use a short to save memory in large arrays, in situations where the memory savings actually matters.
* **int**: By default, the int data type is a 32-bit signed two's complement integer, which has a minimum value of -231 and a maximum value of 231-1. In Java SE 8 and later, you can use the int data type to represent an unsigned 32-bit integer, which has a minimum value of 0 and a maximum value of 232-1. Use the Integer class to use int data type as an unsigned integer. See the section The Number Classes for more information. Static methods like compareUnsigned, divideUnsigned etc have been added to the [Integer](http://docs.oracle.com/javase/8/docs/api/java/lang/Integer.html) class to support the arithmetic operations for unsigned integers.
* **long**: The long data type is a 64-bit two's complement integer. The signed long has a minimum value of -263 and a maximum value of 263-1. In Java SE 8 and later, you can use thelong data type to represent an unsigned 64-bit long, which has a minimum value of 0 and a maximum value of 264-1. Use this data type when you need a range of values wider than those provided by int. The [Long](http://docs.oracle.com/javase/8/docs/api/java/lang/Long.html) class also contains methods like compareUnsigned, divideUnsigned etc to support arithmetic operations for unsigned long.
* **float**: The float data type is a single-precision 32-bit IEEE 754 floating point. Its range of values is beyond the scope of this discussion, but is specified in the [Floating-Point Types, Formats, and Values](http://docs.oracle.com/javase/specs/jls/se7/html/jls-4.html#jls-4.2.3) section of the Java Language Specification. As with the recommendations for byte and short, use a float (instead of double) if you need to save memory in large arrays of floating point numbers. This data type should never be used for precise values, such as currency. For that, you will need to use the [java.math.BigDecimal](http://docs.oracle.com/javase/8/docs/api/java/math/BigDecimal.html) class instead.[Numbers and Strings](http://docs.oracle.com/javase/tutorial/java/data/index.html) covers BigDecimal and other useful classes provided by the Java platform.
* **double**: The double data type is a double-precision 64-bit IEEE 754 floating point. Its range of values is beyond the scope of this discussion, but is specified in the [Floating-Point Types, Formats, and Values](http://docs.oracle.com/javase/specs/jls/se7/html/jls-4.html#jls-4.2.3) section of the Java Language Specification. For decimal values, this data type is generally the default choice. As mentioned above, this data type should never be used for precise values, such as currency.
* **boolean**: The boolean data type has only two possible values: true and false. Use this data type for simple flags that track true/false conditions. This data type represents one bit of information, but its "size" isn't something that's precisely defined.
* **char**: The char data type is a single 16-bit Unicode character. It has a minimum value of '\u0000' (or 0) and a maximum value of '\uffff' (or 65,535 inclusive).

In addition to the eight primitive data types listed above, the Java programming language also provides special support for character strings via the [java.lang.String](http://docs.oracle.com/javase/8/docs/api/java/lang/String.html) class. Enclosing your character string within double quotes will automatically create a new String object; for example, String s = "this is a string";. String objects are *immutable*, which means that once created, their values cannot be changed. The String class is not technically a primitive data type, but considering the special support given to it by the language, you'll probably tend to think of it as such. You'll learn more about the String class in [Simple Data Objects](http://docs.oracle.com/javase/tutorial/java/data/index.html)

## Default Values

It's not always necessary to assign a value when a field is declared. Fields that are declared but not initialized will be set to a reasonable default by the compiler. Generally speaking, this default will be zero or null, depending on the data type. Relying on such default values, however, is generally considered bad programming style.

The following chart summarizes the default values for the above data types.

|  |  |
| --- | --- |
| **Data Type** | **Default Value (for fields)** |
| byte | 0 |
| short | 0 |
| int | 0 |
| long | 0L |
| float | 0.0f |
| double | 0.0d |
| char | '\u0000' |
| String (or any object) | null |
| boolean | false |

Local variables are slightly different; the compiler never assigns a default value to an uninitialized local variable. If you cannot initialize your local variable where it is declared, make sure to assign it a value before you attempt to use it. Accessing an uninitialized local variable will result in a compile-time error.

### Literals

You may have noticed that the new keyword isn't used when initializing a variable of a primitive type. Primitive types are special data types built into the language; they are not objects created from a class. A literal is the source code representation of a fixed value; literals are represented directly in your code without requiring computation. As shown below, it's possible to assign a literal to a variable of a primitive type:

boolean result = true;

char capitalC = 'C';

byte b = 100;

short s = 10000;

int i = 100000;

#### Integer Literals

An integer literal is of type long if it ends with the letter L or l; otherwise it is of type int. It is recommended that you use the upper case letter L because the lower case letter l is hard to distinguish from the digit 1.

Values of the integral types byte, short, int, and long can be created from int literals. Values of type long that exceed the range of int can be created from long literals. Integer literals can be expressed by these number systems:

* Decimal: Base 10, whose digits consists of the numbers 0 through 9; this is the number system you use every day
* Hexadecimal: Base 16, whose digits consist of the numbers 0 through 9 and the letters A through F
* Binary: Base 2, whose digits consists of the numbers 0 and 1 (you can create binary literals in Java SE 7 and later)

For general-purpose programming, the decimal system is likely to be the only number system you'll ever use. However, if you need to use another number system, the following example shows the correct syntax. The prefix 0x indicates hexadecimal and 0b indicates binary:

// The number 26, in decimal

int decVal = 26;

// The number 26, in hexadecimal

int hexVal = 0x1a;

// The number 26, in binary

int binVal = 0b11010;

#### Floating-Point Literals

A floating-point literal is of type float if it ends with the letter F or f; otherwise its type is double and it can optionally end with the letter D or d.

The floating point types (float and double) can also be expressed using E or e (for scientific notation), F or f (32-bit float literal) and D or d (64-bit double literal; this is the default and by convention is omitted).

double d1 = 123.4;

// same value as d1, but in scientific notation

double d2 = 1.234e2;

float f1 = 123.4f;

#### Character and String Literals

Literals of types char and String may contain any Unicode (UTF-16) characters. If your editor and file system allow it, you can use such characters directly in your code. If not, you can use a "Unicode escape" such as '\u0108' (capital C with circumflex), or "S\u00ED Se\u00F1or" (Sí Señor in Spanish). Always use 'single quotes' for char literals and "double quotes" forString literals. Unicode escape sequences may be used elsewhere in a program (such as in field names, for example), not just in char or String literals.

The Java programming language also supports a few special escape sequences for char and String literals: \b (backspace), \t (tab), \n (line feed), \f (form feed), \r (carriage return), \"(double quote), \' (single quote), and \\ (backslash).

There's also a special null literal that can be used as a value for any reference type. null may be assigned to any variable, except variables of primitive types. There's little you can do with anull value beyond testing for its presence. Therefore, null is often used in programs as a marker to indicate that some object is unavailable.

Finally, there's also a special kind of literal called a class literal, formed by taking a type name and appending ".class"; for example, String.class. This refers to the object (of typeClass) that represents the type itself.

## Using Underscore Characters in Numeric Literals

In Java SE 7 and later, any number of underscore characters (\_) can appear anywhere between digits in a numerical literal. This feature enables you, for example. to separate groups of digits in numeric literals, which can improve the readability of your code.

For instance, if your code contains numbers with many digits, you can use an underscore character to separate digits in groups of three, similar to how you would use a punctuation mark like a comma, or a space, as a separator.

The following example shows other ways you can use the underscore in numeric literals:

long creditCardNumber = 1234\_5678\_9012\_3456L;

long socialSecurityNumber = 999\_99\_9999L;

float pi = 3.14\_15F;

long hexBytes = 0xFF\_EC\_DE\_5E;

long hexWords = 0xCAFE\_BABE;

long maxLong = 0x7fff\_ffff\_ffff\_ffffL;

byte nybbles = 0b0010\_0101;

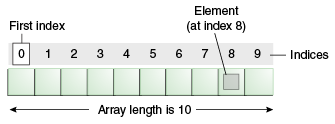
long bytes = 0b11010010\_01101001\_10010100\_10010010;

You can place underscores only between digits; you cannot place underscores in the following places:

* At the beginning or end of a number
* Adjacent to a decimal point in a floating point literal
* Prior to an F or L suffix
* In positions where a string of digits is expected

# Arrays

An *array* is a container object that holds a fixed number of values of a single type. The length of an array is established when the array is created. After creation, its length is fixed. You have seen an example of arrays already, in the main method of the "Hello World!" application. This section discusses arrays in greater detail.



An array of 10 elements.

Each item in an array is called an *element*, and each element is accessed by its numerical *index*. As shown in the preceding illustration, numbering begins with 0. The 9th element, for example, would therefore be accessed at index 8.

The following program, [ArrayDemo](http://docs.oracle.com/javase/tutorial/java/nutsandbolts/examples/ArrayDemo.java), creates an array of integers, puts some values in the array, and prints each value to standard output.

class ArrayDemo {

public static void main(String[] args) {

// declares an array of integers

int[] anArray;

// allocates memory for 10 integers

anArray = new int[10];

// initialize first element

anArray[0] = 100;

// initialize second element

anArray[1] = 200;

// and so forth

anArray[2] = 300;

anArray[3] = 400;

anArray[4] = 500;

anArray[5] = 600;

anArray[6] = 700;

anArray[7] = 800;

anArray[8] = 900;

anArray[9] = 1000;

System.out.println("Element at index 0: "

+ anArray[0]);

System.out.println("Element at index 1: "

+ anArray[1]);

System.out.println("Element at index 2: "

+ anArray[2]);

System.out.println("Element at index 3: "

+ anArray[3]);

System.out.println("Element at index 4: "

+ anArray[4]);

System.out.println("Element at index 5: "

+ anArray[5]);

System.out.println("Element at index 6: "

+ anArray[6]);

System.out.println("Element at index 7: "

+ anArray[7]);

System.out.println("Element at index 8: "

+ anArray[8]);

System.out.println("Element at index 9: "

+ anArray[9]);

}

}

The output from this program is:

Element at index 0: 100

Element at index 1: 200

Element at index 2: 300

Element at index 3: 400

Element at index 4: 500

Element at index 5: 600

Element at index 6: 700

Element at index 7: 800

Element at index 8: 900

Element at index 9: 1000

In a real-world programming situation, you would probably use one of the supported *looping constructs* to iterate through each element of the array, rather than write each line individually as in the preceding example. However, the example clearly illustrates the array syntax. You will learn about the various looping constructs (for, while, and do-while) in the [Control Flow](http://docs.oracle.com/javase/tutorial/java/nutsandbolts/flow.html) section.

## Declaring a Variable to Refer to an Array

The preceding program declares an array (named anArray) with the following line of code:

// declares an array of integers

int[] anArray;

Like declarations for variables of other types, an array declaration has two components: the array's type and the array's name. An array's type is written as type[], where type is the data type of the contained elements; the brackets are special symbols indicating that this variable holds an array. The size of the array is not part of its type (which is why the brackets are empty). An array's name can be anything you want, provided that it follows the rules and conventions as previously discussed in the [naming](http://docs.oracle.com/javase/tutorial/java/nutsandbolts/variables.html#naming) section. As with variables of other types, the declaration does not actually create an array; it simply tells the compiler that this variable will hold an array of the specified type.

Similarly, you can declare arrays of other types:

byte[] anArrayOfBytes;

short[] anArrayOfShorts;

long[] anArrayOfLongs;

float[] anArrayOfFloats;

double[] anArrayOfDoubles;

boolean[] anArrayOfBooleans;

char[] anArrayOfChars;

String[] anArrayOfStrings;

You can also place the brackets after the array's name:

// this form is discouraged

float anArrayOfFloats[];

However, convention discourages this form; the brackets identify the array type and should appear with the type designation.

## Creating, Initializing, and Accessing an Array

One way to create an array is with the new operator. The next statement in the ArrayDemo program allocates an array with enough memory for 10 integer elements and assigns the array to theanArray variable.

// create an array of integers

anArray = new int[10];

If this statement is missing, then the compiler prints an error like the following, and compilation fails:

ArrayDemo.java:4: Variable anArray may not have been initialized.

The next few lines assign values to each element of the array:

anArray[0] = 100; // initialize first element

anArray[1] = 200; // initialize second element

anArray[2] = 300; // and so forth

Each array element is accessed by its numerical index:

System.out.println("Element 1 at index 0: " + anArray[0]);

System.out.println("Element 2 at index 1: " + anArray[1]);

System.out.println("Element 3 at index 2: " + anArray[2]);

Alternatively, you can use the shortcut syntax to create and initialize an array:

int[] anArray = {

100, 200, 300,

400, 500, 600,

700, 800, 900, 1000

};

Here the length of the array is determined by the number of values provided between braces and separated by commas.

You can also declare an array of arrays (also known as a multidimensional array) by using two or more sets of brackets, such as String[][] names. Each element, therefore, must be accessed by a corresponding number of index values.

In the Java programming language, a multidimensional array is an array whose components are themselves arrays. This is unlike arrays in C or Fortran. A consequence of this is that the rows are allowed to vary in length, as shown in the following [MultiDimArrayDemo](http://docs.oracle.com/javase/tutorial/java/nutsandbolts/examples/MultiDimArrayDemo.java) program:

class MultiDimArrayDemo {

public static void main(String[] args) {

String[][] names = {

{"Mr. ", "Mrs. ", "Ms. "},

{"Smith", "Jones"}

};

// Mr. Smith

System.out.println(names[0][0] + names[1][0]);

// Ms. Jones

System.out.println(names[0][2] + names[1][1]);

}

}

The output from this program is:

Mr. Smith

Ms. Jones

Finally, you can use the built-in length property to determine the size of any array. The following code prints the array's size to standard output:

System.out.println(anArray.length);

## Copying Arrays

The System class has an arraycopy method that you can use to efficiently copy data from one array into another:

public static void arraycopy(Object src, int srcPos,

Object dest, int destPos, int length)

The two Object arguments specify the array to copy *from* and the array to copy *to*. The three int arguments specify the starting position in the source array, the starting position in the destination array, and the number of array elements to copy.

The following program, [ArrayCopyDemo](http://docs.oracle.com/javase/tutorial/java/nutsandbolts/examples/ArrayCopyDemo.java), declares an array of char elements, spelling the word "decaffeinated." It uses the System.arraycopy method to copy a subsequence of array components into a second array:

class ArrayCopyDemo {

public static void main(String[] args) {

char[] copyFrom = { 'd', 'e', 'c', 'a', 'f', 'f', 'e',

'i', 'n', 'a', 't', 'e', 'd' };

char[] copyTo = new char[7];

System.arraycopy(copyFrom, 2, copyTo, 0, 7);

System.out.println(new String(copyTo));

}

}

The output from this program is:

caffein

## Array Manipulations

Arrays are a powerful and useful concept used in programming. Java SE provides methods to perform some of the most common manipulations related to arrays. For instance, the[ArrayCopyDemo](http://docs.oracle.com/javase/tutorial/java/nutsandbolts/examples/ArrayCopyDemo.java) example uses the arraycopy method of the System class instead of manually iterating through the elements of the source array and placing each one into the destination array. This is performed behind the scenes, enabling the developer to use just one line of code to call the method.

For your convenience, Java SE provides several methods for performing array manipulations (common tasks, such as copying, sorting and searching arrays) in the [java.util.Arrays](http://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html) class. For instance, the previous example can be modified to use the copyOfRange method of the java.util.Arrays class, as you can see in the [ArrayCopyOfDemo](http://docs.oracle.com/javase/tutorial/java/nutsandbolts/examples/ArrayCopyOfDemo.java) example. The difference is that using the copyOfRange method does not require you to create the destination array before calling the method, because the destination array is returned by the method:

class ArrayCopyOfDemo {

public static void main(String[] args) {

char[] copyFrom = {'d', 'e', 'c', 'a', 'f', 'f', 'e',

'i', 'n', 'a', 't', 'e', 'd'};

char[] copyTo = java.util.Arrays.copyOfRange(copyFrom, 2, 9);

System.out.println(new String(copyTo));

}

}

As you can see, the output from this program is the same (caffein), although it requires fewer lines of code. Note that the second parameter of the copyOfRange method is the initial index of the range to be copied, inclusively, while the third parameter is the final index of the range to be copied, exclusively. In this example, the range to be copied does not include the array element at index 9 (which contains the character a).

Some other useful operations provided by methods in the java.util.Arrays class, are:

* Searching an array for a specific value to get the index at which it is placed (the binarySearch method).
* Comparing two arrays to determine if they are equal or not (the equals method).
* Filling an array to place a specific value at each index (the fill method).
* Sorting an array into ascending order. This can be done either sequentially, using the sort method, or concurrently, using the parallelSort method introduced in Java SE 8. Parallel sorting of large arrays on multiprocessor systems is faster than sequential array sorting.

# Operators

Now that you've learned how to declare and initialize variables, you probably want to know how to *do something* with them. Learning the operators of the Java programming language is a good place to start. Operators are special symbols that perform specific operations on one, two, or three *operands*, and then return a result.

As we explore the operators of the Java programming language, it may be helpful for you to know ahead of time which operators have the highest precedence. The operators in the following table are listed according to precedence order. The closer to the top of the table an operator appears, the higher its precedence. Operators with higher precedence are evaluated before operators with relatively lower precedence. Operators on the same line have equal precedence. When operators of equal precedence appear in the same expression, a rule must govern which is evaluated first. All binary operators except for the assignment operators are evaluated from left to right; assignment operators are evaluated right to left.

|  |  |
| --- | --- |
| **Operator Precedence** | |
| **Operators** | **Precedence** |
| postfix | expr++ expr-- |
| unary | ++expr --expr +expr -expr ~ ! |
| multiplicative | \* / % |
| additive | + - |
| shift | << >> >>> |
| relational | < > <= >= instanceof |
| equality | == != |
| bitwise AND | & |
| bitwise exclusive OR | ^ |
| bitwise inclusive OR | | |
| logical AND | && |
| logical OR | || |
| ternary | ? : |
| assignment | = += -= \*= /= %= &= ^= |= <<= >>= >>>= |

In general-purpose programming, certain operators tend to appear more frequently than others; for example, the assignment operator "=" is far more common than the unsigned right shift operator ">>>". With that in mind, the following discussion focuses first on the operators that you're most likely to use on a regular basis, and ends focusing on those that are less common. Each discussion is accompanied by sample code that you can compile and run. Studying its output will help reinforce what you've just learned.

# Assignment, Arithmetic, and Unary Operators

## The Simple Assignment Operator

One of the most common operators that you'll encounter is the simple assignment operator "=". You saw this operator in the Bicycle class; it assigns the value on its right to the operand on its left:

int cadence = 0;

int speed = 0;

int gear = 1;

This operator can also be used on objects to assign *object references*, as discussed in [Creating Objects](http://docs.oracle.com/javase/tutorial/java/javaOO/objectcreation.html).

## The Arithmetic Operators

The Java programming language provides operators that perform addition, subtraction, multiplication, and division. There's a good chance you'll recognize them by their counterparts in basic mathematics. The only symbol that might look new to you is "%", which divides one operand by another and returns the remainder as its result.

|  |  |
| --- | --- |
| **Operator** | **Description** |
| + | Additive operator (also used for String concatenation) |
| - | Subtraction operator |
| \* | Multiplication operator |
| / | Division operator |
| % | Remainder operator |

The following program, [ArithmeticDemo](http://docs.oracle.com/javase/tutorial/java/nutsandbolts/examples/ArithmeticDemo.java), tests the arithmetic operators.

class ArithmeticDemo {

public static void main (String[] args) {

int result = 1 + 2;

// result is now 3

System.out.println("1 + 2 = " + result);

int original\_result = result;

result = result - 1;

// result is now 2

System.out.println(original\_result + " - 1 = " + result);

original\_result = result;

result = result \* 2;

// result is now 4

System.out.println(original\_result + " \* 2 = " + result);

original\_result = result;

result = result / 2;

// result is now 2

System.out.println(original\_result + " / 2 = " + result);

original\_result = result;

result = result + 8;

// result is now 10

System.out.println(original\_result + " + 8 = " + result);

original\_result = result;

result = result % 7;

// result is now 3

System.out.println(original\_result + " % 7 = " + result);

}

}

This program prints the following:

1 + 2 = 3

3 - 1 = 2

2 \* 2 = 4

4 / 2 = 2

2 + 8 = 10

10 % 7 = 3

You can also combine the arithmetic operators with the simple assignment operator to create *compound assignments*. For example, x+=1; and x=x+1; both increment the value of x by 1.

The + operator can also be used for concatenating (joining) two strings together, as shown in the following [ConcatDemo](http://docs.oracle.com/javase/tutorial/java/nutsandbolts/examples/ConcatDemo.java) program:

class ConcatDemo {

public static void main(String[] args){

String firstString = "This is";

String secondString = " a concatenated string.";

String thirdString = firstString+secondString;

System.out.println(thirdString);

}

}

By the end of this program, the variable thirdString contains "This is a concatenated string.", which gets printed to standard output.

## The Unary Operators

The unary operators require only one operand; they perform various operations such as incrementing/decrementing a value by one, negating an expression, or inverting the value of a boolean.

|  |  |
| --- | --- |
| **Operator** | **Description** |
| + | Unary plus operator; indicates positive value (numbers are positive without this, however) |
| - | Unary minus operator; negates an expression |
| ++ | Increment operator; increments a value by 1 |
| -- | Decrement operator; decrements a value by 1 |
| ! | Logical complement operator; inverts the value of a boolean |

The following program, [UnaryDemo](http://docs.oracle.com/javase/tutorial/java/nutsandbolts/examples/UnaryDemo.java), tests the unary operators:

class UnaryDemo {

public static void main(String[] args) {

int result = +1;

// result is now 1

System.out.println(result);

result--;

// result is now 0

System.out.println(result);

result++;

// result is now 1

System.out.println(result);

result = -result;

// result is now -1

System.out.println(result);

boolean success = false;

// false

System.out.println(success);

// true

System.out.println(!success);

}

}

The increment/decrement operators can be applied before (prefix) or after (postfix) the operand. The code result++; and ++result; will both end in result being incremented by one. The only difference is that the prefix version (++result) evaluates to the incremented value, whereas the postfix version (result++) evaluates to the original value. If you are just performing a simple increment/decrement, it doesn't really matter which version you choose. But if you use this operator in part of a larger expression, the one that you choose may make a significant difference.

The following program, [PrePostDemo](http://docs.oracle.com/javase/tutorial/java/nutsandbolts/examples/PrePostDemo.java), illustrates the prefix/postfix unary increment operator:

class PrePostDemo {

public static void main(String[] args){

int i = 3;

i++;

// prints 4

System.out.println(i);

++i;

// prints 5

System.out.println(i);

// prints 6

System.out.println(++i);

// prints 6

System.out.println(i++);

// prints 7

System.out.println(i);

}

}

# Equality, Relational, and Conditional Operators

## The Equality and Relational Operators

The equality and relational operators determine if one operand is greater than, less than, equal to, or not equal to another operand. The majority of these operators will probably look familiar to you as well. Keep in mind that you must use "==", not "=", when testing if two primitive values are equal.

== equal to

!= not equal to

> greater than

>= greater than or equal to

< less than

<= less than or equal to

The following program, [ComparisonDemo](http://docs.oracle.com/javase/tutorial/java/nutsandbolts/examples/ComparisonDemo.java), tests the comparison operators:

class ComparisonDemo {

public static void main(String[] args){

int value1 = 1;

int value2 = 2;

if(value1 == value2)

System.out.println("value1 == value2");

if(value1 != value2)

System.out.println("value1 != value2");

if(value1 > value2)

System.out.println("value1 > value2");

if(value1 < value2)

System.out.println("value1 < value2");

if(value1 <= value2)

System.out.println("value1 <= value2");

}

}

Output:

value1 != value2

value1 < value2

value1 <= value2

## The Conditional Operators

The && and || operators perform Conditional-AND and Conditional-OR operations on two boolean expressions. These operators exhibit "short-circuiting" behavior, which means that the second operand is evaluated only if needed.

&& Conditional-AND

|| Conditional-OR

The following program, [ConditionalDemo1](http://docs.oracle.com/javase/tutorial/java/nutsandbolts/examples/ConditionalDemo1.java), tests these operators:

class ConditionalDemo1 {

public static void main(String[] args){

int value1 = 1;

int value2 = 2;

if((value1 == 1) && (value2 == 2))

System.out.println("value1 is 1 AND value2 is 2");

if((value1 == 1) || (value2 == 1))

System.out.println("value1 is 1 OR value2 is 1");

}

}

Another conditional operator is ?:, which can be thought of as shorthand for an if-then-else statement (discussed in the [Control Flow Statements](http://docs.oracle.com/javase/tutorial/java/nutsandbolts/flow.html) section of this lesson). This operator is also known as the *ternary operator* because it uses three operands. In the following example, this operator should be read as: "If someCondition is true, assign the value of value1 toresult. Otherwise, assign the value of value2 to result."

The following program, [ConditionalDemo2](http://docs.oracle.com/javase/tutorial/java/nutsandbolts/examples/ConditionalDemo2.java), tests the ?: operator:

class ConditionalDemo2 {

public static void main(String[] args){

int value1 = 1;

int value2 = 2;

int result;

boolean someCondition = true;

result = someCondition ? value1 : value2;

System.out.println(result);

}

}

Because someCondition is true, this program prints "1" to the screen. Use the ?: operator instead of an if-then-else statement if it makes your code more readable; for example, when the expressions are compact and without side-effects (such as assignments).

## The Type Comparison Operator instanceof

The instanceof operator compares an object to a specified type. You can use it to test if an object is an instance of a class, an instance of a subclass, or an instance of a class that implements a particular interface.

The following program, [InstanceofDemo](http://docs.oracle.com/javase/tutorial/java/nutsandbolts/examples/InstanceofDemo.java), defines a parent class (named Parent), a simple interface (named MyInterface), and a child class (named Child) that inherits from the parent and implements the interface.

class InstanceofDemo {

public static void main(String[] args) {

Parent obj1 = new Parent();

Parent obj2 = new Child();

System.out.println("obj1 instanceof Parent: "

+ (obj1 instanceof Parent));

System.out.println("obj1 instanceof Child: "

+ (obj1 instanceof Child));

System.out.println("obj1 instanceof MyInterface: "

+ (obj1 instanceof MyInterface));

System.out.println("obj2 instanceof Parent: "

+ (obj2 instanceof Parent));

System.out.println("obj2 instanceof Child: "

+ (obj2 instanceof Child));

System.out.println("obj2 instanceof MyInterface: "

+ (obj2 instanceof MyInterface));

}

}

class Parent {}

class Child extends Parent implements MyInterface {}

interface MyInterface {}

Output:

obj1 instanceof Parent: true

obj1 instanceof Child: false

obj1 instanceof MyInterface: false

obj2 instanceof Parent: true

obj2 instanceof Child: true

obj2 instanceof MyInterface: true

When using the instanceof operator, keep in mind that null is not an instance of anything.

# Bitwise and Bit Shift Operators

The Java programming language also provides operators that perform bitwise and bit shift operations on integral types. The operators discussed in this section are less commonly used. Therefore, their coverage is brief; the intent is to simply make you aware that these operators exist.

The unary bitwise complement operator "~" inverts a bit pattern; it can be applied to any of the integral types, making every "0" a "1" and every "1" a "0". For example, a byte contains 8 bits; applying this operator to a value whose bit pattern is "00000000" would change its pattern to "11111111".

The signed left shift operator "<<" shifts a bit pattern to the left, and the signed right shift operator ">>" shifts a bit pattern to the right. The bit pattern is given by the left-hand operand, and the number of positions to shift by the right-hand operand. The unsigned right shift operator ">>>" shifts a zero into the leftmost position, while the leftmost position after ">>" depends on sign extension.

The bitwise & operator performs a bitwise AND operation.

The bitwise ^ operator performs a bitwise exclusive OR operation.

The bitwise | operator performs a bitwise inclusive OR operation.

The following program, [BitDemo](http://docs.oracle.com/javase/tutorial/java/nutsandbolts/examples/BitDemo.java), uses the bitwise AND operator to print the number "2" to standard output.

class BitDemo {

public static void main(String[] args) {

int bitmask = 0x000F;

int val = 0x2222;

// prints "2"

System.out.println(val & bitmask);

}

}

# Control Flow Statements

The statements inside your source files are generally executed from top to bottom, in the order that they appear. *Control flow statements*, however, break up the flow of execution by employing decision making, looping, and branching, enabling your program to *conditionally* execute particular blocks of code. This section describes the decision-making statements (if-then, if-then-else, switch), the looping statements (for, while, do-while), and the branching statements (break, continue, return) supported by the Java programming language.

# The if-then and if-then-else Statements

## The if-then Statement

The if-then statement is the most basic of all the control flow statements. It tells your program to execute a certain section of code *only if* a particular test evaluates to true. For example, the Bicycle class could allow the brakes to decrease the bicycle's speed *only if* the bicycle is already in motion. One possible implementation of the applyBrakes method could be as follows:

void applyBrakes() {

// the "if" clause: bicycle must be moving

if (isMoving){

// the "then" clause: decrease current speed

currentSpeed--;

}

}

If this test evaluates to false (meaning that the bicycle is not in motion), control jumps to the end of the if-then statement.

In addition, the opening and closing braces are optional, provided that the "then" clause contains only one statement:

void applyBrakes() {

// same as above, but without braces

if (isMoving)

currentSpeed--;

}

Deciding when to omit the braces is a matter of personal taste. Omitting them can make the code more brittle. If a second statement is later added to the "then" clause, a common mistake would be forgetting to add the newly required braces. The compiler cannot catch this sort of error; you'll just get the wrong results.

## The if-then-else Statement

The if-then-else statement provides a secondary path of execution when an "if" clause evaluates to false. You could use an if-then-else statement in the applyBrakes method to take some action if the brakes are applied when the bicycle is not in motion. In this case, the action is to simply print an error message stating that the bicycle has already stopped.

void applyBrakes() {

if (isMoving) {

currentSpeed--;

} else {

System.err.println("The bicycle has already stopped!");

}

}

The following program, [IfElseDemo](http://docs.oracle.com/javase/tutorial/java/nutsandbolts/examples/IfElseDemo.java), assigns a grade based on the value of a test score: an A for a score of 90% or above, a B for a score of 80% or above, and so on.

class IfElseDemo {

public static void main(String[] args) {

int testscore = 76;

char grade;

if (testscore >= 90) {

grade = 'A';

} else if (testscore >= 80) {

grade = 'B';

} else if (testscore >= 70) {

grade = 'C';

} else if (testscore >= 60) {

grade = 'D';

} else {

grade = 'F';

}

System.out.println("Grade = " + grade);

}

}

The output from the program is:

Grade = C

You may have noticed that the value of testscore can satisfy more than one expression in the compound statement: 76 >= 70 and 76 >= 60. However, once a condition is satisfied, the appropriate statements are executed (grade = 'C';) and the remaining conditions are not evaluated.

# The switch Statement

Unlike if-then and if-then-else statements, the switch statement can have a number of possible execution paths. A switch works with the byte, short, char, and int primitive data types. It also works with *enumerated types* (discussed in [Enum Types](http://docs.oracle.com/javase/tutorial/java/javaOO/enum.html)), the [String](http://docs.oracle.com/javase/8/docs/api/java/lang/String.html) class, and a few special classes that wrap certain primitive types: [Character](http://docs.oracle.com/javase/8/docs/api/java/lang/Character.html), [Byte](http://docs.oracle.com/javase/8/docs/api/java/lang/Byte.html), [Short](http://docs.oracle.com/javase/8/docs/api/java/lang/Short.html), and[Integer](http://docs.oracle.com/javase/8/docs/api/java/lang/Integer.html) (discussed in [Numbers and Strings](http://docs.oracle.com/javase/tutorial/java/data/index.html)).

The following code example, [SwitchDemo](http://docs.oracle.com/javase/tutorial/java/nutsandbolts/examples/SwitchDemo.java), declares an int named month whose value represents a month. The code displays the name of the month, based on the value of month, using the switch statement.

public class SwitchDemo {

public static void main(String[] args) {

int month = 8;

String monthString;

switch (month) {

case 1: monthString = "January";

break;

case 2: monthString = "February";

break;

case 3: monthString = "March";

break;

case 4: monthString = "April";

break;

case 5: monthString = "May";

break;

case 6: monthString = "June";

break;

case 7: monthString = "July";

break;

case 8: monthString = "August";

break;

case 9: monthString = "September";

break;

case 10: monthString = "October";

break;

case 11: monthString = "November";

break;

case 12: monthString = "December";

break;

default: monthString = "Invalid month";

break;

}

System.out.println(monthString);

}

}

In this case, August is printed to standard output.

The body of a switch statement is known as a *switch block*. A statement in the switch block can be labeled with one or more case or default labels. The switch statement evaluates its expression, then executes all statements that follow the matching case label.

You could also display the name of the month with if-then-else statements:

int month = 8;

if (month == 1) {

System.out.println("January");

} else if (month == 2) {

System.out.println("February");

}

... // and so on

Deciding whether to use if-then-else statements or a switch statement is based on readability and the expression that the statement is testing. An if-then-else statement can test expressions based on ranges of values or conditions, whereas a switch statement tests expressions based only on a single integer, enumerated value, or String object.

Another point of interest is the break statement. Each break statement terminates the enclosing switch statement. Control flow continues with the first statement following the switchblock. The break statements are necessary because without them, statements in switch blocks fall through: All statements after the matching case label are executed in sequence, regardless of the expression of subsequent case labels, until a break statement is encountered. The program [SwitchDemoFallThrough](http://docs.oracle.com/javase/tutorial/java/nutsandbolts/examples/SwitchDemoFallThrough.java) shows statements in a switch block that fall through. The program displays the month corresponding to the integer month and the months that follow in the year:

public class SwitchDemoFallThrough {

public static void main(String[] args) {

java.util.ArrayList<String> futureMonths =

new java.util.ArrayList<String>();

int month = 8;

switch (month) {

case 1: futureMonths.add("January");

case 2: futureMonths.add("February");

case 3: futureMonths.add("March");

case 4: futureMonths.add("April");

case 5: futureMonths.add("May");

case 6: futureMonths.add("June");

case 7: futureMonths.add("July");

case 8: futureMonths.add("August");

case 9: futureMonths.add("September");

case 10: futureMonths.add("October");

case 11: futureMonths.add("November");

case 12: futureMonths.add("December");

break;

default: break;

}

if (futureMonths.isEmpty()) {

System.out.println("Invalid month number");

} else {

for (String monthName : futureMonths) {

System.out.println(monthName);

}

}

}

}

This is the output from the code:

August

September

October

November

December

Technically, the final break is not required because flow falls out of the switch statement. Using a break is recommended so that modifying the code is easier and less error prone. Thedefault section handles all values that are not explicitly handled by one of the case sections.

The following code example, [SwitchDemo2](http://docs.oracle.com/javase/tutorial/java/nutsandbolts/examples/SwitchDemo2.java), shows how a statement can have multiple case labels. The code example calculates the number of days in a particular month:

class SwitchDemo2 {

public static void main(String[] args) {

int month = 2;

int year = 2000;

int numDays = 0;

switch (month) {

case 1: case 3: case 5:

case 7: case 8: case 10:

case 12:

numDays = 31;

break;

case 4: case 6:

case 9: case 11:

numDays = 30;

break;

case 2:

if (((year % 4 == 0) &&

!(year % 100 == 0))

|| (year % 400 == 0))

numDays = 29;

else

numDays = 28;

break;

default:

System.out.println("Invalid month.");

break;

}

System.out.println("Number of Days = "

+ numDays);

}

}

This is the output from the code:

Number of Days = 29

## Using Strings in switch Statements

In Java SE 7 and later, you can use a String object in the switch statement's expression. The following code example, [StringSwitchDemo](http://docs.oracle.com/javase/tutorial/java/nutsandbolts/examples/StringSwitchDemo.java), displays the number of the month based on the value of the String named month:

public class StringSwitchDemo {

public static int getMonthNumber(String month) {

int monthNumber = 0;

if (month == null) {

return monthNumber;

}

switch (month.toLowerCase()) {

case "january":

monthNumber = 1;

break;

case "february":

monthNumber = 2;

break;

case "march":

monthNumber = 3;

break;

case "april":

monthNumber = 4;

break;

case "may":

monthNumber = 5;

break;

case "june":

monthNumber = 6;

break;

case "july":

monthNumber = 7;

break;

case "august":

monthNumber = 8;

break;

case "september":

monthNumber = 9;

break;

case "october":

monthNumber = 10;

break;

case "november":

monthNumber = 11;

break;

case "december":

monthNumber = 12;

break;

default:

monthNumber = 0;

break;

}

return monthNumber;

}

public static void main(String[] args) {

String month = "August";

int returnedMonthNumber =

StringSwitchDemo.getMonthNumber(month);

if (returnedMonthNumber == 0) {

System.out.println("Invalid month");

} else {

System.out.println(returnedMonthNumber);

}

}

}

The output from this code is 8.

The String in the switch expression is compared with the expressions associated with each case label as if the [String.equals](http://docs.oracle.com/javase/8/docs/api/java/lang/String.html#equals-java.lang.Object-) method were being used. In order for theStringSwitchDemo example to accept any month regardless of case, month is converted to lowercase (with the [toLowerCase](http://docs.oracle.com/javase/8/docs/api/java/lang/String.html#toLowerCase--) method), and all the strings associated with the case labels are in lowercase.

**Note**: This example checks if the expression in the switch statement is null. Ensure that the expression in any switch statement is not null to prevent a NullPointerException from being thrown.

# The while and do-while Statements

The while statement continually executes a block of statements while a particular condition is true. Its syntax can be expressed as:

while (expression) {

statement(s)

}

The while statement evaluates *expression*, which must return a boolean value. If the expression evaluates to true, the while statement executes the statement(s) in the while block. Thewhile statement continues testing the expression and executing its block until the expression evaluates to false. Using the while statement to print the values from 1 through 10 can be accomplished as in the following [WhileDemo](http://docs.oracle.com/javase/tutorial/java/nutsandbolts/examples/WhileDemo.java) program:

class WhileDemo {

public static void main(String[] args){

int count = 1;

while (count < 11) {

System.out.println("Count is: " + count);

count++;

}

}

}

You can implement an infinite loop using the while statement as follows:

while (true){

// your code goes here

}

The Java programming language also provides a do-while statement, which can be expressed as follows:

do {

statement(s)

} while (expression);

The difference between do-while and while is that do-while evaluates its expression at the bottom of the loop instead of the top. Therefore, the statements within the do block are always executed at least once, as shown in the following [DoWhileDemo](http://docs.oracle.com/javase/tutorial/java/nutsandbolts/examples/DoWhileDemo.java) program:

class DoWhileDemo {

public static void main(String[] args){

int count = 1;

do {

System.out.println("Count is: " + count);

count++;

} while (count < 11);

}

}

# The for Statement

The for statement provides a compact way to iterate over a range of values. Programmers often refer to it as the "for loop" because of the way in which it repeatedly loops until a particular condition is satisfied. The general form of the for statement can be expressed as follows:

for (initialization; termination;

increment) {

statement(s)

}

When using this version of the for statement, keep in mind that:

* The initialization expression initializes the loop; it's executed once, as the loop begins.
* When the termination expression evaluates to false, the loop terminates.
* The increment expression is invoked after each iteration through the loop; it is perfectly acceptable for this expression to increment *or* decrement a value.

The following program, [ForDemo](http://docs.oracle.com/javase/tutorial/java/nutsandbolts/examples/ForDemo.java), uses the general form of the for statement to print the numbers 1 through 10 to standard output:

class ForDemo {

public static void main(String[] args){

for(int i=1; i<11; i++){

System.out.println("Count is: " + i);

}

}

}

The output of this program is:

Count is: 1

Count is: 2

Count is: 3

Count is: 4

Count is: 5

Count is: 6

Count is: 7

Count is: 8

Count is: 9

Count is: 10

Notice how the code declares a variable within the initialization expression. The scope of this variable extends from its declaration to the end of the block governed by the for statement, so it can be used in the termination and increment expressions as well. If the variable that controls a for statement is not needed outside of the loop, it's best to declare the variable in the initialization expression. The names i, j, and k are often used to control for loops; declaring them within the initialization expression limits their life span and reduces errors.

The three expressions of the for loop are optional; an infinite loop can be created as follows:

// infinite loop

for ( ; ; ) {

// your code goes here

}

The for statement also has another form designed for iteration through [Collections](http://docs.oracle.com/javase/tutorial/collections/index.html) and [arrays](http://docs.oracle.com/javase/tutorial/java/nutsandbolts/arrays.html) This form is sometimes referred to as the *enhanced for* statement, and can be used to make your loops more compact and easy to read. To demonstrate, consider the following array, which holds the numbers 1 through 10:

int[] numbers = {1,2,3,4,5,6,7,8,9,10};

The following program, [EnhancedForDemo](http://docs.oracle.com/javase/tutorial/java/nutsandbolts/examples/EnhancedForDemo.java), uses the enhanced for to loop through the array:

class EnhancedForDemo {

public static void main(String[] args){

int[] numbers =

{1,2,3,4,5,6,7,8,9,10};

for (int item : numbers) {

System.out.println("Count is: " + item);

}

}

}

In this example, the variable item holds the current value from the numbers array. The output from this program is the same as before:

Count is: 1

Count is: 2

Count is: 3

Count is: 4

Count is: 5

Count is: 6

Count is: 7

Count is: 8

Count is: 9

Count is: 10

We recommend using this form of the for statement instead of the general form whenever possible.

# Branching Statements

## The break Statement

The break statement has two forms: labeled and unlabeled. You saw the unlabeled form in the previous discussion of the switch statement. You can also use an unlabeled break to terminate a for, while, or do-while loop, as shown in the following [BreakDemo](http://docs.oracle.com/javase/tutorial/java/nutsandbolts/examples/BreakDemo.java) program:

class BreakDemo {

public static void main(String[] args) {

int[] arrayOfInts =

{ 32, 87, 3, 589,

12, 1076, 2000,

8, 622, 127 };

int searchfor = 12;

int i;

boolean foundIt = false;

for (i = 0; i < arrayOfInts.length; i++) {

if (arrayOfInts[i] == searchfor) {

foundIt = true;

**break;**

}

}

if (foundIt) {

System.out.println("Found " + searchfor + " at index " + i);

} else {

System.out.println(searchfor + " not in the array");

}

}

}

This program searches for the number 12 in an array. The break statement, shown in boldface, terminates the for loop when that value is found. Control flow then transfers to the statement after the for loop. This program's output is:

Found 12 at index 4

An unlabeled break statement terminates the innermost switch, for, while, or do-while statement, but a labeled break terminates an outer statement. The following program,[BreakWithLabelDemo](http://docs.oracle.com/javase/tutorial/java/nutsandbolts/examples/BreakWithLabelDemo.java), is similar to the previous program, but uses nested for loops to search for a value in a two-dimensional array. When the value is found, a labeled break terminates the outer for loop (labeled "search"):

class BreakWithLabelDemo {

public static void main(String[] args) {

int[][] arrayOfInts = {

{ 32, 87, 3, 589 },

{ 12, 1076, 2000, 8 },

{ 622, 127, 77, 955 }

};

int searchfor = 12;

int i;

int j = 0;

boolean foundIt = false;

search:

for (i = 0; i < arrayOfInts.length; i++) {

for (j = 0; j < arrayOfInts[i].length;

j++) {

if (arrayOfInts[i][j] == searchfor) {

foundIt = true;

break search;

}

}

}

if (foundIt) {

System.out.println("Found " + searchfor + " at " + i + ", " + j);

} else {

System.out.println(searchfor + " not in the array");

}

}

}

This is the output of the program.

Found 12 at 1, 0

The break statement terminates the labeled statement; it does not transfer the flow of control to the label. Control flow is transferred to the statement immediately following the labeled (terminated) statement.

## The continue Statement

The continue statement skips the current iteration of a for, while , or do-while loop. The unlabeled form skips to the end of the innermost loop's body and evaluates the booleanexpression that controls the loop. The following program, [ContinueDemo](http://docs.oracle.com/javase/tutorial/java/nutsandbolts/examples/ContinueDemo.java) , steps through a String, counting the occurences of the letter "p". If the current character is not a p, the continuestatement skips the rest of the loop and proceeds to the next character. If it *is* a "p", the program increments the letter count.

class ContinueDemo {

public static void main(String[] args) {

String searchMe = "peter piper picked a " + "peck of pickled peppers";

int max = searchMe.length();

int numPs = 0;

for (int i = 0; i < max; i++) {

// interested only in p's

if (searchMe.charAt(i) != 'p')

continue;

// process p's

numPs++;

}

System.out.println("Found " + numPs + " p's in the string.");

}

}

Here is the output of this program:

Found 9 p's in the string.

To see this effect more clearly, try removing the continue statement and recompiling. When you run the program again, the count will be wrong, saying that it found 35 p's instead of 9.

A labeled continue statement skips the current iteration of an outer loop marked with the given label. The following example program, ContinueWithLabelDemo, uses nested loops to search for a substring within another string. Two nested loops are required: one to iterate over the substring and one to iterate over the string being searched. The following program,[ContinueWithLabelDemo](http://docs.oracle.com/javase/tutorial/java/nutsandbolts/examples/ContinueWithLabelDemo.java), uses the labeled form of continue to skip an iteration in the outer loop.

class ContinueWithLabelDemo {

public static void main(String[] args) {

String searchMe = "Look for a substring in me";

String substring = "sub";

boolean foundIt = false;

int max = searchMe.length() -

substring.length();

test:

for (int i = 0; i <= max; i++) {

int n = substring.length();

int j = i;

int k = 0;

while (n-- != 0) {

if (searchMe.charAt(j++) != substring.charAt(k++)) {

continue test;

}

}

foundIt = true;

break test;

}

System.out.println(foundIt ? "Found it" : "Didn't find it");

}

}

Here is the output from this program.

Found it

## The return Statement

The last of the branching statements is the return statement. The return statement exits from the current method, and control flow returns to where the method was invoked. The returnstatement has two forms: one that returns a value, and one that doesn't. To return a value, simply put the value (or an expression that calculates the value) after the return keyword.

return ++count;

The data type of the returned value must match the type of the method's declared return value. When a method is declared void, use the form of return that doesn't return a value.

return;

The [Classes and Objects](http://docs.oracle.com/javase/tutorial/java/javaOO/methods.html) lesson will cover everything you need to know about writing methods.

# Lesson: Classes and Objects

# Classes

The introduction to object-oriented concepts in the lesson titled [Object-oriented Programming Concepts](http://docs.oracle.com/javase/tutorial/java/concepts/index.html)used a bicycle class as an example, with racing bikes, mountain bikes, and tandem bikes as subclasses. Here is sample code for a possible implementation of a Bicycle class, to give you an overview of a class declaration. Subsequent sections of this lesson will back up and explain class declarations step by step. For the moment, don't concern yourself with the details.

public class Bicycle {

// **the Bicycle class has**

// **three *fields***

public int cadence;

public int gear;

public int speed;

// **the Bicycle class has**

// **one *constructor***

public Bicycle(int startCadence, int startSpeed, int startGear) {

gear = startGear;

cadence = startCadence;

speed = startSpeed;

}

// **the Bicycle class has**

// **four *methods***

public void setCadence(int newValue) {

cadence = newValue;

}

public void setGear(int newValue) {

gear = newValue;

}

public void applyBrake(int decrement) {

speed -= decrement;

}

public void speedUp(int increment) {

speed += increment;

}

}

A class declaration for a MountainBike class that is a subclass of Bicycle might look like this:

public class MountainBike extends Bicycle {

// **the MountainBike subclass has**

// **one *field***

public int seatHeight;

// **the MountainBike subclass has**

// **one *constructor***

public MountainBike(int startHeight, int startCadence,

int startSpeed, int startGear) {

super(startCadence, startSpeed, startGear);

seatHeight = startHeight;

}

// **the MountainBike subclass has**

// **one *method***

public void setHeight(int newValue) {

seatHeight = newValue;

}

}

MountainBike inherits all the fields and methods of Bicycle and adds the field seatHeight and a method to set it (mountain bikes have seats that can be moved up and down as the terrain demands).

# Declaring Member Variables

There are several kinds of variables:

* Member variables in a class—these are called *fields*.
* Variables in a method or block of code—these are called *local variables*.
* Variables in method declarations—these are called *parameters*.

The Bicycle class uses the following lines of code to define its fields:

public int cadence;

public int gear;

public int speed;

Field declarations are composed of three components, in order:

1. Zero or more modifiers, such as public or private.
2. The field's type.
3. The field's name.

The fields of Bicycle are named cadence, gear, and speed and are all of data type integer (int). The public keyword identifies these fields as public members, accessible by any object that can access the class.

## Access Modifiers

The first (left-most) modifier used lets you control what other classes have access to a member field. For the moment, consider only public and private. Other access modifiers will be discussed later.

* public modifier—the field is accessible from all classes.
* private modifier—the field is accessible only within its own class.

In the spirit of encapsulation, it is common to make fields private. This means that they can only be *directly* accessed from the Bicycle class. We still need access to these values, however. This can be done *indirectly* by adding public methods that obtain the field values for us:

public class Bicycle {

private int cadence;

private int gear;

private int speed;

public Bicycle(int startCadence, int startSpeed, int startGear) {

gear = startGear;

cadence = startCadence;

speed = startSpeed;

}

public int getCadence() {

return cadence;

}

public void setCadence(int newValue) {

cadence = newValue;

}

public int getGear() {

return gear;

}

public void setGear(int newValue) {

gear = newValue;

}

public int getSpeed() {

return speed;

}

public void applyBrake(int decrement) {

speed -= decrement;

}

public void speedUp(int increment) {

speed += increment;

}

}

## Types

All variables must have a type. You can use primitive types such as int, float, boolean, etc. Or you can use reference types, such as strings, arrays, or objects.

## Variable Names

All variables, whether they are fields, local variables, or parameters, follow the same naming rules and conventions that were covered in the Language Basics lesson, [Variables—Naming](http://docs.oracle.com/javase/tutorial/java/nutsandbolts/variables.html#naming).

In this lesson, be aware that the same naming rules and conventions are used for method and class names, except that

* the first letter of a class name should be capitalized, and
* the first (or only) word in a method name should be a verb.

# Defining Methods

Here is an example of a typical method declaration:

public double calculateAnswer(double wingSpan, int numberOfEngines,

double length, double grossTons) {

//do the calculation here

}

The only required elements of a method declaration are the method's return type, name, a pair of parentheses, (), and a body between braces, {}.

More generally, method declarations have six components, in order:

1. Modifiers—such as public, private, and others you will learn about later.
2. The return type—the data type of the value returned by the method, or void if the method does not return a value.
3. The method name—the rules for field names apply to method names as well, but the convention is a little different.
4. The parameter list in parenthesis—a comma-delimited list of input parameters, preceded by their data types, enclosed by parentheses, (). If there are no parameters, you must use empty parentheses.
5. An exception list—to be discussed later.
6. The method body, enclosed between braces—the method's code, including the declaration of local variables, goes here.

Modifiers, return types, and parameters will be discussed later in this lesson. Exceptions are discussed in a later lesson.

**Definition:** Two of the components of a method declaration comprise the *method signature*—the method's name and the parameter types.

The signature of the method declared above is:

calculateAnswer(double, int, double, double)

## Naming a Method

Although a method name can be any legal identifier, code conventions restrict method names. By convention, method names should be a verb in lowercase or a multi-word name that begins with a verb in lowercase, followed by adjectives, nouns, etc. In multi-word names, the first letter of each of the second and following words should be capitalized. Here are some examples:

run

runFast

getBackground

getFinalData

compareTo

setX

isEmpty

Typically, a method has a unique name within its class. However, a method might have the same name as other methods due to *method overloading*.

## Overloading Methods

The Java programming language supports *overloading* methods, and Java can distinguish between methods with different *method signatures*. This means that methods within a class can have the same name if they have different parameter lists (there are some qualifications to this that will be discussed in the lesson titled "Interfaces and Inheritance").

Suppose that you have a class that can use calligraphy to draw various types of data (strings, integers, and so on) and that contains a method for drawing each data type. It is cumbersome to use a new name for each method—for example, drawString, drawInteger, drawFloat, and so on. In the Java programming language, you can use the same name for all the drawing methods but pass a different argument list to each method. Thus, the data drawing class might declare four methods named draw, each of which has a different parameter list.

public class DataArtist {

...

public void draw(String s) {

...

}

public void draw(int i) {

...

}

public void draw(double f) {

...

}

public void draw(int i, double f) {

...

}

}

Overloaded methods are differentiated by the number and the type of the arguments passed into the method. In the code sample, draw(String s) and draw(int i) are distinct and unique methods because they require different argument types.

You cannot declare more than one method with the same name and the same number and type of arguments, because the compiler cannot tell them apart.

The compiler does not consider return type when differentiating methods, so you cannot declare two methods with the same signature even if they have a different return type.

**Note:** Overloaded methods should be used sparingly, as they can make code much less readable.

# Providing Constructors for Your Classes

A class contains constructors that are invoked to create objects from the class blueprint. Constructor declarations look like method declarations—except that they use the name of the class and have no return type. For example, Bicycle has one constructor:

public Bicycle(int startCadence, int startSpeed, int startGear) {

gear = startGear;

cadence = startCadence;

speed = startSpeed;

}

To create a new Bicycle object called myBike, a constructor is called by the new operator:

Bicycle myBike = new Bicycle(30, 0, 8);

new Bicycle(30, 0, 8) creates space in memory for the object and initializes its fields.

Although Bicycle only has one constructor, it could have others, including a no-argument constructor:

public Bicycle() {

gear = 1;

cadence = 10;

speed = 0;

}

Bicycle yourBike = new Bicycle(); invokes the no-argument constructor to create a new Bicycle object called yourBike.

Both constructors could have been declared in Bicycle because they have different argument lists. As with methods, the Java platform differentiates constructors on the basis of the number of arguments in the list and their types. You cannot write two constructors that have the same number and type of arguments for the same class, because the platform would not be able to tell them apart. Doing so causes a compile-time error.

You don't have to provide any constructors for your class, but you must be careful when doing this. The compiler automatically provides a no-argument, default constructor for any class without constructors. This default constructor will call the no-argument constructor of the superclass. In this situation, the compiler will complain if the superclass doesn't have a no-argument constructor so you must verify that it does. If your class has no explicit superclass, then it has an implicit superclass of Object, which *does* have a no-argument constructor.

You can use a superclass constructor yourself. The MountainBike class at the beginning of this lesson did just that. This will be discussed later, in the lesson on interfaces and inheritance.

You can use access modifiers in a constructor's declaration to control which other classes can call the constructor.

**Note:** If another class cannot call a MyClass constructor, it cannot directly create MyClass objects.

# Passing Information to a Method or a Constructor

The declaration for a method or a constructor declares the number and the type of the arguments for that method or constructor. For example, the following is a method that computes the monthly payments for a home loan, based on the amount of the loan, the interest rate, the length of the loan (the number of periods), and the future value of the loan:

public double computePayment(

double **loanAmt**,

double **rate**,

double **futureValue**,

int **numPeriods**) {

double interest = **rate** / 100.0;

double partial1 = Math.pow((1 + interest),

- **numPeriods**);

double denominator = (1 - partial1) / interest;

double answer = (-**loanAmt** / denominator)

- ((**futureValue** \* partial1) / denominator);

return answer;

}

This method has four parameters: the loan amount, the interest rate, the future value and the number of periods. The first three are double-precision floating point numbers, and the fourth is an integer. The parameters are used in the method body and at runtime will take on the values of the arguments that are passed in.

**Note:** *Parameters* refers to the list of variables in a method declaration. *Arguments* are the actual values that are passed in when the method is invoked. When you invoke a method, the arguments used must match the declaration's parameters in type and order.

## Parameter Types

You can use any data type for a parameter of a method or a constructor. This includes primitive data types, such as doubles, floats, and integers, as you saw in the computePayment method, and reference data types, such as objects and arrays.

Here's an example of a method that accepts an array as an argument. In this example, the method creates a new Polygon object and initializes it from an array of Point objects (assume thatPoint is a class that represents an x, y coordinate):

public Polygon polygonFrom(Point[] corners) {

// method body goes here

}

**Note:** If you want to pass a method into a method, then use a [lambda expression](http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html) or a [method reference](http://docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html).

## Arbitrary Number of Arguments

You can use a construct called varargs to pass an arbitrary number of values to a method. You use varargs when you don't know how many of a particular type of argument will be passed to the method. It's a shortcut to creating an array manually (the previous method could have used varargs rather than an array).

To use varargs, you follow the type of the last parameter by an ellipsis (three dots, ...), then a space, and the parameter name. The method can then be called with any number of that parameter, including none.

public Polygon polygonFrom(Point... corners) {

int numberOfSides = corners.length;

double squareOfSide1, lengthOfSide1;

squareOfSide1 = (corners[1].x - corners[0].x)

\* (corners[1].x - corners[0].x)

+ (corners[1].y - corners[0].y)

\* (corners[1].y - corners[0].y);

lengthOfSide1 = Math.sqrt(squareOfSide1);

// more method body code follows that creates and returns a

// polygon connecting the Points

}

You can see that, inside the method, corners is treated like an array. The method can be called either with an array or with a sequence of arguments. The code in the method body will treat the parameter as an array in either case.

You will most commonly see varargs with the printing methods; for example, this printf method:

public PrintStream printf(String format, Object... args)

allows you to print an arbitrary number of objects. It can be called like this:

System.out.printf("%s: %d, %s%n", name, idnum, address);

or like this

System.out.printf("%s: %d, %s, %s, %s%n", name, idnum, address, phone, email);

or with yet a different number of arguments.

## Parameter Names

When you declare a parameter to a method or a constructor, you provide a name for that parameter. This name is used within the method body to refer to the passed-in argument.

The name of a parameter must be unique in its scope. It cannot be the same as the name of another parameter for the same method or constructor, and it cannot be the name of a local variable within the method or constructor.

A parameter can have the same name as one of the class's fields. If this is the case, the parameter is said to *shadow* the field. Shadowing fields can make your code difficult to read and is conventionally used only within constructors and methods that set a particular field. For example, consider the following Circle class and its setOrigin method:

public class Circle {

private int x, y, radius;

public void setOrigin(int x, int y) {

...

}

}

The Circle class has three fields: x, y, and radius. The setOrigin method has two parameters, each of which has the same name as one of the fields. Each method parameter shadows the field that shares its name. So using the simple names x or y within the body of the method refers to the parameter, *not* to the field. To access the field, you must use a qualified name. This will be discussed later in this lesson in the section titled "Using the this Keyword."

## Passing Primitive Data Type Arguments

Primitive arguments, such as an int or a double, are passed into methods *by value*. This means that any changes to the values of the parameters exist only within the scope of the method. When the method returns, the parameters are gone and any changes to them are lost. Here is an example:

public class PassPrimitiveByValue {

public static void main(String[] args) {

int x = 3;

// invoke passMethod() with

// x as argument

passMethod(x);

// print x to see if its

// value has changed

System.out.println("After invoking passMethod, x = " + x);

}

// change parameter in passMethod()

public static void passMethod(int p) {

p = 10;

}

}

When you run this program, the output is:

After invoking passMethod, x = 3

## Passing Reference Data Type Arguments

Reference data type parameters, such as objects, are also passed into methods *by value*. This means that when the method returns, the passed-in reference still references the same object as before. *However*, the values of the object's fields *can* be changed in the method, if they have the proper access level.

For example, consider a method in an arbitrary class that moves Circle objects:

public void moveCircle(Circle circle, int deltaX, int deltaY) {

// code to move origin of circle to x+deltaX, y+deltaY

circle.setX(circle.getX() + deltaX);

circle.setY(circle.getY() + deltaY);

// code to assign a new reference to circle

circle = new Circle(0, 0);

}

Let the method be invoked with these arguments:

moveCircle(myCircle, 23, 56)

Inside the method, circle initially refers to myCircle. The method changes the x and y coordinates of the object that circle references (i.e., myCircle) by 23 and 56, respectively. These changes will persist when the method returns. Then circle is assigned a reference to a new Circle object with x = y = 0. This reassignment has no permanence, however, because the reference was passed in by value and cannot change. Within the method, the object pointed to by circle has changed, but, when the method returns, myCircle still references the sameCircle object as before the method was called.

# Objects

A typical Java program creates many objects, which as you know, interact by invoking methods. Through these object interactions, a program can carry out various tasks, such as implementing a GUI, running an animation, or sending and receiving information over a network. Once an object has completed the work for which it was created, its resources are recycled for use by other objects.

Here's a small program, called [CreateObjectDemo](http://docs.oracle.com/javase/tutorial/java/javaOO/examples/CreateObjectDemo.java), that creates three objects: one [Point](http://docs.oracle.com/javase/tutorial/java/javaOO/examples/Point.java) object and two [Rectangle](http://docs.oracle.com/javase/tutorial/java/javaOO/examples/Rectangle.java) objects. You will need all three source files to compile this program.

public class CreateObjectDemo {

public static void main(String[] args) {

// Declare and create a point object and two rectangle objects.

Point originOne = new Point(23, 94);

Rectangle rectOne = new Rectangle(originOne, 100, 200);

Rectangle rectTwo = new Rectangle(50, 100);

// display rectOne's width, height, and area

System.out.println("Width of rectOne: " + rectOne.width);

System.out.println("Height of rectOne: " + rectOne.height);

System.out.println("Area of rectOne: " + rectOne.getArea());

// set rectTwo's position

rectTwo.origin = originOne;

// display rectTwo's position

System.out.println("X Position of rectTwo: " + rectTwo.origin.x);

System.out.println("Y Position of rectTwo: " + rectTwo.origin.y);

// move rectTwo and display its new position

rectTwo.move(40, 72);

System.out.println("X Position of rectTwo: " + rectTwo.origin.x);

System.out.println("Y Position of rectTwo: " + rectTwo.origin.y);

}

}

This program creates, manipulates, and displays information about various objects. Here's the output:

Width of rectOne: 100

Height of rectOne: 200

Area of rectOne: 20000

X Position of rectTwo: 23

Y Position of rectTwo: 94

X Position of rectTwo: 40

Y Position of rectTwo: 72

The following three sections use the above example to describe the life cycle of an object within a program. From them, you will learn how to write code that creates and uses objects in your own programs. You will also learn how the system cleans up after an object when its life has ended.

# Creating Objects

As you know, a class provides the blueprint for objects; you create an object from a class. Each of the following statements taken from the [CreateObjectDemo](http://docs.oracle.com/javase/tutorial/java/javaOO/examples/CreateObjectDemo.java) program creates an object and assigns it to a variable:

**Point originOne** = new Point(23, 94);

**Rectangle rectOne** = new Rectangle(originOne, 100, 200);

**Rectangle rectTwo** = new Rectangle(50, 100);

The first line creates an object of the [Point](http://docs.oracle.com/javase/tutorial/java/javaOO/examples/Point.java) class, and the second and third lines each create an object of the [Rectangle](http://docs.oracle.com/javase/tutorial/java/javaOO/examples/Rectangle.java) class.

Each of these statements has three parts (discussed in detail below):

1. **Declaration**: The code set in **bold** are all variable declarations that associate a variable name with an object type.
2. **Instantiation**: The new keyword is a Java operator that creates the object.
3. **Initialization**: The new operator is followed by a call to a constructor, which initializes the new object.

## Declaring a Variable to Refer to an Object

Previously, you learned that to declare a variable, you write:

type name;

This notifies the compiler that you will use name to refer to data whose type is type. With a primitive variable, this declaration also reserves the proper amount of memory for the variable.

You can also declare a reference variable on its own line. For example:

Point originOne;

If you declare originOne like this, its value will be undetermined until an object is actually created and assigned to it. Simply declaring a reference variable does not create an object. For that, you need to use the new operator, as described in the next section. You must assign an object to originOne before you use it in your code. Otherwise, you will get a compiler error.

A variable in this state, which currently references no object, can be illustrated as follows (the variable name, originOne, plus a reference pointing to nothing):



## Instantiating a Class

The new operator instantiates a class by allocating memory for a new object and returning a reference to that memory. The new operator also invokes the object constructor.

**Note:** The phrase "instantiating a class" means the same thing as "creating an object." When you create an object, you are creating an "instance" of a class, therefore "instantiating" a class.

The new operator requires a single, postfix argument: a call to a constructor. The name of the constructor provides the name of the class to instantiate.

The new operator returns a reference to the object it created. This reference is usually assigned to a variable of the appropriate type, like:

Point originOne = new Point(23, 94);

The reference returned by the new operator does not have to be assigned to a variable. It can also be used directly in an expression. For example:

int height = new Rectangle().height;

This statement will be discussed in the next section.

## Initializing an Object

Here's the code for the Point class:

public class Point {

public int x = 0;

public int y = 0;

**//constructor**

**public Point(int a, int b) {**

**x = a;**

**y = b;**

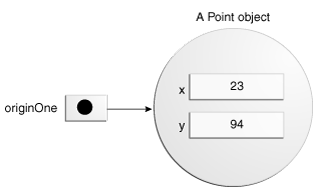
**}**

}

This class contains a single constructor. You can recognize a constructor because its declaration uses the same name as the class and it has no return type. The constructor in the Point class takes two integer arguments, as declared by the code (int a, int b). The following statement provides 23 and 94 as values for those arguments:

Point originOne = new Point(23, 94);

The result of executing this statement can be illustrated in the next figure:



Here's the code for the Rectangle class, which contains four constructors:

public class Rectangle {

public int width = 0;

public int height = 0;

public Point origin;

// four constructors

public Rectangle() {

origin = new Point(0, 0);

}

public Rectangle(Point p) {

origin = p;

}

public Rectangle(int w, int h) {

origin = new Point(0, 0);

width = w;

height = h;

}

public Rectangle(Point p, int w, int h) {

origin = p;

width = w;

height = h;

}

// a method for moving the rectangle

public void move(int x, int y) {

origin.x = x;

origin.y = y;

}

// a method for computing the area of the rectangle

public int getArea() {

return width \* height;

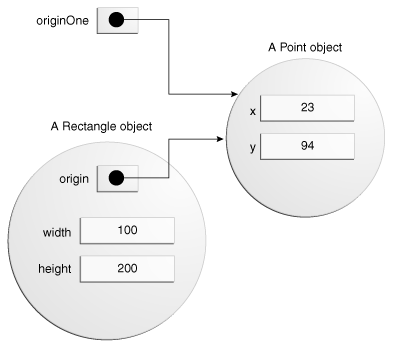
}

}

Each constructor lets you provide initial values for the rectangle's origin, width, and height, using both primitive and reference types. If a class has multiple constructors, they must have different signatures. The Java compiler differentiates the constructors based on the number and the type of the arguments. When the Java compiler encounters the following code, it knows to call the constructor in the Rectangle class that requires a Point argument followed by two integer arguments:

Rectangle rectOne = new Rectangle(originOne, 100, 200);

This calls one of Rectangle's constructors that initializes origin to originOne. Also, the constructor sets width to 100 and height to 200. Now there are two references to the samePoint object—an object can have multiple references to it, as shown in the next figure:



The following line of code calls the Rectangle constructor that requires two integer arguments, which provide the initial values for width and height. If you inspect the code within the constructor, you will see that it creates a new Point object whose x and y values are initialized to 0:

Rectangle rectTwo = new Rectangle(50, 100);

The Rectangle constructor used in the following statement doesn't take any arguments, so it's called a no-argument constructor:

Rectangle rect = new Rectangle();

All classes have at least one constructor. If a class does not explicitly declare any, the Java compiler automatically provides a no-argument constructor, called the default constructor. This default constructor calls the class parent's no-argument constructor, or the Object constructor if the class has no other parent. If the parent has no constructor (Object does have one), the compiler will reject the program.

# Using Objects

Once you've created an object, you probably want to use it for something. You may need to use the value of one of its fields, change one of its fields, or call one of its methods to perform an action.

## Referencing an Object's Fields

Object fields are accessed by their name. You must use a name that is unambiguous.

You may use a simple name for a field within its own class. For example, we can add a statement *within* the Rectangle class that prints the width and height:

System.out.println("Width and height are: " + width + ", " + height);

In this case, width and height are simple names.

Code that is outside the object's class must use an object reference or expression, followed by the dot (.) operator, followed by a simple field name, as in:

objectReference.fieldName

For example, the code in the CreateObjectDemo class is outside the code for the Rectangle class. So to refer to the origin, width, and height fields within the Rectangle object named rectOne, theCreateObjectDemo class must use the names rectOne.origin, rectOne.width, and rectOne.height, respectively. The program uses two of these names to display the width and the height of rectOne:

System.out.println("Width of rectOne: " + rectOne.width);

System.out.println("Height of rectOne: " + rectOne.height);

Attempting to use the simple names width and height from the code in the CreateObjectDemo class doesn't make sense — those fields exist only within an object — and results in a compiler error.

Later, the program uses similar code to display information about rectTwo. Objects of the same type have their own copy of the same instance fields. Thus, each Rectangle object has fields named origin, width, and height. When you access an instance field through an object reference, you reference that particular object's field. The two objects rectOne and rectTwo in theCreateObjectDemo program have different origin, width, and height fields.

To access a field, you can use a named reference to an object, as in the previous examples, or you can use any expression that returns an object reference. Recall that the new operator returns a reference to an object. So you could use the value returned from new to access a new object's fields:

int height = new Rectangle().height;

This statement creates a new Rectangle object and immediately gets its height. In essence, the statement calculates the default height of a Rectangle. Note that after this statement has been executed, the program no longer has a reference to the created Rectangle, because the program never stored the reference anywhere. The object is unreferenced, and its resources are free to be recycled by the Java Virtual Machine.

## Calling an Object's Methods

You also use an object reference to invoke an object's method. You append the method's simple name to the object reference, with an intervening dot operator (.). Also, you provide, within enclosing parentheses, any arguments to the method. If the method does not require any arguments, use empty parentheses.

objectReference.methodName(argumentList);

or:

objectReference.methodName();

The Rectangle class has two methods: getArea() to compute the rectangle's area and move() to change the rectangle's origin. Here's the CreateObjectDemo code that invokes these two methods:

System.out.println("Area of rectOne: " + rectOne.getArea());

...

rectTwo.move(40, 72);

The first statement invokes rectOne's getArea() method and displays the results. The second line moves rectTwo because the move() method assigns new values to the object's origin.x andorigin.y.

As with instance fields, objectReference must be a reference to an object. You can use a variable name, but you also can use any expression that returns an object reference. The new operator returns an object reference, so you can use the value returned from new to invoke a new object's methods:

new Rectangle(100, 50).getArea()

The expression new Rectangle(100, 50) returns an object reference that refers to a Rectangle object. As shown, you can use the dot notation to invoke the new Rectangle's getArea() method to compute the area of the new rectangle.

Some methods, such as getArea(), return a value. For methods that return a value, you can use the method invocation in expressions. You can assign the return value to a variable, use it to make decisions, or control a loop. This code assigns the value returned by getArea() to the variable areaOfRectangle:

int areaOfRectangle = new Rectangle(100, 50).getArea();

Remember, invoking a method on a particular object is the same as sending a message to that object. In this case, the object that getArea() is invoked on is the rectangle returned by the constructor.

## The Garbage Collector

Some object-oriented languages require that you keep track of all the objects you create and that you explicitly destroy them when they are no longer needed. Managing memory explicitly is tedious and error-prone. The Java platform allows you to create as many objects as you want (limited, of course, by what your system can handle), and you don't have to worry about destroying them. The Java runtime environment deletes objects when it determines that they are no longer being used. This process is called garbage collection.

An object is eligible for garbage collection when there are no more references to that object. References that are held in a variable are usually dropped when the variable goes out of scope. Or, you can explicitly drop an object reference by setting the variable to the special value null. Remember that a program can have multiple references to the same object; all references to an object must be dropped before the object is eligible for garbage collection.

The Java runtime environment has a garbage collector that periodically frees the memory used by objects that are no longer referenced. The garbage collector does its job automatically when it determines that the time is right.

# Returning a Value from a Method

A method returns to the code that invoked it when it

* completes all the statements in the method,
* reaches a return statement, or
* throws an exception (covered later),

whichever occurs first.

You declare a method's return type in its method declaration. Within the body of the method, you use the return statement to return the value.

Any method declared void doesn't return a value. It does not need to contain a return statement, but it may do so. In such a case, a return statement can be used to branch out of a control flow block and exit the method and is simply used like this:

return;

If you try to return a value from a method that is declared void, you will get a compiler error.

Any method that is not declared void must contain a return statement with a corresponding return value, like this:

return returnValue;

The data type of the return value must match the method's declared return type; you can't return an integer value from a method declared to return a boolean.

The getArea() method in the Rectangle [Rectangle](http://docs.oracle.com/javase/tutorial/java/javaOO/examples/Rectangle.java) class that was discussed in the sections on objects returns an integer:

// a method for computing the area of the rectangle

public int getArea() {

return width \* height;

}

This method returns the integer that the expression width\*height evaluates to.

The getArea method returns a primitive type. A method can also return a reference type. For example, in a program to manipulate Bicycle objects, we might have a method like this:

public Bicycle seeWhosFastest(Bicycle myBike, Bicycle yourBike,

Environment env) {

Bicycle fastest;

// code to calculate which bike is

// faster, given each bike's gear

// and cadence and given the

// environment (terrain and wind)

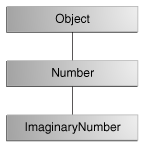
return fastest;

}

## Returning a Class or Interface

If this section confuses you, skip it and return to it after you have finished the lesson on interfaces and inheritance.

When a method uses a class name as its return type, such as whosFastest does, the class of the type of the returned object must be either a subclass of, or the exact class of, the return type. Suppose that you have a class hierarchy in which ImaginaryNumber is a subclass of java.lang.Number, which is in turn a subclass of Object, as illustrated in the following figure.



The class hierarchy for ImaginaryNumber

Now suppose that you have a method declared to return a Number:

public Number returnANumber() {

...

}

The returnANumber method can return an ImaginaryNumber but not an Object. ImaginaryNumber is a Number because it's a subclass of Number. However, an Object is not necessarily a Number — it could be a String or another type.

You can override a method and define it to return a subclass of the original method, like this:

public ImaginaryNumber returnANumber() {

...

}

This technique, called covariant return type, means that the return type is allowed to vary in the same direction as the subclass.

**Note:** You also can use interface names as return types. In this case, the object returned must implement the specified interface.

# Controlling Access to Members of a Class

Access level modifiers determine whether other classes can use a particular field or invoke a particular method. There are two levels of access control:

* At the top level—public, or *package-private* (no explicit modifier).
* At the member level—public, private, protected, or *package-private* (no explicit modifier).

A class may be declared with the modifier public, in which case that class is visible to all classes everywhere. If a class has no modifier (the default, also known as *package-private*), it is visible only within its own package (packages are named groups of related classes — you will learn about them in a later lesson.)

At the member level, you can also use the public modifier or no modifier (*package-private*) just as with top-level classes, and with the same meaning. For members, there are two additional access modifiers: private and protected. The private modifier specifies that the member can only be accessed in its own class. The protected modifier specifies that the member can only be accessed within its own package (as with *package-private*) and, in addition, by a subclass of its class in another package.

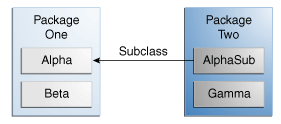
The following table shows the access to members permitted by each modifier.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Access Levels** | | | | |
| **Modifier** | **Class** | **Package** | **Subclass** | **World** |
| public | Y | Y | Y | Y |
| protected | Y | Y | Y | N |
| *no modifier* | Y | Y | N | N |
| private | Y | N | N | N |

The first data column indicates whether the class itself has access to the member defined by the access level. As you can see, a class always has access to its own members. The second column indicates whether classes in the same package as the class (regardless of their parentage) have access to the member. The third column indicates whether subclasses of the class declared outside this package have access to the member. The fourth column indicates whether all classes have access to the member.

Access levels affect you in two ways. First, when you use classes that come from another source, such as the classes in the Java platform, access levels determine which members of those classes your own classes can use. Second, when you write a class, you need to decide what access level every member variable and every method in your class should have.

Let's look at a collection of classes and see how access levels affect visibility. The following figure shows the four classes in this example and how they are related.



Classes and Packages of the Example Used to Illustrate Access Levels

The following table shows where the members of the Alpha class are visible for each of the access modifiers that can be applied to them.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Visibility** | | | | |
| **Modifier** | **Alpha** | **Beta** | **Alphasub** | **Gamma** |
| public | Y | Y | Y | Y |
| protected | Y | Y | Y | N |
| *no modifier* | Y | Y | N | N |
| private | Y | N | N | N |

**Tips on Choosing an Access Level:**

If other programmers use your class, you want to ensure that errors from misuse cannot happen. Access levels can help you do this.

* Use the most restrictive access level that makes sense for a particular member. Use private unless you have a good reason not to.

## Avoid public fields except for constants. (Many of the examples in the tutorial use public fields. This may help to illustrate some points concisely, but is not recoClass Variables

When a number of objects are created from the same class blueprint, they each have their own distinct copies of *instance variables*. In the case of the Bicycle class, the instance variables arecadence, gear, and speed. Each Bicycle object has its own values for these variables, stored in different memory locations.

Sometimes, you want to have variables that are common to all objects. This is accomplished with the static modifier. Fields that have the static modifier in their declaration are called*static fields* or *class variables*. They are associated with the class, rather than with any object. Every instance of the class shares a class variable, which is in one fixed location in memory. Any object can change the value of a class variable, but class variables can also be manipulated without creating an instance of the class.

For example, suppose you want to create a number of Bicycle objects and assign each a serial number, beginning with 1 for the first object. This ID number is unique to each object and is therefore an instance variable. At the same time, you need a field to keep track of how many Bicycle objects have been created so that you know what ID to assign to the next one. Such a field is not related to any individual object, but to the class as a whole. For this you need a class variable, numberOfBicycles, as follows:

public class Bicycle {

private int cadence;

private int gear;

private int speed;

// **add an instance variable for the object ID**

private int id;

// **add a class variable for the**

// **number of Bicycle objects instantiated**

private **static** int numberOfBicycles = 0;

...

}

Class variables are referenced by the class name itself, as in

Bicycle.numberOfBicycles

This makes it clear that they are class variables.

**Note:** You can also refer to static fields with an object reference like

myBike.numberOfBicycles

but this is discouraged because it does not make it clear that they are class variables.

You can use the Bicycle constructor to set the id instance variable and increment the numberOfBicycles class variable:

public class Bicycle {

private int cadence;

private int gear;

private int speed;

private int id;

private static int numberOfBicycles = 0;

public Bicycle(int startCadence, int startSpeed, int startGear){

gear = startGear;

cadence = startCadence;

speed = startSpeed;

// **increment number of Bicycles**

// **and assign ID number**

**id = ++numberOfBicycles;**

}

// **new method to return the ID instance variable**

public int getID() {

return id;

}

...

}

**Class Methods**

The Java programming language supports static methods as well as static variables. Static methods, which have the static modifier in their declarations, should be invoked with the class name, without the need for creating an instance of the class, as in

ClassName.methodName(args)

**Note:** You can also refer to static methods with an object reference like

instanceName.methodName(args)

but this is discouraged because it does not make it clear that they are class methods.

A common use for static methods is to access static fields. For example, we could add a static method to the Bicycle class to access the numberOfBicycles static field:

public **static** int getNumberOfBicycles() {

return numberOfBicycles;

}

Not all combinations of instance and class variables and methods are allowed:

* Instance methods can access instance variables and instance methods directly.
* Instance methods can access class variables and class methods directly.
* Class methods can access class variables and class methods directly.
* Class methods ***cannot*** access instance variables or instance methods directly—they must use an object reference. Also, class methods cannot use the this keyword as there is no instance for this to refer to.

**Constants**

The static modifier, in combination with the final modifier, is also used to define constants. The final modifier indicates that the value of this field cannot change.

For example, the following variable declaration defines a constant named PI, whose value is an approximation of pi (the ratio of the circumference of a circle to its diameter):

static final double PI = 3.141592653589793;

Constants defined in this way cannot be reassigned, and it is a compile-time error if your program tries to do so. By convention, the names of constant values are spelled in uppercase letters. If the name is composed of more than one word, the words are separated by an underscore (\_).

**Note:** If a primitive type or a string is defined as a constant and the value is known at compile time, the compiler replaces the constant name everywhere in the code with its value. This is called a *compile-time constant*. If the value of the constant in the outside world changes (for example, if it is legislated that pi actually should be 3.975), you will need to recompile any classes that use this constant to get the current value.

**The Bicycle Class**

After all the modifications made in this section, the Bicycle class is now:

public class Bicycle {

private int cadence;

private int gear;

private int speed;

private int id;

private **static** int numberOfBicycles = 0;

public Bicycle(int startCadence,

int startSpeed,

int startGear){

gear = startGear;

cadence = startCadence;

speed = startSpeed;

id = ++numberOfBicycles;

}

public int getID() {

return id;

}

public static int getNumberOfBicycles() {

return numberOfBicycles;

}

public int getCadence(){

return cadence;

}

public void setCadence(int newValue){

cadence = newValue;

}

public int getGear(){

return gear;

}

public void setGear(int newValue){

gear = newValue;

}

public int getSpeed(){

return speed;

}

public void applyBrake(int decrement){

speed -= decrement;

}

public void speedUp(int increment){

speed += increment;

}

}

* mmended for production code.) Public fields tend to link you to a particular implementation and limit your flexibility in changing your code.

# Initializing Fields

As you have seen, you can often provide an initial value for a field in its declaration:

public class BedAndBreakfast {

// initialize to 10

public static int capacity = 10;

// initialize to false

private boolean full = false;

}

This works well when the initialization value is available and the initialization can be put on one line. However, this form of initialization has limitations because of its simplicity. If initialization requires some logic (for example, error handling or a for loop to fill a complex array), simple assignment is inadequate. Instance variables can be initialized in constructors, where error handling or other logic can be used. To provide the same capability for class variables, the Java programming language includes *static initialization blocks*.

**Note:** It is not necessary to declare fields at the beginning of the class definition, although this is the most common practice. It is only necessary that they be declared and initialized before they are used.

## Static Initialization Blocks

A *static initialization block* is a normal block of code enclosed in braces, { }, and preceded by the static keyword. Here is an example:

static {

// whatever code is needed for initialization goes here

}

A class can have any number of static initialization blocks, and they can appear anywhere in the class body. The runtime system guarantees that static initialization blocks are called in the order that they appear in the source code.

There is an alternative to static blocks — you can write a private static method:

class Whatever {

public static varType myVar = initializeClassVariable();

private static varType initializeClassVariable() {

// initialization code goes here

}

}

The advantage of private static methods is that they can be reused later if you need to reinitialize the class variable.

## Initializing Instance Members

Normally, you would put code to initialize an instance variable in a constructor. There are two alternatives to using a constructor to initialize instance variables: initializer blocks and final methods.

Initializer blocks for instance variables look just like static initializer blocks, but without the static keyword:

{

// whatever code is needed for initialization goes here

}

The Java compiler copies initializer blocks into every constructor. Therefore, this approach can be used to share a block of code between multiple constructors.

A *final method* cannot be overridden in a subclass. This is discussed in the lesson on interfaces and inheritance. Here is an example of using a final method for initializing an instance variable:

class Whatever {

private varType myVar = initializeInstanceVariable();

protected final varType initializeInstanceVariable() {

// initialization code goes here

}

}

This is especially useful if subclasses might want to reuse the initialization method. The method is final because calling non-final methods during instance initialization can cause problems.

# Nested Classes

The Java programming language allows you to define a class within another class. Such a class is called a *nested class* and is illustrated here:

class OuterClass {

...

class NestedClass {

...

}

}

**Terminology:** Nested classes are divided into two categories: static and non-static. Nested classes that are declared static are called *static nested classes*. Non-static nested classes are called *inner classes*.

class OuterClass {

...

static class StaticNestedClass {

...

}

class InnerClass {

...

}

}

A nested class is a member of its enclosing class. Non-static nested classes (inner classes) have access to other members of the enclosing class, even if they are declared private. Static nested classes do not have access to other members of the enclosing class. As a member of the OuterClass, a nested class can be declared private, public, protected, or *package private*. (Recall that outer classes can only be declared public or *package private*.)

## Why Use Nested Classes?

Compelling reasons for using nested classes include the following:

* **It is a way of logically grouping classes that are only used in one place**: If a class is useful to only one other class, then it is logical to embed it in that class and keep the two together. Nesting such "helper classes" makes their package more streamlined.
* **It increases encapsulation**: Consider two top-level classes, A and B, where B needs access to members of A that would otherwise be declared private. By hiding class B within class A, A's members can be declared private and B can access them. In addition, B itself can be hidden from the outside world.
* **It can lead to more readable and maintainable code**: Nesting small classes within top-level classes places the code closer to where it is used.

## Static Nested Classes

As with class methods and variables, a static nested class is associated with its outer class. And like static class methods, a static nested class cannot refer directly to instance variables or methods defined in its enclosing class: it can use them only through an object reference.

**Note:** A static nested class interacts with the instance members of its outer class (and other classes) just like any other top-level class. In effect, a static nested class is behaviorally a top-level class that has been nested in another top-level class for packaging convenience.

Static nested classes are accessed using the enclosing class name:

OuterClass.StaticNestedClass

For example, to create an object for the static nested class, use this syntax:

OuterClass.StaticNestedClass nestedObject =

new OuterClass.StaticNestedClass();

## Inner Classes

As with instance methods and variables, an inner class is associated with an instance of its enclosing class and has direct access to that object's methods and fields. Also, because an inner class is associated with an instance, it cannot define any static members itself.

Objects that are instances of an inner class exist *within* an instance of the outer class. Consider the following classes:

class OuterClass {

...

class InnerClass {

...

}

}

An instance of InnerClass can exist only within an instance of OuterClass and has direct access to the methods and fields of its enclosing instance.

To instantiate an inner class, you must first instantiate the outer class. Then, create the inner object within the outer object with this syntax:

OuterClass.InnerClass innerObject = outerObject.new InnerClass();

There are two special kinds of inner classes: [local classes](http://docs.oracle.com/javase/tutorial/java/javaOO/localclasses.html) and [anonymous classes](http://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html).

## Shadowing

If a declaration of a type (such as a member variable or a parameter name) in a particular scope (such as an inner class or a method definition) has the same name as another declaration in the enclosing scope, then the declaration shadows the declaration of the enclosing scope. You cannot refer to a shadowed declaration by its name alone. The following example, [ShadowTest](http://docs.oracle.com/javase/tutorial/java/javaOO/examples/ShadowTest.java), demonstrates this:

public class ShadowTest {

public int x = 0;

class FirstLevel {

public int x = 1;

void methodInFirstLevel(int x) {

System.out.println("x = " + x);

System.out.println("this.x = " + this.x);

System.out.println("ShadowTest.this.x = " + ShadowTest.this.x);

}

}

public static void main(String... args) {

ShadowTest st = new ShadowTest();

ShadowTest.FirstLevel fl = st.new FirstLevel();

fl.methodInFirstLevel(23);

}

}

The following is the output of this example:

x = 23

this.x = 1

ShadowTest.this.x = 0

This example defines three variables named x: the member variable of the class ShadowTest, the member variable of the inner class FirstLevel, and the parameter in the methodmethodInFirstLevel. The variable x defined as a parameter of the method methodInFirstLevel shadows the variable of the inner class FirstLevel. Consequently, when you use the variable x in the method methodInFirstLevel, it refers to the method parameter. To refer to the member variable of the inner class FirstLevel, use the keyword this to represent the enclosing scope:

System.out.println("this.x = " + this.x);

Refer to member variables that enclose larger scopes by the class name to which they belong. For example, the following statement accesses the member variable of the class ShadowTestfrom the method methodInFirstLevel:

System.out.println("ShadowTest.this.x = " + ShadowTest.this.x);

## Serialization

[Serialization](http://docs.oracle.com/javase/tutorial/jndi/objects/serial.html) of inner classes, including [local](http://docs.oracle.com/javase/tutorial/java/javaOO/localclasses.html) and [anonymous](http://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html) classes, is strongly discouraged. When the Java compiler compiles certain constructs, such as inner classes, it creates synthetic constructs; these are classes, methods, fields, and other constructs that do not have a corresponding construct in the source code. Synthetic constructs enable Java compilers to implement new Java language features without changes to the JVM. However, synthetic constructs can vary among different Java compiler implementations, which means that .class files can vary among different implementations as well. Consequently, you may have compatibility issues if you serialize an inner class and then deserialize it with a different JRE implementation. See the section[Implicit and Synthetic Parameters](http://docs.oracle.com/javase/tutorial/reflect/member/methodparameterreflection.html#implcit_and_synthetic) in the section [Obtaining Names of Method Parameters](http://docs.oracle.com/javase/tutorial/reflect/member/methodparameterreflection.html) for more information about the synthetic constructs generated when an inner class is compiled.

# Inner Class Example

To see an inner class in use, first consider an array. In the following example, you create an array, fill it with integer values, and then output only values of even indices of the array in ascending order.

The [DataStructure.java](http://docs.oracle.com/javase/tutorial/java/javaOO/examples/DataStructure.java) example that follows consists of:

* The DataStructure outer class, which includes a constructor to create an instance of DataStructure containing an array filled with consecutive integer values (0, 1, 2, 3, and so on) and a method that prints elements of the array that have an even index value.
* The EvenIterator inner class, which implements the DataStructureIterator interface, which extends the [Iterator](http://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html)< [Integer](http://docs.oracle.com/javase/8/docs/api/java/lang/Integer.html)> interface. Iterators are used to step through a data structure and typically have methods to test for the last element, retrieve the current element, and move to the next element.
* A main method that instantiates a DataStructure object (ds), then invokes the printEven method to print elements of the array arrayOfInts that have an even index value.

public class DataStructure {

// Create an array

private final static int SIZE = 15;

private int[] arrayOfInts = new int[SIZE];

public DataStructure() {

// fill the array with ascending integer values

for (int i = 0; i < SIZE; i++) {

arrayOfInts[i] = i;

}

}

public void printEven() {

// Print out values of even indices of the array

DataStructureIterator iterator = this.new EvenIterator();

while (iterator.hasNext()) {

System.out.print(iterator.next() + " ");

}

System.out.println();

}

interface DataStructureIterator extends java.util.Iterator<Integer> { }

// Inner class implements the DataStructureIterator interface,

// which extends the Iterator<Integer> interface

private class EvenIterator implements DataStructureIterator {

// Start stepping through the array from the beginning

private int nextIndex = 0;

public boolean hasNext() {

// Check if the current element is the last in the array

return (nextIndex <= SIZE - 1);

}

public Integer next() {

// Record a value of an even index of the array

Integer retValue = Integer.valueOf(arrayOfInts[nextIndex]);

// Get the next even element

nextIndex += 2;

return retValue;

}

}

public static void main(String s[]) {

// Fill the array with integer values and print out only

// values of even indices

DataStructure ds = new DataStructure();

ds.printEven();

}

}

The output is:

0 2 4 6 8 10 12 14

Note that the EvenIterator class refers directly to the arrayOfInts instance variable of the DataStructure object.

You can use inner classes to implement helper classes such as the one shown in the this example. To handle user interface events, you must know how to use inner classes, because the event-handling mechanism makes extensive use of them.

## Local and Anonymous Classes

There are two additional types of inner classes. You can declare an inner class within the body of a method. These classes are known as [local classes](http://docs.oracle.com/javase/tutorial/java/javaOO/localclasses.html). You can also declare an inner class within the body of a method without naming the class. These classes are known as [anonymous classes](http://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html).

## Modifiers

You can use the same modifiers for inner classes that you use for other members of the outer class. For example, you can use the access specifiers private, public, and protected to restrict access to inner classes, just as you use them to restrict access do to other class members.

# Lambda Expressions

One issue with anonymous classes is that if the implementation of your anonymous class is very simple, such as an interface that contains only one method, then the syntax of anonymous classes may seem unwieldy and unclear. In these cases, you're usually trying to pass functionality as an argument to another method, such as what action should be taken when someone clicks a button. Lambda expressions enable you to do this, to treat functionality as method argument, or code as data.

The previous section, [Anonymous Classes](http://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html), shows you how to implement a base class without giving it a name. Although this is often more concise than a named class, for classes with only one method, even an anonymous class seems a bit excessive and cumbersome. Lambda expressions let you express instances of single-method classes more compactly.

This section covers the following topics:

* [Ideal Use Case for Lambda Expressions](http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html#use-case)
  + [Approach 1: Create Methods That Search for Members That Match One Characteristic](http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html#approach1)
  + [Approach 2: Create More Generalized Search Methods](http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html#approach2)
  + [Approach 3: Specify Search Criteria Code in a Local Class](http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html#approach3)
  + [Approach 4: Specify Search Criteria Code in an Anonymous Class](http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html#approach4)
  + [Approach 5: Specify Search Criteria Code with a Lambda Expression](http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html#approach5)
  + [Approach 6: Use Standard Functional Interfaces with Lambda Expressions](http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html#approach6)
  + [Approach 7: Use Lambda Expressions Throughout Your Application](http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html#approach7)
  + [Approach 8: Use Generics More Extensively](http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html#approach8)
  + [Approach 9: Use Aggregate Operations That Accept Lambda Expressions as Parameters](http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html#approach9)
* [Lambda Expressions in GUI Applications](http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html#lambda-expressions-in-gui-applications)
* [Syntax of Lambda Expressions](http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html#syntax)
* [Accessing Local Variables of the Enclosing Scope](http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html#accessing-local-variables)
* [Target Typing](http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html#target-typing)
  + [Target Types and Method Arguments](http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html#target-types-and-method-arguments)
* [Serialization](http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html#serialization)

## Ideal Use Case for Lambda Expressions

Suppose that you are creating a social networking application. You want to create a feature that enables an administrator to perform any kind of action, such as sending a message, on members of the social networking application that satisfy certain criteria. The following table describes this use case in detail:

|  |  |
| --- | --- |
| **Field** | **Description** |
| Name | Perform action on selected members |
| Primary Actor | Administrator |
| Preconditions | Administrator is logged in to the system. |
| Postconditions | Action is performed only on members that fit the specified criteria. |
| Main Success Scenario | 1. Administrator specifies criteria of members on which to perform a certain action. 2. Administrator specifies an action to perform on those selected members. 3. Administrator selects the **Submit** button. 4. The system finds all members that match the specified criteria. 5. The system performs the specified action on all matching members. |
| Extensions | 1a. Administrator has an option to preview those members who match the specified criteria before he or she specifies the action to be performed or before selecting the**Submit** button. |
| Frequency of Occurrence | Many times during the day. |

Suppose that members of this social networking application are represented by the following [Person](http://docs.oracle.com/javase/tutorial/java/javaOO/examples/Person.java) class:

public class Person {

public enum Sex {

MALE, FEMALE

}

String name;

LocalDate birthday;

Sex gender;

String emailAddress;

public int getAge() {

// ...

}

public void printPerson() {

// ...

}

}

Suppose that the members of your social networking application are stored in a List<Person> instance.

This section begins with a naive approach to this use case. It improves upon this approach with local and anonymous classes, and then finishes with an efficient and concise approach using lambda expressions. Find the code excerpts described in this section in the example [RosterTest](http://docs.oracle.com/javase/tutorial/java/javaOO/examples/RosterTest.java).

### Approach 1: Create Methods That Search for Members That Match One Characteristic

One simplistic approach is to create several methods; each method searches for members that match one characteristic, such as gender or age. The following method prints members that are older than a specified age:

public static void printPersonsOlderThan(List<Person> roster, int age) {

for (Person p : roster) {

if (p.getAge() >= age) {

p.printPerson();

}

}

}

**Note**: A [List](http://docs.oracle.com/javase/8/docs/api/java/util/List.html) is an ordered [Collection](http://docs.oracle.com/javase/8/docs/api/java/util/Collection.html). A collection is an object that groups multiple elements into a single unit. Collections are used to store, retrieve, manipulate, and communicate aggregate data. For more information about collections, see the [Collections](http://docs.oracle.com/javase/tutorial/collections/index.html) trail.

This approach can potentially make your application brittle, which is the likelihood of an application not working because of the introduction of updates (such as newer data types). Suppose that you upgrade your application and change the structure of the Person class such that it contains different member variables; perhaps the class records and measures ages with a different data type or algorithm. You would have to rewrite a lot of your API to accommodate this change. In addition, this approach is unnecessarily restrictive; what if you wanted to print members younger than a certain age, for example?

### Approach 2: Create More Generalized Search Methods

The following method is more generic than printPersonsOlderThan; it prints members within a specified range of ages:

public static void printPersonsWithinAgeRange(

List<Person> roster, int low, int high) {

for (Person p : roster) {

if (low <= p.getAge() && p.getAge() < high) {

p.printPerson();

}

}

}

What if you want to print members of a specified sex, or a combination of a specified gender and age range? What if you decide to change the Person class and add other attributes such as relationship status or geographical location? Although this method is more generic than printPersonsOlderThan, trying to create a separate method for each possible search query can still lead to brittle code. You can instead separate the code that specifies the criteria for which you want to search in a different class.

### Approach 3: Specify Search Criteria Code in a Local Class

The following method prints members that match search criteria that you specify:

public static void printPersons(

List<Person> roster, CheckPerson tester) {

for (Person p : roster) {

if (tester.test(p)) {

p.printPerson();

}

}

}

This method checks each Person instance contained in the List parameter roster whether it satisfies the search criteria specified in the CheckPerson parameter tester by invoking the method tester.test. If the method tester.test returns a true value, then the method printPersons is invoked on the Person instance.

To specify the search criteria, you implement the CheckPerson interface:

interface CheckPerson {

boolean test(Person p);

}

The following class implements the CheckPerson interface by specifying an implementation for the method test. This method filters members that are eligible for Selective Service in the United States: it returns a true value if its Person parameter is male and between the ages of 18 and 25:

class CheckPersonEligibleForSelectiveService implements CheckPerson {

public boolean test(Person p) {

return p.gender == Person.Sex.MALE &&

p.getAge() >= 18 &&

p.getAge() <= 25;

}

}

To use this class, you create a new instance of it and invoke the printPersons method:

printPersons(

roster, new CheckPersonEligibleForSelectiveService());

Although this approach is less brittle—you don't have to rewrite methods if you change the structure of the Person—you still have additional code: a new interface and a local class for each search you plan to perform in your application. Because CheckPersonEligibleForSelectiveService implements an interface, you can use an anonymous class instead of a local class and bypass the need to declare a new class for each search.

### Approach 4: Specify Search Criteria Code in an Anonymous Class

One of the arguments of the following invocation of the method printPersons is an anonymous class that filters members that are eligible for Selective Service in the United States: those who are male and between the ages of 18 and 25:

printPersons(

roster,

new CheckPerson() {

public boolean test(Person p) {

return p.getGender() == Person.Sex.MALE

&& p.getAge() >= 18

&& p.getAge() <= 25;

}

}

);

This approach reduces the amount of code required because you don't have to create a new class for each search that you want to perform. However, the syntax of anonymous classes is bulky considering that the CheckPerson interface contains only one method. In this case, you can use a lambda expression instead of an anonymous class, as described in the next section.

### Approach 5: Specify Search Criteria Code with a Lambda Expression

The CheckPerson interface is a *functional interface*. A functional interface is any interface that contains only one [abstract method](http://docs.oracle.com/javase/tutorial/java/IandI/abstract.html). (A functional interface may contain one or more [default methods](http://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html) or [static methods](http://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html#static).) Because a functional interface contains only one abstract method, you can omit the name of that method when you implement it. To do this, instead of using an anonymous class expression, you use a *lambda expression*, which is highlighted in the following method invocation:

printPersons(

roster,

**(Person p) -> p.getGender() == Person.Sex.MALE**

**&& p.getAge() >= 18**

**&& p.getAge() <= 25**

);

See [Syntax of Lambda Expressions](http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html#syntax) for information about how to define lambda expressions.

You can use a standard functional interface in place of the interface CheckPerson, which reduces even further the amount of code required.

### Approach 6: Use Standard Functional Interfaces with Lambda Expressions

Reconsider the CheckPerson interface:

interface CheckPerson {

boolean test(Person p);

}

This is a very simple interface. It's a functional interface because it contains only one abstract method. This method takes one parameter and returns a boolean value. The method is so simple that it might not be worth it to define one in your application. Consequently, the JDK defines several standard functional interfaces, which you can find in the package java.util.function.

For example, you can use the Predicate<T> interface in place of CheckPerson. This interface contains the method boolean test(T t):

interface Predicate<T> {

boolean test(T t);

}

The interface Predicate<T> is an example of a generic interface. (For more information about generics, see the [Generics (Updated)](http://docs.oracle.com/javase/tutorial/java/generics/index.html) lesson.) Generic types (such as generic interfaces) specify one or more type parameters within angle brackets (<>). This interface contains only one type parameter, T. When you declare or instantiate a generic type with actual type arguments, you have a parameterized type. For example, the parameterized type Predicate<Person> is the following:

interface Predicate<Person> {

boolean test(Person t);

}

This parameterized type contains a method that has the same return type and parameters as CheckPerson.boolean test(Person p). Consequently, you can use Predicate<T> in place of CheckPerson as the following method demonstrates:

public static void printPersonsWithPredicate(

List<Person> roster, Predicate<Person> tester) {

for (Person p : roster) {

if (tester.test(p)) {

p.printPerson();

}

}

}

As a result, the following method invocation is the same as when you invoked printPersons in [Approach 3: Specify Search Criteria Code in a Local Class](http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html#approach3) to obtain members who are eligible for Selective Service:

printPersonsWithPredicate(

roster,

p -> p.getGender() == Person.Sex.MALE

&& p.getAge() >= 18

&& p.getAge() <= 25

);

This is not the only possible place in this method to use a lambda expression. The following approach suggests other ways to use lambda expressions.

### Approach 7: Use Lambda Expressions Throughout Your Application

Reconsider the method printPersonsWithPredicate to see where else you could use lambda expressions:

public static void printPersonsWithPredicate(

List<Person> roster, Predicate<Person> tester) {

for (Person p : roster) {

if (tester.test(p)) {

p.printPerson();

}

}

}

This method checks each Person instance contained in the List parameter roster whether it satisfies the criteria specified in the Predicate parameter tester. If the Person instance does satisfy the criteria specified by tester, the method printPersron is invoked on the Person instance.

Instead of invoking the method printPerson, you can specify a different action to perform on those Person instances that satisfy the criteria specified by tester. You can specify this action with a lambda expression. Suppose you want a lambda expression similar to printPerson, one that takes one argument (an object of type Person) and returns void. Remember, to use a lambda expression, you need to implement a functional interface. In this case, you need a functional interface that contains an abstract method that can take one argument of type Person and returns void. The Consumer<T> interface contains the method void accept(T t), which has these characteristics. The following method replaces the invocation p.printPerson() with an instance of Consumer<Person> that invokes the method accept:

public static void processPersons(

List<Person> roster,

Predicate<Person> tester,

**Consumer<Person> block**) {

for (Person p : roster) {

if (tester.test(p)) {

**block.accept(p);**

}

}

}

As a result, the following method invocation is the same as when you invoked printPersons in [Approach 3: Specify Search Criteria Code in a Local Class](http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html#approach3) to obtain members who are eligible for Selective Service. The lambda expression used to print members is highlighted:

processPersons(

roster,

p -> p.getGender() == Person.Sex.MALE

&& p.getAge() >= 18

&& p.getAge() <= 25,

**p -> p.printPerson()**

);

What if you want to do more with your members' profiles than printing them out. Suppose that you want to validate the members' profiles or retrieve their contact information? In this case, you need a functional interface that contains an abstract method that returns a value. The Function<T,R> interface contains the method R apply(T t). The following method retrieves the data specified by the parameter mapper, and then performs an action on it specified by the parameter block:

public static void processPersonsWithFunction(

List<Person> roster,

Predicate<Person> tester,

Function<Person, String> mapper,

Consumer<String> block) {

for (Person p : roster) {

if (tester.test(p)) {

String data = mapper.apply(p);

block.accept(data);

}

}

}

The following method retrieves the email address from each member contained in roster who is eligible for Selective Service and then prints it:

processPersonsWithFunction(

roster,

p -> p.getGender() == Person.Sex.MALE

&& p.getAge() >= 18

&& p.getAge() <= 25,

p -> p.getEmailAddress(),

email -> System.out.println(email)

);

### Approach 8: Use Generics More Extensively

Reconsider the method processPersonsWithFunction. The following is a generic version of it that accepts, as a parameter, a collection that contains elements of any data type:

public static <X, Y> void processElements(

Iterable<X> source,

Predicate<X> tester,

Function <X, Y> mapper,

Consumer<Y> block) {

for (X p : source) {

if (tester.test(p)) {

Y data = mapper.apply(p);

block.accept(data);

}

}

}

To print the e-mail address of members who are eligible for Selective Service, invoke the processElements method as follows:

processElements(

roster,

p -> p.getGender() == Person.Sex.MALE

&& p.getAge() >= 18

&& p.getAge() <= 25,

p -> p.getEmailAddress(),

email -> System.out.println(email)

);

This method invocation performs the following actions:

1. Obtains a source of objects from the collection source. In this example, it obtains a source of Person objects from the collection roster. Notice that the collection roster, which is a collection of type List, is also an object of type Iterable.
2. Filters objects that match the Predicate object tester. In this example, the Predicate object is a lambda expression that specifies which members would be eligible for Selective Service.
3. Maps each filtered object to a value as specified by the Function object mapper. In this example, the Function object is a lambda expression that returns the e-mail address of a member.
4. Performs an action on each mapped object as specified by the Consumer object block. In this example, the Consumer object is a lambda expression that prints a string, which is the e-mail address returned by the Function object.

You can replace each of these actions with an aggregate operation.

### Approach 9: Use Aggregate Operations That Accept Lambda Expressions as Parameters

The following example uses aggregate operations to print the e-mail addresses of those members contained in the collection roster who are eligible for Selective Service:

roster

.stream()

.filter(

p -> p.getGender() == Person.Sex.MALE

&& p.getAge() >= 18

&& p.getAge() <= 25)

.map(p -> p.getEmailAddress())

.forEach(email -> System.out.println(email));

The following table maps each of the operations the method processElements performs with the corresponding aggregate operation:

|  |  |
| --- | --- |
| **processElements Action** | **Aggregate Operation** |
| Obtain a source of objects | Stream<E> **stream**() |
| Filter objects that match a Predicate object | Stream<T> **filter**(Predicate<? super T> predicate) |
| Map objects to another value as specified by a Function object | <R> Stream<R> **map**(Function<? super T,? extends R> mapper) |
| Perform an action as specified by a Consumer object | void **forEach**(Consumer<? super T> action) |

The operations filter, map, and forEach are aggregate operations. Aggregate operations process elements from a stream, not directly from a collection (which is the reason why the first method invoked in this example is stream). A stream is a sequence of elements. Unlike a collection, it is not a data structure that stores elements. Instead, a stream carries values from a source, such as collection, through a pipeline. A pipeline is a sequence of stream operations, which in this example is filter- map-forEach. In addition, aggregate operations typically accept lambda expressions as parameters, enabling you to customize how they behave.

For a more thorough discussion of aggregate operations, see the [Aggregate Operations](http://docs.oracle.com/javase/tutorial/collections/streams/index.html) lesson.

## Lambda Expressions in GUI Applications

To process events in a graphical user interface (GUI) application, such as keyboard actions, mouse actions, and scroll actions, you typically create event handlers, which usually involves implementing a particular interface. Often, event handler interfaces are functional interfaces; they tend to have only one method.

In the JavaFX example [HelloWorld.java](http://docs.oracle.com/javase/8/javafx/get-started-tutorial/hello_world.htm) (discussed in the previous section [Anonymous Classes](http://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html)), you can replace the highlighted anonymous class with a lambda expression in this statement:

btn.setOnAction(**new EventHandler<ActionEvent>() {**

**@Override**

**public void handle(ActionEvent event) {**

**System.out.println("Hello World!");**

**}**

**}**);

The method invocation btn.setOnAction specifies what happens when you select the button represented by the btn object. This method requires an object of typeEventHandler<ActionEvent>. The EventHandler<ActionEvent> interface contains only one method, void handle(T event). This interface is a functional interface, so you could use the following highlighted lambda expression to replace it:

btn.setOnAction(

**event -> System.out.println("Hello World!")**

);

## Syntax of Lambda Expressions

A lambda expression consists of the following:

* A comma-separated list of formal parameters enclosed in parentheses. The CheckPerson.test method contains one parameter, p, which represents an instance of the Person class.

**Note**: You can omit the data type of the parameters in a lambda expression. In addition, you can omit the parentheses if there is only one parameter. For example, the following lambda expression is also valid:

p -> p.getGender() == Person.Sex.MALE

&& p.getAge() >= 18

&& p.getAge() <= 25

* The arrow token, ->
* A body, which consists of a single expression or a statement block. This example uses the following expression:
* p.getGender() == Person.Sex.MALE
* && p.getAge() >= 18

&& p.getAge() <= 25

If you specify a single expression, then the Java runtime evaluates the expression and then returns its value. Alternatively, you can use a return statement:

p -> {

return p.getGender() == Person.Sex.MALE

&& p.getAge() >= 18

&& p.getAge() <= 25;

}

A return statement is not an expression; in a lambda expression, you must enclose statements in braces ({}). However, you do not have to enclose a void method invocation in braces. For example, the following is a valid lambda expression:

email -> System.out.println(email)

Note that a lambda expression looks a lot like a method declaration; you can consider lambda expressions as anonymous methods—methods without a name.

The following example, [Calculator](http://docs.oracle.com/javase/tutorial/java/javaOO/examples/Calculator.java), is an example of lambda expressions that take more than one formal parameter:

public class Calculator {

interface IntegerMath {

int operation(int a, int b);

}

public int operateBinary(int a, int b, IntegerMath op) {

return op.operation(a, b);

}

public static void main(String... args) {

Calculator myApp = new Calculator();

IntegerMath addition = (a, b) -> a + b;

IntegerMath subtraction = (a, b) -> a - b;

System.out.println("40 + 2 = " +

myApp.operateBinary(40, 2, addition));

System.out.println("20 - 10 = " +

myApp.operateBinary(20, 10, subtraction));

}

}

The method operateBinary performs a mathematical operation on two integer operands. The operation itself is specified by an instance of IntegerMath. The example defines two operations with lambda expressions, addition and subtraction. The example prints the following:

40 + 2 = 42

20 - 10 = 10

## Accessing Local Variables of the Enclosing Scope

Like local and anonymous classes, lambda expressions can [capture variables](http://docs.oracle.com/javase/tutorial/java/javaOO/localclasses.html#accessing-members-of-an-enclosing-class); they have the same access to local variables of the enclosing scope. However, unlike local and anonymous classes, lambda expressions do not have any shadowing issues (see [Shadowing](http://docs.oracle.com/javase/tutorial/java/javaOO/nested.html#shadowing) for more information). Lambda expressions are lexically scoped. This means that they do not inherit any names from a supertype or introduce a new level of scoping. Declarations in a lambda expression are interpreted just as they are in the enclosing environment. The following example, [LambdaScopeTest](http://docs.oracle.com/javase/tutorial/java/javaOO/examples/LambdaScopeTest.java), demonstrates this:

import java.util.function.Consumer;

public class LambdaScopeTest {

public int x = 0;

class FirstLevel {

public int x = 1;

void methodInFirstLevel(int x) {

// The following statement causes the compiler to generate

// the error "local variables referenced from a lambda expression

// must be final or effectively final" in statement A:

//

// x = 99;

Consumer<Integer> myConsumer = (y) ->

{

System.out.println("x = " + x); // Statement A

System.out.println("y = " + y);

System.out.println("this.x = " + this.x);

System.out.println("LambdaScopeTest.this.x = " +

LambdaScopeTest.this.x);

};

myConsumer.accept(x);

}

}

public static void main(String... args) {

LambdaScopeTest st = new LambdaScopeTest();

LambdaScopeTest.FirstLevel fl = st.new FirstLevel();

fl.methodInFirstLevel(23);

}

}

This example generates the following output:

x = 23

y = 23

this.x = 1

LambdaScopeTest.this.x = 0

If you substitute the parameter x in place of y in the declaration of the lambda expression myConsumer, then the compiler generates an error:

Consumer<Integer> myConsumer = (x) -> {

// ...

}

The compiler generates the error "variable x is already defined in method methodInFirstLevel(int)" because the lambda expression does not introduce a new level of scoping. Consequently, you can directly access fields, methods, and local variables of the enclosing scope. For example, the lambda expression directly accesses the parameter x of the method methodInFirstLevel. To access variables in the enclosing class, use the keyword this. In this example, this.x refers to the member variable FirstLevel.x.

However, like local and anonymous classes, a lambda expression can only access local variables and parameters of the enclosing block that are final or effectively final. For example, suppose that you add the following assignment statement immediately after the methodInFirstLevel definition statement:

void methodInFirstLevel(int x) {

**x = 99;**

// ...

}

Because of this assignment statement, the variable FirstLevel.x is not effectively final anymore. As a result, the Java compiler generates an error message similar to "local variables referenced from a lambda expression must be final or effectively final" where the lambda expression myConsumer tries to access the FirstLevel.x variable:

System.out.println("x = " + x);

## Target Typing

How do you determine the type of a lambda expression? Recall the lambda expression that selected members who are male and between the ages 18 and 25 years:

p -> p.getGender() == Person.Sex.MALE

&& p.getAge() >= 18

&& p.getAge() <= 25

This lambda expression was used in the following two methods:

* public static void printPersons(List<Person> roster, CheckPerson tester) in [Approach 3: Specify Search Criteria Code in a Local Class](http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html#approach3)
* public void printPersonsWithPredicate(List<Person> roster, Predicate<Person> tester) in [Approach 6: Use Standard Functional Interfaces with Lambda Expressions](http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html#approach6)

When the Java runtime invokes the method printPersons, it's expecting a data type of CheckPerson, so the lambda expression is of this type. However, when the Java runtime invokes the method printPersonsWithPredicate, it's expecting a data type of Predicate<Person>, so the lambda expression is of this type. The data type that these methods expect is called thetarget type. To determine the type of a lambda expression, the Java compiler uses the target type of the context or situation in which the lambda expression was found. It follows that you can only use lambda expressions in situations in which the Java compiler can determine a target type:

* Variable declarations
* Assignments
* Return statements
* Array initializers
* Method or constructor arguments
* Lambda expression bodies
* Conditional expressions, ?:
* Cast expressions

### Target Types and Method Arguments

For method arguments, the Java compiler determines the target type with two other language features: overload resolution and type argument inference.

Consider the following two functional interfaces ( [java.lang.Runnable](http://docs.oracle.com/javase/8/docs/api/java/lang/Runnable.html) and [java.util.concurrent.Callable<V>](http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Callable.html)):

public interface Runnable {

void run();

}

public interface Callable<V> {

V call();

}

The method Runnable.run does not return a value, whereas Callable<V>.call does.

Suppose that you have overloaded the method invoke as follows (see [Defining Methods](http://docs.oracle.com/javase/tutorial/java/javaOO/methods.html) for more information about overloading methods):

void invoke(Runnable r) {

r.run();

}

<T> T invoke(Callable<T> c) {

return c.call();

}

Which method will be invoked in the following statement?

String s = invoke(() -> "done");

The method invoke(Callable<T>) will be invoked because that method returns a value; the method invoke(Runnable) does not. In this case, the type of the lambda expression () -> "done" is Callable<T>.

## Serialization

You can [serialize](http://docs.oracle.com/javase/tutorial/jndi/objects/serial.html) a lambda expression if its target type and its captured arguments are serializable. However, like [inner classes](http://docs.oracle.com/javase/tutorial/java/javaOO/nested.html#serialization), the serialization of lambda expressions is strongly discouraged.

# Local Classes

Local classes are classes that are defined in a block, which is a group of zero or more statements between balanced braces. You typically find local classes defined in the body of a method.

This section covers the following topics:

* [Declaring Local Classes](http://docs.oracle.com/javase/tutorial/java/javaOO/localclasses.html#declaring-local-classes)
* [Accessing Members of an Enclosing Class](http://docs.oracle.com/javase/tutorial/java/javaOO/localclasses.html#accessing-members-of-an-enclosing-class)
  + [Shadowing and Local Classes](http://docs.oracle.com/javase/tutorial/java/javaOO/localclasses.html#shadowing-and-local-classes)
* [Local Classes Are Similar To Inner Classes](http://docs.oracle.com/javase/tutorial/java/javaOO/localclasses.html#local-classes-are-similar-to-inner-classes)

## Declaring Local Classes

You can define a local class inside any block (see [Expressions, Statements, and Blocks](http://docs.oracle.com/javase/tutorial/java/nutsandbolts/expressions.html) for more information). For example, you can define a local class in a method body, a for loop, or an ifclause.

The following example, [LocalClassExample](http://docs.oracle.com/javase/tutorial/java/javaOO/examples/LocalClassExample.java), validates two phone numbers. It defines the local class PhoneNumber in the method validatePhoneNumber:

public class LocalClassExample {

static String regularExpression = "[^0-9]";

public static void validatePhoneNumber(

String phoneNumber1, String phoneNumber2) {

final int numberLength = 10;

// Valid in JDK 8 and later:

// int numberLength = 10;

class PhoneNumber {

String formattedPhoneNumber = null;

PhoneNumber(String phoneNumber){

// numberLength = 7;

String currentNumber = phoneNumber.replaceAll(

regularExpression, "");

if (currentNumber.length() == numberLength)

formattedPhoneNumber = currentNumber;

else

formattedPhoneNumber = null;

}

public String getNumber() {

return formattedPhoneNumber;

}

// Valid in JDK 8 and later:

// public void printOriginalNumbers() {

// System.out.println("Original numbers are " + phoneNumber1 +

// " and " + phoneNumber2);

// }

}

PhoneNumber myNumber1 = new PhoneNumber(phoneNumber1);

PhoneNumber myNumber2 = new PhoneNumber(phoneNumber2);

// Valid in JDK 8 and later:

// myNumber1.printOriginalNumbers();

if (myNumber1.getNumber() == null)

System.out.println("First number is invalid");

else

System.out.println("First number is " + myNumber1.getNumber());

if (myNumber2.getNumber() == null)

System.out.println("Second number is invalid");

else

System.out.println("Second number is " + myNumber2.getNumber());

}

public static void main(String... args) {

validatePhoneNumber("123-456-7890", "456-7890");

}

}

The example validates a phone number by first removing all characters from the phone number except the digits 0 through 9. After, it checks whether the phone number contains exactly ten digits (the length of a phone number in North America). This example prints the following:

First number is 1234567890

Second number is invalid

## Accessing Members of an Enclosing Class

A local class has access to the members of its enclosing class. In the previous example, the PhoneNumber constructor accesses the member LocalClassExample.regularExpression.

In addition, a local class has access to local variables. However, a local class can only access local variables that are declared final. When a local class accesses a local variable or parameter of the enclosing block, it captures that variable or parameter. For example, the PhoneNumber constructor can access the local variable numberLength because it is declared final;numberLength is a captured variable.

However, starting in Java SE 8, a local class can access local variables and parameters of the enclosing block that are final or effectively final. A variable or parameter whose value is never changed after it is initialized is effectively final. For example, suppose that the variable numberLength is not declared final, and you add the highlighted assignment statement in thePhoneNumber constructor:

PhoneNumber(String phoneNumber) {

**numberLength = 7;**

String currentNumber = phoneNumber.replaceAll(

regularExpression, "");

if (currentNumber.length() == numberLength)

formattedPhoneNumber = currentNumber;

else

formattedPhoneNumber = null;

}

Because of this assignment statement, the variable numberLength is not effectively final anymore. As a result, the Java compiler generates an error message similar to "local variables referenced from an inner class must be final or effectively final" where the inner class PhoneNumber tries to access the numberLength variable:

if (currentNumber.length() == numberLength)

Starting in Java SE 8, if you declare the local class in a method, it can access the method's parameters. For example, you can define the following method in the PhoneNumber local class:

public void printOriginalNumbers() {

System.out.println("Original numbers are " + phoneNumber1 +

" and " + phoneNumber2);

}

The method printOriginalNumbers accesses the parameters phoneNumber1 and phoneNumber2 of the method validatePhoneNumber.

### Shadowing and Local Classes

Declarations of a type (such as a variable) in a local class shadow declarations in the enclosing scope that have the same name. See [Shadowing](http://docs.oracle.com/javase/tutorial/java/javaOO/nested.html#shadowing) for more information.

## Local Classes Are Similar To Inner Classes

Local classes are similar to inner classes because they cannot define or declare any static members. Local classes in static methods, such as the class PhoneNumber, which is defined in the static method validatePhoneNumber, can only refer to static members of the enclosing class. For example, if you do not define the member variable regularExpression as static, then the Java compiler generates an error similar to "non-static variable regularExpression cannot be referenced from a static context."

Local classes are non-static because they have access to instance members of the enclosing block. Consequently, they cannot contain most kinds of static declarations.

You cannot declare an interface inside a block; interfaces are inherently static. For example, the following code excerpt does not compile because the interface HelloThere is defined inside the body of the method greetInEnglish:

public void greetInEnglish() {

interface HelloThere {

public void greet();

}

class EnglishHelloThere implements HelloThere {

public void greet() {

System.out.println("Hello " + name);

}

}

HelloThere myGreeting = new EnglishHelloThere();

myGreeting.greet();

}

You cannot declare static initializers or member interfaces in a local class. The following code excerpt does not compile because the method EnglishGoodbye.sayGoodbye is declaredstatic. The compiler generates an error similar to "modifier 'static' is only allowed in constant variable declaration" when it encounters this method definition:

public void sayGoodbyeInEnglish() {

class EnglishGoodbye {

public static void sayGoodbye() {

System.out.println("Bye bye");

}

}

EnglishGoodbye.sayGoodbye();

}

A local class can have static members provided that they are constant variables. (A constant variable is a variable of primitive type or type String that is declared final and initialized with a compile-time constant expression. A compile-time constant expression is typically a string or an arithmetic expression that can be evaluated at compile time. See [Understanding Class Members](http://docs.oracle.com/javase/tutorial/java/javaOO/classvars.html)for more information.) The following code excerpt compiles because the static member EnglishGoodbye.farewell is a constant variable:

public void sayGoodbyeInEnglish() {

class EnglishGoodbye {

public static final String farewell = "Bye bye";

public void sayGoodbye() {

System.out.println(farewell);

}

}

EnglishGoodbye myEnglishGoodbye = new EnglishGoodbye();

myEnglishGoodbye.sayGoodbye();

}

# Anonymous Classes

Anonymous classes enable you to make your code more concise. They enable you to declare and instantiate a class at the same time. They are like local classes except that they do not have a name. Use them if you need to use a local class only once.

This section covers the following topics:

* [Declaring Anonymous Classes](http://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html#declaring-anonymous-classes)
* [Syntax of Anonymous Classes](http://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html#syntax-of-anonymous-classes)
* [Accessing Local Variables of the Enclosing Scope, and Declaring and Accessing Members of the Anonymous Class](http://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html#accessing)
* [Examples of Anonymous Classes](http://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html#examples-of-anonymous-classes)

## Declaring Anonymous Classes

While local classes are class declarations, anonymous classes are expressions, which means that you define the class in another expression. The following example,[HelloWorldAnonymousClasses](http://docs.oracle.com/javase/tutorial/java/javaOO/examples/HelloWorldAnonymousClasses.java), uses anonymous classes in the initialization statements of the local variables frenchGreeting and spanishGreeting, but uses a local class for the initialization of the variable englishGreeting:

public class HelloWorldAnonymousClasses {

interface HelloWorld {

public void greet();

public void greetSomeone(String someone);

}

public void sayHello() {

class EnglishGreeting implements HelloWorld {

String name = "world";

public void greet() {

greetSomeone("world");

}

public void greetSomeone(String someone) {

name = someone;

System.out.println("Hello " + name);

}

}

HelloWorld englishGreeting = new EnglishGreeting();

HelloWorld frenchGreeting = new HelloWorld() {

String name = "tout le monde";

public void greet() {

greetSomeone("tout le monde");

}

public void greetSomeone(String someone) {

name = someone;

System.out.println("Salut " + name);

}

};

HelloWorld spanishGreeting = new HelloWorld() {

String name = "mundo";

public void greet() {

greetSomeone("mundo");

}

public void greetSomeone(String someone) {

name = someone;

System.out.println("Hola, " + name);

}

};

englishGreeting.greet();

frenchGreeting.greetSomeone("Fred");

spanishGreeting.greet();

}

public static void main(String... args) {

HelloWorldAnonymousClasses myApp =

new HelloWorldAnonymousClasses();

myApp.sayHello();

}

}

## Syntax of Anonymous Classes

As mentioned previously, an anonymous class is an expression. The syntax of an anonymous class expression is like the invocation of a constructor, except that there is a class definition contained in a block of code.

Consider the instantiation of the frenchGreeting object:

HelloWorld frenchGreeting = new HelloWorld() {

String name = "tout le monde";

public void greet() {

greetSomeone("tout le monde");

}

public void greetSomeone(String someone) {

name = someone;

System.out.println("Salut " + name);

}

};

The anonymous class expression consists of the following:

* The new operator
* The name of an interface to implement or a class to extend. In this example, the anonymous class is implementing the interface HelloWorld.
* Parentheses that contain the arguments to a constructor, just like a normal class instance creation expression. **Note**: When you implement an interface, there is no constructor, so you use an empty pair of parentheses, as in this example.
* A body, which is a class declaration body. More specifically, in the body, method declarations are allowed but statements are not.

Because an anonymous class definition is an expression, it must be part of a statement. In this example, the anonymous class expression is part of the statement that instantiates thefrenchGreeting object. (This explains why there is a semicolon after the closing brace.)

## Accessing Local Variables of the Enclosing Scope, and Declaring and Accessing Members of the Anonymous Class

Like local classes, anonymous classes can [capture variables](http://docs.oracle.com/javase/tutorial/java/javaOO/localclasses.html#accessing-members-of-an-enclosing-class); they have the same access to local variables of the enclosing scope:

* An anonymous class has access to the members of its enclosing class.
* An anonymous class cannot access local variables in its enclosing scope that are not declared as final or effectively final.
* Like a nested class, a declaration of a type (such as a variable) in an anonymous class shadows any other declarations in the enclosing scope that have the same name. See [Shadowing](http://docs.oracle.com/javase/tutorial/java/javaOO/nested.html#shadowing)for more information.

Anonymous classes also have the same restrictions as local classes with respect to their members:

* You cannot declare static initializers or member interfaces in an anonymous class.
* An anonymous class can have static members provided that they are constant variables.

Note that you can declare the following in anonymous classes:

* Fields
* Extra methods (even if they do not implement any methods of the supertype)
* Instance initializers
* Local classes

However, you cannot declare constructors in an anonymous class.

## Examples of Anonymous Classes

Anonymous classes are often used in graphical user interface (GUI) applications.

Consider the JavaFX example [HelloWorld.java](http://docs.oracle.com/javase/8/javafx/get-started-tutorial/hello_world.htm) (from the section [Hello World, JavaFX Style](http://docs.oracle.com/javase/8/javafx/get-started-tutorial/hello_world.htm) from [Getting Started with JavaFX](http://docs.oracle.com/javase/8/javafx/get-started-tutorial/javafx_get_started.htm)). This sample creates a frame that contains a **Say 'Hello World'** button. The anonymous class expression is highlighted:

import javafx.event.ActionEvent;

import javafx.event.EventHandler;

import javafx.scene.Scene;

import javafx.scene.control.Button;

import javafx.scene.layout.StackPane;

import javafx.stage.Stage;

public class HelloWorld extends Application {

public static void main(String[] args) {

launch(args);

}

@Override

public void start(Stage primaryStage) {

primaryStage.setTitle("Hello World!");

Button btn = new Button();

btn.setText("Say 'Hello World'");

btn.setOnAction(**new EventHandler<ActionEvent>() {**

**@Override**

**public void handle(ActionEvent event) {**

**System.out.println("Hello World!");**

**}**

**}**);

StackPane root = new StackPane();

root.getChildren().add(btn);

primaryStage.setScene(new Scene(root, 300, 250));

primaryStage.show();

}

}

In this example, the method invocation btn.setOnAction specifies what happens when you select the **Say 'Hello World'** button. This method requires an object of typeEventHandler<ActionEvent>. The EventHandler<ActionEvent> interface contains only one method, handle. Instead of implementing this method with a new class, the example uses an anonymous class expression. Notice that this expression is the argument passed to the btn.setOnAction method.

Because the EventHandler<ActionEvent> interface contains only one method, you can use a lambda expression instead of an anonymous class expression. See the section [Lambda Expressions](http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html) for more information.

Anonymous classes are ideal for implementing an interface that contains two or more methods. The following JavaFX example is from the section [Customization of UI Controls](http://docs.oracle.com/javase/8/javafx/user-interface-tutorial/custom.htm). The highlighted code creates a text field that only accepts numeric values. It redefines the default implementation of the TextField class with an anonymous class by overriding the replaceText andreplaceSelection methods inherited from the TextInputControl class.

import javafx.application.Application;

import javafx.event.ActionEvent;

import javafx.event.EventHandler;

import javafx.geometry.Insets;

import javafx.scene.Group;

import javafx.scene.Scene;

import javafx.scene.control.\*;

import javafx.scene.layout.GridPane;

import javafx.scene.layout.HBox;

import javafx.stage.Stage;

public class CustomTextFieldSample extends Application {

final static Label label = new Label();

@Override

public void start(Stage stage) {

Group root = new Group();

Scene scene = new Scene(root, 300, 150);

stage.setScene(scene);

stage.setTitle("Text Field Sample");

GridPane grid = new GridPane();

grid.setPadding(new Insets(10, 10, 10, 10));

grid.setVgap(5);

grid.setHgap(5);

scene.setRoot(grid);

final Label dollar = new Label("$");

GridPane.setConstraints(dollar, 0, 0);

grid.getChildren().add(dollar);

final TextField sum = **new TextField() {**

**@Override**

**public void replaceText(int start, int end, String text) {**

**if (!text.matches("[a-z, A-Z]")) {**

**super.replaceText(start, end, text);**

**}**

**label.setText("Enter a numeric value");**

**}**

**@Override**

**public void replaceSelection(String text) {**

**if (!text.matches("[a-z, A-Z]")) {**

**super.replaceSelection(text);**

**}**

**}**

**};**

sum.setPromptText("Enter the total");

sum.setPrefColumnCount(10);

GridPane.setConstraints(sum, 1, 0);

grid.getChildren().add(sum);

Button submit = new Button("Submit");

GridPane.setConstraints(submit, 2, 0);

grid.getChildren().add(submit);

submit.setOnAction(new EventHandler<ActionEvent>() {

@Override

public void handle(ActionEvent e) {

label.setText(null);

}

});

GridPane.setConstraints(label, 0, 1);

GridPane.setColumnSpan(label, 3);

grid.getChildren().add(label);

scene.setRoot(grid);

stage.show();

}

public static void main(String[] args) {

launch(args);

}

}

# Annotations Basics

## The Format of an Annotation

In its simplest form, an annotation looks like the following:

@Entity

The at sign character (@) indicates to the compiler that what follows is an annotation. In the following example, the annotation's name is Override:

@Override

void mySuperMethod() { ... }

The annotation can include elements, which can be named or unnamed, and there are values for those elements:

@Author(

name = "Benjamin Franklin",

date = "3/27/2003"

)

class MyClass() { ... }

or

@SuppressWarnings(value = "unchecked")

void myMethod() { ... }

If there is just one element named value, then the name can be omitted, as in:

@SuppressWarnings("unchecked")

void myMethod() { ... }

If the annotation has no elements, then the parentheses can be omitted, as shown in the previous @Override example.

It is also possible to use multiple annotations on the same declaration:

@Author(name = "Jane Doe")

@EBook

class MyClass { ... }

If the annotations have the same type, then this is called a repeating annotation:

@Author(name = "Jane Doe")

@Author(name = "John Smith")

class MyClass { ... }

Repeating annotations are supported as of the Java SE 8 release. For more information, see [Repeating Annotations](http://docs.oracle.com/javase/tutorial/java/annotations/repeating.html).

The annotation type can be one of the types that are defined in the java.lang or java.lang.annotation packages of the Java SE API. In the previous examples, Override andSuppressWarnings are [predefined Java annotations](http://docs.oracle.com/javase/tutorial/java/annotations/predefined.html). It is also possible to define your own annotation type. The Author and Ebook annotations in the previous example are custom annotation types.

## Where Annotations Can Be Used

Annotations can be applied to declarations: declarations of classes, fields, methods, and other program elements. When used on a declaration, each annotation often appears, by convention, on its own line.

As of the Java SE 8 release, annotations can also be applied to the use of types. Here are some examples:

* Class instance creation expression:
* new @Interned MyObject();
* Type cast:
* myString = (@NonNull String) str;
* implements clause:
* class UnmodifiableList<T> implements
* @Readonly List<@Readonly T> { ... }
* Thrown exception declaration:
* void monitorTemperature() throws
* @Critical TemperatureException { ... }

This form of annotation is called a type annotation. For more information, see [Type Annotations and Pluggable Type Systems](http://docs.oracle.com/javase/tutorial/java/annotations/type_annotations.html).

# Declaring an Annotation Type

Many annotations replace comments in code.

Suppose that a software group traditionally starts the body of every class with comments providing important information:

public class Generation3List extends Generation2List {

// Author: John Doe

// Date: 3/17/2002

// Current revision: 6

// Last modified: 4/12/2004

// By: Jane Doe

// Reviewers: Alice, Bill, Cindy

// class code goes here

}

To add this same metadata with an annotation, you must first define the *annotation type*. The syntax for doing this is:

@interface ClassPreamble {

String author();

String date();

int currentRevision() default 1;

String lastModified() default "N/A";

String lastModifiedBy() default "N/A";

// Note use of array

String[] reviewers();

}

The annotation type definition looks similar to an interface definition where the keyword interface is preceded by the at sign (@) (@ = AT, as in annotation type). Annotation types are a form of *interface*, which will be covered in a later lesson. For the moment, you do not need to understand interfaces.

The body of the previous annotation definition contains *annotation type element* declarations, which look a lot like methods. Note that they can define optional default values.

After the annotation type is defined, you can use annotations of that type, with the values filled in, like this:

@ClassPreamble (

author = "John Doe",

date = "3/17/2002",

currentRevision = 6,

lastModified = "4/12/2004",

lastModifiedBy = "Jane Doe",

// Note array notation

reviewers = {"Alice", "Bob", "Cindy"}

)

public class Generation3List extends Generation2List {

// class code goes here

}

**Note:** To make the information in @ClassPreamble appear in Javadoc-generated documentation, you must annotate the @ClassPreamble definition with the @Documentedannotation:

// import this to use @Documented

import java.lang.annotation.\*;

@Documented

@interface ClassPreamble {

// Annotation element definitions

}

# Predefined Annotation Types

A set of annotation types are predefined in the Java SE API. Some annotation types are used by the Java compiler, and some apply to other annotations.

## Annotation Types Used by the Java Language

The predefined annotation types defined in java.lang are @Deprecated, @Override, and @SuppressWarnings.

**@Deprecated** [@Deprecated](http://docs.oracle.com/javase/8/docs/api/java/lang/Deprecated.html) annotation indicates that the marked element is *deprecated* and should no longer be used. The compiler generates a warning whenever a program uses a method, class, or field with the @Deprecated annotation. When an element is deprecated, it should also be documented using the Javadoc @deprecated tag, as shown in the following example. The use of the at sign (@) in both Javadoc comments and in annotations is not coincidental: they are related conceptually. Also, note that the Javadoc tag starts with a lowercase d and the annotation starts with an uppercase D.

// Javadoc comment follows

/\*\*

\* *@deprecated*

\* *explanation of why it was deprecated*

\*/

**@Deprecated**

static void deprecatedMethod() { }

}

**@Override** [@Override](http://docs.oracle.com/javase/8/docs/api/java/lang/Override.html) annotation informs the compiler that the element is meant to override an element declared in a superclass. Overriding methods will be discussed in [Interfaces and Inheritance](http://docs.oracle.com/javase/tutorial/java/IandI/index.html).

// *mark method as a superclass method*

// *that has been overridden*

**@Override**

int overriddenMethod() { }

While it is not required to use this annotation when overriding a method, it helps to prevent errors. If a method marked with @Override fails to correctly override a method in one of its superclasses, the compiler generates an error.

**@SuppressWarnings** [@SuppressWarnings](http://docs.oracle.com/javase/8/docs/api/java/lang/SuppressWarnings.html) annotation tells the compiler to suppress specific warnings that it would otherwise generate. In the following example, a deprecated method is used, and the compiler usually generates a warning. In this case, however, the annotation causes the warning to be suppressed.

// *use a deprecated method and tell*

// *compiler not to generate a warning*

**@SuppressWarnings("deprecation")**

void useDeprecatedMethod() {

// deprecation warning

// - suppressed

objectOne.deprecatedMethod();

}

Every compiler warning belongs to a category. The Java Language Specification lists two categories: deprecation and unchecked. The unchecked warning can occur when interfacing with legacy code written before the advent of [generics](http://docs.oracle.com/javase/tutorial/java/generics/index.html). To suppress multiple categories of warnings, use the following syntax:

@SuppressWarnings({"unchecked", "deprecation"})

**@SafeVarargs** [@SafeVarargs](http://docs.oracle.com/javase/8/docs/api/java/lang/SafeVarargs.html) annotation, when applied to a method or constructor, asserts that the code does not perform potentially unsafe operations on its varargs parameter. When this annotation type is used, unchecked warnings relating to varargs usage are suppressed.

**@FunctionalInterface** [@FunctionalInterface](http://docs.oracle.com/javase/8/docs/api/java/lang/FunctionalInterface.html) annotation, introduced in Java SE 8, indicates that the type declaration is intended to be a functional interface, as defined by the Java Language Specification.

## Annotations That Apply to Other Annotations

Annotations that apply to other annotations are called meta-annotations. There are several meta-annotation types defined in java.lang.annotation.

**@Retention** [@Retention](http://docs.oracle.com/javase/8/docs/api/java/lang/annotation/Retention.html) annotation specifies how the marked annotation is stored:

* RetentionPolicy.SOURCE – The marked annotation is retained only in the source level and is ignored by the compiler.
* RetentionPolicy.CLASS – The marked annotation is retained by the compiler at compile time, but is ignored by the Java Virtual Machine (JVM).
* RetentionPolicy.RUNTIME – The marked annotation is retained by the JVM so it can be used by the runtime environment.

**@Documented** [@Documented](http://docs.oracle.com/javase/8/docs/api/java/lang/annotation/Documented.html) annotation indicates that whenever the specified annotation is used those elements should be documented using the Javadoc tool. (By default, annotations are not included in Javadoc.) For more information, see the [Javadoc tools page](http://docs.oracle.com/javase/8/docs/technotes/guides/javadoc/index.html).

**@Target** [@Target](http://docs.oracle.com/javase/8/docs/api/java/lang/annotation/Target.html) annotation marks another annotation to restrict what kind of Java elements the annotation can be applied to. A target annotation specifies one of the following element types as its value:

* ElementType.ANNOTATION\_TYPE can be applied to an annotation type.
* ElementType.CONSTRUCTOR can be applied to a constructor.
* ElementType.FIELD can be applied to a field or property.
* ElementType.LOCAL\_VARIABLE can be applied to a local variable.
* ElementType.METHOD can be applied to a method-level annotation.
* ElementType.PACKAGE can be applied to a package declaration.
* ElementType.PARAMETER can be applied to the parameters of a method.
* ElementType.TYPE can be applied to any element of a class.

**@Inherited** [@Inherited](http://docs.oracle.com/javase/8/docs/api/java/lang/annotation/Inherited.html) annotation indicates that the annotation type can be inherited from the super class. (This is not true by default.) When the user queries the annotation type and the class has no annotation for this type, the class' superclass is queried for the annotation type. This annotation applies only to class declarations.

**@Repeatable** [@Repeatable](http://docs.oracle.com/javase/8/docs/api/java/lang/annotation/Repeatable.html) annotation, introduced in Java SE 8, indicates that the marked annotation can be applied more than once to the same declaration or type use. For more information, see [Repeating Annotations](http://docs.oracle.com/javase/tutorial/java/annotations/repeating.html).

# Type Annotations and Pluggable Type Systems

Before the Java SE 8 release, annotations could only be applied to declarations. As of the Java SE 8 release, annotations can also be applied to any type use. This means that annotations can be used anywhere you use a type. A few examples of where types are used are class instance creation expressions (new), casts, implements clauses, and throws clauses. This form of annotation is called a type annotation and several examples are provided in [Annotations Basics](http://docs.oracle.com/javase/tutorial/java/annotations/basics.html).

Type annotations were created to support improved analysis of Java programs way of ensuring stronger type checking. The Java SE 8 release does not provide a type checking framework, but it allows you to write (or download) a type checking framework that is implemented as one or more pluggable modules that are used in conjunction with the Java compiler.

For example, you want to ensure that a particular variable in your program is never assigned to null; you want to avoid triggering a NullPointerException. You can write a custom plug-in to check for this. You would then modify your code to annotate that particular variable, indicating that it is never assigned to null. The variable declaration might look like this:

**@NonNull** String str;

When you compile the code, including the NonNull module at the command line, the compiler prints a warning if it detects a potential problem, allowing you to modify the code to avoid the error. After you correct the code to remove all warnings, this particular error will not occur when the program runs.

You can use multiple type-checking modules where each module checks for a different kind of error. In this way, you can build on top of the Java type system, adding specific checks when and where you want them.

With the judicious use of type annotations and the presence of pluggable type checkers, you can write code that is stronger and less prone to error.

In many cases, you do not have to write your own type checking modules. There are third parties who have done the work for you. For example, you might want to take advantage of the Checker Framework created by the University of Washington. This framework includes a NonNull module, as well as a regular expression module, and a mutex lock module. For more information, see the [Checker Framework](http://types.cs.washington.edu/checker-framework/).

# Interfaces

There are a number of situations in software engineering when it is important for disparate groups of programmers to agree to a "contract" that spells out how their software interacts. Each group should be able to write their code without any knowledge of how the other group's code is written. Generally speaking, *interfaces* are such contracts.

For example, imagine a futuristic society where computer-controlled robotic cars transport passengers through city streets without a human operator. Automobile manufacturers write software (Java, of course) that operates the automobile—stop, start, accelerate, turn left, and so forth. Another industrial group, electronic guidance instrument manufacturers, make computer systems that receive GPS (Global Positioning System) position data and wireless transmission of traffic conditions and use that information to drive the car.

The auto manufacturers must publish an industry-standard interface that spells out in detail what methods can be invoked to make the car move (any car, from any manufacturer). The guidance manufacturers can then write software that invokes the methods described in the interface to command the car. Neither industrial group needs to know *how* the other group's software is implemented. In fact, each group considers its software highly proprietary and reserves the right to modify it at any time, as long as it continues to adhere to the published interface.

## Interfaces in Java

In the Java programming language, an *interface* is a reference type, similar to a class, that can contain *only* constants, method signatures, default methods, static methods, and nested types. Method bodies exist only for default methods and static methods. Interfaces cannot be instantiated—they can only be *implemented* by classes or *extended* by other interfaces. Extension is discussed later in this lesson.

Defining an interface is similar to creating a new class:

public interface OperateCar {

// constant declarations, if any

// method signatures

// An enum with values RIGHT, LEFT

int turn(Direction direction,

double radius,

double startSpeed,

double endSpeed);

int changeLanes(Direction direction,

double startSpeed,

double endSpeed);

int signalTurn(Direction direction,

boolean signalOn);

int getRadarFront(double distanceToCar,

double speedOfCar);

int getRadarRear(double distanceToCar,

double speedOfCar);

......

// more method signatures

}

Note that the method signatures have no braces and are terminated with a semicolon.

To use an interface, you write a class that *implements* the interface. When an instantiable class implements an interface, it provides a method body for each of the methods declared in the interface. For example,

public class OperateBMW760i implements OperateCar {

// the OperateCar method signatures, with implementation --

// for example:

int signalTurn(Direction direction, boolean signalOn) {

// code to turn BMW's LEFT turn indicator lights on

// code to turn BMW's LEFT turn indicator lights off

// code to turn BMW's RIGHT turn indicator lights on

// code to turn BMW's RIGHT turn indicator lights off

}

// other members, as needed -- for example, helper classes not

// visible to clients of the interface

}

In the robotic car example above, it is the automobile manufacturers who will implement the interface. Chevrolet's implementation will be substantially different from that of Toyota, of course, but both manufacturers will adhere to the same interface. The guidance manufacturers, who are the clients of the interface, will build systems that use GPS data on a car's location, digital street maps, and traffic data to drive the car. In so doing, the guidance systems will invoke the interface methods: turn, change lanes, brake, accelerate, and so forth.

## Interfaces as APIs

The robotic car example shows an interface being used as an industry standard *Application Programming Interface (API)*. APIs are also common in commercial software products. Typically, a company sells a software package that contains complex methods that another company wants to use in its own software product. An example would be a package of digital image processing methods that are sold to companies making end-user graphics programs. The image processing company writes its classes to implement an interface, which it makes public to its customers. The graphics company then invokes the image processing methods using the signatures and return types defined in the interface. While the image processing company's API is made public (to its customers), its implementation of the API is kept as a closely guarded secret—in fact, it may revise the implementation at a later date as long as it continues to implement the original interface that its customers have relied on.

# Defining an Interface

An interface declaration consists of modifiers, the keyword interface, the interface name, a comma-separated list of parent interfaces (if any), and the interface body. For example:

public interface GroupedInterface extends Interface1, Interface2, Interface3 {

// constant declarations

// base of natural logarithms

double E = 2.718282;

// method signatures

void doSomething (int i, double x);

int doSomethingElse(String s);

}

The public access specifier indicates that the interface can be used by any class in any package. If you do not specify that the interface is public, then your interface is accessible only to classes defined in the same package as the interface.

An interface can extend other interfaces, just as a class subclass or extend another class. However, whereas a class can extend only one other class, an interface can extend any number of interfaces. The interface declaration includes a comma-separated list of all the interfaces that it extends.

## The Interface Body

The interface body can contain [abstract methods](http://docs.oracle.com/javase/tutorial/java/IandI/abstract.html), [default methods](http://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html), and [static methods](http://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html#static). An abstract method within an interface is followed by a semicolon, but no braces (an abstract method does not contain an implementation). Default methods are defined with the default modifier, and static methods with the static keyword. All abstract, default, and static methods in an interface are implicitly public, so you can omit the public modifier.

In addition, an interface can contain constant declarations. All constant values defined in an interface are implicitly public, static, and final. Once again, you can omit these modifiers.

# Implementing an Interface

To declare a class that implements an interface, you include an implements clause in the class declaration. Your class can implement more than one interface, so the implements keyword is followed by a comma-separated list of the interfaces implemented by the class. By convention, the implements clause follows the extends clause, if there is one.

## A Sample Interface, Relatable

Consider an interface that defines how to compare the size of objects.

public interface Relatable {

// this (object calling isLargerThan)

// and other must be instances of

// the same class returns 1, 0, -1

// if this is greater than,

// equal to, or less than other

public int isLargerThan(Relatable other);

}

If you want to be able to compare the size of similar objects, no matter what they are, the class that instantiates them should implement Relatable.

Any class can implement Relatable if there is some way to compare the relative "size" of objects instantiated from the class. For strings, it could be number of characters; for books, it could be number of pages; for students, it could be weight; and so forth. For planar geometric objects, area would be a good choice (see the RectanglePlus class that follows), while volume would work for three-dimensional geometric objects. All such classes can implement the isLargerThan() method.

If you know that a class implements Relatable, then you know that you can compare the size of the objects instantiated from that class.

## Implementing the Relatable Interface

Here is the Rectangle class that was presented in the [Creating Objects](http://docs.oracle.com/javase/tutorial/java/javaOO/objectcreation.html) section, rewritten to implement Relatable.

public class RectanglePlus

implements Relatable {

public int width = 0;

public int height = 0;

public Point origin;

// four constructors

public RectanglePlus() {

origin = new Point(0, 0);

}

public RectanglePlus(Point p) {

origin = p;

}

public RectanglePlus(int w, int h) {

origin = new Point(0, 0);

width = w;

height = h;

}

public RectanglePlus(Point p, int w, int h) {

origin = p;

width = w;

height = h;

}

// a method for moving the rectangle

public void move(int x, int y) {

origin.x = x;

origin.y = y;

}

// a method for computing

// the area of the rectangle

public int getArea() {

return width \* height;

}

// a method required to implement

// the Relatable interface

public int isLargerThan(Relatable other) {

**RectanglePlus otherRect**

**= (RectanglePlus)other;**

if (this.getArea() < otherRect.getArea())

return -1;

else if (this.getArea() > otherRect.getArea())

return 1;

else

return 0;

}

}

Because RectanglePlus implements Relatable, the size of any two RectanglePlus objects can be compared.

**Note:** The isLargerThan method, as defined in the Relatable interface, takes an object of type Relatable. The line of code, shown in bold in the previous example, castsother to a RectanglePlus instance. Type casting tells the compiler what the object really is. Invoking getArea directly on the other instance (other.getArea()) would fail to compile because the compiler does not understand that other is actually an instance of RectanglePlus.

# Using an Interface as a Type

When you define a new interface, you are defining a new reference data type. You can use interface names anywhere you can use any other data type name. If you define a reference variable whose type is an interface, any object you assign to it *must* be an instance of a class that implements the interface.

As an example, here is a method for finding the largest object in a pair of objects, for *any* objects that are instantiated from a class that implements Relatable:

public Object findLargest(Object object1, Object object2) {

Relatable obj1 = (Relatable)object1;

Relatable obj2 = (Relatable)object2;

if ((obj1).isLargerThan(obj2) > 0)

return object1;

else

return object2;

}

By casting object1 to a Relatable type, it can invoke the isLargerThan method.

If you make a point of implementing Relatable in a wide variety of classes, the objects instantiated from *any* of those classes can be compared with the findLargest() method—provided that both objects are of the same class. Similarly, they can all be compared with the following methods:

public Object findSmallest(Object object1, Object object2) {

Relatable obj1 = (Relatable)object1;

Relatable obj2 = (Relatable)object2;

if ((obj1).isLargerThan(obj2) < 0)

return object1;

else

return object2;

}

public boolean isEqual(Object object1, Object object2) {

Relatable obj1 = (Relatable)object1;

Relatable obj2 = (Relatable)object2;

if ( (obj1).isLargerThan(obj2) == 0)

return true;

else

return false;

}

These methods work for any "relatable" objects, no matter what their class inheritance is. When they implement Relatable, they can be of both their own class (or superclass) type and aRelatable type. This gives them some of the advantages of multiple inheritance, where they can have behavior from both a superclass and an interface.

# Evolving Interfaces

Consider an interface that you have developed called DoIt:

public interface DoIt {

void doSomething(int i, double x);

int doSomethingElse(String s);

}

Suppose that, at a later time, you want to add a third method to DoIt, so that the interface now becomes:

public interface DoIt {

void doSomething(int i, double x);

int doSomethingElse(String s);

boolean didItWork(int i, double x, String s);

}

If you make this change, then all classes that implement the old DoIt interface will break because they no longer implement the old interface. Programmers relying on this interface will protest loudly.

Try to anticipate all uses for your interface and specify it completely from the beginning. If you want to add additional methods to an interface, you have several options. You could create aDoItPlus interface that extends DoIt:

public interface DoItPlus extends DoIt {

boolean didItWork(int i, double x, String s);

}

Now users of your code can choose to continue to use the old interface or to upgrade to the new interface.

Alternatively, you can define your new methods as [default methods](http://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html). The following example defines a default method named didItWork:

public interface DoIt {

void doSomething(int i, double x);

int doSomethingElse(String s);

**default boolean didItWork(int i, double x, String s) {**

**// Method body**

**}**

}

Note that you must provide an implementation for default methods. You could also define new [static methods](http://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html#static) to existing interfaces. Users who have classes that implement interfaces enhanced with new default or static methods do not have to modify or recompile them to accommodate the additional methods.

# Default Methods

The section [Interfaces](http://docs.oracle.com/javase/tutorial/java/IandI/createinterface.html) describes an example that involves manufacturers of computer-controlled cars who publish industry-standard interfaces that describe which methods can be invoked to operate their cars. What if those computer-controlled car manufacturers add new functionality, such as flight, to their cars? These manufacturers would need to specify new methods to enable other companies (such as electronic guidance instrument manufacturers) to adapt their software to flying cars. Where would these car manufacturers declare these new flight-related methods? If they add them to their original interfaces, then programmers who have implemented those interfaces would have to rewrite their implementations. If they add them as static methods, then programmers would regard them as utility methods, not as essential, core methods.

Default methods enable you to add new functionality to the interfaces of your libraries and ensure binary compatibility with code written for older versions of those interfaces.

Consider the following interface, [TimeClient](http://docs.oracle.com/javase/tutorial/java/IandI/examples/TimeClient.java), as described in [Answers to Questions and Exercises: Interfaces](http://docs.oracle.com/javase/tutorial/java/IandI/QandE/interfaces-answers.html):

import java.time.\*;

public interface TimeClient {

void setTime(int hour, int minute, int second);

void setDate(int day, int month, int year);

void setDateAndTime(int day, int month, int year,

int hour, int minute, int second);

LocalDateTime getLocalDateTime();

}

The following class, [SimpleTimeClient](http://docs.oracle.com/javase/tutorial/java/IandI/examples/defaultmethods/SimpleTimeClient.java), implements TimeClient:

package defaultmethods;

import java.time.\*;

import java.lang.\*;

import java.util.\*;

public class SimpleTimeClient implements TimeClient {

private LocalDateTime dateAndTime;

public SimpleTimeClient() {

dateAndTime = LocalDateTime.now();

}

public void setTime(int hour, int minute, int second) {

LocalDate currentDate = LocalDate.from(dateAndTime);

LocalTime timeToSet = LocalTime.of(hour, minute, second);

dateAndTime = LocalDateTime.of(currentDate, timeToSet);

}

public void setDate(int day, int month, int year) {

LocalDate dateToSet = LocalDate.of(day, month, year);

LocalTime currentTime = LocalTime.from(dateAndTime);

dateAndTime = LocalDateTime.of(dateToSet, currentTime);

}

public void setDateAndTime(int day, int month, int year,

int hour, int minute, int second) {

LocalDate dateToSet = LocalDate.of(day, month, year);

LocalTime timeToSet = LocalTime.of(hour, minute, second);

dateAndTime = LocalDateTime.of(dateToSet, timeToSet);

}

public LocalDateTime getLocalDateTime() {

return dateAndTime;

}

public String toString() {

return dateAndTime.toString();

}

public static void main(String... args) {

TimeClient myTimeClient = new SimpleTimeClient();

System.out.println(myTimeClient.toString());

}

}

Suppose that you want to add new functionality to the TimeClient interface, such as the ability to specify a time zone through a [ZonedDateTime](http://docs.oracle.com/javase/8/docs/api/java/time/ZonedDateTime.html) object (which is like a [LocalDateTime](http://docs.oracle.com/javase/8/docs/api/java/time/LocalDateTime.html)object except that it stores time zone information):

public interface TimeClient {

void setTime(int hour, int minute, int second);

void setDate(int day, int month, int year);

void setDateAndTime(int day, int month, int year,

int hour, int minute, int second);

LocalDateTime getLocalDateTime();

**ZonedDateTime getZonedDateTime(String zoneString);**

}

Following this modification to the TimeClient interface, you would also have to modify the class SimpleTimeClient and implement the method getZonedDateTime. However, rather than leaving getZonedDateTime as abstract (as in the previous example), you can instead define a default implementation. (Remember that an [abstract method](http://docs.oracle.com/javase/tutorial/java/IandI/abstract.html) is a method declared without an implementation.)

package defaultmethods;

import java.time.\*;

public interface TimeClient {

void setTime(int hour, int minute, int second);

void setDate(int day, int month, int year);

void setDateAndTime(int day, int month, int year,

int hour, int minute, int second);

LocalDateTime getLocalDateTime();

static ZoneId getZoneId (String zoneString) {

try {

return ZoneId.of(zoneString);

} catch (DateTimeException e) {

System.err.println("Invalid time zone: " + zoneString +

"; using default time zone instead.");

return ZoneId.systemDefault();

}

}

default ZonedDateTime getZonedDateTime(String zoneString) {

return ZonedDateTime.of(getLocalDateTime(), getZoneId(zoneString));

}

}

You specify that a method definition in an interface is a default method with the default keyword at the beginning of the method signature. All method declarations in an interface, including default methods, are implicitly public, so you can omit the public modifier.

With this interface, you do not have to modify the class SimpleTimeClient, and this class (and any class that implements the interface TimeClient), will have the methodgetZonedDateTime already defined. The following example, [TestSimpleTimeClient](http://docs.oracle.com/javase/tutorial/java/IandI/examples/defaultmethods/TestSimpleTimeClient.java), invokes the method getZonedDateTime from an instance of SimpleTimeClient:

package defaultmethods;

import java.time.\*;

import java.lang.\*;

import java.util.\*;

public class TestSimpleTimeClient {

public static void main(String... args) {

TimeClient myTimeClient = new SimpleTimeClient();

System.out.println("Current time: " + myTimeClient.toString());

System.out.println("Time in California: " +

myTimeClient.getZonedDateTime("Blah blah").toString());

}

}

## Extending Interfaces That Contain Default Methods

When you extend an interface that contains a default method, you can do the following:

* Not mention the default method at all, which lets your extended interface inherit the default method.
* Redeclare the default method, which makes it abstract.
* Redefine the default method, which overrides it.

Suppose that you extend the interface TimeClient as follows:

public interface AnotherTimeClient extends TimeClient { }

Any class that implements the interface AnotherTimeClient will have the implementation specified by the default method TimeClient.getZonedDateTime.

Suppose that you extend the interface TimeClient as follows:

public interface AbstractZoneTimeClient extends TimeClient {

public ZonedDateTime getZonedDateTime(String zoneString);

}

Any class that implements the interface AbstractZoneTimeClient will have to implement the method getZonedDateTime; this method is an abstract method like all other nondefault (and nonstatic) methods in an interface.

Suppose that you extend the interface TimeClient as follows:

public interface HandleInvalidTimeZoneClient extends TimeClient {

default public ZonedDateTime getZonedDateTime(String zoneString) {

try {

return ZonedDateTime.of(getLocalDateTime(),ZoneId.of(zoneString));

} catch (DateTimeException e) {

System.err.println("Invalid zone ID: " + zoneString +

"; using the default time zone instead.");

return ZonedDateTime.of(getLocalDateTime(),ZoneId.systemDefault());

}

}

}

Any class that implements the interface HandleInvalidTimeZoneClient will use the implementation of getZonedDateTime specified by this interface instead of the one specified by the interface TimeClient.

## Static Methods

In addition to default methods, you can define [static methods](http://docs.oracle.com/javase/tutorial/java/javaOO/classvars.html)in interfaces. (A static method is a method that is associated with the class in which it is defined rather than with any object. Every instance of the class shares its static methods.) This makes it easier for you to organize helper methods in your libraries; you can keep static methods specific to an interface in the same interface rather than in a separate class. The following example defines a static method that retrieves a [ZoneId](http://docs.oracle.com/javase/8/docs/api/java/time/ZoneId.html) object corresponding to a time zone identifier; it uses the system default time zone if there is no ZoneId object corresponding to the given identifier. (As a result, you can simplify the method getZonedDateTime):

public interface TimeClient {

// ...

static public ZoneId getZoneId (String zoneString) {

try {

return ZoneId.of(zoneString);

} catch (DateTimeException e) {

System.err.println("Invalid time zone: " + zoneString +

"; using default time zone instead.");

return ZoneId.systemDefault();

}

}

default public ZonedDateTime getZonedDateTime(String zoneString) {

return ZonedDateTime.of(getLocalDateTime(), getZoneId(zoneString));

}

}

Like static methods in classes, you specify that a method definition in an interface is a static method with the static keyword at the beginning of the method signature. All method declarations in an interface, including static methods, are implicitly public, so you can omit the public modifier.

## Integrating Default Methods into Existing Libraries

Default methods enable you to add new functionality to existing interfaces and ensure binary compatibility with code written for older versions of those interfaces. In particular, default methods enable you to add methods that accept lambda expressions as parameters to existing interfaces. This section demonstrates how the [Comparator](http://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html) interface has been enhanced with default and static methods.

Consider the Card and Deck classes as described in [Questions and Exercises: Classes](http://docs.oracle.com/javase/tutorial/java/javaOO/QandE/creating-questions.html). This example rewrites the [Card](http://docs.oracle.com/javase/tutorial/java/IandI/examples/defaultmethods/Card.java) and [Deck](http://docs.oracle.com/javase/tutorial/java/IandI/examples/defaultmethods/Deck.java) classes as interfaces. The Card interface contains two enumtypes (Suit and Rank) and two abstract methods (getSuit and getRank):

package defaultmethods;

public interface Card extends Comparable<Card> {

public enum Suit {

DIAMONDS (1, "Diamonds"),

CLUBS (2, "Clubs" ),

HEARTS (3, "Hearts" ),

SPADES (4, "Spades" );

private final int value;

private final String text;

Suit(int value, String text) {

this.value = value;

this.text = text;

}

public int value() {return value;}

public String text() {return text;}

}

public enum Rank {

DEUCE (2 , "Two" ),

THREE (3 , "Three"),

FOUR (4 , "Four" ),

FIVE (5 , "Five" ),

SIX (6 , "Six" ),

SEVEN (7 , "Seven"),

EIGHT (8 , "Eight"),

NINE (9 , "Nine" ),

TEN (10, "Ten" ),

JACK (11, "Jack" ),

QUEEN (12, "Queen"),

KING (13, "King" ),

ACE (14, "Ace" );

private final int value;

private final String text;

Rank(int value, String text) {

this.value = value;

this.text = text;

}

public int value() {return value;}

public String text() {return text;}

}

public Card.Suit getSuit();

public Card.Rank getRank();

}

The Deck interface contains various methods that manipulate cards in a deck:

package defaultmethods;

import java.util.\*;

import java.util.stream.\*;

import java.lang.\*;

public interface Deck {

List<Card> getCards();

Deck deckFactory();

int size();

void addCard(Card card);

void addCards(List<Card> cards);

void addDeck(Deck deck);

void shuffle();

void sort();

void sort(Comparator<Card> c);

String deckToString();

Map<Integer, Deck> deal(int players, int numberOfCards)

throws IllegalArgumentException;

}

The class [PlayingCard](http://docs.oracle.com/javase/tutorial/java/IandI/examples/defaultmethods/PlayingCard.java) implements the interface Card, and the class [StandardDeck](http://docs.oracle.com/javase/tutorial/java/IandI/examples/defaultmethods/StandardDeck.java) implements the interface Deck.

The class StandardDeck implements the abstract method Deck.sort as follows:

public class StandardDeck implements Deck {

private List<Card> entireDeck;

// ...

public void sort() {

Collections.sort(entireDeck);

}

// ...

}

The method Collections.sort sorts an instance of List whose element type implements the interface [Comparable](http://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html). The member entireDeck is an instance of List whose elements are of the type Card, which extends Comparable. The class PlayingCard implements the [Comparable.compareTo](http://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html#compareTo-T-) method as follows:

public int hashCode() {

return ((suit.value()-1)\*13)+rank.value();

}

public int compareTo(Card o) {

return this.hashCode() - o.hashCode();

}

The method compareTo causes the method StandardDeck.sort() to sort the deck of cards first by suit, and then by rank.

What if you want to sort the deck first by rank, then by suit? You would need to implement the [Comparator](http://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html) interface to specify new sorting criteria, and use the method [sort(List<T> list, Comparator<? super T> c)](http://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#sort-java.util.List-java.util.Comparator-) (the version of the sort method that includes a Comparator parameter). You can define the following method in the class StandardDeck:

public void sort(Comparator<Card> c) {

Collections.sort(entireDeck, c);

}

With this method, you can specify how the method Collections.sort sorts instances of the Card class. One way to do this is to implement the Comparator interface to specify how you want the cards sorted. The example [SortByRankThenSuit](http://docs.oracle.com/javase/tutorial/java/IandI/examples/defaultmethods/SortByRankThenSuit.java) does this:

package defaultmethods;

import java.util.\*;

import java.util.stream.\*;

import java.lang.\*;

public class SortByRankThenSuit implements Comparator<Card> {

public int compare(Card firstCard, Card secondCard) {

int compVal =

firstCard.getRank().value() - secondCard.getRank().value();

if (compVal != 0)

return compVal;

else

return firstCard.getSuit().value() - secondCard.getSuit().value();

}

}

The following invocation sorts the deck of playing cards first by rank, then by suit:

StandardDeck myDeck = new StandardDeck();

myDeck.shuffle();

myDeck.sort(new SortByRankThenSuit());

However, this approach is too verbose; it would be better if you could specify what you want to sort, not how you want to sort. Suppose that you are the developer who wrote the Comparatorinterface. What default or static methods could you add to the Comparator interface to enable other developers to more easily specify sort criteria?

To start, suppose that you want to sort the deck of playing cards by rank, regardless of suit. You can invoke the StandardDeck.sort method as follows:

StandardDeck myDeck = new StandardDeck();

myDeck.shuffle();

myDeck.sort(

(firstCard, secondCard) ->

firstCard.getRank().value() - secondCard.getRank().value()

);

Because the interface Comparator is a [functional interface](http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html#approach6), you can use a lambda expression as an argument for the sort method. In this example, the lambda expression compares two integer values.

It would be simpler for your developers if they could create a Comparator instance by invoking the method Card.getRank only. In particular, it would be helpful if your developers could create a Comparator instance that compares any object that can return a numerical value from a method such as getValue or hashCode. The Comparator interface has been enhanced with this ability with the static method [comparing](http://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html#comparing-java.util.function.Function-java.util.Comparator-):

myDeck.sort(Comparator.comparing((card) -> card.getRank()));

In this example, you can use a [method reference](http://docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html) instead:

myDeck.sort(Comparator.comparing(Card::getRank));

This invocation better demonstrates what to sort rather than how to do it.

The Comparator interface has been enhanced with other versions of the static method comparing such as [comparingDouble](http://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html#comparingDouble-java.util.function.ToDoubleFunction-java.util.Comparator-) and [comparingLong](http://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html#comparingLong-java.util.function.ToLongFunction-) that enable you to createComparator instances that compare other data types.

Suppose that your developers would like to create a Comparator instance that could compare objects with more than one criteria. For example, how would you sort the deck of playing cards first by rank, and then by suit? As before, you could use a lambda expression to specify these sort criteria:

StandardDeck myDeck = new StandardDeck();

myDeck.shuffle();

myDeck.sort(

(firstCard, secondCard) -> {

int compare =

firstCard.getRank().value() - secondCard.getRank().value();

if (compare != 0)

return compare;

else

return firstCard.getSuit().value() - secondCard.getSuit().value();

}

);

It would be simpler for your developers if they could build a Comparator instance from a series of Comparator instances. The Comparator interface has been enhanced with this ability with the default method [thenComparing](http://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html#thenComparing-java.util.Comparator-):

myDeck.sort(

Comparator

.comparing(Card::getRank)

.thenComparing(Comparator.comparing(Card::getSuit)));

The Comparator interface has been enhanced with other versions of the default method thenComparing (such as [thenComparingDouble](http://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html#thenComparingDouble-java.util.function.ToDoubleFunction-) and [thenComparingLong](http://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html#thenComparingLong-java.util.function.ToLongFunction-)) that enable you to build Comparator instances that compare other data types.

Suppose that your developers would like to create a Comparator instance that enables them to sort a collection of objects in reverse order. For example, how would you sort the deck of playing cards first by descending order of rank, from Ace to Two (instead of from Two to Ace)? As before, you could specify another lambda expression. However, it would be simpler for your developers if they could reverse an existing Comparator by invoking a method. The Comparator interface has been enhanced with this ability with the default method [reversed](http://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html#reversed--):

myDeck.sort(

Comparator.comparing(Card::getRank)

.reversed()

.thenComparing(Comparator.comparing(Card::getSuit)));

This example demonstrates how the Comparator interface has been enhanced with default methods, static methods, lambda expressions, and method references to create more expressive library methods whose functionality programmers can quickly deduce by looking at how they are invoked. Use these constructs to enhance the interfaces in your libraries.

# Default Methods

The section [Interfaces](http://docs.oracle.com/javase/tutorial/java/IandI/createinterface.html) describes an example that involves manufacturers of computer-controlled cars who publish industry-standard interfaces that describe which methods can be invoked to operate their cars. What if those computer-controlled car manufacturers add new functionality, such as flight, to their cars? These manufacturers would need to specify new methods to enable other companies (such as electronic guidance instrument manufacturers) to adapt their software to flying cars. Where would these car manufacturers declare these new flight-related methods? If they add them to their original interfaces, then programmers who have implemented those interfaces would have to rewrite their implementations. If they add them as static methods, then programmers would regard them as utility methods, not as essential, core methods.

Default methods enable you to add new functionality to the interfaces of your libraries and ensure binary compatibility with code written for older versions of those interfaces.

Consider the following interface, [TimeClient](http://docs.oracle.com/javase/tutorial/java/IandI/examples/TimeClient.java), as described in [Answers to Questions and Exercises: Interfaces](http://docs.oracle.com/javase/tutorial/java/IandI/QandE/interfaces-answers.html):

import java.time.\*;

public interface TimeClient {

void setTime(int hour, int minute, int second);

void setDate(int day, int month, int year);

void setDateAndTime(int day, int month, int year,

int hour, int minute, int second);

LocalDateTime getLocalDateTime();

}

The following class, [SimpleTimeClient](http://docs.oracle.com/javase/tutorial/java/IandI/examples/defaultmethods/SimpleTimeClient.java), implements TimeClient:

package defaultmethods;

import java.time.\*;

import java.lang.\*;

import java.util.\*;

public class SimpleTimeClient implements TimeClient {

private LocalDateTime dateAndTime;

public SimpleTimeClient() {

dateAndTime = LocalDateTime.now();

}

public void setTime(int hour, int minute, int second) {

LocalDate currentDate = LocalDate.from(dateAndTime);

LocalTime timeToSet = LocalTime.of(hour, minute, second);

dateAndTime = LocalDateTime.of(currentDate, timeToSet);

}

public void setDate(int day, int month, int year) {

LocalDate dateToSet = LocalDate.of(day, month, year);

LocalTime currentTime = LocalTime.from(dateAndTime);

dateAndTime = LocalDateTime.of(dateToSet, currentTime);

}

public void setDateAndTime(int day, int month, int year,

int hour, int minute, int second) {

LocalDate dateToSet = LocalDate.of(day, month, year);

LocalTime timeToSet = LocalTime.of(hour, minute, second);

dateAndTime = LocalDateTime.of(dateToSet, timeToSet);

}

public LocalDateTime getLocalDateTime() {

return dateAndTime;

}

public String toString() {

return dateAndTime.toString();

}

public static void main(String... args) {

TimeClient myTimeClient = new SimpleTimeClient();

System.out.println(myTimeClient.toString());

}

}

Suppose that you want to add new functionality to the TimeClient interface, such as the ability to specify a time zone through a [ZonedDateTime](http://docs.oracle.com/javase/8/docs/api/java/time/ZonedDateTime.html) object (which is like a [LocalDateTime](http://docs.oracle.com/javase/8/docs/api/java/time/LocalDateTime.html)object except that it stores time zone information):

public interface TimeClient {

void setTime(int hour, int minute, int second);

void setDate(int day, int month, int year);

void setDateAndTime(int day, int month, int year,

int hour, int minute, int second);

LocalDateTime getLocalDateTime();

**ZonedDateTime getZonedDateTime(String zoneString);**

}

Following this modification to the TimeClient interface, you would also have to modify the class SimpleTimeClient and implement the method getZonedDateTime. However, rather than leaving getZonedDateTime as abstract (as in the previous example), you can instead define a default implementation. (Remember that an [abstract method](http://docs.oracle.com/javase/tutorial/java/IandI/abstract.html) is a method declared without an implementation.)

package defaultmethods;

import java.time.\*;

public interface TimeClient {

void setTime(int hour, int minute, int second);

void setDate(int day, int month, int year);

void setDateAndTime(int day, int month, int year,

int hour, int minute, int second);

LocalDateTime getLocalDateTime();

static ZoneId getZoneId (String zoneString) {

try {

return ZoneId.of(zoneString);

} catch (DateTimeException e) {

System.err.println("Invalid time zone: " + zoneString +

"; using default time zone instead.");

return ZoneId.systemDefault();

}

}

default ZonedDateTime getZonedDateTime(String zoneString) {

return ZonedDateTime.of(getLocalDateTime(), getZoneId(zoneString));

}

}

You specify that a method definition in an interface is a default method with the default keyword at the beginning of the method signature. All method declarations in an interface, including default methods, are implicitly public, so you can omit the public modifier.

With this interface, you do not have to modify the class SimpleTimeClient, and this class (and any class that implements the interface TimeClient), will have the methodgetZonedDateTime already defined. The following example, [TestSimpleTimeClient](http://docs.oracle.com/javase/tutorial/java/IandI/examples/defaultmethods/TestSimpleTimeClient.java), invokes the method getZonedDateTime from an instance of SimpleTimeClient:

package defaultmethods;

import java.time.\*;

import java.lang.\*;

import java.util.\*;

public class TestSimpleTimeClient {

public static void main(String... args) {

TimeClient myTimeClient = new SimpleTimeClient();

System.out.println("Current time: " + myTimeClient.toString());

System.out.println("Time in California: " +

myTimeClient.getZonedDateTime("Blah blah").toString());

}

}

## Extending Interfaces That Contain Default Methods

When you extend an interface that contains a default method, you can do the following:

* Not mention the default method at all, which lets your extended interface inherit the default method.
* Redeclare the default method, which makes it abstract.
* Redefine the default method, which overrides it.

Suppose that you extend the interface TimeClient as follows:

public interface AnotherTimeClient extends TimeClient { }

Any class that implements the interface AnotherTimeClient will have the implementation specified by the default method TimeClient.getZonedDateTime.

Suppose that you extend the interface TimeClient as follows:

public interface AbstractZoneTimeClient extends TimeClient {

public ZonedDateTime getZonedDateTime(String zoneString);

}

Any class that implements the interface AbstractZoneTimeClient will have to implement the method getZonedDateTime; this method is an abstract method like all other nondefault (and nonstatic) methods in an interface.

Suppose that you extend the interface TimeClient as follows:

public interface HandleInvalidTimeZoneClient extends TimeClient {

default public ZonedDateTime getZonedDateTime(String zoneString) {

try {

return ZonedDateTime.of(getLocalDateTime(),ZoneId.of(zoneString));

} catch (DateTimeException e) {

System.err.println("Invalid zone ID: " + zoneString +

"; using the default time zone instead.");

return ZonedDateTime.of(getLocalDateTime(),ZoneId.systemDefault());

}

}

}

Any class that implements the interface HandleInvalidTimeZoneClient will use the implementation of getZonedDateTime specified by this interface instead of the one specified by the interface TimeClient.

## Static Methods

In addition to default methods, you can define [static methods](http://docs.oracle.com/javase/tutorial/java/javaOO/classvars.html)in interfaces. (A static method is a method that is associated with the class in which it is defined rather than with any object. Every instance of the class shares its static methods.) This makes it easier for you to organize helper methods in your libraries; you can keep static methods specific to an interface in the same interface rather than in a separate class. The following example defines a static method that retrieves a [ZoneId](http://docs.oracle.com/javase/8/docs/api/java/time/ZoneId.html) object corresponding to a time zone identifier; it uses the system default time zone if there is no ZoneId object corresponding to the given identifier. (As a result, you can simplify the method getZonedDateTime):

public interface TimeClient {

// ...

static public ZoneId getZoneId (String zoneString) {

try {

return ZoneId.of(zoneString);

} catch (DateTimeException e) {

System.err.println("Invalid time zone: " + zoneString +

"; using default time zone instead.");

return ZoneId.systemDefault();

}

}

default public ZonedDateTime getZonedDateTime(String zoneString) {

return ZonedDateTime.of(getLocalDateTime(), getZoneId(zoneString));

}

}

Like static methods in classes, you specify that a method definition in an interface is a static method with the static keyword at the beginning of the method signature. All method declarations in an interface, including static methods, are implicitly public, so you can omit the public modifier.

## Integrating Default Methods into Existing Libraries

Default methods enable you to add new functionality to existing interfaces and ensure binary compatibility with code written for older versions of those interfaces. In particular, default methods enable you to add methods that accept lambda expressions as parameters to existing interfaces. This section demonstrates how the [Comparator](http://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html) interface has been enhanced with default and static methods.

Consider the Card and Deck classes as described in [Questions and Exercises: Classes](http://docs.oracle.com/javase/tutorial/java/javaOO/QandE/creating-questions.html). This example rewrites the [Card](http://docs.oracle.com/javase/tutorial/java/IandI/examples/defaultmethods/Card.java) and [Deck](http://docs.oracle.com/javase/tutorial/java/IandI/examples/defaultmethods/Deck.java) classes as interfaces. The Card interface contains two enumtypes (Suit and Rank) and two abstract methods (getSuit and getRank):

package defaultmethods;

public interface Card extends Comparable<Card> {

public enum Suit {

DIAMONDS (1, "Diamonds"),

CLUBS (2, "Clubs" ),

HEARTS (3, "Hearts" ),

SPADES (4, "Spades" );

private final int value;

private final String text;

Suit(int value, String text) {

this.value = value;

this.text = text;

}

public int value() {return value;}

public String text() {return text;}

}

public enum Rank {

DEUCE (2 , "Two" ),

THREE (3 , "Three"),

FOUR (4 , "Four" ),

FIVE (5 , "Five" ),

SIX (6 , "Six" ),

SEVEN (7 , "Seven"),

EIGHT (8 , "Eight"),

NINE (9 , "Nine" ),

TEN (10, "Ten" ),

JACK (11, "Jack" ),

QUEEN (12, "Queen"),

KING (13, "King" ),

ACE (14, "Ace" );

private final int value;

private final String text;

Rank(int value, String text) {

this.value = value;

this.text = text;

}

public int value() {return value;}

public String text() {return text;}

}

public Card.Suit getSuit();

public Card.Rank getRank();

}

The Deck interface contains various methods that manipulate cards in a deck:

package defaultmethods;

import java.util.\*;

import java.util.stream.\*;

import java.lang.\*;

public interface Deck {

List<Card> getCards();

Deck deckFactory();

int size();

void addCard(Card card);

void addCards(List<Card> cards);

void addDeck(Deck deck);

void shuffle();

void sort();

void sort(Comparator<Card> c);

String deckToString();

Map<Integer, Deck> deal(int players, int numberOfCards)

throws IllegalArgumentException;

}

The class [PlayingCard](http://docs.oracle.com/javase/tutorial/java/IandI/examples/defaultmethods/PlayingCard.java) implements the interface Card, and the class [StandardDeck](http://docs.oracle.com/javase/tutorial/java/IandI/examples/defaultmethods/StandardDeck.java) implements the interface Deck.

The class StandardDeck implements the abstract method Deck.sort as follows:

public class StandardDeck implements Deck {

private List<Card> entireDeck;

// ...

public void sort() {

Collections.sort(entireDeck);

}

// ...

}

The method Collections.sort sorts an instance of List whose element type implements the interface [Comparable](http://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html). The member entireDeck is an instance of List whose elements are of the type Card, which extends Comparable. The class PlayingCard implements the [Comparable.compareTo](http://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html#compareTo-T-) method as follows:

public int hashCode() {

return ((suit.value()-1)\*13)+rank.value();

}

public int compareTo(Card o) {

return this.hashCode() - o.hashCode();

}

The method compareTo causes the method StandardDeck.sort() to sort the deck of cards first by suit, and then by rank.

What if you want to sort the deck first by rank, then by suit? You would need to implement the [Comparator](http://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html) interface to specify new sorting criteria, and use the method [sort(List<T> list, Comparator<? super T> c)](http://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#sort-java.util.List-java.util.Comparator-) (the version of the sort method that includes a Comparator parameter). You can define the following method in the class StandardDeck:

public void sort(Comparator<Card> c) {

Collections.sort(entireDeck, c);

}

With this method, you can specify how the method Collections.sort sorts instances of the Card class. One way to do this is to implement the Comparator interface to specify how you want the cards sorted. The example [SortByRankThenSuit](http://docs.oracle.com/javase/tutorial/java/IandI/examples/defaultmethods/SortByRankThenSuit.java) does this:

package defaultmethods;

import java.util.\*;

import java.util.stream.\*;

import java.lang.\*;

public class SortByRankThenSuit implements Comparator<Card> {

public int compare(Card firstCard, Card secondCard) {

int compVal =

firstCard.getRank().value() - secondCard.getRank().value();

if (compVal != 0)

return compVal;

else

return firstCard.getSuit().value() - secondCard.getSuit().value();

}

}

The following invocation sorts the deck of playing cards first by rank, then by suit:

StandardDeck myDeck = new StandardDeck();

myDeck.shuffle();

myDeck.sort(new SortByRankThenSuit());

However, this approach is too verbose; it would be better if you could specify what you want to sort, not how you want to sort. Suppose that you are the developer who wrote the Comparatorinterface. What default or static methods could you add to the Comparator interface to enable other developers to more easily specify sort criteria?

To start, suppose that you want to sort the deck of playing cards by rank, regardless of suit. You can invoke the StandardDeck.sort method as follows:

StandardDeck myDeck = new StandardDeck();

myDeck.shuffle();

myDeck.sort(

(firstCard, secondCard) ->

firstCard.getRank().value() - secondCard.getRank().value()

);

Because the interface Comparator is a [functional interface](http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html#approach6), you can use a lambda expression as an argument for the sort method. In this example, the lambda expression compares two integer values.

It would be simpler for your developers if they could create a Comparator instance by invoking the method Card.getRank only. In particular, it would be helpful if your developers could create a Comparator instance that compares any object that can return a numerical value from a method such as getValue or hashCode. The Comparator interface has been enhanced with this ability with the static method [comparing](http://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html#comparing-java.util.function.Function-java.util.Comparator-):

myDeck.sort(Comparator.comparing((card) -> card.getRank()));

In this example, you can use a [method reference](http://docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html) instead:

myDeck.sort(Comparator.comparing(Card::getRank));

This invocation better demonstrates what to sort rather than how to do it.

The Comparator interface has been enhanced with other versions of the static method comparing such as [comparingDouble](http://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html#comparingDouble-java.util.function.ToDoubleFunction-java.util.Comparator-) and [comparingLong](http://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html#comparingLong-java.util.function.ToLongFunction-) that enable you to createComparator instances that compare other data types.

Suppose that your developers would like to create a Comparator instance that could compare objects with more than one criteria. For example, how would you sort the deck of playing cards first by rank, and then by suit? As before, you could use a lambda expression to specify these sort criteria:

StandardDeck myDeck = new StandardDeck();

myDeck.shuffle();

myDeck.sort(

(firstCard, secondCard) -> {

int compare =

firstCard.getRank().value() - secondCard.getRank().value();

if (compare != 0)

return compare;

else

return firstCard.getSuit().value() - secondCard.getSuit().value();

}

);

It would be simpler for your developers if they could build a Comparator instance from a series of Comparator instances. The Comparator interface has been enhanced with this ability with the default method [thenComparing](http://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html#thenComparing-java.util.Comparator-):

myDeck.sort(

Comparator

.comparing(Card::getRank)

.thenComparing(Comparator.comparing(Card::getSuit)));

The Comparator interface has been enhanced with other versions of the default method thenComparing (such as [thenComparingDouble](http://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html#thenComparingDouble-java.util.function.ToDoubleFunction-) and [thenComparingLong](http://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html#thenComparingLong-java.util.function.ToLongFunction-)) that enable you to build Comparator instances that compare other data types.

Suppose that your developers would like to create a Comparator instance that enables them to sort a collection of objects in reverse order. For example, how would you sort the deck of playing cards first by descending order of rank, from Ace to Two (instead of from Two to Ace)? As before, you could specify another lambda expression. However, it would be simpler for your developers if they could reverse an existing Comparator by invoking a method. The Comparator interface has been enhanced with this ability with the default method [reversed](http://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html#reversed--):

myDeck.sort(

Comparator.comparing(Card::getRank)

.reversed()

.thenComparing(Comparator.comparing(Card::getSuit)));

This example demonstrates how the Comparator interface has been enhanced with default methods, static methods, lambda expressions, and method references to create more expressive library methods whose functionality programmers can quickly deduce by looking at how they are invoked. Use these constructs to enhance the interfaces in your libraries.

# Inheritance

In the preceding lessons, you have seen *inheritance* mentioned several times. In the Java language, classes can be *derived* from other classes, thereby *inheriting* fields and methods from those classes.

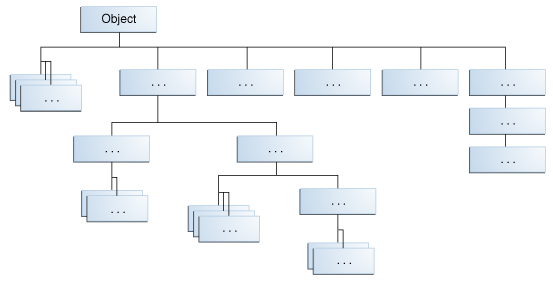
**Definitions:** A class that is derived from another class is called a *subclass* (also a *derived class*, *extended class*, or *child class*). The class from which the subclass is derived is called a*superclass* (also a *base class* or a *parent class*).  
  
Excepting Object, which has no superclass, every class has one and only one direct superclass (single inheritance). In the absence of any other explicit superclass, every class is implicitly a subclass of Object.  
  
Classes can be derived from classes that are derived from classes that are derived from classes, and so on, and ultimately derived from the topmost class, Object. Such a class is said to be *descended* from all the classes in the inheritance chain stretching back to Object.

The idea of inheritance is simple but powerful: When you want to create a new class and there is already a class that includes some of the code that you want, you can derive your new class from the existing class. In doing this, you can reuse the fields and methods of the existing class without having to write (and debug!) them yourself.

A subclass inherits all the *members* (fields, methods, and nested classes) from its superclass. Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass.

## The Java Platform Class Hierarchy

The [Object](http://docs.oracle.com/javase/8/docs/api/java/lang/Object.html) class, defined in the java.lang package, defines and implements behavior common to all classes—including the ones that you write. In the Java platform, many classes derive directly from Object, other classes derive from some of those classes, and so on, forming a hierarchy of classes.



All Classes in the Java Platform are Descendants of Object

At the top of the hierarchy, Object is the most general of all classes. Classes near the bottom of the hierarchy provide more specialized behavior.

## An Example of Inheritance

Here is the sample code for a possible implementation of a Bicycle class that was presented in the Classes and Objects lesson:

public class Bicycle {

// **the Bicycle class has three *fields***

public int cadence;

public int gear;

public int speed;

// **the Bicycle class has one *constructor***

public Bicycle(int startCadence, int startSpeed, int startGear) {

gear = startGear;

cadence = startCadence;

speed = startSpeed;

}

// **the Bicycle class has four *methods***

public void setCadence(int newValue) {

cadence = newValue;

}

public void setGear(int newValue) {

gear = newValue;

}

public void applyBrake(int decrement) {

speed -= decrement;

}

public void speedUp(int increment) {

speed += increment;

}

}

A class declaration for a MountainBike class that is a subclass of Bicycle might look like this:

public class MountainBike extends Bicycle {

// **the MountainBike subclass adds one *field***

public int seatHeight;

// **the MountainBike subclass has one *constructor***

public MountainBike(int startHeight,

int startCadence,

int startSpeed,

int startGear) {

super(startCadence, startSpeed, startGear);

seatHeight = startHeight;

}

// **the MountainBike subclass adds one *method***

public void setHeight(int newValue) {

seatHeight = newValue;

}

}

MountainBike inherits all the fields and methods of Bicycle and adds the field seatHeight and a method to set it. Except for the constructor, it is as if you had written a newMountainBike class entirely from scratch, with four fields and five methods. However, you didn't have to do all the work. This would be especially valuable if the methods in the Bicycleclass were complex and had taken substantial time to debug.

## What You Can Do in a Subclass

A subclass inherits all of the *public* and *protected* members of its parent, no matter what package the subclass is in. If the subclass is in the same package as its parent, it also inherits the*package-private* members of the parent. You can use the inherited members as is, replace them, hide them, or supplement them with new members:

* The inherited fields can be used directly, just like any other fields.
* You can declare a field in the subclass with the same name as the one in the superclass, thus *hiding* it (not recommended).
* You can declare new fields in the subclass that are not in the superclass.
* The inherited methods can be used directly as they are.
* You can write a new *instance* method in the subclass that has the same signature as the one in the superclass, thus *overriding* it.
* You can write a new *static* method in the subclass that has the same signature as the one in the superclass, thus *hiding* it.
* You can declare new methods in the subclass that are not in the superclass.
* You can write a subclass constructor that invokes the constructor of the superclass, either implicitly or by using the keyword super.

The following sections in this lesson will expand on these topics.

## Private Members in a Superclass

A subclass does not inherit the private members of its parent class. However, if the superclass has public or protected methods for accessing its private fields, these can also be used by the subclass.

A nested class has access to all the private members of its enclosing class—both fields and methods. Therefore, a public or protected nested class inherited by a subclass has indirect access to all of the private members of the superclass.

## Casting Objects

We have seen that an object is of the data type of the class from which it was instantiated. For example, if we write

public MountainBike myBike = new MountainBike();

then myBike is of type MountainBike.

MountainBike is descended from Bicycle and Object. Therefore, a MountainBike is a Bicycle and is also an Object, and it can be used wherever Bicycle or Object objects are called for.

The reverse is not necessarily true: a Bicycle *may be* a MountainBike, but it isn't necessarily. Similarly, an Object *may be* a Bicycle or a MountainBike, but it isn't necessarily.

*Casting* shows the use of an object of one type in place of another type, among the objects permitted by inheritance and implementations. For example, if we write

Object obj = new MountainBike();

then obj is both an Object and a MountainBike (until such time as obj is assigned another object that is *not* a MountainBike). This is called *implicit casting*.

If, on the other hand, we write

MountainBike myBike = obj;

we would get a compile-time error because obj is not known to the compiler to be a MountainBike. However, we can *tell* the compiler that we promise to assign a MountainBike to obj by*explicit casting:*

MountainBike myBike = (MountainBike)obj;

This cast inserts a runtime check that obj is assigned a MountainBike so that the compiler can safely assume that obj is a MountainBike. If obj is not a MountainBike at runtime, an exception will be thrown.

**Note:** You can make a logical test as to the type of a particular object using the instanceof operator. This can save you from a runtime error owing to an improper cast. For example:

if (obj instanceof MountainBike) {

MountainBike myBike = (MountainBike)obj;

}

Here the instanceof operator verifies that obj refers to a MountainBike so that we can make the cast with knowledge that there will be no runtime exception thrown.

# Multiple Inheritance of State, Implementation, and Type

One significant difference between classes and interfaces is that classes can have fields whereas interfaces cannot. In addition, you can instantiate a class to create an object, which you cannot do with interfaces. As explained in the section [What Is an Object?](http://docs.oracle.com/javase/tutorial/java/concepts/object.html), an object stores its state in fields, which are defined in classes. One reason why the Java programming language does not permit you to extend more than one class is to avoid the issues of multiple inheritance of state, which is the ability to inherit fields from multiple classes. For example, suppose that you are able to define a new class that extends multiple classes. When you create an object by instantiating that class, that object will inherit fields from all of the class's superclasses. What if methods or constructors from different superclasses instantiate the same field? Which method or constructor will take precedence? Because interfaces do not contain fields, you do not have to worry about problems that result from multiple inheritance of state.

Multiple inheritance of implementation is the ability to inherit method definitions from multiple classes. Problems arise with this type of multiple inheritance, such as name conflicts and ambiguity. When compilers of programming languages that support this type of multiple inheritance encounter superclasses that contain methods with the same name, they sometimes cannot determine which member or method to access or invoke. In addition, a programmer can unwittingly introduce a name conflict by adding a new method to a superclass. [Default methods](http://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html) introduce one form of multiple inheritance of implementation. A class can implement more than one interface, which can contain default methods that have the same name. The Java compiler provides some rules to determine which default method a particular class uses.

The Java programming language supports multiple inheritance of type, which is the ability of a class to implement more than one interface. An object can have multiple types: the type of its own class and the types of all the interfaces that the class implements. This means that if a variable is declared to be the type of an interface, then its value can reference any object that is instantiated from any class that implements the interface. This is discussed in the section [Using an Interface as a Type](http://docs.oracle.com/javase/tutorial/java/IandI/interfaceAsType.html).

As with multiple inheritance of implementation, a class can inherit different implementations of a method defined (as default or static) in the interfaces that it extends. In this case, the compiler or the user must decide which one to use.

# Overriding and Hiding Methods

## Instance Methods

An instance method in a subclass with the same signature (name, plus the number and the type of its parameters) and return type as an instance method in the superclass *overrides* the superclass's method.

The ability of a subclass to override a method allows a class to inherit from a superclass whose behavior is "close enough" and then to modify behavior as needed. The overriding method has the same name, number and type of parameters, and return type as the method that it overrides. An overriding method can also return a subtype of the type returned by the overridden method. This subtype is called a *covariant return type*.

When overriding a method, you might want to use the @Override annotation that instructs the compiler that you intend to override a method in the superclass. If, for some reason, the compiler detects that the method does not exist in one of the superclasses, then it will generate an error. For more information on @Override, see [Annotations](http://docs.oracle.com/javase/tutorial/java/annotations/index.html).

## Static Methods

If a subclass defines a static method with the same signature as a static method in the superclass, then the method in the subclass *hides* the one in the superclass.

The distinction between hiding a static method and overriding an instance method has important implications:

* The version of the overridden instance method that gets invoked is the one in the subclass.
* The version of the hidden static method that gets invoked depends on whether it is invoked from the superclass or the subclass.

Consider an example that contains two classes. The first is Animal, which contains one instance method and one static method:

public class Animal {

public static void testClassMethod() {

System.out.println("The static method in Animal");

}

public void testInstanceMethod() {

System.out.println("The instance method in Animal");

}

}

The second class, a subclass of Animal, is called Cat:

public class Cat extends Animal {

public static void testClassMethod() {

System.out.println("The static method in Cat");

}

public void testInstanceMethod() {

System.out.println("The instance method in Cat");

}

public static void main(String[] args) {

Cat myCat = new Cat();

Animal myAnimal = myCat;

Animal.testClassMethod();

myAnimal.testInstanceMethod();

}

}

The Cat class overrides the instance method in Animal and hides the static method in Animal. The main method in this class creates an instance of Cat and invokes testClassMethod()on the class and testInstanceMethod() on the instance.

The output from this program is as follows:

The static method in Animal

The instance method in Cat

As promised, the version of the hidden static method that gets invoked is the one in the superclass, and the version of the overridden instance method that gets invoked is the one in the subclass.

## Interface Methods

[Default methods](http://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html) and [abstract methods](http://docs.oracle.com/javase/tutorial/java/IandI/abstract.html) in interfaces are inherited like instance methods. However, when the supertypes of a class or interface provide multiple default methods with the same signature, the Java compiler follows inheritance rules to resolve the name conflict. These rules are driven by the following two principles:

* Instance methods are preferred over interface default methods.

Consider the following classes and interfaces:

public class Horse {

public String identifyMyself() {

return "I am a horse.";

}

}

public interface Flyer {

default public String identifyMyself() {

return "I am able to fly.";

}

}

public interface Mythical {

default public String identifyMyself() {

return "I am a mythical creature.";

}

}

public class Pegasus extends Horse implements Flyer, Mythical {

public static void main(String... args) {

Pegasus myApp = new Pegasus();

System.out.println(myApp.identifyMyself());

}

}

The method Pegasus.identifyMyself returns the string I am a horse.

* Methods that are already overridden by other candidates are ignored. This circumstance can arise when supertypes share a common ancestor.

Consider the following interfaces and classes:

public interface Animal {

default public String identifyMyself() {

return "I am an animal.";

}

}

public interface EggLayer extends Animal {

default public String identifyMyself() {

return "I am able to lay eggs.";

}

}

public interface FireBreather extends Animal { }

public class Dragon implements EggLayer, FireBreather {

public static void main (String... args) {

Dragon myApp = new Dragon();

System.out.println(myApp.identifyMyself());

}

}

The method Dragon.identifyMyself returns the string I am able to lay eggs.

If two or more independently defined default methods conflict, or a default method conflicts with an abstract method, then the Java compiler produces a compiler error. You must explicitly override the supertype methods.

Consider the example about computer-controlled cars that can now fly. You have two interfaces (OperateCar and FlyCar) that provide default implementations for the same method, (startEngine):

public interface OperateCar {

// ...

default public int startEngine(EncryptedKey key) {

// Implementation

}

}

public interface FlyCar {

// ...

default public int startEngine(EncryptedKey key) {

// Implementation

}

}

A class that implements both OperateCar and FlyCar must override the method startEngine. You could invoke any of the of the default implementations with the super keyword.

public class FlyingCar implements OperateCar, FlyCar {

// ...

public int startEngine(EncryptedKey key) {

FlyCar.super.startEngine(key);

OperateCar.super.startEngine(key);

}

}

The name preceding super (in this example, FlyCar or OperateCar) must refer to a direct superinterface that defines or inherits a default for the invoked method. This form of method invocation is not restricted to differentiating between multiple implemented interfaces that contain default methods with the same signature. You can use the super keyword to invoke a default method in both classes and interfaces.

Inherited instance methods from classes can override abstract interface methods. Consider the following interfaces and classes:

public interface Mammal {

String identifyMyself();

}

public class Horse {

public String identifyMyself() {

return "I am a horse.";

}

}

public class Mustang extends Horse implements Mammal {

public static void main(String... args) {

Mustang myApp = new Mustang();

System.out.println(myApp.identifyMyself());

}

}

The method Mustang.identifyMyself returns the string I am a horse. The class Mustang inherits the method identifyMyself from the class Horse, which overrides the abstract method of the same name in the interface Mammal.

**Note**: Static methods in interfaces are never inherited.

## Modifiers

The access specifier for an overriding method can allow more, but not less, access than the overridden method. For example, a protected instance method in the superclass can be made public, but not private, in the subclass.

You will get a compile-time error if you attempt to change an instance method in the superclass to a static method in the subclass, and vice versa.

## Summary

The following table summarizes what happens when you define a method with the same signature as a method in a superclass.

|  |  |  |
| --- | --- | --- |
| **Defining a Method with the Same Signature as a Superclass's Method** | | |
|  | **Superclass Instance Method** | **Superclass Static Method** |
| **Subclass Instance Method** | Overrides | Generates a compile-time error |
| **Subclass Static Method** | Generates a compile-time error | Hides |

**Note:** In a subclass, you can overload the methods inherited from the superclass. Such overloaded methods neither hide nor override the superclass instance methods—they are new methods, unique to the subclass.

# Polymorphism

The dictionary definition of polymorphism refers to a principle in biology in which an organism or species can have many different forms or stages. This principle can also be applied to object-oriented programming and languages like the Java language. Subclasses of a class can define their own unique behaviors and yet share some of the same functionality of the parent class.

Polymorphism can be demonstrated with a minor modification to the Bicycle class. For example, a printDescription method could be added to the class that displays all the data currently stored in an instance.

public void printDescription(){

System.out.println("\nBike is " + "in gear " + this.gear

+ " with a cadence of " + this.cadence +

" and travelling at a speed of " + this.speed + ". ");

}

To demonstrate polymorphic features in the Java language, extend the Bicycle class with a MountainBike and a RoadBike class. For MountainBike, add a field for suspension, which is a String value that indicates if the bike has a front shock absorber, Front. Or, the bike has a front and back shock absorber, Dual.

Here is the updated class:

public class MountainBike extends Bicycle {

private String suspension;

public MountainBike(

int startCadence,

int startSpeed,

int startGear,

String suspensionType){

super(startCadence,

startSpeed,

startGear);

this.setSuspension(suspensionType);

}

public String getSuspension(){

return this.suspension;

}

public void setSuspension(String suspensionType) {

this.suspension = suspensionType;

}

public void printDescription() {

super.printDescription();

System.out.println("The " + "MountainBike has a" +

getSuspension() + " suspension.");

}

}

Note the overridden printDescription method. In addition to the information provided before, additional data about the suspension is included to the output.

Next, create the RoadBike class. Because road or racing bikes have skinny tires, add an attribute to track the tire width. Here is the RoadBike class:

public class RoadBike extends Bicycle{

// In millimeters (mm)

private int tireWidth;

public RoadBike(int startCadence,

int startSpeed,

int startGear,

int newTireWidth){

super(startCadence,

startSpeed,

startGear);

this.setTireWidth(newTireWidth);

}

public int getTireWidth(){

return this.tireWidth;

}

public void setTireWidth(int newTireWidth){

this.tireWidth = newTireWidth;

}

public void printDescription(){

super.printDescription();

System.out.println("The RoadBike" + " has " + getTireWidth() +

" MM tires.");

}

}

Note that once again, the printDescription method has been overridden. This time, information about the tire width is displayed.

To summarize, there are three classes: Bicycle, MountainBike, and RoadBike. The two subclasses override the printDescription method and print unique information.

Here is a test program that creates three Bicycle variables. Each variable is assigned to one of the three bicycle classes. Each variable is then printed.

public class TestBikes {

public static void main(String[] args){

Bicycle bike01, bike02, bike03;

bike01 = new Bicycle(20, 10, 1);

bike02 = new MountainBike(20, 10, 5, "Dual");

bike03 = new RoadBike(40, 20, 8, 23);

bike01.printDescription();

bike02.printDescription();

bike03.printDescription();

}

}

The following is the output from the test program:

Bike is in gear 1 with a cadence of 20 and travelling at a speed of 10.

Bike is in gear 5 with a cadence of 20 and travelling at a speed of 10.

The MountainBike has a Dual suspension.

Bike is in gear 8 with a cadence of 40 and travelling at a speed of 20.

The RoadBike has 23 MM tires.

The Java virtual machine (JVM) calls the appropriate method for the object that is referred to in each variable. It does not call the method that is defined by the variable's type. This behavior is referred to as virtual method invocation and demonstrates an aspect of the important polymorphism features in the Java language.

# Hiding Fields

Within a class, a field that has the same name as a field in the superclass hides the superclass's field, even if their types are different. Within the subclass, the field in the superclass cannot be referenced by its simple name. Instead, the field must be accessed through super, which is covered in the next section. Generally speaking, we don't recommend hiding fields as it makes code difficult to read.

# Using the Keyword super

## Accessing Superclass Members

If your method overrides one of its superclass's methods, you can invoke the overridden method through the use of the keyword super. You can also use super to refer to a hidden field (although hiding fields is discouraged). Consider this class, Superclass:

public class Superclass {

public void printMethod() {

System.out.println("Printed in Superclass.");

}

}

Here is a subclass, called Subclass, that overrides printMethod():

public class Subclass extends Superclass {

// overrides printMethod in Superclass

public void printMethod() {

super.printMethod();

System.out.println("Printed in Subclass");

}

public static void main(String[] args) {

Subclass s = new Subclass();

s.printMethod();

}

}

Within Subclass, the simple name printMethod() refers to the one declared in Subclass, which overrides the one in Superclass. So, to refer to printMethod() inherited fromSuperclass, Subclass must use a qualified name, using super as shown. Compiling and executing Subclass prints the following:

Printed in Superclass.

Printed in Subclass

## Subclass Constructors

The following example illustrates how to use the super keyword to invoke a superclass's constructor. Recall from the [Bicycle](http://docs.oracle.com/javase/tutorial/java/IandI/subclasses.html)example that MountainBike is a subclass of Bicycle. Here is the MountainBike (subclass) constructor that calls the superclass constructor and then adds initialization code of its own:

public MountainBike(int startHeight,

int startCadence,

int startSpeed,

int startGear) {

super(startCadence, startSpeed, startGear);

seatHeight = startHeight;

}

Invocation of a superclass constructor must be the first line in the subclass constructor.

The syntax for calling a superclass constructor is

super();

or:

super(parameter list);

With super(), the superclass no-argument constructor is called. With super(parameter list), the superclass constructor with a matching parameter list is called.

**Note:** If a constructor does not explicitly invoke a superclass constructor, the Java compiler automatically inserts a call to the no-argument constructor of the superclass. If the super class does not have a no-argument constructor, you will get a compile-time error. Object *does* have such a constructor, so if Object is the only superclass, there is no problem.

If a subclass constructor invokes a constructor of its superclass, either explicitly or implicitly, you might think that there will be a whole chain of constructors called, all the way back to the constructor of Object. In fact, this is the case. It is called *constructor chaining*, and you need to be aware of it when there is a long line of class descent.

# Object as a Superclass

The [Object](http://docs.oracle.com/javase/8/docs/api/java/lang/Object.html) class, in the java.lang package, sits at the top of the class hierarchy tree. Every class is a descendant, direct or indirect, of the Object class. Every class you use or write inherits the instance methods of Object. You need not use any of these methods, but, if you choose to do so, you may need to override them with code that is specific to your class. The methods inherited from Object that are discussed in this section are:

* protected Object clone() throws CloneNotSupportedException  
        Creates and returns a copy of this object.
* public boolean equals(Object obj)  
        Indicates whether some other object is "equal to" this one.
* protected void finalize() throws Throwable  
        Called by the garbage collector on an object when garbage  
        collection determines that there are no more references to the object
* public final Class getClass()  
        Returns the runtime class of an object.
* public int hashCode()  
        Returns a hash code value for the object.
* public String toString()  
        Returns a string representation of the object.

The notify, notifyAll, and wait methods of Object all play a part in synchronizing the activities of independently running threads in a program, which is discussed in a later lesson and won't be covered here. There are five of these methods:

* public final void notify()
* public final void notifyAll()
* public final void wait()
* public final void wait(long timeout)
* public final void wait(long timeout, int nanos)

**Note:** There are some subtle aspects to a number of these methods, especially the clone method.

## The clone() Method

If a class, or one of its superclasses, implements the Cloneable interface, you can use the clone() method to create a copy from an existing object. To create a clone, you write:

*aCloneableObject*.clone();

Object's implementation of this method checks to see whether the object on which clone() was invoked implements the Cloneable interface. If the object does not, the method throws aCloneNotSupportedException exception. Exception handling will be covered in a later lesson. For the moment, you need to know that clone() must be declared as

protected Object clone() throws CloneNotSupportedException

or:

public Object clone() throws CloneNotSupportedException

if you are going to write a clone() method to override the one in Object.

If the object on which clone() was invoked does implement the Cloneable interface, Object's implementation of the clone() method creates an object of the same class as the original object and initializes the new object's member variables to have the same values as the original object's corresponding member variables.

The simplest way to make your class cloneable is to add implements Cloneable to your class's declaration. then your objects can invoke the clone() method.

For some classes, the default behavior of Object's clone() method works just fine. If, however, an object contains a reference to an external object, say ObjExternal, you may need to override clone() to get correct behavior. Otherwise, a change in ObjExternal made by one object will be visible in its clone also. This means that the original object and its clone are not independent—to decouple them, you must override clone() so that it clones the object *and* ObjExternal. Then the original object references ObjExternal and the clone references a clone of ObjExternal, so that the object and its clone are truly independent.

## The equals() Method

The equals() method compares two objects for equality and returns true if they are equal. The equals() method provided in the Object class uses the identity operator (==) to determine whether two objects are equal. For primitive data types, this gives the correct result. For objects, however, it does not. The equals() method provided by Object tests whether the object*references* are equal—that is, if the objects compared are the exact same object.

To test whether two objects are equal in the sense of *equivalency* (containing the same information), you must override the equals() method. Here is an example of a Book class that overrides equals():

public class Book {

...

public boolean equals(Object obj) {

if (obj instanceof Book)

return ISBN.equals((Book)obj.getISBN());

else

return false;

}

}

Consider this code that tests two instances of the Book class for equality:

// Swing Tutorial, 2nd edition

Book firstBook = new Book("0201914670");

Book secondBook = new Book("0201914670");

if (firstBook.equals(secondBook)) {

System.out.println("objects are equal");

} else {

System.out.println("objects are not equal");

}

This program displays objects are equal even though firstBook and secondBook reference two distinct objects. They are considered equal because the objects compared contain the same ISBN number.

You should always override the equals() method if the identity operator is not appropriate for your class.

**Note:** If you override equals(), you must override hashCode() as well.

## The finalize() Method

The Object class provides a callback method, finalize(), that *may be* invoked on an object when it becomes garbage. Object's implementation of finalize() does nothing—you can override finalize() to do cleanup, such as freeing resources.

The finalize() method may be called automatically by the system, but when it is called, or even if it is called, is uncertain. Therefore, you should not rely on this method to do your cleanup for you. For example, if you don't close file descriptors in your code after performing I/O and you expect finalize() to close them for you, you may run out of file descriptors.

## The getClass() Method

You cannot override getClass.

The getClass() method returns a Class object, which has methods you can use to get information about the class, such as its name (getSimpleName()), its superclass (getSuperclass()), and the interfaces it implements (getInterfaces()). For example, the following method gets and displays the class name of an object:

void printClassName(Object obj) {

System.out.println("The object's" + " class is " +

obj.getClass().getSimpleName());

}

The [Class](http://docs.oracle.com/javase/8/docs/api/java/lang/Class.html) class, in the java.lang package, has a large number of methods (more than 50). For example, you can test to see if the class is an annotation (isAnnotation()), an interface (isInterface()), or an enumeration (isEnum()). You can see what the object's fields are (getFields()) or what its methods are (getMethods()), and so on.

## The hashCode() Method

The value returned by hashCode() is the object's hash code, which is the object's memory address in hexadecimal.

By definition, if two objects are equal, their hash code *must also* be equal. If you override the equals() method, you change the way two objects are equated and Object's implementation ofhashCode() is no longer valid. Therefore, if you override the equals() method, you must also override the hashCode() method as well.

## The toString() Method

You should always consider overriding the toString() method in your classes.

The Object's toString() method returns a String representation of the object, which is very useful for debugging. The String representation for an object depends entirely on the object, which is why you need to override toString() in your classes.

You can use toString() along with System.out.println() to display a text representation of an object, such as an instance of Book:

System.out.println(firstBook.toString());

which would, for a properly overridden toString() method, print something useful, like this:

ISBN: 0201914670; The Swing Tutorial; A Guide to Constructing GUIs, 2nd Edition

# Writing Final Classes and Methods

You can declare some or all of a class's methods *final*. You use the final keyword in a method declaration to indicate that the method cannot be overridden by subclasses. The Object class does this—a number of its methods are final.

You might wish to make a method final if it has an implementation that should not be changed and it is critical to the consistent state of the object. For example, you might want to make thegetFirstPlayer method in this ChessAlgorithm class final:

class ChessAlgorithm {

enum ChessPlayer { WHITE, BLACK }

...

**final** ChessPlayer getFirstPlayer() {

return ChessPlayer.WHITE;

}

...

}

Methods called from constructors should generally be declared final. If a constructor calls a non-final method, a subclass may redefine that method with surprising or undesirable results.

Note that you can also declare an entire class final. A class that is declared final cannot be subclassed. This is particularly useful, for example, when creating an immutable class like the Stringclass.

# Abstract Methods and Classes

Abstraction implements through abstract class and interface.

f  any class has any abstract method then it can not be instantiated  
  
\*Abstract method can not be used in normal class ,but used in abstract class  
  
\*Abstract class may have abstract functions.  
  
\*Any child class has to override all the abstract class's abstract function othrwise child has to     
  become abstract.  
  
\*We never call the abstract function explicitly.automatically runs the code of the abstract   
     function.  
  
\* It is not mandatory that abstract class should have atleast one abstract function   
  
\* Cant mark class as abstract and final both ....both have opposite meaning

An *abstract class* is a class that is declared abstract—it may or may not include abstract methods. Abstract classes cannot be instantiated, but they can be subclassed.

An *abstract method* is a method that is declared without an implementation (without braces, and followed by a semicolon), like this:

abstract void moveTo(double deltaX, double deltaY);

If a class includes abstract methods, then the class itself *must* be declared abstract, as in:

public abstract class GraphicObject {

// declare fields

// declare nonabstract methods

abstract void draw();

}

When an abstract class is subclassed, the subclass usually provides implementations for all of the abstract methods in its parent class. However, if it does not, then the subclass must also be declared abstract.

**Note:** Methods in an *interface* (see the [Interfaces](http://docs.oracle.com/javase/tutorial/java/IandI/createinterface.html) section) that are not declared as default or static are *implicitly* abstract, so the abstract modifier is not used with interface methods. (It can be used, but it is unnecessary.)

## Abstract Classes Compared to Interfaces

Abstract classes are similar to interfaces. You cannot instantiate them, and they may contain a mix of methods declared with or without an implementation. However, with abstract classes, you can declare fields that are not static and final, and define public, protected, and private concrete methods. With interfaces, all fields are automatically public, static, and final, and all methods that you declare or define (as default methods) are public. In addition, you can extend only one class, whether or not it is abstract, whereas you can implement any number of interfaces.

Which should you use, abstract classes or interfaces?

* Consider using abstract classes if any of these statements apply to your situation:
  + You want to share code among several closely related classes.
  + You expect that classes that extend your abstract class have many common methods or fields, or require access modifiers other than public (such as protected and private).
  + You want to declare non-static or non-final fields. This enables you to define methods that can access and modify the state of the object to which they belong.
* Consider using interfaces if any of these statements apply to your situation:
  + You expect that unrelated classes would implement your interface. For example, the interfaces [Comparable](http://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html) and [Cloneable](http://docs.oracle.com/javase/8/docs/api/java/lang/Cloneable.html) are implemented by many unrelated classes.
  + You want to specify the behavior of a particular data type, but not concerned about who implements its behavior.
  + You want to take advantage of multiple inheritance of type.

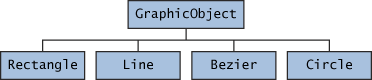
An example of an abstract class in the JDK is [AbstractMap](http://docs.oracle.com/javase/8/docs/api/java/util/AbstractMap.html), which is part of the Collections Framework. Its subclasses (which include HashMap, TreeMap, and ConcurrentHashMap) share many methods (including get, put, isEmpty, containsKey, and containsValue) that AbstractMap defines.

An example of a class in the JDK that implements several interfaces is [HashMap](http://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html), which implements the interfaces Serializable, Cloneable, and Map<K, V>. By reading this list of interfaces, you can infer that an instance of HashMap (regardless of the developer or company who implemented the class) can be cloned, is serializable (which means that it can be converted into a byte stream; see the section [Serializable Objects](http://docs.oracle.com/javase/tutorial/jndi/objects/serial.html)), and has the functionality of a map. In addition, the Map<K, V> interface has been enhanced with many default methods such asmerge and forEach that older classes that have implemented this interface do not have to define.

Note that many software libraries use both abstract classes and interfaces; the HashMap class implements several interfaces and also extends the abstract class AbstractMap.

## An Abstract Class Example

In an object-oriented drawing application, you can draw circles, rectangles, lines, Bezier curves, and many other graphic objects. These objects all have certain states (for example: position, orientation, line color, fill color) and behaviors (for example: moveTo, rotate, resize, draw) in common. Some of these states and behaviors are the same for all graphic objects (for example: position, fill color, and moveTo). Others require different implementations (for example, resize or draw). All GraphicObjects must be able to draw or resize themselves; they just differ in how they do it. This is a perfect situation for an abstract superclass. You can take advantage of the similarities and declare all the graphic objects to inherit from the same abstract parent object (for example, GraphicObject) as shown in the following figure.



Classes Rectangle, Line, Bezier, and Circle Inherit from GraphicObject

First, you declare an abstract class, GraphicObject, to provide member variables and methods that are wholly shared by all subclasses, such as the current position and the moveTo method.GraphicObject also declares abstract methods for methods, such as draw or resize, that need to be implemented by all subclasses but must be implemented in different ways. TheGraphicObject class can look something like this:

abstract class GraphicObject {

int x, y;

...

void moveTo(int newX, int newY) {

...

}

abstract void draw();

abstract void resize();

}

Each nonabstract subclass of GraphicObject, such as Circle and Rectangle, must provide implementations for the draw and resize methods:

class Circle extends GraphicObject {

void draw() {

...

}

void resize() {

...

}

}

class Rectangle extends GraphicObject {

void draw() {

...

}

void resize() {

...

}

}

## When an Abstract Class Implements an Interface

In the section on [Interfaces](http://docs.oracle.com/javase/tutorial/java/IandI/createinterface.html), it was noted that a class that implements an interface must implement *all* of the interface's methods. It is possible, however, to define a class that does not implement all of the interface's methods, provided that the class is declared to be abstract. For example,

abstract class X implements Y {

// implements all but one method of Y

}

class XX extends X {

// implements the remaining method in Y

}

In this case, class X must be abstract because it does not fully implement Y, but class XX does, in fact, implement Y.

## Class Members

An abstract class may have static fields and static methods. You can use these static members with a class reference (for example, AbstractClass.staticMethod()) as you would with any other class.

# Why Use Generics?

In a nutshell, generics enable types (classes and interfaces) to be parameters when defining classes, interfaces and methods. Much like the more familiar formal parameters used in method declarations, type parameters provide a way for you to re-use the same code with different inputs. The difference is that the inputs to formal parameters are values, while the inputs to type parameters are types.

Code that uses generics has many benefits over non-generic code:

* Stronger type checks at compile time.  
  A Java compiler applies strong type checking to generic code and issues errors if the code violates type safety. Fixing compile-time errors is easier than fixing runtime errors, which can be difficult to find.
* Elimination of casts.  
  The following code snippet without generics requires casting:
* List list = new ArrayList();
* list.add("hello");
* String s = **(String)** list.get(0);

When re-written to use generics, the code does not require casting:

List<String> list = new ArrayList<String>();

list.add("hello");

String s = list.get(0); // no cast

* Enabling programmers to implement generic algorithms.  
  By using generics, programmers can implement generic algorithms that work on collections of different types, can be customized, and are type safe and easier to read.

# Generic Types

A generic type is a generic class or interface that is parameterized over types. The following Box class will be modified to demonstrate the concept.

## A Simple Box Class

Begin by examining a non-generic Box class that operates on objects of any type. It needs only to provide two methods: set, which adds an object to the box, and get, which retrieves it:

public class Box {

private Object object;

public void set(Object object) { this.object = object; }

public Object get() { return object; }

}

Since its methods accept or return an Object, you are free to pass in whatever you want, provided that it is not one of the primitive types. There is no way to verify, at compile time, how the class is used. One part of the code may place an Integer in the box and expect to get Integers out of it, while another part of the code may mistakenly pass in a String, resulting in a runtime error.

## A Generic Version of the Box Class

A generic class is defined with the following format:

class name<T1, T2, ..., Tn> { /\* ... \*/ }

The type parameter section, delimited by angle brackets (<>), follows the class name. It specifies the type parameters (also called type variables) T1, T2, ..., and Tn.

To update the Box class to use generics, you create a generic type declaration by changing the code "public class Box" to "public class Box<T>". This introduces the type variable, T, that can be used anywhere inside the class.

With this change, the Box class becomes:

/\*\*

\* Generic version of the Box class.

\* @param <T> the type of the value being boxed

\*/

public class Box<T> {

// T stands for "Type"

private T t;

public void set(T t) { this.t = t; }

public T get() { return t; }

}

As you can see, all occurrences of Object are replaced by T. A type variable can be any **non-primitive** type you specify: any class type, any interface type, any array type, or even another type variable.

This same technique can be applied to create generic interfaces.

## Type Parameter Naming Conventions

By convention, type parameter names are single, uppercase letters. This stands in sharp contrast to the variable [naming](http://docs.oracle.com/javase/tutorial/java/nutsandbolts/variables.html#naming) conventions that you already know about, and with good reason: Without this convention, it would be difficult to tell the difference between a type variable and an ordinary class or interface name.

The most commonly used type parameter names are:

* E - Element (used extensively by the Java Collections Framework)
* K - Key
* N - Number
* T - Type
* V - Value
* S,U,V etc. - 2nd, 3rd, 4th types

You'll see these names used throughout the Java SE API and the rest of this lesson.

## Invoking and Instantiating a Generic Type

To reference the generic Box class from within your code, you must perform a generic type invocation, which replaces T with some concrete value, such as Integer:

Box<Integer> integerBox;

You can think of a generic type invocation as being similar to an ordinary method invocation, but instead of passing an argument to a method, you are passing a type argument — Integer in this case — to the Box class itself.

**Type Parameter and Type Argument Terminology:** Many developers use the terms "type parameter" and "type argument" interchangeably, but these terms are not the same. When coding, one provides type arguments in order to create a parameterized type. Therefore, the T in Foo<T> is a type parameter and the String in Foo<String> f is a type argument. This lesson observes this definition when using these terms.

Like any other variable declaration, this code does not actually create a new Box object. It simply declares that integerBox will hold a reference to a "Box of Integer", which is how Box<Integer> is read.

An invocation of a generic type is generally known as a parameterized type.

To instantiate this class, use the new keyword, as usual, but place <Integer> between the class name and the parenthesis:

Box<Integer> integerBox = new Box<Integer>();

## The Diamond

In Java SE 7 and later, you can replace the type arguments required to invoke the constructor of a generic class with an empty set of type arguments (<>) as long as the compiler can determine, or infer, the type arguments from the context. This pair of angle brackets, <>, is informally called the diamond. For example, you can create an instance of Box<Integer> with the following statement:

Box<Integer> integerBox = new Box<>();

For more information on diamond notation and type inference, see [Type Inference](http://docs.oracle.com/javase/tutorial/java/generics/genTypeInference.html).

## Multiple Type Parameters

As mentioned previously, a generic class can have multiple type parameters. For example, the generic OrderedPair class, which implements the generic Pair interface:

public interface Pair<K, V> {

public K getKey();

public V getValue();

}

public class OrderedPair<K, V> implements Pair<K, V> {

private K key;

private V value;

public OrderedPair(K key, V value) {

this.key = key;

this.value = value;

}

public K getKey() { return key; }

public V getValue() { return value; }

}

The following statements create two instantiations of the OrderedPair class:

Pair<String, Integer> p1 = new OrderedPair<String, Integer>("Even", 8);

Pair<String, String> p2 = new OrderedPair<String, String>("hello", "world");

The code, new OrderedPair<String, Integer>, instantiates K as a String and V as an Integer. Therefore, the parameter types of OrderedPair's constructor are String and Integer, respectively. Due to[autoboxing](http://docs.oracle.com/javase/tutorial/java/data/autoboxing.html), it is valid to pass a String and an int to the class.

As mentioned in [The Diamond](http://docs.oracle.com/javase/tutorial/java/generics/types.html#diamond), because a Java compiler can infer the K and V types from the declaration OrderedPair<String, Integer>, these statements can be shortened using diamond notation:

OrderedPair<String, Integer> p1 = new OrderedPair**<>**("Even", 8);

OrderedPair<String, String> p2 = new OrderedPair**<>**("hello", "world");

To create a generic interface, follow the same conventions as for creating a generic class.

## Parameterized Types

You can also substitute a type parameter (i.e., K or V) with a parameterized type (i.e., List<String>). For example, using the OrderedPair<K, V> example:

OrderedPair<String, **Box<Integer>**> p = new OrderedPair<>("primes", new Box<Integer>(...));

# Raw Types

A raw type is the name of a generic class or interface without any type arguments. For example, given the generic Box class:

public class Box<T> {

public void set(T t) { /\* ... \*/ }

// ...

}

To create a parameterized type of Box<T>, you supply an actual type argument for the formal type parameter T:

Box<Integer> intBox = new Box<>();

If the actual type argument is omitted, you create a raw type of Box<T>:

Box rawBox = new Box();

Therefore, Box is the raw type of the generic type Box<T>. However, a non-generic class or interface type is not a raw type.

Raw types show up in legacy code because lots of API classes (such as the Collections classes) were not generic prior to JDK 5.0. When using raw types, you essentially get pre-generics behavior — a Box gives you Objects. For backward compatibility, assigning a parameterized type to its raw type is allowed:

Box<String> stringBox = new Box<>();

Box rawBox = stringBox; // OK

But if you assign a raw type to a parameterized type, you get a warning:

Box rawBox = new Box(); // rawBox is a raw type of Box<T>

Box<Integer> intBox = rawBox; // warning: unchecked conversion

You also get a warning if you use a raw type to invoke generic methods defined in the corresponding generic type:

Box<String> stringBox = new Box<>();

Box rawBox = stringBox;

rawBox.set(8); // warning: unchecked invocation to set(T)

The warning shows that raw types bypass generic type checks, deferring the catch of unsafe code to runtime. Therefore, you should avoid using raw types.

The [Type Erasure](http://docs.oracle.com/javase/tutorial/java/generics/erasure.html) section has more information on how the Java compiler uses raw types.

## Unchecked Error Messages

As mentioned previously, when mixing legacy code with generic code, you may encounter warning messages similar to the following:

Note: Example.java uses unchecked or unsafe operations.

Note: Recompile with -Xlint:unchecked for details.

This can happen when using an older API that operates on raw types, as shown in the following example:

public class WarningDemo {

public static void main(String[] args){

Box<Integer> bi;

bi = createBox();

}

static Box createBox(){

return new Box();

}

}

The term "unchecked" means that the compiler does not have enough type information to perform all type checks necessary to ensure type safety. The "unchecked" warning is disabled, by default, though the compiler gives a hint. To see all "unchecked" warnings, recompile with -Xlint:unchecked.

Recompiling the previous example with -Xlint:unchecked reveals the following additional information:

WarningDemo.java:4: warning: [unchecked] unchecked conversion

found : Box

required: Box<java.lang.Integer>

bi = createBox();

^

1 warning

To completely disable unchecked warnings, use the -Xlint:-unchecked flag. The @SuppressWarnings("unchecked") annotation suppresses unchecked warnings. If you are unfamiliar with the@SuppressWarnings syntax, see [Annotations](http://docs.oracle.com/javase/tutorial/java/annotations/index.html).

# Generic Methods

Generic methods are methods that introduce their own type parameters. This is similar to declaring a generic type, but the type parameter's scope is limited to the method where it is declared. Static and non-static generic methods are allowed, as well as generic class constructors.

The syntax for a generic method includes a type parameter, inside angle brackets, and appears before the method's return type. For static generic methods, the type parameter section must appear before the method's return type.

The Util class includes a generic method, compare, which compares two Pair objects:

public class Util {

// Generic static method

**public static <K, V> boolean compare(Pair<K, V> p1, Pair<K, V> p2)** {

return p1.getKey().equals(p2.getKey()) &&

p1.getValue().equals(p2.getValue());

}

}

public class Pair<K, V> {

private K key;

private V value;

// Generic constructor

public Pair(K key, V value) {

this.key = key;

this.value = value;

}

// Generic methods

public void setKey(K key) { this.key = key; }

public void setValue(V value) { this.value = value; }

public K getKey() { return key; }

public V getValue() { return value; }

}

The complete syntax for invoking this method would be:

Pair<Integer, String> p1 = new Pair<>(1, "apple");

Pair<Integer, String> p2 = new Pair<>(2, "pear");

boolean same = Util.**<Integer, String>**compare(p1, p2);

The type has been explicitly provided, as shown in bold. Generally, this can be left out and the compiler will infer the type that is needed:

Pair<Integer, String> p1 = new Pair<>(1, "apple");

Pair<Integer, String> p2 = new Pair<>(2, "pear");

boolean same = Util.compare(p1, p2);

This feature, known as type inference, allows you to invoke a generic method as an ordinary method, without specifying a type between angle brackets. This topic is further discussed in the following section, [Type Inference](http://docs.oracle.com/javase/tutorial/java/generics/genTypeInference.html).

# Bounded Type Parameters

There may be times when you want to restrict the types that can be used as type arguments in a parameterized type. For example, a method that operates on numbers might only want to accept instances of Number or its subclasses. This is what *bounded type parameters* are for.

To declare a bounded type parameter, list the type parameter's name, followed by the extends keyword, followed by its *upper bound*, which in this example is Number. Note that, in this context, extends is used in a general sense to mean either "extends" (as in classes) or "implements" (as in interfaces).

public class Box<T> {

private T t;

public void set(T t) {

this.t = t;

}

public T get() {

return t;

}

public <U **extends Number**> void inspect(U u){

System.out.println("T: " + t.getClass().getName());

System.out.println("U: " + u.getClass().getName());

}

public static void main(String[] args) {

Box<Integer> integerBox = new Box<Integer>();

integerBox.set(new Integer(10));

integerBox.inspect("some text"); // **error: this is still String!**

}

}

By modifying our generic method to include this bounded type parameter, compilation will now fail, since our invocation of inspect still includes a String:

Box.java:21: <U>inspect(U) in Box<java.lang.Integer> cannot

be applied to (java.lang.String)

integerBox.inspect("10");

^

1 error

In addition to limiting the types you can use to instantiate a generic type, bounded type parameters allow you to invoke methods defined in the bounds:

public class NaturalNumber<T extends Integer> {

private T n;

public NaturalNumber(T n) { this.n = n; }

public boolean isEven() {

return **n.intValue()** % 2 == 0;

}

// ...

}

The isEven method invokes the intValue method defined in the Integer class through n.

## Multiple Bounds

The preceding example illustrates the use of a type parameter with a single bound, but a type parameter can have multiple bounds:

<T extends B1 & B2 & B3>

A type variable with multiple bounds is a subtype of all the types listed in the bound. If one of the bounds is a class, it must be specified first. For example:

Class A { /\* ... \*/ }

interface B { /\* ... \*/ }

interface C { /\* ... \*/ }

class D <T extends A & B & C> { /\* ... \*/ }

If bound A is not specified first, you get a compile-time error:

class D <T extends B & A & C> { /\* ... \*/ } // compile-time error

# Generic Methods and Bounded Type Parameters

Bounded type parameters are key to the implementation of generic algorithms. Consider the following method that counts the number of elements in an array T[] that are greater than a specified element elem.

public static <T> int countGreaterThan(T[] anArray, T elem) {

int count = 0;

for (T e : anArray)

if (e > elem) // compiler error

++count;

return count;

}

The implementation of the method is straightforward, but it does not compile because the greater than operator (>) applies only to primitive types such as short, int, double, long, float, byte, andchar. You cannot use the > operator to compare objects. To fix the problem, use a type parameter bounded by the Comparable<T> interface:

public interface Comparable<T> {

public int compareTo(T o);

}

The resulting code will be:

public static <T extends Comparable<T>> int countGreaterThan(T[] anArray, T elem) {

int count = 0;

for (T e : anArray)

if (e.compareTo(elem) > 0)

++count;

return count;

}

# Generics, Inheritance, and Subtypes

As you already know, it is possible to assign an object of one type to an object of another type provided that the types are compatible. For example, you can assign an Integer to an Object, sinceObject is one of Integer's supertypes:

Object someObject = new Object();

Integer someInteger = new Integer(10);

someObject = someInteger; // OK

In object-oriented terminology, this is called an "is a" relationship. Since an Integer is a kind of Object, the assignment is allowed. But Integer is also a kind of Number, so the following code is valid as well:

public void someMethod(Number n) { /\* ... \*/ }

someMethod(new Integer(10)); // OK

someMethod(new Double(10.1)); // OK

The same is also true with generics. You can perform a generic type invocation, passing Number as its type argument, and any subsequent invocation of add will be allowed if the argument is compatible with Number:

Box<Number> box = new Box<Number>();

box.add(new Integer(10)); // OK

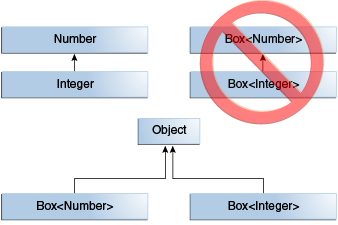
box.add(new Double(10.1)); // OK

Now consider the following method:

public void boxTest(Box<Number> n) { /\* ... \*/ }

What type of argument does it accept? By looking at its signature, you can see that it accepts a single argument whose type is Box<Number>. But what does that mean? Are you allowed to pass inBox<Integer> or Box<Double>, as you might expect? The answer is "no", because Box<Integer> and Box<Double> are not subtypes of Box<Number>.

This is a common misunderstanding when it comes to programming with generics, but it is an important concept to learn.



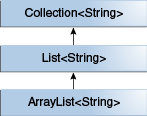
Box<Integer> is not a subtype of Box<Number> even though Integer is a subtype of Number.

**Note:** Given two concrete types A and B (for example, Number and Integer), MyClass<A> has no relationship to MyClass<B>, regardless of whether or not A and B are related. The common parent of MyClass<A> and MyClass<B> is Object.  
  
For information on how to create a subtype-like relationship between two generic classes when the type parameters are related, see [Wildcards and Subtyping](http://docs.oracle.com/javase/tutorial/java/generics/subtyping.html).

## Generic Classes and Subtyping

You can subtype a generic class or interface by extending or implementing it. The relationship between the type parameters of one class or interface and the type parameters of another are determined by the extends and implements clauses.

Using the Collections classes as an example, ArrayList<E> implements List<E>, and List<E> extends Collection<E>. So ArrayList<String> is a subtype of List<String>, which is a subtype ofCollection<String>. So long as you do not vary the type argument, the subtyping relationship is preserved between the types.



A sample Collections hierarchy

Now imagine we want to define our own list interface, PayloadList, that associates an optional value of generic type P with each element. Its declaration might look like:

interface PayloadList<E,P> extends List<E> {

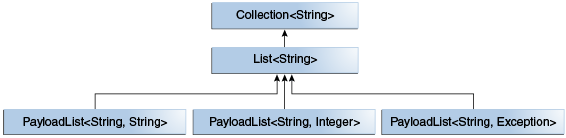
void setPayload(int index, P val);

...

}

The following parameterizations of PayloadList are subtypes of List<String>:

* PayloadList<String,String>
* PayloadList<String,Integer>
* PayloadList<String,Exception>



A sample PayloadList hierarchy

# Type Inference

Type inference is a Java compiler's ability to look at each method invocation and corresponding declaration to determine the type argument (or arguments) that make the invocation applicable. The inference algorithm determines the types of the arguments and, if available, the type that the result is being assigned, or returned. Finally, the inference algorithm tries to find the most specific type that works with all of the arguments.

To illustrate this last point, in the following example, inference determines that the second argument being passed to the pick method is of type Serializable:

static <T> T pick(T a1, T a2) { return a2; }

Serializable s = pick("d", new ArrayList<String>());

## Type Inference and Generic Methods

[Generic Methods](http://docs.oracle.com/javase/tutorial/java/generics/methods.html) introduced you to type inference, which enables you to invoke a generic method as you would an ordinary method, without specifying a type between angle brackets. Consider the following example, [BoxDemo](http://docs.oracle.com/javase/tutorial/java/generics/examples/BoxDemo.java), which requires the [Box](http://docs.oracle.com/javase/tutorial/java/generics/examples/Box.java) class:

public class BoxDemo {

public static <U> void addBox(U u,

java.util.List<Box<U>> boxes) {

Box<U> box = new Box<>();

box.set(u);

boxes.add(box);

}

public static <U> void outputBoxes(java.util.List<Box<U>> boxes) {

int counter = 0;

for (Box<U> box: boxes) {

U boxContents = box.get();

System.out.println("Box #" + counter + " contains [" +

boxContents.toString() + "]");

counter++;

}

}

public static void main(String[] args) {

java.util.ArrayList<Box<Integer>> listOfIntegerBoxes =

new java.util.ArrayList<>();

BoxDemo.<Integer>addBox(Integer.valueOf(10), listOfIntegerBoxes);

BoxDemo.addBox(Integer.valueOf(20), listOfIntegerBoxes);

BoxDemo.addBox(Integer.valueOf(30), listOfIntegerBoxes);

BoxDemo.outputBoxes(listOfIntegerBoxes);

}

}

The following is the output from this example:

Box #0 contains [10]

Box #1 contains [20]

Box #2 contains [30]

The generic method addBox defines one type parameter named U. Generally, a Java compiler can infer the type parameters of a generic method call. Consequently, in most cases, you do not have to specify them. For example, to invoke the generic method addBox, you can specify the type parameter with a type witness as follows:

BoxDemo.**<Integer>**addBox(Integer.valueOf(10), listOfIntegerBoxes);

Alternatively, if you omit the type witness,a Java compiler automatically infers (from the method's arguments) that the type parameter is Integer:

BoxDemo.addBox(Integer.valueOf(20), listOfIntegerBoxes);

## Type Inference and Instantiation of Generic Classes

You can replace the type arguments required to invoke the constructor of a generic class with an empty set of type parameters (<>) as long as the compiler can infer the type arguments from the context. This pair of angle brackets is informally called [the diamond](http://docs.oracle.com/javase/tutorial/java/generics/types.html#diamond).

For example, consider the following variable declaration:

Map<String, List<String>> myMap = new HashMap<String, List<String>>();

You can substitute the parameterized type of the constructor with an empty set of type parameters (<>):

Map<String, List<String>> myMap = new HashMap<>();

Note that to take advantage of type inference during generic class instantiation, you must use the diamond. In the following example, the compiler generates an unchecked conversion warning because the HashMap() constructor refers to the HashMap raw type, not the Map<String, List<String>> type:

Map<String, List<String>> myMap = new HashMap(); // unchecked conversion warning

## Type Inference and Generic Constructors of Generic and Non-Generic Classes

Note that constructors can be generic (in other words, declare their own formal type parameters) in both generic and non-generic classes. Consider the following example:

class MyClass<X> {

<T> MyClass(T t) {

// ...

}

}

Consider the following instantiation of the class MyClass:

new MyClass<Integer>("")

This statement creates an instance of the parameterized type MyClass<Integer>; the statement explicitly specifies the type Integer for the formal type parameter, X, of the generic classMyClass<X>. Note that the constructor for this generic class contains a formal type parameter, T. The compiler infers the type String for the formal type parameter, T, of the constructor of this generic class (because the actual parameter of this constructor is a String object).

Compilers from releases prior to Java SE 7 are able to infer the actual type parameters of generic constructors, similar to generic methods. However, compilers in Java SE 7 and later can infer the actual type parameters of the generic class being instantiated if you use the diamond (<>). Consider the following example:

MyClass<Integer> myObject = new MyClass<>("");

In this example, the compiler infers the type Integer for the formal type parameter, X, of the generic class MyClass<X>. It infers the type String for the formal type parameter, T, of the constructor of this generic class.

**Note:** It is important to note that the inference algorithm uses only invocation arguments, target types, and possibly an obvious expected return type to infer types. The inference algorithm does not use results from later in the program.

## Target Types

The Java compiler takes advantage of target typing to infer the type parameters of a generic method invocation. The target type of an expression is the data type that the Java compiler expects depending on where the expression appears. Consider the method Collections.emptyList, which is declared as follows:

static <T> List<T> emptyList();

Consider the following assignment statement:

List<String> listOne = Collections.emptyList();

This statement is expecting an instance of List<String>; this data type is the target type. Because the method emptyList returns a value of type List<T>, the compiler infers that the type argument T must be the value String. This works in both Java SE 7 and 8. Alternatively, you could use a type witness and specify the value of T as follows:

List<String> listOne = Collections.<String>emptyList();

However, this is not necessary in this context. It was necessary in other contexts, though. Consider the following method:

void processStringList(List<String> stringList) {

// process stringList

}

Suppose you want to invoke the method processStringList with an empty list. In Java SE 7, the following statement does not compile:

processStringList(Collections.emptyList());

The Java SE 7 compiler generates an error message similar to the following:

List<Object> cannot be converted to List<String>

The compiler requires a value for the type argument T so it starts with the value Object. Consequently, the invocation of Collections.emptyList returns a value of type List<Object>, which is incompatible with the method processStringList. Thus, in Java SE 7, you must specify the value of the value of the type argument as follows:

processStringList(Collections.<String>emptyList());

This is no longer necessary in Java SE 8. The notion of what is a target type has been expanded to include method arguments, such as the argument to the method processStringList. In this case, processStringList requires an argument of type List<String>. The method Collections.emptyList returns a value of List<T>, so using the target type ofList<String>, the compiler infers that the type argument T has a value of String. Thus, in Java SE 8, the following statement compiles:

processStringList(Collections.emptyList());

See [Target Typing](http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html#target-typing) in [Lambda Expressions](http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html) for more information.

# Wildcards

In generic code, the question mark (?), called the wildcard, represents an unknown type. The wildcard can be used in a variety of situations: as the type of a parameter, field, or local variable; sometimes as a return type (though it is better programming practice to be more specific). The wildcard is never used as a type argument for a generic method invocation, a generic class instance creation, or a supertype.

The following sections discuss wildcards in more detail, including upper bounded wildcards, lower bounded wildcards, and wildcard capture.

# Upper Bounded Wildcards

You can use an upper bounded wildcard to relax the restrictions on a variable. For example, say you want to write a method that works on List<Integer>, List<Double>, and List<Number>; you can achieve this by using an upper bounded wildcard.

To declare an upper-bounded wildcard, use the wildcard character ('?'), followed by the extends keyword, followed by its upper bound. Note that, in this context, extends is used in a general sense to mean either "extends" (as in classes) or "implements" (as in interfaces).

To write the method that works on lists of Number and the subtypes of Number, such as Integer, Double, and Float, you would specify List<? extends Number>. The term List<Number> is more restrictive than List<? extends Number> because the former matches a list of type Number only, whereas the latter matches a list of type Number or any of its subclasses.

Consider the following process method:

public static void process(List**<? extends Foo>** list) { /\* ... \*/ }

The upper bounded wildcard, <? extends Foo>, where Foo is any type, matches Foo and any subtype of Foo. The process method can access the list elements as type Foo:

public static void process(List<? extends Foo> list) {

for (Foo elem : list) {

// ...

}

}

In the foreach clause, the elem variable iterates over each element in the list. Any method defined in the Foo class can now be used on elem.

The sumOfList method returns the sum of the numbers in a list:

public static double sumOfList(List<? extends Number> list) {

double s = 0.0;

for (Number n : list)

s += n.doubleValue();

return s;

}

The following code, using a list of Integer objects, prints sum = 6.0:

List<Integer> li = Arrays.asList(1, 2, 3);

System.out.println("sum = " + sumOfList(li));

A list of Double values can use the same sumOfList method. The following code prints sum = 7.0:

List<Double> ld = Arrays.asList(1.2, 2.3, 3.5);

System.out.println("sum = " + sumOfList(ld));

# Unbounded Wildcards

The unbounded wildcard type is specified using the wildcard character (?), for example, List<?>. This is called a list of unknown type. There are two scenarios where an unbounded wildcard is a useful approach:

* If you are writing a method that can be implemented using functionality provided in the Object class.
* When the code is using methods in the generic class that don't depend on the type parameter. For example, List.size or List.clear. In fact, Class<?> is so often used because most of the methods in Class<T> do not depend on T.

Consider the following method, printList:

public static void printList(List<Object> list) {

for (Object elem : list)

System.out.println(elem + " ");

System.out.println();

}

The goal of printList is to print a list of any type, but it fails to achieve that goal — it prints only a list of Object instances; it cannot print List<Integer>, List<String>, List<Double>, and so on, because they are not subtypes of List<Object>. To write a generic printList method, use List<?>:

public static void printList(List<?> list) {

for (Object elem: list)

System.out.print(elem + " ");

System.out.println();

}

Because for any concrete type A, List<A> is a subtype of List<?>, you can use printList to print a list of any type:

List<Integer> li = Arrays.asList(1, 2, 3);

List<String> ls = Arrays.asList("one", "two", "three");

printList(li);

printList(ls);

**Note:** The [Arrays.asList](http://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html#asList-T...-) method is used in examples throughout this lesson. This static factory method converts the specified array and returns a fixed-size list.

It's important to note that List<Object> and List<?> are not the same. You can insert an Object, or any subtype of Object, into a List<Object>. But you can only insert null into a List<?>. The[Guidelines for Wildcard Use](http://docs.oracle.com/javase/tutorial/java/generics/wildcardGuidelines.html) section has more information on how to determine what kind of wildcard, if any, should be used in a given situation.

# Lower Bounded Wildcards

The [Upper Bounded Wildcards](http://docs.oracle.com/javase/tutorial/java/generics/upperBounded.html) section shows that an upper bounded wildcard restricts the unknown type to be a specific type or a subtype of that type and is represented using the extendskeyword. In a similar way, a lower bounded wildcard restricts the unknown type to be a specific type or a super type of that type.

A lower bounded wildcard is expressed using the wildcard character ('?'), following by the super keyword, followed by its lower bound: <? super A>.

**Note:** You can specify an upper bound for a wildcard, or you can specify a lower bound, but you cannot specify both.

Say you want to write a method that puts Integer objects into a list. To maximize flexibility, you would like the method to work on List<Integer>, List<Number>, and List<Object> — anything that can hold Integer values.

To write the method that works on lists of Integer and the supertypes of Integer, such as Integer, Number, and Object, you would specify List<? super Integer>. The term List<Integer> is more restrictive than List<? super Integer> because the former matches a list of type Integer only, whereas the latter matches a list of any type that is a supertype of Integer.

The following code adds the numbers 1 through 10 to the end of a list:

public static void addNumbers(List<? super Integer> list) {

for (int i = 1; i <= 10; i++) {

list.add(i);

}

}

The [Guidelines for Wildcard Use](http://docs.oracle.com/javase/tutorial/java/generics/wildcardGuidelines.html) section provides guidance on when to use upper bounded wildcards and when to use lower bounded wildcards.

# Wildcards and Subtyping

As described in [Generics, Inheritance, and Subtypes](http://docs.oracle.com/javase/tutorial/java/generics/inheritance.html), generic classes or interfaces are not related merely because there is a relationship between their types. However, you can use wildcards to create a relationship between generic classes or interfaces.

Given the following two regular (non-generic) classes:

class A { /\* ... \*/ }

class B extends A { /\* ... \*/ }

It would be reasonable to write the following code:

B b = new B();

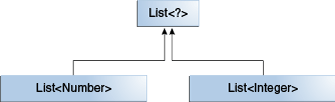
A a = b;

This example shows that inheritance of regular classes follows this rule of subtyping: class B is a subtype of class A if B extends A. This rule does not apply to generic types:

List<B> lb = new ArrayList<>();

List<A> la = lb; // compile-time error

Given that Integer is a subtype of Number, what is the relationship between List<Integer> and List<Number>?



The common parent is List<?>.

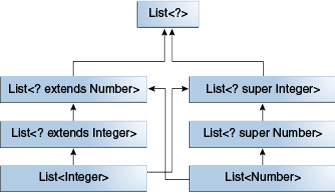
Although Integer is a subtype of Number, List<Integer> is not a subtype of List<Number> and, in fact, these two types are not related. The common parent of List<Number> and List<Integer> is List<?>.

In order to create a relationship between these classes so that the code can access Number's methods through List<Integer>'s elements, use an upper bounded wildcard:

List<? extends Integer> intList = new ArrayList<>();

List<? extends Number> numList = intList; // OK. List<? extends Integer> is a subtype of List<? extends Number>

Because Integer is a subtype of Number, and numList is a list of Number objects, a relationship now exists between intList (a list of Integer objects) and numList. The following diagram shows the relationships between several List classes declared with both upper and lower bounded wildcards.



A hierarchy of several generic List class declarations.

The [Guidelines for Wildcard Use](http://docs.oracle.com/javase/tutorial/java/generics/wildcardGuidelines.html) section has more information about the ramifications of using upper and lower bounded wildcards

# Wildcard Capture and Helper Methods

In some cases, the compiler infers the type of a wildcard. For example, a list may be defined as List<?> but, when evaluating an expression, the compiler infers a particular type from the code. This scenario is known as wildcard capture.

For the most part, you don't need to worry about wildcard capture, except when you see an error message that contains the phrase "capture of".

The [WildcardError](http://docs.oracle.com/javase/tutorial/java/generics/examples/WildcardError.java) example produces a capture error when compiled:

import java.util.List;

public class WildcardError {

void foo(List<?> i) {

i.set(0, i.get(0));

}

}

In this example, the compiler processes the i input parameter as being of type Object. When the foo method invokes [List.set(int, E)](http://docs.oracle.com/javase/8/docs/api/java/util/List.html#set-int-E-), the compiler is not able to confirm the type of object that is being inserted into the list, and an error is produced. When this type of error occurs it typically means that the compiler believes that you are assigning the wrong type to a variable. Generics were added to the Java language for this reason — to enforce type safety at compile time.

The WildcardError example generates the following error when compiled by Oracle's JDK 7 javac implementation:

WildcardError.java:6: error: method set in interface List<E> cannot be applied to given types;

i.set(0, i.get(0));

^

required: int,CAP#1

found: int,Object

reason: actual argument Object cannot be converted to CAP#1 by method invocation conversion

where E is a type-variable:

E extends Object declared in interface List

where CAP#1 is a fresh type-variable:

CAP#1 extends Object from capture of ?

1 error

In this example, the code is attempting to perform a safe operation, so how can you work around the compiler error? You can fix it by writing a private helper method which captures the wildcard. In this case, you can work around the problem by creating the private helper method, fooHelper, as shown in [WildcardFixed](http://docs.oracle.com/javase/tutorial/java/generics/examples/WildcardFixed.java):

public class WildcardFixed {

void foo(List<?> i) {

fooHelper(i);

}

// Helper method created so that the wildcard can be captured

// through type inference.

**private <T> void fooHelper(List<T> l) {**

**l.set(0, l.get(0));**

**}**

}

Thanks to the helper method, the compiler uses inference to determine that T is CAP#1, the capture variable, in the invocation. The example now compiles successfully.

By convention, helper methods are generally named originalMethodNameHelper.

Now consider a more complex example, [WildcardErrorBad](http://docs.oracle.com/javase/tutorial/java/generics/examples/WildcardErrorBad.java):

import java.util.List;

public class WildcardErrorBad {

void swapFirst(List<? extends Number> l1, List<? extends Number> l2) {

Number temp = l1.get(0);

l1.set(0, l2.get(0)); // expected a CAP#1 extends Number,

// got a CAP#2 extends Number;

// same bound, but different types

l2.set(0, temp); // expected a CAP#1 extends Number,

// got a Number

}

}

In this example, the code is attempting an unsafe operation. For example, consider the following invocation of the swapFirst method:

List<Integer> li = Arrays.asList(1, 2, 3);

List<Double> ld = Arrays.asList(10.10, 20.20, 30.30);

swapFirst(li, ld);

While List<Integer> and List<Double> both fulfill the criteria of List<? extends Number>, it is clearly incorrect to take an item from a list of Integer values and attempt to place it into a list of Doublevalues.

Compiling the code with Oracle's JDK javac compiler produces the following error:

WildcardErrorBad.java:7: error: method set in interface List<E> cannot be applied to given types;

l1.set(0, l2.get(0)); // expected a CAP#1 extends Number,

^

required: int,CAP#1

found: int,Number

reason: actual argument Number cannot be converted to CAP#1 by method invocation conversion

where E is a type-variable:

E extends Object declared in interface List

where CAP#1 is a fresh type-variable:

CAP#1 extends Number from capture of ? extends Number

WildcardErrorBad.java:10: error: method set in interface List<E> cannot be applied to given types;

l2.set(0, temp); // expected a CAP#1 extends Number,

^

required: int,CAP#1

found: int,Number

reason: actual argument Number cannot be converted to CAP#1 by method invocation conversion

where E is a type-variable:

E extends Object declared in interface List

where CAP#1 is a fresh type-variable:

CAP#1 extends Number from capture of ? extends Number

WildcardErrorBad.java:15: error: method set in interface List<E> cannot be applied to given types;

i.set(0, i.get(0));

^

required: int,CAP#1

found: int,Object

reason: actual argument Object cannot be converted to CAP#1 by method invocation conversion

where E is a type-variable:

E extends Object declared in interface List

where CAP#1 is a fresh type-variable:

CAP#1 extends Object from capture of ?

3 errors

There is no helper method to work around the problem, because the code is fundamentally wrong.

# Guidelines for Wildcard Use

One of the more confusing aspects when learning to program with generics is determining when to use an upper bounded wildcard and when to use a lower bounded wildcard. This page provides some guidelines to follow when designing your code.

For purposes of this discussion, it is helpful to think of variables as providing one of two functions:

**An "In" Variable**

An "in" variable serves up data to the code. Imagine a copy method with two arguments: copy(src, dest). The src argument provides the data to be copied, so it is the "in" parameter.

**An "Out" Variable**

An "out" variable holds data for use elsewhere. In the copy example, copy(src, dest), the dest argument accepts data, so it is the "out" parameter.

Of course, some variables are used both for "in" and "out" purposes — this scenario is also addressed in the guidelines.

You can use the "in" and "out" principle when deciding whether to use a wildcard and what type of wildcard is appropriate. The following list provides the guidelines to follow:

**Wildcard Guidelines:**

* An "in" variable is defined with an upper bounded wildcard, using the extends keyword.
* An "out" variable is defined with a lower bounded wildcard, using the super keyword.
* In the case where the "in" variable can be accessed using methods defined in the Object class, use an unbounded wildcard.
* In the case where the code needs to access the variable as both an "in" and an "out" variable, do not use a wildcard.

These guidelines do not apply to a method's return type. Using a wildcard as a return type should be avoided because it forces programmers using the code to deal with wildcards.

A list defined by List<? extends ...> can be informally thought of as read-only, but that is not a strict guarantee. Suppose you have the following two classes:

class NaturalNumber {

private int i;

public NaturalNumber(int i) { this.i = i; }

// ...

}

class EvenNumber extends NaturalNumber {

public EvenNumber(int i) { super(i); }

// ...

}

Consider the following code:

List<EvenNumber> le = new ArrayList<>();

List<? extends NaturalNumber> ln = le;

ln.add(new NaturalNumber(35)); // compile-time error

Because List<EvenNumber> is a subtype of List<? extends NaturalNumber>, you can assign le to ln. But you cannot use ln to add a natural number to a list of even numbers. The following operations on the list are possible:

* You can add null.
* You can invoke clear.
* You can get the iterator and invoke remove.
* You can capture the wildcard and write elements that you've read from the list.

You can see that the list defined by List<? extends NaturalNumber> is not read-only in the strictest sense of the word, but you might think of it that way because you cannot store a new element or change an existing element in the list.

# Erasure of Generic Types

During the type erasure process, the Java compiler erases all type parameters and replaces each with its first bound if the type parameter is bounded, or Object if the type parameter is unbounded.

Consider the following generic class that represents a node in a singly linked list:

public class Node<T> {

private T data;

private Node<T> next;

public Node(T data, Node<T> next) }

this.data = data;

this.next = next;

}

public T getData() { return data; }

// ...

}

Because the type parameter T is unbounded, the Java compiler replaces it with Object:

public class Node {

private Object data;

private Node next;

public Node(Object data, Node next) {

this.data = data;

this.next = next;

}

public Object getData() { return data; }

// ...

}

In the following example, the generic Node class uses a bounded type parameter:

public class Node<T extends Comparable<T>> {

private T data;

private Node<T> next;

public Node(T data, Node<T> next) {

this.data = data;

this.next = next;

}

public T getData() { return data; }

// ...

}

The Java compiler replaces the bounded type parameter T with the first bound class, Comparable:

public class Node {

private Comparable data;

private Node next;

public Node(Comparable data, Node next) {

this.data = data;

this.next = next;

}

public Comparable getData() { return data; }

// ...

}

# Erasure of Generic Methods

The Java compiler also erases type parameters in generic method arguments. Consider the following generic method:

// Counts the number of occurrences of elem in anArray.

//

public static <T> int count(T[] anArray, T elem) {

int cnt = 0;

for (T e : anArray)

if (e.equals(elem))

++cnt;

return cnt;

}

Because T is unbounded, the Java compiler replaces it with Object:

public static int count(Object[] anArray, Object elem) {

int cnt = 0;

for (Object e : anArray)

if (e.equals(elem))

++cnt;

return cnt;

}

Suppose the following classes are defined:

class Shape { /\* ... \*/ }

class Circle extends Shape { /\* ... \*/ }

class Rectangle extends Shape { /\* ... \*/ }

You can write a generic method to draw different shapes:

public static <T extends Shape> void draw(T shape) { /\* ... \*/ }

The Java compiler replaces T with Shape:

public static void draw(Shape shape) { /\* ... \*/ }

# Effects of Type Erasure and Bridge Methods

Sometimes type erasure causes a situation that you may not have anticipated. The following example shows how this can occur. The example (described in [Bridge Methods](http://docs.oracle.com/javase/tutorial/java/generics/bridgeMethods.html#bridgeMethods)) shows how a compiler sometimes creates a synthetic method, called a bridge method, as part of the type erasure process.

Given the following two classes:

public class Node<T> {

private T data;

public Node(T data) { this.data = data; }

public void setData(T data) {

System.out.println("Node.setData");

this.data = data;

}

}

public class MyNode extends Node<Integer> {

public MyNode(Integer data) { super(data); }

public void setData(Integer data) {

System.out.println("MyNode.setData");

super.setData(data);

}

}

Consider the following code:

MyNode mn = new MyNode(5);

Node n = mn; // A raw type - compiler throws an unchecked warning

n.setData("Hello"); // Causes a ClassCastException to be thrown.

Integer x = mn.data;

After type erasure, this code becomes:

MyNode mn = new MyNode(5);

Node n = (MyNode)mn; // A raw type - compiler throws an unchecked warning

n.setData("Hello");

Integer x = (String)mn.data; // Causes a ClassCastException to be thrown.

Here is what happens as the code is executed:

* n.setData("Hello"); causes the method setData(Object) to be executed on the object of class MyNode. (The MyNode class inherited setData(Object) from Node.)
* In the body of setData(Object), the data field of the object referenced by n is assigned to a String.
* The data field of that same object, referenced via mn, can be accessed and is expected to be an integer (since mn is a MyNode which is a Node<Integer>.
* Trying to assign a String to an Integer causes a ClassCastException from a cast inserted at the assignment by a Java compiler.

## Bridge Methods

When compiling a class or interface that extends a parameterized class or implements a parameterized interface, the compiler may need to create a synthetic method, called a bridge method, as part of the type erasure process. You normally don't need to worry about bridge methods, but you might be puzzled if one appears in a stack trace.

After type erasure, the Node and MyNode classes become:

public class Node {

private Object data;

public Node(Object data) { this.data = data; }

public void setData(Object data) {

System.out.println("Node.setData");

this.data = data;

}

}

public class MyNode extends Node {

public MyNode(Integer data) { super(data); }

public void setData(Integer data) {

System.out.println(Integer data);

super.setData(data);

}

}

After type erasure, the method signatures do not not match. The Node method becomes setData(Object) and the MyNode method becomes setData(Integer). Therefore, the MyNode setData method does not override the Node setData method.

To solve this problem and preserve the [polymorphism](http://docs.oracle.com/javase/tutorial/java/IandI/polymorphism.html) of generic types after type erasure, a Java compiler generates a bridge method to ensure that subtyping works as expected. For the MyNodeclass, the compiler generates the following bridge method for setData:

class MyNode extends Node {

**// Bridge method generated by the compiler**

**//**

**public void setData(Object data) {**

**setData((Integer) data);**

**}**

public void setData(Integer data) {

System.out.println("MyNode.setData");

super.setData(data);

}

// ...

}

As you can see, the bridge method, which has the same method signature as the Node class's setData method after type erasure, delegates to the original setData method

# Non-Reifiable Types

The section [Type Erasure](http://docs.oracle.com/javase/tutorial/java/generics/erasure.html) discusses the process where the compiler removes information related to type parameters and type arguments. Type erasure has consequences related to variable arguments (also known as varargs ) methods whose varargs formal parameter has a non-reifiable type. See the section [Arbitrary Number of Arguments](http://docs.oracle.com/javase/tutorial/java/javaOO/arguments.html#varargs) in [Passing Information to a Method or a Constructor](http://docs.oracle.com/javase/tutorial/java/javaOO/arguments.html) for more information about varargs methods.

This page covers the following topics:

* [Non-Reifiable Types](http://docs.oracle.com/javase/tutorial/java/generics/nonReifiableVarargsType.html#non-reifiable-types)
* [Heap Pollution](http://docs.oracle.com/javase/tutorial/java/generics/nonReifiableVarargsType.html#heap_pollution)
* [Potential Vulnerabilities of Varargs Methods with Non-Reifiable Formal Parameters](http://docs.oracle.com/javase/tutorial/java/generics/nonReifiableVarargsType.html#vulnerabilities)
* [Preventing Warnings from Varargs Methods with Non-Reifiable Formal Parameters](http://docs.oracle.com/javase/tutorial/java/generics/nonReifiableVarargsType.html#suppressing)

## Non-Reifiable Types

A reifiable type is a type whose type information is fully available at runtime. This includes primitives, non-generic types, raw types, and invocations of unbound wildcards.

Non-reifiable types are types where information has been removed at compile-time by type erasure — invocations of generic types that are not defined as unbounded wildcards. A non-reifiable type does not have all of its information available at runtime. Examples of non-reifiable types are List<String> and List<Number>; the JVM cannot tell the difference between these types at runtime. As shown in [Restrictions on Generics](http://docs.oracle.com/javase/tutorial/java/generics/restrictions.html), there are certain situations where non-reifiable types cannot be used: in an instanceof expression, for example, or as an element in an array.

## Heap Pollution

Heap pollution occurs when a variable of a parameterized type refers to an object that is not of that parameterized type. This situation occurs if the program performed some operation that gives rise to an unchecked warning at compile-time. An unchecked warning is generated if, either at compile-time (within the limits of the compile-time type checking rules) or at runtime, the correctness of an operation involving a parameterized type (for example, a cast or method call) cannot be verified. For example, heap pollution occurs when mixing raw types and parameterized types, or when performing unchecked casts.

In normal situations, when all code is compiled at the same time, the compiler issues an unchecked warning to draw your attention to potential heap pollution. If you compile sections of your code separately, it is difficult to detect the potential risk of heap pollution. If you ensure that your code compiles without warnings, then no heap pollution can occur.

## Potential Vulnerabilities of Varargs Methods with Non-Reifiable Formal Parameters

Generic methods that include vararg input parameters can cause heap pollution.

Consider the following ArrayBuilder class:

public class ArrayBuilder {

public static <T> void addToList (List<T> listArg, T... elements) {

for (T x : elements) {

listArg.add(x);

}

}

public static void faultyMethod(List<String>... l) {

Object[] objectArray = l; // Valid

objectArray[0] = Arrays.asList(42);

String s = l[0].get(0); // ClassCastException thrown here

}

}

The following example, HeapPollutionExample uses the ArrayBuiler class:

public class HeapPollutionExample {

public static void main(String[] args) {

List<String> stringListA = new ArrayList<String>();

List<String> stringListB = new ArrayList<String>();

ArrayBuilder.addToList(stringListA, "Seven", "Eight", "Nine");

ArrayBuilder.addToList(stringListB, "Ten", "Eleven", "Twelve");

List<List<String>> listOfStringLists =

new ArrayList<List<String>>();

ArrayBuilder.addToList(listOfStringLists,

stringListA, stringListB);

ArrayBuilder.faultyMethod(Arrays.asList("Hello!"), Arrays.asList("World!"));

}

}

When compiled, the following warning is produced by the definition of the ArrayBuilder.addToList method:

warning: [varargs] Possible heap pollution from parameterized vararg type T

When the compiler encounters a varargs method, it translates the varargs formal parameter into an array. However, the Java programming language does not permit the creation of arrays of parameterized types. In the method ArrayBuilder.addToList, the compiler translates the varargs formal parameter T... elements to the formal parameter T[] elements, an array. However, because of type erasure, the compiler converts the varargs formal parameter to Object[] elements. Consequently, there is a possibility of heap pollution.

The following statement assigns the varargs formal parameter l to the Object array objectArgs:

Object[] objectArray = l;

This statement can potentially introduce heap pollution. A value that does match the parameterized type of the varargs formal parameter l can be assigned to the variable objectArray, and thus can be assigned to l. However, the compiler does not generate an unchecked warning at this statement. The compiler has already generated a warning when it translated the varargs formal parameter List<String>... l to the formal parameter List[] l. This statement is valid; the variable l has the type List[], which is a subtype of Object[].

Consequently, the compiler does not issue a warning or error if you assign a List object of any type to any array component of the objectArray array as shown by this statement:

objectArray[0] = Arrays.asList(42);

This statement assigns to the first array component of the objectArray array with a List object that contains one object of type Integer.

Suppose you invoke ArrayBuilder.faultyMethod with the following statement:

ArrayBuilder.faultyMethod(Arrays.asList("Hello!"), Arrays.asList("World!"));

At runtime, the JVM throws a ClassCastException at the following statement:

// ClassCastException thrown here

String s = l[0].get(0);

The object stored in the first array component of the variable l has the type List<Integer>, but this statement is expecting an object of type List<String>.

## Prevent Warnings from Varargs Methods with Non-Reifiable Formal Parameters

If you declare a varargs method that has parameters of a parameterized type, and you ensure that the body of the method does not throw a ClassCastException or other similar exception due to improper handling of the varargs formal parameter, you can prevent the warning that the compiler generates for these kinds of varargs methods by adding the following annotation to static and non-constructor method declarations:

@SafeVarargs

The @SafeVarargs annotation is a documented part of the method's contract; this annotation asserts that the implementation of the method will not improperly handle the varargs formal parameter.

It is also possible, though less desirable, to suppress such warnings by adding the following to the method declaration:

@SuppressWarnings({"unchecked", "varargs"})

However, this approach does not suppress warnings generated from the method's call site. If you are unfamiliar with the @SuppressWarnings syntax, see [Annotations](http://docs.oracle.com/javase/tutorial/java/annotations/index.html).

# Restrictions on Generics

To use Java generics effectively, you must consider the following restrictions:

* [Cannot Instantiate Generic Types with Primitive Types](http://docs.oracle.com/javase/tutorial/java/generics/restrictions.html#instantiate)
* [Cannot Create Instances of Type Parameters](http://docs.oracle.com/javase/tutorial/java/generics/restrictions.html#createObjects)
* [Cannot Declare Static Fields Whose Types are Type Parameters](http://docs.oracle.com/javase/tutorial/java/generics/restrictions.html#createStatic)
* [Cannot Use Casts or instanceof With Parameterized Types](http://docs.oracle.com/javase/tutorial/java/generics/restrictions.html#cannotCast)
* [Cannot Create Arrays of Parameterized Types](http://docs.oracle.com/javase/tutorial/java/generics/restrictions.html#createArrays)
* [Cannot Create, Catch, or Throw Objects of Parameterized Types](http://docs.oracle.com/javase/tutorial/java/generics/restrictions.html#cannotCatch)
* [Cannot Overload a Method Where the Formal Parameter Types of Each Overload Erase to the Same Raw Type](http://docs.oracle.com/javase/tutorial/java/generics/restrictions.html#cannotOverload)

## Cannot Instantiate Generic Types with Primitive Types

Consider the following parameterized type:

class Pair<K, V> {

private K key;

private V value;

public Pair(K key, V value) {

this.key = key;

this.value = value;

}

// ...

}

When creating a Pair object, you cannot substitute a primitive type for the type parameter K or V:

Pair<**int, char**> p = new Pair<>(8, 'a'); // compile-time error

You can substitute only non-primitive types for the type parameters K and V:

Pair<**Integer, Character**> p = new Pair<>(8, 'a');

Note that the Java compiler autoboxes 8 to Integer.valueOf(8) and 'a' to Character('a'):

Pair<Integer, Character> p = new Pair<>(Integer.valueOf(8), new Character('a'));

For more information on autoboxing, see [Autoboxing and Unboxing](http://docs.oracle.com/javase/tutorial/java/data/autoboxing.html) in the [Numbers and Strings](http://docs.oracle.com/javase/tutorial/java/data/index.html) lesson.

## Cannot Create Instances of Type Parameters

You cannot create an instance of a type parameter. For example, the following code causes a compile-time error:

public static <E> void append(List<E> list) {

E elem = new E(); // compile-time error

list.add(elem);

}

As a workaround, you can create an object of a type parameter through reflection:

public static <E> void append(List<E> list, Class<E> cls) throws Exception {

E elem = cls.newInstance(); // OK

list.add(elem);

}

You can invoke the append method as follows:

List<String> ls = new ArrayList<>();

append(ls, String.class);

## Cannot Declare Static Fields Whose Types are Type Parameters

A class's static field is a class-level variable shared by all non-static objects of the class. Hence, static fields of type parameters are not allowed. Consider the following class:

public class MobileDevice<T> {

private static T os;

// ...

}

If static fields of type parameters were allowed, then the following code would be confused:

MobileDevice<Smartphone> phone = new MobileDevice<>();

MobileDevice<Pager> pager = new MobileDevice<>();

MobileDevice<TabletPC> pc = new MobileDevice<>();

Because the static field os is shared by phone, pager, and pc, what is the actual type of os? It cannot be Smartphone, Pager, and TabletPC at the same time. You cannot, therefore, create static fields of type parameters.

## Cannot Use Casts or instanceof with Parameterized Types

Because the Java compiler erases all type parameters in generic code, you cannot verify which parameterized type for a generic type is being used at runtime:

public static <E> void rtti(List<E> list) {

if (list instanceof ArrayList<Integer>) { // compile-time error

// ...

}

}

The set of parameterized types passed to the rtti method is:

S = { ArrayList<Integer>, ArrayList<String> LinkedList<Character>, ... }

The runtime does not keep track of type parameters, so it cannot tell the difference between an ArrayList<Integer> and an ArrayList<String>. The most you can do is to use an unbounded wildcard to verify that the list is an ArrayList:

public static void rtti(List<?> list) {

if (list instanceof ArrayList<?>) { // OK; instanceof requires a reifiable type

// ...

}

}

Typically, you cannot cast to a parameterized type unless it is parameterized by unbounded wildcards. For example:

List<Integer> li = new ArrayList<>();

List<Number> ln = (List<Number>) li; // compile-time error

However, in some cases the compiler knows that a type parameter is always valid and allows the cast. For example:

List<String> l1 = ...;

ArrayList<String> l2 = (ArrayList<String>)l1; // OK

## Cannot Create Arrays of Parameterized Types

You cannot create arrays of parameterized types. For example, the following code does not compile:

List<Integer>[] arrayOfLists = new List<Integer>[2]; // compile-time error

The following code illustrates what happens when different types are inserted into an array:

Object[] strings = new String[2];

strings[0] = "hi"; // OK

strings[1] = 100; // An ArrayStoreException is thrown.

If you try the same thing with a generic list, there would be a problem:

Object[] stringLists = new List<String>[]; // compiler error, but pretend it's allowed

stringLists[0] = new ArrayList<String>(); // OK

stringLists[1] = new ArrayList<Integer>(); // An ArrayStoreException should be thrown,

// but the runtime can't detect it.

If arrays of parameterized lists were allowed, the previous code would fail to throw the desired ArrayStoreException.

## Cannot Create, Catch, or Throw Objects of Parameterized Types

A generic class cannot extend the Throwable class directly or indirectly. For example, the following classes will not compile:

// Extends Throwable indirectly

class MathException<T> extends Exception { /\* ... \*/ } // compile-time error

// Extends Throwable directly

class QueueFullException<T> extends Throwable { /\* ... \*/ // compile-time error

A method cannot catch an instance of a type parameter:

public static <T extends Exception, J> void execute(List<J> jobs) {

try {

for (J job : jobs)

// ...

} catch (T e) { // compile-time error

// ...

}

}

You can, however, use a type parameter in a throws clause:

class Parser<T extends Exception> {

public void parse(File file) throws T { // OK

// ...

}

}

## Cannot Overload a Method Where the Formal Parameter Types of Each Overload Erase to the Same Raw Type

A class cannot have two overloaded methods that will have the same signature after type erasure.

public class Example {

public void print(Set<String> strSet) { }

public void print(Set<Integer> intSet) { }

}

The overloads would all share the same classfile representation and will generate a compile-time error.

# Creating a Package

To create a package, you choose a name for the package (naming conventions are discussed in the next section) and put a package statement with that name at the top of *every source file*that contains the types (classes, interfaces, enumerations, and annotation types) that you want to include in the package.

The package statement (for example, package graphics;) must be the first line in the source file. There can be only one package statement in each source file, and it applies to all types in the file.

**Note:** If you put multiple types in a single source file, only one can be public, and it must have the same name as the source file. For example, you can define public class Circle in the file Circle.java, define public interface Draggable in the file Draggable.java, define public enum Day in the file Day.java, and so forth.  
  
You can include non-public types in the same file as a public type (this is strongly discouraged, unless the non-public types are small and closely related to the public type), but only the public type will be accessible from outside of the package. All the top-level, non-public types will be *package private*.

If you put the graphics interface and classes listed in the preceding section in a package called graphics, you would need six source files, like this:

//in the Draggable.java file

package graphics;

public interface Draggable {

. . .

}

//in the Graphic.java file

package graphics;

public abstract class Graphic {

. . .

}

//in the Circle.java file

package graphics;

public class Circle extends Graphic

implements Draggable {

. . .

}

//in the Rectangle.java file

package graphics;

public class Rectangle extends Graphic

implements Draggable {

. . .

}

//in the Point.java file

package graphics;

public class Point extends Graphic

implements Draggable {

. . .

}

//in the Line.java file

package graphics;

public class Line extends Graphic

implements Draggable {

. . .

}

If you do not use a package statement, your type ends up in an unnamed package. Generally speaking, an unnamed package is only for small or temporary applications or when you are just beginning the development process. Otherwise, classes and interfaces belong in named packages.

# Naming a Package

With programmers worldwide writing classes and interfaces using the Java programming language, it is likely that many programmers will use the same name for different types. In fact, the previous example does just that: It defines a Rectangle class when there is already a Rectangle class in the java.awt package. Still, the compiler allows both classes to have the same name if they are in different packages. The fully qualified name of each Rectangle class includes the package name. That is, the fully qualified name of the Rectangle class in thegraphics package is graphics.Rectangle, and the fully qualified name of the Rectangle class in the java.awt package is java.awt.Rectangle.

This works well unless two independent programmers use the same name for their packages. What prevents this problem? Convention.

## Naming Conventions

Package names are written in all lower case to avoid conflict with the names of classes or interfaces.

Companies use their reversed Internet domain name to begin their package names—for example, com.example.mypackage for a package named mypackage created by a programmer atexample.com.

Name collisions that occur within a single company need to be handled by convention within that company, perhaps by including the region or the project name after the company name (for example, com.example.region.mypackage).

Packages in the Java language itself begin with java. or javax.

In some cases, the internet domain name may not be a valid package name. This can occur if the domain name contains a hyphen or other special character, if the package name begins with a digit or other character that is illegal to use as the beginning of a Java name, or if the package name contains a reserved Java keyword, such as "int". In this event, the suggested convention is to add an underscore. For example:

|  |  |
| --- | --- |
| **Legalizing Package Names** | |
| **Domain Name** | **Package Name Prefix** |
| hyphenated-name.example.org | org.example.hyphenated\_name |
| example.int | int\_.example |
| 123name.example.com | com.example.\_123name |

# Using Package Members

The types that comprise a package are known as the *package members*.

To use a public package member from outside its package, you must do one of the following:

* Refer to the member by its fully qualified name
* Import the package member
* Import the member's entire package

Each is appropriate for different situations, as explained in the sections that follow.

## Referring to a Package Member by Its Qualified Name

So far, most of the examples in this tutorial have referred to types by their simple names, such as Rectangle and StackOfInts. You can use a package member's simple name if the code you are writing is in the same package as that member or if that member has been imported.

However, if you are trying to use a member from a different package and that package has not been imported, you must use the member's fully qualified name, which includes the package name. Here is the fully qualified name for the Rectangle class declared in the graphics package in the previous example.

graphics.Rectangle

You could use this qualified name to create an instance of graphics.Rectangle:

graphics.Rectangle myRect = new graphics.Rectangle();

Qualified names are all right for infrequent use. When a name is used repetitively, however, typing the name repeatedly becomes tedious and the code becomes difficult to read. As an alternative, you can *import* the member or its package and then use its simple name.

## Importing a Package Member

To import a specific member into the current file, put an import statement at the beginning of the file before any type definitions but after the package statement, if there is one. Here's how you would import the Rectangle class from the graphics package created in the previous section.

import graphics.Rectangle;

Now you can refer to the Rectangle class by its simple name.

Rectangle myRectangle = new Rectangle();

This approach works well if you use just a few members from the graphics package. But if you use many types from a package, you should import the entire package.

## Importing an Entire Package

To import all the types contained in a particular package, use the import statement with the asterisk (\*) wildcard character.

import graphics.\*;

Now you can refer to any class or interface in the graphics package by its simple name.

Circle myCircle = new Circle();

Rectangle myRectangle = new Rectangle();

The asterisk in the import statement can be used only to specify all the classes within a package, as shown here. It cannot be used to match a subset of the classes in a package. For example, the following does not match all the classes in the graphics package that begin with A.

// does not work

import graphics.A\*;

Instead, it generates a compiler error. With the import statement, you generally import only a single package member or an entire package.

**Note:** Another, less common form of import allows you to import the public nested classes of an enclosing class. For example, if the graphics.Rectangle class contained useful nested classes, such as Rectangle.DoubleWide and Rectangle.Square, you could import Rectangle and its nested classes by using the following *two* statements.

import graphics.Rectangle;

import graphics.Rectangle.\*;

Be aware that the second import statement will *not* import Rectangle.  
  
Another less common form of import, the *static import statement*, will be discussed at the end of this section.

For convenience, the Java compiler automatically imports two entire packages for each source file: (1) the java.lang package and (2) the current package (the package for the current file).

## Apparent Hierarchies of Packages

At first, packages appear to be hierarchical, but they are not. For example, the Java API includes a java.awt package, a java.awt.color package, a java.awt.font package, and many others that begin with java.awt. However, the java.awt.color package, the java.awt.font package, and other java.awt.xxxx packages are *not included* in the java.awtpackage. The prefix java.awt (the Java Abstract Window Toolkit) is used for a number of related packages to make the relationship evident, but not to show inclusion.

Importing java.awt.\* imports all of the types in the java.awt package, but it *does not import* java.awt.color, java.awt.font, or any other java.awt.xxxx packages. If you plan to use the classes and other types in java.awt.color as well as those in java.awt, you must import both packages with all their files:

import java.awt.\*;

import java.awt.color.\*;

## Name Ambiguities

If a member in one package shares its name with a member in another package and both packages are imported, you must refer to each member by its qualified name. For example, thegraphics package defined a class named Rectangle. The java.awt package also contains a Rectangle class. If both graphics and java.awt have been imported, the following is ambiguous.

Rectangle rect;

In such a situation, you have to use the member's fully qualified name to indicate exactly which Rectangle class you want. For example,

graphics.Rectangle rect;

## The Static Import Statement

There are situations where you need frequent access to static final fields (constants) and static methods from one or two classes. Prefixing the name of these classes over and over can result in cluttered code. The *static import* statement gives you a way to import the constants and static methods that you want to use so that you do not need to prefix the name of their class.

The java.lang.Math class defines the PI constant and many static methods, including methods for calculating sines, cosines, tangents, square roots, maxima, minima, exponents, and many more. For example,

public static final double PI

= 3.141592653589793;

public static double cos(double a)

{

...

}

Ordinarily, to use these objects from another class, you prefix the class name, as follows.

double r = Math.cos(Math.PI \* theta);

You can use the static import statement to import the static members of java.lang.Math so that you don't need to prefix the class name, Math. The static members of Math can be imported either individually:

import **static** java.lang.Math.PI;

or as a group:

import **static** java.lang.Math.\*;

Once they have been imported, the static members can be used without qualification. For example, the previous code snippet would become:

double r = cos(PI \* theta);

Obviously, you can write your own classes that contain constants and static methods that you use frequently, and then use the static import statement. For example,

import **static** mypackage.MyConstants.\*;

**Note:** Use static import very sparingly. Overusing static import can result in code that is difficult to read and maintain, because readers of the code won't know which class defines a particular static object. Used properly, static import makes code more readable by removing class name repetition.

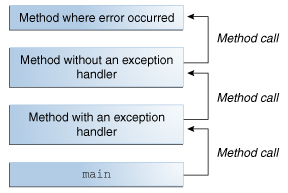
# What Is an Exception?

The term exception is shorthand for the phrase "exceptional event."

**Definition:** An exception is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.

When an error occurs within a method, the method creates an object and hands it off to the runtime system. The object, called an exception object, contains information about the error, including its type and the state of the program when the error occurred. Creating an exception object and handing it to the runtime system is called throwing an exception.

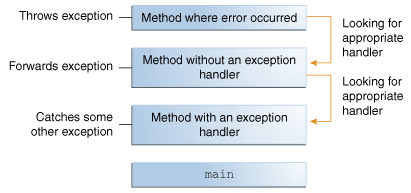
After a method throws an exception, the runtime system attempts to find something to handle it. The set of possible "somethings" to handle the exception is the ordered list of methods that had been called to get to the method where the error occurred. The list of methods is known as the call stack (see the next figure).



The call stack.

The runtime system searches the call stack for a method that contains a block of code that can handle the exception. This block of code is called an exception handler. The search begins with the method in which the error occurred and proceeds through the call stack in the reverse order in which the methods were called. When an appropriate handler is found, the runtime system passes the exception to the handler. An exception handler is considered appropriate if the type of the exception object thrown matches the type that can be handled by the handler.

The exception handler chosen is said to catch the exception. If the runtime system exhaustively searches all the methods on the call stack without finding an appropriate exception handler, as shown in the next figure, the runtime system (and, consequently, the program) terminates.



Searching the call stack for the exception handler.

Using exceptions to manage errors has some advantages over traditional error-management techniques. You can learn more in the [Advantages of Exceptions](http://docs.oracle.com/javase/tutorial/essential/exceptions/advantages.html) section.

# The Catch or Specify Requirement

Valid Java programming language code must honor the *Catch or Specify Requirement*. This means that code that might throw certain exceptions must be enclosed by either of the following:

* A try statement that catches the exception. The try must provide a handler for the exception, as described in [Catching and Handling Exceptions](http://docs.oracle.com/javase/tutorial/essential/exceptions/handling.html).
* A method that specifies that it can throw the exception. The method must provide a throws clause that lists the exception, as described in [Specifying the Exceptions Thrown by a Method](http://docs.oracle.com/javase/tutorial/essential/exceptions/declaring.html).

Code that fails to honor the Catch or Specify Requirement will not compile.

Not all exceptions are subject to the Catch or Specify Requirement. To understand why, we need to look at the three basic categories of exceptions, only one of which is subject to the Requirement.

## The Three Kinds of Exceptions

The first kind of exception is the *checked exception*. These are exceptional conditions that a well-written application should anticipate and recover from. For example, suppose an application prompts a user for an input file name, then opens the file by passing the name to the constructor for java.io.FileReader. Normally, the user provides the name of an existing, readable file, so the construction of the FileReader object succeeds, and the execution of the application proceeds normally. But sometimes the user supplies the name of a nonexistent file, and the constructor throws java.io.FileNotFoundException. A well-written program will catch this exception and notify the user of the mistake, possibly prompting for a corrected file name.

Checked exceptions *are subject* to the Catch or Specify Requirement. All exceptions are checked exceptions, except for those indicated by Error, RuntimeException, and their subclasses.

The second kind of exception is the *error*. These are exceptional conditions that are external to the application, and that the application usually cannot anticipate or recover from. For example, suppose that an application successfully opens a file for input, but is unable to read the file because of a hardware or system malfunction. The unsuccessful read will throw java.io.IOError. An application might choose to catch this exception, in order to notify the user of the problem — but it also might make sense for the program to print a stack trace and exit.

Errors *are not subject* to the Catch or Specify Requirement. Errors are those exceptions indicated by Error and its subclasses.

The third kind of exception is the *runtime exception*. These are exceptional conditions that are internal to the application, and that the application usually cannot anticipate or recover from. These usually indicate programming bugs, such as logic errors or improper use of an API. For example, consider the application described previously that passes a file name to the constructor for FileReader. If a logic error causes a null to be passed to the constructor, the constructor will throw NullPointerException. The application can catch this exception, but it probably makes more sense to eliminate the bug that caused the exception to occur.

Runtime exceptions *are not subject* to the Catch or Specify Requirement. Runtime exceptions are those indicated by RuntimeException and its subclasses.

Errors and runtime exceptions are collectively known as *unchecked exceptions*.

## Bypassing Catch or Specify

Some programmers consider the Catch or Specify Requirement a serious flaw in the exception mechanism and bypass it by using unchecked exceptions in place of checked exceptions. In general, this is not recommended. The section [Unchecked Exceptions — The Controversy](http://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html) talks about when it is appropriate to use unchecked exceptions.

# Catching and Handling Exceptions

This section describes how to use the three exception handler components — the try, catch, and finally blocks — to write an exception handler. Then, the try-with-resources statement, introduced in Java SE 7, is explained. The try-with-resources statement is particularly suited to situations that use Closeable resources, such as streams.

The last part of this section walks through an example and analyzes what occurs during various scenarios.

The following example defines and implements a class named ListOfNumbers. When constructed, ListOfNumbers creates an ArrayList that contains 10 Integer elements with sequential values 0 through 9. The ListOfNumbers class also defines a method named writeList, which writes the list of numbers into a text file called OutFile.txt. This example uses output classes defined in java.io, which are covered in [Basic I/O](http://docs.oracle.com/javase/tutorial/essential/io/index.html).

// Note: This class will not compile yet.

import java.io.\*;

import java.util.List;

import java.util.ArrayList;

public class ListOfNumbers {

private List<Integer> list;

private static final int SIZE = 10;

public ListOfNumbers () {

list = new ArrayList<Integer>(SIZE);

for (int i = 0; i < SIZE; i++) {

list.add(new Integer(i));

}

}

public void writeList() {

// The FileWriter constructor throws IOException, which must be caught.

PrintWriter out = new PrintWriter(**new FileWriter("OutFile.txt")**);

for (int i = 0; i < SIZE; i++) {

// The get(int) method throws IndexOutOfBoundsException, which must be caught.

out.println("Value at: " + i + " = " + **list.get(i)**);

}

out.close();

}

}

The first line in boldface is a call to a constructor. The constructor initializes an output stream on a file. If the file cannot be opened, the constructor throws an IOException. The second boldface line is a call to the ArrayList class's get method, which throws an IndexOutOfBoundsException if the value of its argument is too small (less than 0) or too large (more than the number of elements currently contained by the ArrayList).

If you try to compile the [ListOfNumbers](http://docs.oracle.com/javase/tutorial/essential/exceptions/examples/ListOfNumbers.java) class, the compiler prints an error message about the exception thrown by the FileWriter constructor. However, it does not display an error message about the exception thrown by get. The reason is that the exception thrown by the constructor, IOException, is a checked exception, and the one thrown by the get method, IndexOutOfBoundsException, is an unchecked exception.

Now that you're familiar with the ListOfNumbers class and where the exceptions can be thrown within it, you're ready to write exception handlers to catch and handle those exceptions.

# The try Block

The first step in constructing an exception handler is to enclose the code that might throw an exception within a try block. In general, a try block looks like the following:

try {

code

}

catch and finally blocks . . .

The segment in the example labeled code contains one or more legal lines of code that could throw an exception. (The catch and finally blocks are explained in the next two subsections.)

To construct an exception handler for the writeList method from the ListOfNumbers class, enclose the exception-throwing statements of the writeList method within a try block. There is more than one way to do this. You can put each line of code that might throw an exception within its own try block and provide separate exception handlers for each. Or, you can put all the writeList code within a single try block and associate multiple handlers with it. The following listing uses one try block for the entire method because the code in question is very short.

private List<Integer> list;

private static final int SIZE = 10;

public void writeList() {

PrintWriter out = null;

try {

System.out.println("Entered try statement");

out = new PrintWriter(new FileWriter("OutFile.txt"));

for (int i = 0; i < SIZE; i++) {

out.println("Value at: " + i + " = " + list.get(i));

}

}

catch and finally blocks . . .

}

If an exception occurs within the try block, that exception is handled by an exception handler associated with it. To associate an exception handler with a try block, you must put a catch block after it; the next section, [The catch Blocks](http://docs.oracle.com/javase/tutorial/essential/exceptions/catch.html), shows you how.

# The catch Blocks

You associate exception handlers with a try block by providing one or more catch blocks directly after the try block. No code can be between the end of the try block and the beginning of the first catch block.

try {

} catch (*ExceptionType name*) {

} catch (*ExceptionType name*) {

}

Each catch block is an exception handler that handles the type of exception indicated by its argument. The argument type, ExceptionType, declares the type of exception that the handler can handle and must be the name of a class that inherits from the Throwable class. The handler can refer to the exception with name.

The catch block contains code that is executed if and when the exception handler is invoked. The runtime system invokes the exception handler when the handler is the first one in the call stack whose ExceptionType matches the type of the exception thrown. The system considers it a match if the thrown object can legally be assigned to the exception handler's argument.

The following are two exception handlers for the writeList method:

try {

} catch (IndexOutOfBoundsException e) {

System.err.println("IndexOutOfBoundsException: " + e.getMessage());

} catch (IOException e) {

System.err.println("Caught IOException: " + e.getMessage());

}

Exception handlers can do more than just print error messages or halt the program. They can do error recovery, prompt the user to make a decision, or propagate the error up to a higher-level handler using chained exceptions, as described in the [Chained Exceptions](http://docs.oracle.com/javase/tutorial/essential/exceptions/chained.html) section.

## Catching More Than One Type of Exception with One Exception Handler

In Java SE 7 and later, a single catch block can handle more than one type of exception. This feature can reduce code duplication and lessen the temptation to catch an overly broad exception.

In the catch clause, specify the types of exceptions that block can handle, and separate each exception type with a vertical bar (|):

catch (IOException|SQLException ex) {

logger.log(ex);

throw ex;

}

**Note**: If a catch block handles more than one exception type, then the catch parameter is implicitly final. In this example, the catch parameter ex is final and therefore you cannot assign any values to it within the catch block.

# The finally Block

The finally block *always* executes when the try block exits. This ensures that the finally block is executed even if an unexpected exception occurs. But finally is useful for more than just exception handling — it allows the programmer to avoid having cleanup code accidentally bypassed by a return, continue, or break. Putting cleanup code in a finally block is always a good practice, even when no exceptions are anticipated.

**Note:** If the JVM exits while the try or catch code is being executed, then the finally block may not execute. Likewise, if the thread executing the try or catch code is interrupted or killed, the finally block may not execute even though the application as a whole continues.

The try block of the writeList method that you've been working with here opens a PrintWriter. The program should close that stream before exiting the writeList method. This poses a somewhat complicated problem because writeList's try block can exit in one of three ways.

1. The new FileWriter statement fails and throws an IOException.
2. The list.get(i) statement fails and throws an IndexOutOfBoundsException.
3. Everything succeeds and the try block exits normally.

The runtime system always executes the statements within the finally block regardless of what happens within the try block. So it's the perfect place to perform cleanup.

The following finally block for the writeList method cleans up and then closes the PrintWriter.

finally {

if (out != null) {

System.out.println("Closing PrintWriter");

out.close();

} else {

System.out.println("PrintWriter not open");

}

}

**Important:** The finally block is a key tool for preventing resource leaks. When closing a file or otherwise recovering resources, place the code in a finally block to ensure that resource is *always* recovered.  
  
Consider using the try-with-resources statement in these situations, which automatically releases system resources when no longer needed. The [The try-with-resources Statement](http://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html) section has more information.

# The try-with-resources Statement

The try-with-resources statement is a try statement that declares one or more resources. A resource is an object that must be closed after the program is finished with it. The try-with-resources statement ensures that each resource is closed at the end of the statement. Any object that implements java.lang.AutoCloseable, which includes all objects which implement java.io.Closeable, can be used as a resource.

The following example reads the first line from a file. It uses an instance of BufferedReader to read data from the file. BufferedReader is a resource that must be closed after the program is finished with it:

static String readFirstLineFromFile(String path) throws IOException {

**try (BufferedReader br =**

**new BufferedReader(new FileReader(path)))** {

return br.readLine();

}

}

In this example, the resource declared in the try-with-resources statement is a BufferedReader. The declaration statement appears within parentheses immediately after the try keyword. The class BufferedReader, in Java SE 7 and later, implements the interface java.lang.AutoCloseable. Because the BufferedReader instance is declared in a try-with-resource statement, it will be closed regardless of whether the try statement completes normally or abruptly (as a result of the method BufferedReader.readLine throwing an IOException).

Prior to Java SE 7, you can use a finally block to ensure that a resource is closed regardless of whether the try statement completes normally or abruptly. The following example uses a finally block instead of a try-with-resources statement:

static String readFirstLineFromFileWithFinallyBlock(String path)

throws IOException {

BufferedReader br = new BufferedReader(new FileReader(path));

try {

return br.readLine();

} finally {

if (br != null) br.close();

}

}

However, in this example, if the methods readLine and close both throw exceptions, then the method readFirstLineFromFileWithFinallyBlock throws the exception thrown from the finally block; the exception thrown from the try block is suppressed. In contrast, in the example readFirstLineFromFile, if exceptions are thrown from both the try block and the try-with-resources statement, then the method readFirstLineFromFile throws the exception thrown from the try block; the exception thrown from the try-with-resources block is suppressed. In Java SE 7 and later, you can retrieve suppressed exceptions; see the section [Suppressed Exceptions](http://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html#suppressed-exceptions#suppressed-exceptions) for more information.

You may declare one or more resources in a try-with-resources statement. The following example retrieves the names of the files packaged in the zip file zipFileName and creates a text file that contains the names of these files:

public static void writeToFileZipFileContents(String zipFileName,

String outputFileName)

throws java.io.IOException {

java.nio.charset.Charset charset =

java.nio.charset.StandardCharsets.US\_ASCII;

java.nio.file.Path outputFilePath =

java.nio.file.Paths.get(outputFileName);

// Open zip file and create output file with

// try-with-resources statement

**try (**

**java.util.zip.ZipFile zf =**

**new java.util.zip.ZipFile(zipFileName);**

**java.io.BufferedWriter writer =**

**java.nio.file.Files.newBufferedWriter(outputFilePath, charset)**

**)** {

// Enumerate each entry

for (java.util.Enumeration entries =

zf.entries(); entries.hasMoreElements();) {

// Get the entry name and write it to the output file

String newLine = System.getProperty("line.separator");

String zipEntryName =

((java.util.zip.ZipEntry)entries.nextElement()).getName() +

newLine;

writer.write(zipEntryName, 0, zipEntryName.length());

}

}

}

In this example, the try-with-resources statement contains two declarations that are separated by a semicolon: ZipFile and BufferedWriter. When the block of code that directly follows it terminates, either normally or because of an exception, the close methods of the BufferedWriter and ZipFile objects are automatically called in this order. Note that the close methods of resources are called in the opposite order of their creation.

The following example uses a try-with-resources statement to automatically close a java.sql.Statement object:

public static void viewTable(Connection con) throws SQLException {

String query = "select COF\_NAME, SUP\_ID, PRICE, SALES, TOTAL from COFFEES";

**try (Statement stmt = con.createStatement())** {

ResultSet rs = stmt.executeQuery(query);

while (rs.next()) {

String coffeeName = rs.getString("COF\_NAME");

int supplierID = rs.getInt("SUP\_ID");

float price = rs.getFloat("PRICE");

int sales = rs.getInt("SALES");

int total = rs.getInt("TOTAL");

System.out.println(coffeeName + ", " + supplierID + ", " +

price + ", " + sales + ", " + total);

}

} catch (SQLException e) {

JDBCTutorialUtilities.printSQLException(e);

}

}

The resource java.sql.Statement used in this example is part of the JDBC 4.1 and later API.

**Note**: A try-with-resources statement can have catch and finally blocks just like an ordinary try statement. In a try-with-resources statement, any catch or finally block is run after the resources declared have been closed.

## Suppressed Exceptions

An exception can be thrown from the block of code associated with the try-with-resources statement. In the example writeToFileZipFileContents, an exception can be thrown from the try block, and up to two exceptions can be thrown from the try-with-resources statement when it tries to close the ZipFile and BufferedWriter objects. If an exception is thrown from the try block and one or more exceptions are thrown from the try-with-resources statement, then those exceptions thrown from the try-with-resources statement are suppressed, and the exception thrown by the block is the one that is thrown by the writeToFileZipFileContents method. You can retrieve these suppressed exceptions by calling the Throwable.getSuppressed method from the exception thrown by the try block.

## Classes That Implement the AutoCloseable or Closeable Interface

See the Javadoc of the [AutoCloseable](http://docs.oracle.com/javase/8/docs/api/java/lang/AutoCloseable.html) and [Closeable](http://docs.oracle.com/javase/8/docs/api/java/io/Closeable.html) interfaces for a list of classes that implement either of these interfaces. The Closeable interface extends the AutoCloseable interface. The close method of the Closeable interface throws exceptions of type IOException while the close method of the AutoCloseable interface throws exceptions of type Exception. Consequently, subclasses of the AutoCloseable interface can override this behavior of the close method to throw specialized exceptions, such as IOException, or no exception at all.

# Specifying the Exceptions Thrown by a Method

The previous section showed how to write an exception handler for the writeList method in the ListOfNumbers class. Sometimes, it's appropriate for code to catch exceptions that can occur within it. In other cases, however, it's better to let a method further up the call stack handle the exception. For example, if you were providing the ListOfNumbers class as part of a package of classes, you probably couldn't anticipate the needs of all the users of your package. In this case, it's better to not catch the exception and to allow a method further up the call stack to handle it.

If the writeList method doesn't catch the checked exceptions that can occur within it, the writeList method must specify that it can throw these exceptions. Let's modify the original writeList method to specify the exceptions it can throw instead of catching them. To remind you, here's the original version of the writeList method that won't compile.

public void writeList() {

PrintWriter out = new PrintWriter(new FileWriter("OutFile.txt"));

for (int i = 0; i < SIZE; i++) {

out.println("Value at: " + i + " = " + list.get(i));

}

out.close();

}

To specify that writeList can throw two exceptions, add a throws clause to the method declaration for the writeList method. The throws clause comprises the throws keyword followed by a comma-separated list of all the exceptions thrown by that method. The clause goes after the method name and argument list and before the brace that defines the scope of the method; here's an example.

public void writeList() **throws IOException, IndexOutOfBoundsException** {

Remember that IndexOutOfBoundsException is an unchecked exception; including it in the throws clause is not mandatory. You could just write the following.

public void writeList() **throws IOException** {

# How to Throw Exceptions

Before you can catch an exception, some code somewhere must throw one. Any code can throw an exception: your code, code from a package written by someone else such as the packages that come with the Java platform, or the Java runtime environment. Regardless of what throws the exception, it's always thrown with the throw statement.

As you have probably noticed, the Java platform provides numerous exception classes. All the classes are descendants of the [Throwable](http://docs.oracle.com/javase/8/docs/api/java/lang/Throwable.html) class, and all allow programs to differentiate among the various types of exceptions that can occur during the execution of a program.

You can also create your own exception classes to represent problems that can occur within the classes you write. In fact, if you are a package developer, you might have to create your own set of exception classes to allow users to differentiate an error that can occur in your package from errors that occur in the Java platform or other packages.

You can also create chained exceptions. For more information, see the [Chained Exceptions](http://docs.oracle.com/javase/tutorial/essential/exceptions/chained.html) section.

## The throw Statement

All methods use the throw statement to throw an exception. The throw statement requires a single argument: a throwable object. Throwable objects are instances of any subclass of the Throwable class. Here's an example of a throw statement.

throw *someThrowableObject*;

Let's look at the throw statement in context. The following pop method is taken from a class that implements a common stack object. The method removes the top element from the stack and returns the object.

public Object pop() {

Object obj;

if (size == 0) {

**throw new EmptyStackException();**

}

obj = objectAt(size - 1);

setObjectAt(size - 1, null);

size--;

return obj;

}

The pop method checks to see whether any elements are on the stack. If the stack is empty (its size is equal to 0), pop instantiates a new EmptyStackException object (a member of java.util) and throws it. The [Creating Exception Classes](http://docs.oracle.com/javase/tutorial/essential/exceptions/creating.html) section in this chapter explains how to create your own exception classes. For now, all you need to remember is that you can throw only objects that inherit from the java.lang.Throwable class.

Note that the declaration of the pop method does not contain a throws clause. EmptyStackException is not a checked exception, so pop is not required to state that it might occur.

## Throwable Class and Its Subclasses

The objects that inherit from the Throwable class include direct descendants (objects that inherit directly from the Throwable class) and indirect descendants (objects that inherit from children or grandchildren of the Throwable class). The figure below illustrates the class hierarchy of the Throwable class and its most significant subclasses. As you can see, Throwable has two direct descendants: [Error](http://docs.oracle.com/javase/8/docs/api/java/lang/Error.html) and [Exception](http://docs.oracle.com/javase/8/docs/api/java/lang/Exception.html)

# Chained Exceptions

An application often responds to an exception by throwing another exception. In effect, the first exception *causes* the second exception. It can be very helpful to know when one exception causes another. *Chained Exceptions* help the programmer do this.

The following are the methods and constructors in Throwable that support chained exceptions.

Throwable getCause()

Throwable initCause(Throwable)

Throwable(String, Throwable)

Throwable(Throwable)

The Throwable argument to initCause and the Throwable constructors is the exception that caused the current exception. getCause returns the exception that caused the current exception, and initCause sets the current exception's cause.

The following example shows how to use a chained exception.

try {

} catch (IOException e) {

throw new SampleException("Other IOException", e);

}

In this example, when an IOException is caught, a new SampleException exception is created with the original cause attached and the chain of exceptions is thrown up to the next higher level exception handler.

## Accessing Stack Trace Information

Now let's suppose that the higher-level exception handler wants to dump the stack trace in its own format.

**Definition:** A stack trace provides information on the execution history of the current thread and lists the names of the classes and methods that were called at the point when the exception occurred. A stack trace is a useful debugging tool that you'll normally take advantage of when an exception has been thrown.

The following code shows how to call the getStackTrace method on the exception object.

catch (Exception cause) {

StackTraceElement elements[] = cause.getStackTrace();

for (int i = 0, n = elements.length; i < n; i++) {

System.err.println(elements[i].getFileName()

+ ":" + elements[i].getLineNumber()

+ ">> "

+ elements[i].getMethodName() + "()");

}

}

## Logging API

The next code snippet logs where an exception occurred from within the catch block. However, rather than manually parsing the stack trace and sending the output to System.err(), it sends the output to a file using the logging facility in the [java.util.logging](http://docs.oracle.com/javase/8/docs/api/java/util/logging/package-summary.html) package.

try {

Handler handler = new FileHandler("OutFile.log");

Logger.getLogger("").addHandler(handler);

} catch (IOException e) {

Logger logger = Logger.getLogger("package.name");

StackTraceElement elements[] = e.getStackTrace();

for (int i = 0, n = elements.length; i < n; i++) {

logger.log(Level.WARNING, elements[i].getMethodName());

}

}

# Creating Exception Classes

When faced with choosing the type of exception to throw, you can either use one written by someone else — the Java platform provides a lot of exception classes you can use — or you can write one of your own. You should write your own exception classes if you answer yes to any of the following questions; otherwise, you can probably use someone else's.

* Do you need an exception type that isn't represented by those in the Java platform?
* Would it help users if they could differentiate your exceptions from those thrown by classes written by other vendors?
* Does your code throw more than one related exception?
* If you use someone else's exceptions, will users have access to those exceptions? A similar question is, should your package be independent and self-contained?

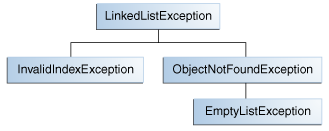
## An Example

Suppose you are writing a linked list class. The class supports the following methods, among others:

* **objectAt(int n)** — Returns the object in the nth position in the list. Throws an exception if the argument is less than 0 or more than the number of objects currently in the list.
* **firstObject()** — Returns the first object in the list. Throws an exception if the list contains no objects.
* **indexOf(Object o)** — Searches the list for the specified Object and returns its position in the list. Throws an exception if the object passed into the method is not in the list.

The linked list class can throw multiple exceptions, and it would be convenient to be able to catch all exceptions thrown by the linked list with one exception handler. Also, if you plan to distribute your linked list in a package, all related code should be packaged together. Thus, the linked list should provide its own set of exception classes.

The next figure illustrates one possible class hierarchy for the exceptions thrown by the linked list.



Example exception class hierarchy.

## Choosing a Superclass

Any Exception subclass can be used as the parent class of LinkedListException. However, a quick perusal of those subclasses shows that they are inappropriate because they are either too specialized or completely unrelated to LinkedListException. Therefore, the parent class of LinkedListException should be Exception.

Most applets and applications you write will throw objects that are Exceptions. Errors are normally used for serious, hard errors in the system, such as those that prevent the JVM from running.

**Note:** For readable code, it's good practice to append the string Exception to the names of all classes that inherit (directly or indirectly) from the Exception class.

# Unchecked Exceptions — The Controversy

Because the Java programming language does not require methods to catch or to specify unchecked exceptions (RuntimeException, Error, and their subclasses), programmers may be tempted to write code that throws only unchecked exceptions or to make all their exception subclasses inherit from RuntimeException. Both of these shortcuts allow programmers to write code without bothering with compiler errors and without bothering to specify or to catch any exceptions. Although this may seem convenient to the programmer, it sidesteps the intent of the catch or specify requirement and can cause problems for others using your classes.

Why did the designers decide to force a method to specify all uncaught checked exceptions that can be thrown within its scope? Any Exception that can be thrown by a method is part of the method's public programming interface. Those who call a method must know about the exceptions that a method can throw so that they can decide what to do about them. These exceptions are as much a part of that method's programming interface as its parameters and return value.

The next question might be: "If it's so good to document a method's API, including the exceptions it can throw, why not specify runtime exceptions too?" Runtime exceptions represent problems that are the result of a programming problem, and as such, the API client code cannot reasonably be expected to recover from them or to handle them in any way. Such problems include arithmetic exceptions, such as dividing by zero; pointer exceptions, such as trying to access an object through a null reference; and indexing exceptions, such as attempting to access an array element through an index that is too large or too small.

Runtime exceptions can occur anywhere in a program, and in a typical one they can be very numerous. Having to add runtime exceptions in every method declaration would reduce a program's clarity. Thus, the compiler does not require that you catch or specify runtime exceptions (although you can).

One case where it is common practice to throw a RuntimeException is when the user calls a method incorrectly. For example, a method can check if one of its arguments is incorrectly null. If an argument is null, the method might throw a NullPointerException, which is an unchecked exception.

Generally speaking, do not throw a RuntimeException or create a subclass of RuntimeException simply because you don't want to be bothered with specifying the exceptions your methods can throw.

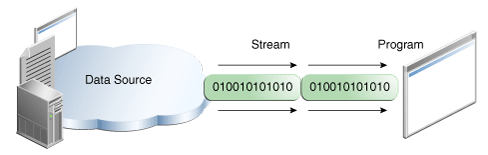
Here's the bottom line guideline: If a client can reasonably be expected to recover from an exception, make it a checked exception. If a client cannot do anything to recover from the exception, make it an unchecked exception.

# I/O Streams

An I/O Stream represents an input source or an output destination. A stream can represent many different kinds of sources and destinations, including disk files, devices, other programs, and memory arrays.

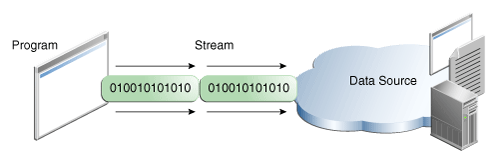
Streams support many different kinds of data, including simple bytes, primitive data types, localized characters, and objects. Some streams simply pass on data; others manipulate and transform the data in useful ways.

No matter how they work internally, all streams present the same simple model to programs that use them: A stream is a sequence of data. A program uses an *input stream* to read data from a source, one item at a time:



Reading information into a program.

A program uses an *output stream* to write data to a destination, one item at time:



Writing information from a program.

In this lesson, we'll see streams that can handle all kinds of data, from primitive values to advanced objects.

The data source and data destination pictured above can be anything that holds, generates, or consumes data. Obviously this includes disk files, but a source or destination can also be another program, a peripheral device, a network socket, or an array.

In the next section, we'll use the most basic kind of streams, byte streams, to demonstrate the common operations of Stream I/O. For sample input, we'll use the example file [xanadu.txt](https://docs.oracle.com/javase/tutorial/essential/io/examples/xanadu.txt), which contains the following verse:

In Xanadu did Kubla Khan

A stately pleasure-dome decree:

Where Alph, the sacred river, ran

Through caverns measureless to man

Down to a sunless sea.

# Byte Streams

Programs use byte streams to perform input and output of 8-bit bytes. All byte stream classes are descended from [InputStream](http://docs.oracle.com/javase/8/docs/api/java/io/InputStream.html) and [OutputStream](http://docs.oracle.com/javase/8/docs/api/java/io/OutputStream.html).

There are many byte stream classes. To demonstrate how byte streams work, we'll focus on the file I/O byte streams, [FileInputStream](http://docs.oracle.com/javase/8/docs/api/java/io/FileInputStream.html) and [FileOutputStream](http://docs.oracle.com/javase/8/docs/api/java/io/FileOutputStream.html). Other kinds of byte streams are used in much the same way; they differ mainly in the way they are constructed.

## Using Byte Streams

We'll explore FileInputStream and FileOutputStream by examining an example program named [CopyBytes](https://docs.oracle.com/javase/tutorial/essential/io/examples/CopyBytes.java), which uses byte streams to copy xanadu.txt, one byte at a time.

import java.io.FileInputStream;

import java.io.FileOutputStream;

import java.io.IOException;

public class CopyBytes {

public static void main(String[] args) throws IOException {

FileInputStream in = null;

FileOutputStream out = null;

try {

in = new FileInputStream("xanadu.txt");

out = new FileOutputStream("outagain.txt");

int c;

while ((c = in.read()) != -1) {

out.write(c);

}

} finally {

if (in != null) {

in.close();

}

if (out != null) {

out.close();

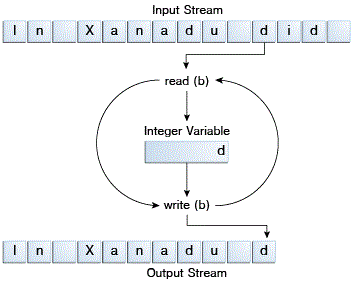
}

}

}

}

CopyBytes spends most of its time in a simple loop that reads the input stream and writes the output stream, one byte at a time, as shown in the following figure.



Simple byte stream input and output.

## Always Close Streams

Closing a stream when it's no longer needed is very important — so important that CopyBytes uses a finally block to guarantee that both streams will be closed even if an error occurs. This practice helps avoid serious resource leaks.

One possible error is that CopyBytes was unable to open one or both files. When that happens, the stream variable corresponding to the file never changes from its initial null value. That's why CopyBytes makes sure that each stream variable contains an object reference before invoking close.

## When Not to Use Byte Streams

CopyBytes seems like a normal program, but it actually represents a kind of low-level I/O that you should avoid. Since xanadu.txt contains character data, the best approach is to use [character streams](https://docs.oracle.com/javase/tutorial/essential/io/charstreams.html), as discussed in the next section. There are also streams for more complicated data types. Byte streams should only be used for the most primitive I/O.

So why talk about byte streams? Because all other stream types are built on byte streams.

# Character Streams

The Java platform stores character values using Unicode conventions. Character stream I/O automatically translates this internal format to and from the local character set. In Western locales, the local character set is usually an 8-bit superset of ASCII.

For most applications, I/O with character streams is no more complicated than I/O with byte streams. Input and output done with stream classes automatically translates to and from the local character set. A program that uses character streams in place of byte streams automatically adapts to the local character set and is ready for internationalization — all without extra effort by the programmer.

If internationalization isn't a priority, you can simply use the character stream classes without paying much attention to character set issues. Later, if internationalization becomes a priority, your program can be adapted without extensive recoding. See the [Internationalization](https://docs.oracle.com/javase/tutorial/i18n/index.html) trail for more information.

## Using Character Streams

All character stream classes are descended from [Reader](http://docs.oracle.com/javase/8/docs/api/java/io/Reader.html) and [Writer](http://docs.oracle.com/javase/8/docs/api/java/io/Writer.html). As with byte streams, there are character stream classes that specialize in file I/O: [FileReader](http://docs.oracle.com/javase/8/docs/api/java/io/FileReader.html) and [FileWriter](http://docs.oracle.com/javase/8/docs/api/java/io/FileWriter.html). The [CopyCharacters](https://docs.oracle.com/javase/tutorial/essential/io/examples/CopyCharacters.java) example illustrates these classes.

import java.io.FileReader;

import java.io.FileWriter;

import java.io.IOException;

public class CopyCharacters {

public static void main(String[] args) throws IOException {

FileReader inputStream = null;

FileWriter outputStream = null;

try {

inputStream = new FileReader("xanadu.txt");

outputStream = new FileWriter("characteroutput.txt");

int c;

while ((c = inputStream.read()) != -1) {

outputStream.write(c);

}

} finally {

if (inputStream != null) {

inputStream.close();

}

if (outputStream != null) {

outputStream.close();

}

}

}

}

CopyCharacters is very similar to CopyBytes. The most important difference is that CopyCharacters uses FileReader and FileWriter for input and output in place of FileInputStream and FileOutputStream. Notice that both CopyBytes and CopyCharacters use an int variable to read to and write from. However, in CopyCharacters, the int variable holds a character value in its last 16 bits; in CopyBytes, the int variable holds a byte value in its last 8 bits.

### Character Streams that Use Byte Streams

Character streams are often "wrappers" for byte streams. The character stream uses the byte stream to perform the physical I/O, while the character stream handles translation between characters and bytes. FileReader, for example, uses FileInputStream, while FileWriter uses FileOutputStream.

There are two general-purpose byte-to-character "bridge" streams: [InputStreamReader](http://docs.oracle.com/javase/8/docs/api/java/io/InputStreamReader.html) and [OutputStreamWriter](http://docs.oracle.com/javase/8/docs/api/java/io/OutputStreamWriter.html). Use them to create character streams when there are no prepackaged character stream classes that meet your needs. The [sockets lesson](https://docs.oracle.com/javase/tutorial/networking/sockets/readingWriting.html) in the [networking trail](https://docs.oracle.com/javase/tutorial/networking/index.html) shows how to create character streams from the byte streams provided by socket classes.

## Line-Oriented I/O

Character I/O usually occurs in bigger units than single characters. One common unit is the line: a string of characters with a line terminator at the end. A line terminator can be a carriage-return/line-feed sequence ("\r\n"), a single carriage-return ("\r"), or a single line-feed ("\n"). Supporting all possible line terminators allows programs to read text files created on any of the widely used operating systems.

Let's modify the CopyCharacters example to use line-oriented I/O. To do this, we have to use two classes we haven't seen before, [BufferedReader](http://docs.oracle.com/javase/8/docs/api/java/io/BufferedReader.html) and [PrintWriter](http://docs.oracle.com/javase/8/docs/api/java/io/PrintWriter.html). We'll explore these classes in greater depth in [Buffered I/O](https://docs.oracle.com/javase/tutorial/essential/io/buffers.html) and [Formatting](https://docs.oracle.com/javase/tutorial/essential/io/formatting.html). Right now, we're just interested in their support for line-oriented I/O.

The [CopyLines](https://docs.oracle.com/javase/tutorial/essential/io/examples/CopyLines.java) example invokes BufferedReader.readLine and PrintWriter.println to do input and output one line at a time.

import java.io.FileReader;

import java.io.FileWriter;

import java.io.BufferedReader;

import java.io.PrintWriter;

import java.io.IOException;

public class CopyLines {

public static void main(String[] args) throws IOException {

BufferedReader inputStream = null;

PrintWriter outputStream = null;

try {

inputStream = new BufferedReader(new FileReader("xanadu.txt"));

outputStream = new PrintWriter(new FileWriter("characteroutput.txt"));

String l;

while ((l = inputStream.readLine()) != null) {

outputStream.println(l);

}

} finally {

if (inputStream != null) {

inputStream.close();

}

if (outputStream != null) {

outputStream.close();

}

}

}

}

Invoking readLine returns a line of text with the line. CopyLines outputs each line using println, which appends the line terminator for the current operating system. This might not be the same line terminator that was used in the input file.

There are many ways to structure text input and output beyond characters and lines. For more information, see [Scanning and Formatting](https://docs.oracle.com/javase/tutorial/essential/io/scanfor.html).

# Buffered Streams

Most of the examples we've seen so far use *unbuffered* I/O. This means each read or write request is handled directly by the underlying OS. This can make a program much less efficient, since each such request often triggers disk access, network activity, or some other operation that is relatively expensive.

To reduce this kind of overhead, the Java platform implements *buffered* I/O streams. Buffered input streams read data from a memory area known as a *buffer*; the native input API is called only when the buffer is empty. Similarly, buffered output streams write data to a buffer, and the native output API is called only when the buffer is full.

A program can convert an unbuffered stream into a buffered stream using the wrapping idiom we've used several times now, where the unbuffered stream object is passed to the constructor for a buffered stream class. Here's how you might modify the constructor invocations in the CopyCharacters example to use buffered I/O:

inputStream = new BufferedReader(new FileReader("xanadu.txt"));

outputStream = new BufferedWriter(new FileWriter("characteroutput.txt"));

There are four buffered stream classes used to wrap unbuffered streams: [BufferedInputStream](http://docs.oracle.com/javase/8/docs/api/java/io/BufferedInputStream.html) and [BufferedOutputStream](http://docs.oracle.com/javase/8/docs/api/java/io/BufferedOutputStream.html) create buffered byte streams, while [BufferedReader](http://docs.oracle.com/javase/8/docs/api/java/io/BufferedReader.html) and [BufferedWriter](http://docs.oracle.com/javase/8/docs/api/java/io/BufferedWriter.html) create buffered character streams.

## Flushing Buffered Streams

It often makes sense to write out a buffer at critical points, without waiting for it to fill. This is known as *flushing* the buffer.

Some buffered output classes support *autoflush*, specified by an optional constructor argument. When autoflush is enabled, certain key events cause the buffer to be flushed. For example, an autoflush PrintWriter object flushes the buffer on every invocation of println or format. See [Formatting](https://docs.oracle.com/javase/tutorial/essential/io/formatting.html) for more on these methods.

To flush a stream manually, invoke its flush method. The flush method is valid on any output stream, but has no effect unless the stream is buffered.

# Scanning and Formatting

Programming I/O often involves translating to and from the neatly formatted data humans like to work with. To assist you with these chores, the Java platform provides two APIs. The [scanner](https://docs.oracle.com/javase/tutorial/essential/io/scanning.html) API breaks input into individual tokens associated with bits of data. The [formatting](https://docs.oracle.com/javase/tutorial/essential/io/formatting.html) API assembles data into nicely formatted, human-readable form.

# Scanning

Objects of type [Scanner](http://docs.oracle.com/javase/8/docs/api/java/util/Scanner.html) are useful for breaking down formatted input into tokens and translating individual tokens according to their data type.

## Breaking Input into Tokens

By default, a scanner uses white space to separate tokens. (White space characters include blanks, tabs, and line terminators. For the full list, refer to the documentation for [Character.isWhitespace](http://docs.oracle.com/javase/8/docs/api/java/lang/Character.html#isWhitespace-char-).) To see how scanning works, let's look at [ScanXan](https://docs.oracle.com/javase/tutorial/essential/io/examples/ScanXan.java), a program that reads the individual words in xanadu.txt and prints them out, one per line.

import java.io.\*;

import java.util.Scanner;

public class ScanXan {

public static void main(String[] args) throws IOException {

Scanner s = null;

try {

s = new Scanner(new BufferedReader(new FileReader("xanadu.txt")));

while (s.hasNext()) {

System.out.println(s.next());

}

} finally {

if (s != null) {

s.close();

}

}

}

}

Notice that ScanXan invokes Scanner's close method when it is done with the scanner object. Even though a scanner is not a stream, you need to close it to indicate that you're done with its underlying stream.

The output of ScanXan looks like this:

In

Xanadu

did

Kubla

Khan

A

stately

pleasure-dome

...

To use a different token separator, invoke useDelimiter(), specifying a regular expression. For example, suppose you wanted the token separator to be a comma, optionally followed by white space. You would invoke,

s.useDelimiter(",\\s\*");

## Translating Individual Tokens

The ScanXan example treats all input tokens as simple String values. Scanner also supports tokens for all of the Java language's primitive types (except for char), as well as BigInteger and BigDecimal. Also, numeric values can use thousands separators. Thus, in a US locale, Scanner correctly reads the string "32,767" as representing an integer value.

We have to mention the locale, because thousands separators and decimal symbols are locale specific. So, the following example would not work correctly in all locales if we didn't specify that the scanner should use the US locale. That's not something you usually have to worry about, because your input data usually comes from sources that use the same locale as you do. But this example is part of the Java Tutorial and gets distributed all over the world.

The [ScanSum](https://docs.oracle.com/javase/tutorial/essential/io/examples/ScanSum.java) example reads a list of double values and adds them up. Here's the source:

import java.io.FileReader;

import java.io.BufferedReader;

import java.io.IOException;

import java.util.Scanner;

import java.util.Locale;

public class ScanSum {

public static void main(String[] args) throws IOException {

Scanner s = null;

double sum = 0;

try {

s = new Scanner(new BufferedReader(new FileReader("usnumbers.txt")));

s.useLocale(Locale.US);

while (s.hasNext()) {

if (s.hasNextDouble()) {

sum += s.nextDouble();

} else {

s.next();

}

}

} finally {

s.close();

}

System.out.println(sum);

}

}

And here's the sample input file, [usnumbers.txt](https://docs.oracle.com/javase/tutorial/essential/io/examples/usnumbers.txt)

8.5

32,767

3.14159

1,000,000.1

The output string is "1032778.74159". The period will be a different character in some locales, because System.out is a PrintStream object, and that class doesn't provide a way to override the default locale. We could override the locale for the whole program — or we could just use formatting, as described in the next topic, [Formatting](https://docs.oracle.com/javase/tutorial/essential/io/formatting.html).

# Formatting

Stream objects that implement formatting are instances of either [PrintWriter](http://docs.oracle.com/javase/8/docs/api/java/io/PrintWriter.html), a character stream class, or [PrintStream](http://docs.oracle.com/javase/8/docs/api/java/io/PrintStream.html), a byte stream class.

**Note:** The only PrintStream objects you are likely to need are [System.out](http://docs.oracle.com/javase/8/docs/api/java/lang/System.html#out) and [System.err](http://docs.oracle.com/javase/8/docs/api/java/lang/System.html#err). (See [I/O from the Command Line](https://docs.oracle.com/javase/tutorial/essential/io/cl.html) for more on these objects.) When you need to create a formatted output stream, instantiate PrintWriter, not PrintStream.

Like all byte and character stream objects, instances of PrintStream and PrintWriter implement a standard set of write methods for simple byte and character output. In addition, both PrintStream and PrintWriter implement the same set of methods for converting internal data into formatted output. Two levels of formatting are provided:

* print and println format individual values in a standard way.
* format formats almost any number of values based on a format string, with many options for precise formatting.

## The print and println Methods

Invoking print or println outputs a single value after converting the value using the appropriate toString method. We can see this in the [Root](https://docs.oracle.com/javase/tutorial/essential/io/examples/Root.java) example:

public class Root {

public static void main(String[] args) {

int i = 2;

double r = Math.sqrt(i);

System.out.print("The square root of ");

System.out.print(i);

System.out.print(" is ");

System.out.print(r);

System.out.println(".");

i = 5;

r = Math.sqrt(i);

System.out.println("The square root of " + i + " is " + r + ".");

}

}

Here is the output of Root:

The square root of 2 is 1.4142135623730951.

The square root of 5 is 2.23606797749979.

The i and r variables are formatted twice: the first time using code in an overload of print, the second time by conversion code automatically generated by the Java compiler, which also utilizes toString. You can format any value this way, but you don't have much control over the results.

## The format Method

The format method formats multiple arguments based on a *format string*. The format string consists of static text embedded with *format specifiers*; except for the format specifiers, the format string is output unchanged.

Format strings support many features. In this tutorial, we'll just cover some basics. For a complete description, see [format string syntax](http://docs.oracle.com/javase/8/docs/api/java/util/Formatter.html#syntax) in the API specification.

The [Root2](https://docs.oracle.com/javase/tutorial/essential/io/examples/Root2.java) example formats two values with a single format invocation:

public class Root2 {

public static void main(String[] args) {

int i = 2;

double r = Math.sqrt(i);

System.out.format("The square root of %d is %f.%n", i, r);

}

}

Here is the output:

The square root of 2 is 1.414214.

Like the three used in this example, all format specifiers begin with a % and end with a 1- or 2-character *conversion* that specifies the kind of formatted output being generated. The three conversions used here are:

* d formats an integer value as a decimal value.
* f formats a floating point value as a decimal value.
* n outputs a platform-specific line terminator.

Here are some other conversions:

* x formats an integer as a hexadecimal value.
* s formats any value as a string.
* tB formats an integer as a locale-specific month name.

There are many other conversions.

**Note:**

Except for %% and %n, all format specifiers must match an argument. If they don't, an exception is thrown.

In the Java programming language, the \n escape always generates the linefeed character (\u000A). Don't use \n unless you specifically want a linefeed character. To get the correct line separator for the local platform, use %n.

In addition to the conversion, a format specifier can contain several additional elements that further customize the formatted output. Here's an example, [Format](https://docs.oracle.com/javase/tutorial/essential/io/examples/Format.java), that uses every possible kind of element.

public class Format {

public static void main(String[] args) {

System.out.format("%f, %1$+020.10f %n", Math.PI);

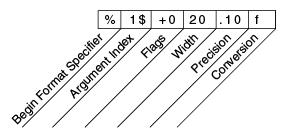
}

}

Here's the output:

3.141593, +00000003.1415926536

The additional elements are all optional. The following figure shows how the longer specifier breaks down into elements.



Elements of a Format Specifier.

The elements must appear in the order shown. Working from the right, the optional elements are:

* **Precision**. For floating point values, this is the mathematical precision of the formatted value. For s and other general conversions, this is the maximum width of the formatted value; the value is right-truncated if necessary.
* **Width**. The minimum width of the formatted value; the value is padded if necessary. By default the value is left-padded with blanks.
* **Flags** specify additional formatting options. In the Format example, the + flag specifies that the number should always be formatted with a sign, and the 0 flag specifies that 0 is the padding character. Other flags include - (pad on the right) and , (format number with locale-specific thousands separators). Note that some flags cannot be used with certain other flags or with certain conversions.
* The **Argument Index** allows you to explicitly match a designated argument. You can also specify < to match the same argument as the previous specifier. Thus the example could have said: System.out.format("%f, %<+020.10f %n", Math.PI);

# I/O from the Command Line

A program is often run from the command line and interacts with the user in the command line environment. The Java platform supports this kind of interaction in two ways: through the Standard Streams and through the Console.

## Standard Streams

Standard Streams are a feature of many operating systems. By default, they read input from the keyboard and write output to the display. They also support I/O on files and between programs, but that feature is controlled by the command line interpreter, not the program.

The Java platform supports three Standard Streams: Standard Input, accessed through System.in; Standard Output, accessed through System.out; and Standard Error, accessed through System.err. These objects are defined automatically and do not need to be opened. Standard Output and Standard Error are both for output; having error output separately allows the user to divert regular output to a file and still be able to read error messages. For more information, refer to the documentation for your command line interpreter.

You might expect the Standard Streams to be character streams, but, for historical reasons, they are byte streams. System.out and System.err are defined as [PrintStream](http://docs.oracle.com/javase/8/docs/api/java/io/PrintStream.html) objects. Although it is technically a byte stream, PrintStream utilizes an internal character stream object to emulate many of the features of character streams.

By contrast, System.in is a byte stream with no character stream features. To use Standard Input as a character stream, wrap System.in in InputStreamReader.

InputStreamReader cin = new InputStreamReader(System.in);

## The Console

A more advanced alternative to the Standard Streams is the Console. This is a single, predefined object of type [Console](http://docs.oracle.com/javase/8/docs/api/java/io/Console.html) that has most of the features provided by the Standard Streams, and others besides. The Console is particularly useful for secure password entry. The Console object also provides input and output streams that are true character streams, through its reader and writer methods.

Before a program can use the Console, it must attempt to retrieve the Console object by invoking System.console(). If the Console object is available, this method returns it. If System.console returns NULL, then Console operations are not permitted, either because the OS doesn't support them or because the program was launched in a noninteractive environment.

The Console object supports secure password entry through its readPassword method. This method helps secure password entry in two ways. First, it suppresses echoing, so the password is not visible on the user's screen. Second, readPassword returns a character array, not a String, so the password can be overwritten, removing it from memory as soon as it is no longer needed.

The [Password](https://docs.oracle.com/javase/tutorial/essential/io/examples/Password.java) example is a prototype program for changing a user's password. It demonstrates several Console methods.

import java.io.Console;

import java.util.Arrays;

import java.io.IOException;

public class Password {

public static void main (String args[]) throws IOException {

Console c = System.console();

if (c == null) {

System.err.println("No console.");

System.exit(1);

}

String login = c.readLine("Enter your login: ");

char [] oldPassword = c.readPassword("Enter your old password: ");

if (verify(login, oldPassword)) {

boolean noMatch;

do {

char [] newPassword1 = c.readPassword("Enter your new password: ");

char [] newPassword2 = c.readPassword("Enter new password again: ");

noMatch = ! Arrays.equals(newPassword1, newPassword2);

if (noMatch) {

c.format("Passwords don't match. Try again.%n");

} else {

change(login, newPassword1);

c.format("Password for %s changed.%n", login);

}

Arrays.fill(newPassword1, ' ');

Arrays.fill(newPassword2, ' ');

} while (noMatch);

}

Arrays.fill(oldPassword, ' ');

}

// Dummy change method.

static boolean verify(String login, char[] password) {

// This method always returns

// true in this example.

// Modify this method to verify

// password according to your rules.

return true;

}

// Dummy change method.

static void change(String login, char[] password) {

// Modify this method to change

// password according to your rules.

}

}

The Password class follows these steps:

1. Attempt to retrieve the Console object. If the object is not available, abort.
2. Invoke Console.readLine to prompt for and read the user's login name.
3. Invoke Console.readPassword to prompt for and read the user's existing password.
4. Invoke verify to confirm that the user is authorized to change the password. (In this example, verify is a dummy method that always returns true.)
5. Repeat the following steps until the user enters the same password twice:
   1. Invoke Console.readPassword twice to prompt for and read a new password.
   2. If the user entered the same password both times, invoke change to change it. (Again, change is a dummy method.)
   3. Overwrite both passwords with blanks.
6. Overwrite the old password with blanks.

# I/O from the Command Line

A program is often run from the command line and interacts with the user in the command line environment. The Java platform supports this kind of interaction in two ways: through the Standard Streams and through the Console.

## Standard Streams

Standard Streams are a feature of many operating systems. By default, they read input from the keyboard and write output to the display. They also support I/O on files and between programs, but that feature is controlled by the command line interpreter, not the program.

The Java platform supports three Standard Streams: Standard Input, accessed through System.in; Standard Output, accessed through System.out; and Standard Error, accessed through System.err. These objects are defined automatically and do not need to be opened. Standard Output and Standard Error are both for output; having error output separately allows the user to divert regular output to a file and still be able to read error messages. For more information, refer to the documentation for your command line interpreter.

You might expect the Standard Streams to be character streams, but, for historical reasons, they are byte streams. System.out and System.err are defined as [PrintStream](http://docs.oracle.com/javase/8/docs/api/java/io/PrintStream.html) objects. Although it is technically a byte stream, PrintStream utilizes an internal character stream object to emulate many of the features of character streams.

By contrast, System.in is a byte stream with no character stream features. To use Standard Input as a character stream, wrap System.in in InputStreamReader.

InputStreamReader cin = new InputStreamReader(System.in);

## The Console

A more advanced alternative to the Standard Streams is the Console. This is a single, predefined object of type [Console](http://docs.oracle.com/javase/8/docs/api/java/io/Console.html) that has most of the features provided by the Standard Streams, and others besides. The Console is particularly useful for secure password entry. The Console object also provides input and output streams that are true character streams, through its reader and writer methods.

Before a program can use the Console, it must attempt to retrieve the Console object by invoking System.console(). If the Console object is available, this method returns it. If System.console returns NULL, then Console operations are not permitted, either because the OS doesn't support them or because the program was launched in a noninteractive environment.

The Console object supports secure password entry through its readPassword method. This method helps secure password entry in two ways. First, it suppresses echoing, so the password is not visible on the user's screen. Second, readPassword returns a character array, not a String, so the password can be overwritten, removing it from memory as soon as it is no longer needed.

The [Password](https://docs.oracle.com/javase/tutorial/essential/io/examples/Password.java) example is a prototype program for changing a user's password. It demonstrates several Console methods.

import java.io.Console;

import java.util.Arrays;

import java.io.IOException;

public class Password {

public static void main (String args[]) throws IOException {

Console c = System.console();

if (c == null) {

System.err.println("No console.");

System.exit(1);

}

String login = c.readLine("Enter your login: ");

char [] oldPassword = c.readPassword("Enter your old password: ");

if (verify(login, oldPassword)) {

boolean noMatch;

do {

char [] newPassword1 = c.readPassword("Enter your new password: ");

char [] newPassword2 = c.readPassword("Enter new password again: ");

noMatch = ! Arrays.equals(newPassword1, newPassword2);

if (noMatch) {

c.format("Passwords don't match. Try again.%n");

} else {

change(login, newPassword1);

c.format("Password for %s changed.%n", login);

}

Arrays.fill(newPassword1, ' ');

Arrays.fill(newPassword2, ' ');

} while (noMatch);

}

Arrays.fill(oldPassword, ' ');

}

// Dummy change method.

static boolean verify(String login, char[] password) {

// This method always returns

// true in this example.

// Modify this method to verify

// password according to your rules.

return true;

}

// Dummy change method.

static void change(String login, char[] password) {

// Modify this method to change

// password according to your rules.

}

}

The Password class follows these steps:

1. Attempt to retrieve the Console object. If the object is not available, abort.
2. Invoke Console.readLine to prompt for and read the user's login name.
3. Invoke Console.readPassword to prompt for and read the user's existing password.
4. Invoke verify to confirm that the user is authorized to change the password. (In this example, verify is a dummy method that always returns true.)
5. Repeat the following steps until the user enters the same password twice:
   1. Invoke Console.readPassword twice to prompt for and read a new password.
   2. If the user entered the same password both times, invoke change to change it. (Again, change is a dummy method.)
   3. Overwrite both passwords with blanks.
6. Overwrite the old password with blanks.

# Data Streams

Data streams support binary I/O of primitive data type values (boolean, char, byte, short, int, long, float, and double) as well as String values. All data streams implement either the [DataInput](http://docs.oracle.com/javase/8/docs/api/java/io/DataInput.html) interface or the [DataOutput](http://docs.oracle.com/javase/8/docs/api/java/io/DataOutput.html) interface. This section focuses on the most widely-used implementations of these interfaces, [DataInputStream](http://docs.oracle.com/javase/8/docs/api/java/io/DataInputStream.html) and [DataOutputStream](http://docs.oracle.com/javase/8/docs/api/java/io/DataOutputStream.html).

The [DataStreams](https://docs.oracle.com/javase/tutorial/essential/io/examples/DataStreams.java) example demonstrates data streams by writing out a set of data records, and then reading them in again. Each record consists of three values related to an item on an invoice, as shown in the following table:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Order in record** | **Data type** | **Data description** | **Output Method** | **Input Method** | **Sample Value** |
| 1 | double | Item price | DataOutputStream.writeDouble | DataInputStream.readDouble | 19.99 |
| 2 | int | Unit count | DataOutputStream.writeInt | DataInputStream.readInt | 12 |
| 3 | String | Item description | DataOutputStream.writeUTF | DataInputStream.readUTF | "Java T-Shirt" |

Let's examine crucial code in DataStreams. First, the program defines some constants containing the name of the data file and the data that will be written to it:

static final String dataFile = "invoicedata";

static final double[] prices = { 19.99, 9.99, 15.99, 3.99, 4.99 };

static final int[] units = { 12, 8, 13, 29, 50 };

static final String[] descs = {

"Java T-shirt",

"Java Mug",

"Duke Juggling Dolls",

"Java Pin",

"Java Key Chain"

};

Then DataStreams opens an output stream. Since a DataOutputStream can only be created as a wrapper for an existing byte stream object, DataStreams provides a buffered file output byte stream.

out = new DataOutputStream(new BufferedOutputStream(

new FileOutputStream(dataFile)));

DataStreams writes out the records and closes the output stream.

for (int i = 0; i < prices.length; i ++) {

out.writeDouble(prices[i]);

out.writeInt(units[i]);

out.writeUTF(descs[i]);

}

The writeUTF method writes out String values in a modified form of UTF-8. This is a variable-width character encoding that only needs a single byte for common Western characters.

Now DataStreams reads the data back in again. First it must provide an input stream, and variables to hold the input data. Like DataOutputStream, DataInputStream must be constructed as a wrapper for a byte stream.

in = new DataInputStream(new

BufferedInputStream(new FileInputStream(dataFile)));

double price;

int unit;

String desc;

double total = 0.0;

Now DataStreams can read each record in the stream, reporting on the data it encounters.

try {

while (true) {

price = in.readDouble();

unit = in.readInt();

desc = in.readUTF();

System.out.format("You ordered %d" + " units of %s at $%.2f%n",

unit, desc, price);

total += unit \* price;

}

} catch (EOFException e) {

}

Notice that DataStreams detects an end-of-file condition by catching [EOFException](http://docs.oracle.com/javase/8/docs/api/java/io/EOFException.html), instead of testing for an invalid return value. All implementations of DataInput methods use EOFException instead of return values.

Also notice that each specialized write in DataStreams is exactly matched by the corresponding specialized read. It is up to the programmer to make sure that output types and input types are matched in this way: The input stream consists of simple binary data, with nothing to indicate the type of individual values, or where they begin in the stream.

DataStreams uses one very bad programming technique: it uses floating point numbers to represent monetary values. In general, floating point is bad for precise values. It's particularly bad for decimal fractions, because common values (such as 0.1) do not have a binary representation.

The correct type to use for currency values is [java.math.BigDecimal](http://docs.oracle.com/javase/8/docs/api/java/math/BigDecimal.html). Unfortunately, BigDecimal is an object type, so it won't work with data streams. However, BigDecimal *will* work with object streams, which are covered in the next section.

# Object Streams

Just as data streams support I/O of primitive data types, object streams support I/O of objects. Most, but not all, standard classes support serialization of their objects. Those that do implement the marker interface [Serializable](http://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html).

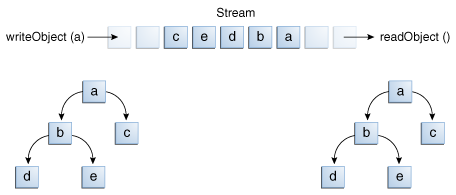
The object stream classes are [ObjectInputStream](http://docs.oracle.com/javase/8/docs/api/java/io/ObjectInputStream.html) and [ObjectOutputStream](http://docs.oracle.com/javase/8/docs/api/java/io/ObjectOutputStream.html). These classes implement [ObjectInput](http://docs.oracle.com/javase/8/docs/api/java/io/ObjectInput.html) and [ObjectOutput](http://docs.oracle.com/javase/8/docs/api/java/io/ObjectOutput.html), which are subinterfaces of DataInput and DataOutput. That means that all the primitive data I/O methods covered in [Data Streams](https://docs.oracle.com/javase/tutorial/essential/io/datastreams.html) are also implemented in object streams. So an object stream can contain a mixture of primitive and object values. The [ObjectStreams](https://docs.oracle.com/javase/tutorial/essential/io/examples/ObjectStreams.java) example illustrates this. ObjectStreams creates the same application as DataStreams, with a couple of changes. First, prices are now [BigDecimal](http://docs.oracle.com/javase/8/docs/api/java/math/BigDecimal.html)objects, to better represent fractional values. Second, a [Calendar](http://docs.oracle.com/javase/8/docs/api/java/util/Calendar.html) object is written to the data file, indicating an invoice date.

If readObject() doesn't return the object type expected, attempting to cast it to the correct type may throw a [ClassNotFoundException](http://docs.oracle.com/javase/8/docs/api/java/lang/ClassNotFoundException.html). In this simple example, that can't happen, so we don't try to catch the exception. Instead, we notify the compiler that we're aware of the issue by adding ClassNotFoundException to the main method's throws clause.

## Output and Input of Complex Objects

The writeObject and readObject methods are simple to use, but they contain some very sophisticated object management logic. This isn't important for a class like Calendar, which just encapsulates primitive values. But many objects contain references to other objects. If readObject is to reconstitute an object from a stream, it has to be able to reconstitute all of the objects the original object referred to. These additional objects might have their own references, and so on. In this situation, writeObject traverses the entire web of object references and writes all objects in that web onto the stream. Thus a single invocation of writeObject can cause a large number of objects to be written to the stream.

This is demonstrated in the following figure, where writeObject is invoked to write a single object named **a**. This object contains references to objects **b** and **c**, while **b** contains references to **d** and **e**. Invoking writeobject(a) writes not just **a**, but all the objects necessary to reconstitute **a**, so the other four objects in this web are written also. When **a** is read back by readObject, the other four objects are read back as well, and all the original object references are preserved.



I/O of multiple referred-to objects

You might wonder what happens if two objects on the same stream both contain references to a single object. Will they both refer to a single object when they're read back? The answer is "yes." A stream can only contain one copy of an object, though it can contain any number of references to it. Thus if you explicitly write an object to a stream twice, you're really writing only the reference twice. For example, if the following code writes an object ob twice to a stream:

Object ob = new Object();

out.writeObject(ob);

out.writeObject(ob);

Each writeObject has to be matched by a readObject, so the code that reads the stream back will look something like this:

Object ob1 = in.readObject();

Object ob2 = in.readObject();

This results in two variables, ob1 and ob2, that are references to a single object.

However, if a single object is written to two different streams, it is effectively duplicated — a single program reading both streams back will see two distinct objects.

# What Is a Path? (And Other File System Facts)

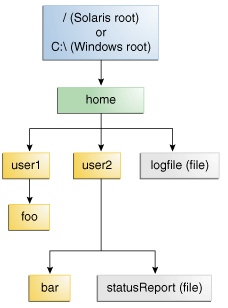
A file system stores and organizes files on some form of media, generally one or more hard drives, in such a way that they can be easily retrieved. Most file systems in use today store the files in a tree (or hierarchical) structure. At the top of the tree is one (or more) root nodes. Under the root node, there are files and directories (folders in Microsoft Windows). Each directory can contain files and subdirectories, which in turn can contain files and subdirectories, and so on, potentially to an almost limitless depth.

This section covers the following:

* [What Is a Path?](https://docs.oracle.com/javase/tutorial/essential/io/path.html#path#path)
* [Relative or Absolute?](https://docs.oracle.com/javase/tutorial/essential/io/path.html#relative#relative)
* [Symbolic Links](https://docs.oracle.com/javase/tutorial/essential/io/path.html#symlink#symlink)

## What Is a Path?

The following figure shows a sample directory tree containing a single root node. Microsoft Windows supports multiple root nodes. Each root node maps to a volume, such as C:\ or D:\. The Solaris OS supports a single root node, which is denoted by the slash character, /.



Sample Directory Structure

A file is identified by its path through the file system, beginning from the root node. For example, the statusReport file in the previous figure is described by the following notation in the Solaris OS:

/home/sally/statusReport

In Microsoft Windows, statusReport is described by the following notation:

C:\home\sally\statusReport

The character used to separate the directory names (also called the delimiter) is specific to the file system: The Solaris OS uses the forward slash (/), and Microsoft Windows uses the backslash slash (\).

## Relative or Absolute?

A path is either relative or absolute. An absolute path always contains the root element and the complete directory list required to locate the file. For example, /home/sally/statusReport is an absolute path. All of the information needed to locate the file is contained in the path string.

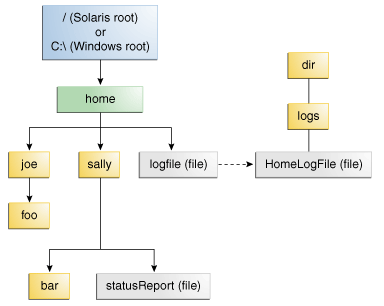
A relative path needs to be combined with another path in order to access a file. For example, joe/foo is a relative path. Without more information, a program cannot reliably locate the joe/foo directory in the file system.

## Symbolic Links

File system objects are most typically directories or files. Everyone is familiar with these objects. But some file systems also support the notion of symbolic links. A symbolic link is also referred to as a symlink or a soft link.

A symbolic link is a special file that serves as a reference to another file. For the most part, symbolic links are transparent to applications, and operations on symbolic links are automatically redirected to the target of the link. (The file or directory being pointed to is called the target of the link.) Exceptions are when a symbolic link is deleted, or renamed in which case the link itself is deleted, or renamed and not the target of the link.

In the following figure, logFile appears to be a regular file to the user, but it is actually a symbolic link to dir/logs/HomeLogFile. HomeLogFile is the target of the link.



Example of a Symbolic Link.

A symbolic link is usually transparent to the user. Reading or writing to a symbolic link is the same as reading or writing to any other file or directory.

The phrase resolving a link means to substitute the actual location in the file system for the symbolic link. In the example, resolving logFile yields dir/logs/HomeLogFile.

In real-world scenarios, most file systems make liberal use of symbolic links. Occasionally, a carelessly created symbolic link can cause a circular reference. A circular reference occurs when the target of a link points back to the original link. The circular reference might be indirect: directory a points to directory b, which points to directory c, which contains a subdirectory pointing back to directory a. Circular references can cause havoc when a program is recursively walking a directory structure. However, this scenario has been accounted for and will not cause your program to loop infinitely.

The next page discusses the heart of file I/O support in the Java programming language, the Path class.

# What Is a Path? (And Other File System Facts)

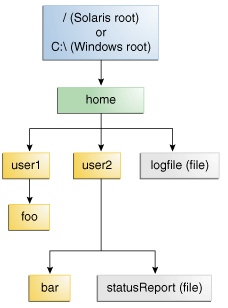
A file system stores and organizes files on some form of media, generally one or more hard drives, in such a way that they can be easily retrieved. Most file systems in use today store the files in a tree (or hierarchical) structure. At the top of the tree is one (or more) root nodes. Under the root node, there are files and directories (folders in Microsoft Windows). Each directory can contain files and subdirectories, which in turn can contain files and subdirectories, and so on, potentially to an almost limitless depth.

This section covers the following:

* [What Is a Path?](https://docs.oracle.com/javase/tutorial/essential/io/path.html#path#path)
* [Relative or Absolute?](https://docs.oracle.com/javase/tutorial/essential/io/path.html#relative#relative)
* [Symbolic Links](https://docs.oracle.com/javase/tutorial/essential/io/path.html#symlink#symlink)

## What Is a Path?

The following figure shows a sample directory tree containing a single root node. Microsoft Windows supports multiple root nodes. Each root node maps to a volume, such as C:\ or D:\. The Solaris OS supports a single root node, which is denoted by the slash character, /.



Sample Directory Structure

A file is identified by its path through the file system, beginning from the root node. For example, the statusReport file in the previous figure is described by the following notation in the Solaris OS:

/home/sally/statusReport

In Microsoft Windows, statusReport is described by the following notation:

C:\home\sally\statusReport

The character used to separate the directory names (also called the delimiter) is specific to the file system: The Solaris OS uses the forward slash (/), and Microsoft Windows uses the backslash slash (\).

## Relative or Absolute?

A path is either relative or absolute. An absolute path always contains the root element and the complete directory list required to locate the file. For example, /home/sally/statusReport is an absolute path. All of the information needed to locate the file is contained in the path string.

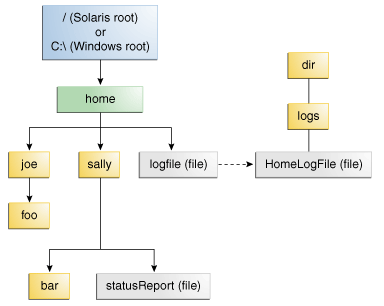
A relative path needs to be combined with another path in order to access a file. For example, joe/foo is a relative path. Without more information, a program cannot reliably locate the joe/foo directory in the file system.

## Symbolic Links

File system objects are most typically directories or files. Everyone is familiar with these objects. But some file systems also support the notion of symbolic links. A symbolic link is also referred to as a symlink or a soft link.

A symbolic link is a special file that serves as a reference to another file. For the most part, symbolic links are transparent to applications, and operations on symbolic links are automatically redirected to the target of the link. (The file or directory being pointed to is called the target of the link.) Exceptions are when a symbolic link is deleted, or renamed in which case the link itself is deleted, or renamed and not the target of the link.

In the following figure, logFile appears to be a regular file to the user, but it is actually a symbolic link to dir/logs/HomeLogFile. HomeLogFile is the target of the link.



Example of a Symbolic Link.

A symbolic link is usually transparent to the user. Reading or writing to a symbolic link is the same as reading or writing to any other file or directory.

The phrase resolving a link means to substitute the actual location in the file system for the symbolic link. In the example, resolving logFile yields dir/logs/HomeLogFile.

In real-world scenarios, most file systems make liberal use of symbolic links. Occasionally, a carelessly created symbolic link can cause a circular reference. A circular reference occurs when the target of a link points back to the original link. The circular reference might be indirect: directory a points to directory b, which points to directory c, which contains a subdirectory pointing back to directory a. Circular references can cause havoc when a program is recursively walking a directory structure. However, this scenario has been accounted for and will not cause your program to loop infinitely.

The next page discusses the heart of file I/O support in the Java programming language, the Path class.

# The Path Class

The [Path](http://docs.oracle.com/javase/8/docs/api/java/nio/file/Path.html) class, introduced in the Java SE 7 release, is one of the primary entrypoints of the [java.nio.file](http://docs.oracle.com/javase/8/docs/api/java/nio/file/package-summary.html) package. If your application uses file I/O, you will want to learn about the powerful features of this class.

**Version Note:** If you have pre-JDK7 code that uses java.io.File, you can still take advantage of the Path class functionality by using the [File.toPath](http://docs.oracle.com/javase/8/docs/api/java/io/File.html#toPath--) method. See [Legacy File I/O Code](https://docs.oracle.com/javase/tutorial/essential/io/legacy.html) for more information.

As its name implies, the Path class is a programmatic representation of a path in the file system. A Path object contains the file name and directory list used to construct the path, and is used to examine, locate, and manipulate files.

A Path instance reflects the underlying platform. In the Solaris OS, a Path uses the Solaris syntax (/home/joe/foo) and in Microsoft Windows, a Path uses the Windows syntax (C:\home\joe\foo). A Path is not system independent. You cannot compare a Path from a Solaris file system and expect it to match a Path from a Windows file system, even if the directory structure is identical and both instances locate the same relative file.

The file or directory corresponding to the Path might not exist. You can create a Path instance and manipulate it in various ways: you can append to it, extract pieces of it, compare it to another path. At the appropriate time, you can use the methods in the [Files](http://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html) class to check the existence of the file corresponding to the Path, create the file, open it, delete it, change its permissions, and so on.

The next page examines the Path class in detail.

# Path Operations

The [Path](http://docs.oracle.com/javase/8/docs/api/java/nio/file/Path.html) class includes various methods that can be used to obtain information about the path, access elements of the path, convert the path to other forms, or extract portions of a path. There are also methods for matching the path string and methods for removing redundancies in a path. This lesson addresses these Path methods, sometimes called syntactic operations, because they operate on the path itself and don't access the file system.

This section covers the following:

* [Creating a Path](https://docs.oracle.com/javase/tutorial/essential/io/pathOps.html#create#create)
* [Retrieving Information About a Path](https://docs.oracle.com/javase/tutorial/essential/io/pathOps.html#info#info)
* [Removing Redundancies from a Path](https://docs.oracle.com/javase/tutorial/essential/io/pathOps.html#normal#normal)
* [Converting a Path](https://docs.oracle.com/javase/tutorial/essential/io/pathOps.html#convert#convert)
* [Joining Two Paths](https://docs.oracle.com/javase/tutorial/essential/io/pathOps.html#resolve#resolve)
* [Creating a Path Between Two Paths](https://docs.oracle.com/javase/tutorial/essential/io/pathOps.html#relativize#relativize)
* [Comparing Two Paths](https://docs.oracle.com/javase/tutorial/essential/io/pathOps.html#compare#compare)

## Creating a Path

A Path instance contains the information used to specify the location of a file or directory. At the time it is defined, a Path is provided with a series of one or more names. A root element or a file name might be included, but neither are required. A Path might consist of just a single directory or file name.

You can easily create a Path object by using one of the following get methods from the [Paths](http://docs.oracle.com/javase/8/docs/api/java/nio/file/Paths.html) (note the plural) helper class:

Path p1 = Paths.get("/tmp/foo");

Path p2 = Paths.get(args[0]);

Path p3 = Paths.get(URI.create("file:///Users/joe/FileTest.java"));

The Paths.get method is shorthand for the following code:

Path p4 = FileSystems.getDefault().getPath("/users/sally");

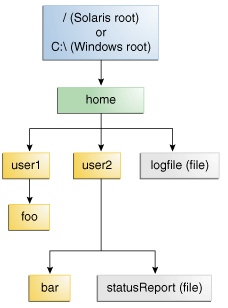
The following example creates /u/joe/logs/foo.log assuming your home directory is /u/joe, or C:\joe\logs\foo.log if you are on Windows.

Path p5 = Paths.get(System.getProperty("user.home"),"logs", "foo.log");

## Retrieving Information about a Path

You can think of the Path as storing these name elements as a sequence. The highest element in the directory structure would be located at index 0. The lowest element in the directory structure would be located at index [n-1], where n is the number of name elements in the Path. Methods are available for retrieving individual elements or a subsequence of the Path using these indexes.

The examples in this lesson use the following directory structure.



Sample Directory Structure

The following code snippet defines a Path instance and then invokes several methods to obtain information about the path:

// None of these methods requires that the file corresponding

// to the Path exists.

// Microsoft Windows syntax

Path path = Paths.get("C:\\home\\joe\\foo");

// Solaris syntax

Path path = Paths.get("/home/joe/foo");

System.out.format("toString: %s%n", path.toString());

System.out.format("getFileName: %s%n", path.getFileName());

System.out.format("getName(0): %s%n", path.getName(0));

System.out.format("getNameCount: %d%n", path.getNameCount());

System.out.format("subpath(0,2): %s%n", path.subpath(0,2));

System.out.format("getParent: %s%n", path.getParent());

System.out.format("getRoot: %s%n", path.getRoot());

Here is the output for both Windows and the Solaris OS:

|  |  |  |  |
| --- | --- | --- | --- |
| **Method Invoked** | **Returns in the Solaris OS** | **Returns in Microsoft Windows** | **Comment** |
| toString | /home/joe/foo | C:\home\joe\foo | Returns the string representation of the Path. If the path was created using Filesystems.getDefault().getPath(String) or Paths.get (the latter is a convenience method for getPath), the method performs minor syntactic cleanup. For example, in a UNIX operating system, it will correct the input string //home/joe/foo to /home/joe/foo. |
| getFileName | foo | foo | Returns the file name or the last element of the sequence of name elements. |
| getName(0) | home | home | Returns the path element corresponding to the specified index. The 0th element is the path element closest to the root. |
| getNameCount | 3 | 3 | Returns the number of elements in the path. |
| subpath(0,2) | home/joe | home\joe | Returns the subsequence of the Path (not including a root element) as specified by the beginning and ending indexes. |
| getParent | /home/joe | \home\joe | Returns the path of the parent directory. |
| getRoot | / | C:\ | Returns the root of the path. |

The previous example shows the output for an absolute path. In the following example, a relative path is specified:

// Solaris syntax

Path path = Paths.get("sally/bar");

or

// Microsoft Windows syntax

Path path = Paths.get("sally\\bar");

Here is the output for Windows and the Solaris OS:

|  |  |  |
| --- | --- | --- |
| **Method Invoked** | **Returns in the Solaris OS** | **Returns in Microsoft Windows** |
| toString | sally/bar | sally\bar |
| getFileName | bar | bar |
| getName(0) | sally | sally |
| getNameCount | 2 | 2 |
| subpath(0,1) | sally | sally |
| getParent | sally | sally |
| getRoot | null | null |

## Removing Redundancies From a Path

Many file systems use "." notation to denote the current directory and ".." to denote the parent directory. You might have a situation where a Path contains redundant directory information. Perhaps a server is configured to save its log files in the "/dir/logs/." directory, and you want to delete the trailing "/." notation from the path.

The following examples both include redundancies:

/home/./joe/foo

/home/sally/../joe/foo

The normalize method removes any redundant elements, which includes any "." or "directory/.." occurrences. Both of the preceding examples normalize to /home/joe/foo.

It is important to note that normalize doesn't check at the file system when it cleans up a path. It is a purely syntactic operation. In the second example, if sally were a symbolic link, removing sally/.. might result in a Path that no longer locates the intended file.

To clean up a path while ensuring that the result locates the correct file, you can use the toRealPath method. This method is described in the next section, [Converting a Path](https://docs.oracle.com/javase/tutorial/essential/io/pathOps.html#convert#convert).

## Converting a Path

You can use three methods to convert the Path. If you need to convert the path to a string that can be opened from a browser, you can use [toUri](http://docs.oracle.com/javase/8/docs/api/java/nio/file/Path.html#toUri--). For example:

Path p1 = Paths.get("/home/logfile");

// Result is **file:///home/logfile**

System.out.format("%s%n", p1.toUri());

The [toAbsolutePath](http://docs.oracle.com/javase/8/docs/api/java/nio/file/Path.html#toAbsolutePath--) method converts a path to an absolute path. If the passed-in path is already absolute, it returns the same Path object. The toAbsolutePath method can be very helpful when processing user-entered file names. For example:

public class FileTest {

public static void main(String[] args) {

if (args.length < 1) {

System.out.println("usage: FileTest file");

System.exit(-1);

}

// Converts the input string to a Path object.

Path inputPath = Paths.get(args[0]);

**// Converts the input Path**

**// to an absolute path.**

**// Generally, this means prepending**

**// the current working**

**// directory. If this example**

**// were called like this:**

**// java FileTest foo**

**// the getRoot and getParent methods**

**// would return null**

**// on the original "inputPath"**

**// instance. Invoking getRoot and**

**// getParent on the "fullPath"**

**// instance returns expected values.**

**Path fullPath = inputPath.toAbsolutePath();**

}

}

The toAbsolutePath method converts the user input and returns a Path that returns useful values when queried. The file does not need to exist for this method to work.

The [toRealPath](http://docs.oracle.com/javase/8/docs/api/java/nio/file/Path.html#toRealPath-java.nio.file.LinkOption...-) method returns the real path of an existing file. This method performs several operations in one:

* If true is passed to this method and the file system supports symbolic links, this method resolves any symbolic links in the path.
* If the Path is relative, it returns an absolute path.
* If the Path contains any redundant elements, it returns a path with those elements removed.

This method throws an exception if the file does not exist or cannot be accessed. You can catch the exception when you want to handle any of these cases. For example:

try {

Path fp = path.toRealPath();

} catch (NoSuchFileException x) {

System.err.format("%s: no such" + " file or directory%n", path);

// Logic for case when file doesn't exist.

} catch (IOException x) {

System.err.format("%s%n", x);

// Logic for other sort of file error.

}

## Joining Two Paths

You can combine paths by using the resolve method. You pass in a partial path , which is a path that does not include a root element, and that partial path is appended to the original path.

For example, consider the following code snippet:

// Solaris

Path p1 = Paths.get("/home/joe/foo");

// Result is **/home/joe/foo/bar**

System.out.format("%s%n", p1.resolve("bar"));

or

// Microsoft Windows

Path p1 = Paths.get("C:\\home\\joe\\foo");

// Result is **C:\home\joe\foo\bar**

System.out.format("%s%n", p1.resolve("bar"));

Passing an absolute path to the resolve method returns the passed-in path:

// Result is **/home/joe**

Paths.get("foo").resolve("/home/joe");

## Creating a Path Between Two Paths

A common requirement when you are writing file I/O code is the capability to construct a path from one location in the file system to another location. You can meet this using the relativize method. This method constructs a path originating from the original path and ending at the location specified by the passed-in path. The new path is relative to the original path.

For example, consider two relative paths defined as joe and sally:

Path p1 = Paths.get("joe");

Path p2 = Paths.get("sally");

In the absence of any other information, it is assumed that joe and sally are siblings, meaning nodes that reside at the same level in the tree structure. To navigate from joe to sally, you would expect to first navigate one level up to the parent node and then down to sally:

// Result is **../sally**

Path p1\_to\_p2 = p1.relativize(p2);

// Result is **../joe**

Path p2\_to\_p1 = p2.relativize(p1);

Consider a slightly more complicated example:

Path p1 = Paths.get("home");

Path p3 = Paths.get("home/sally/bar");

// Result is **sally/bar**

Path p1\_to\_p3 = p1.relativize(p3);

// Result is **../..**

Path p3\_to\_p1 = p3.relativize(p1);

In this example, the two paths share the same node, home. To navigate from home to bar, you first navigate one level down to sally and then one more level down to bar. Navigating from bar to home requires moving up two levels.

A relative path cannot be constructed if only one of the paths includes a root element. If both paths include a root element, the capability to construct a relative path is system dependent.

The recursive [Copy](https://docs.oracle.com/javase/tutorial/essential/io/examples/Copy.java) example uses the relativize and resolve methods.

## Comparing Two Paths

The Path class supports [equals](http://docs.oracle.com/javase/8/docs/api/java/nio/file/Path.html#equals-java.lang.Object-), enabling you to test two paths for equality. The [startsWith](http://docs.oracle.com/javase/8/docs/api/java/nio/file/Path.html#startsWith-java.nio.file.Path-) and [endsWith](http://docs.oracle.com/javase/8/docs/api/java/nio/file/Path.html#endsWith-java.nio.file.Path-) methods enable you to test whether a path begins or ends with a particular string. These methods are easy to use. For example:

Path path = ...;

Path otherPath = ...;

Path beginning = Paths.get("/home");

Path ending = Paths.get("foo");

if (path.equals(otherPath)) {

// equality logic here

} else if (path.startsWith(beginning)) {

// path begins with "/home"

} else if (path.endsWith(ending)) {

// path ends with "foo"

}

The Path class implements the [Iterable](http://docs.oracle.com/javase/8/docs/api/java/lang/Iterable.html) interface. The [iterator](http://docs.oracle.com/javase/8/docs/api/java/nio/file/Path.html#iterator--) method returns an object that enables you to iterate over the name elements in the path. The first element returned is that closest to the root in the directory tree. The following code snippet iterates over a path, printing each name element:

Path path = ...;

for (Path name: path) {

System.out.println(name);

}

The Path class also implements the [Comparable](http://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html) interface. You can compare Path objects by using compareTo which is useful for sorting.

You can also put Path objects into a Collection. See the [Collections](https://docs.oracle.com/javase/tutorial/collections/index.html) trail for more information about this powerful feature.

When you want to verify that two Path objects locate the same file, you can use the isSameFile method, as described in [Checking Whether Two Paths Locate the Same File](https://docs.oracle.com/javase/tutorial/essential/io/check.html#same).

# File Operations

The [Files](http://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html) class is the other primary entrypoint of the java.nio.file package. This class offers a rich set of static methods for reading, writing, and manipulating files and directories. The Files methods work on instances of Path objects. Before proceeding to the remaining sections, you should familiarize yourself with the following common concepts:

* [Releasing System Resources](https://docs.oracle.com/javase/tutorial/essential/io/fileOps.html#resources#resources)
* [Catching Exceptions](https://docs.oracle.com/javase/tutorial/essential/io/fileOps.html#exception#exception)
* [Varargs](https://docs.oracle.com/javase/tutorial/essential/io/fileOps.html#varargs#varargs)
* [Atomic Operations](https://docs.oracle.com/javase/tutorial/essential/io/fileOps.html#atomic#atomic)
* [Method Chaining](https://docs.oracle.com/javase/tutorial/essential/io/fileOps.html#chaining#chaining)
* [What Is a Glob?](https://docs.oracle.com/javase/tutorial/essential/io/fileOps.html#glob#glob)
* [Link Awareness](https://docs.oracle.com/javase/tutorial/essential/io/fileOps.html#linkaware#linkaware)

## Releasing System Resources

Many of the resources that are used in this API, such as streams or channels, implement or extend the [java.io.Closeable](http://docs.oracle.com/javase/8/docs/api/java/io/Closeable.html) interface. A requirement of a Closeable resource is that the close method must be invoked to release the resource when no longer required. Neglecting to close a resource can have a negative implication on an application's performance. The try-with-resources statement, described in the next section, handles this step for you.

## Catching Exceptions

With file I/O, unexpected conditions are a fact of life: a file exists (or doesn't exist) when expected, the program doesn't have access to the file system, the default file system implementation does not support a particular function, and so on. Numerous errors can be encountered.

All methods that access the file system can throw an IOException. It is best practice to catch these exceptions by embedding these methods into a try-with-resources statement, introduced in the Java SE 7 release. The try-with-resources statement has the advantage that the compiler automatically generates the code to close the resource(s) when no longer required. The following code shows how this might look:

Charset charset = Charset.forName("US-ASCII");

String s = ...;

try (BufferedWriter writer = Files.newBufferedWriter(file, charset)) {

writer.write(s, 0, s.length());

} catch (IOException x) {

System.err.format("IOException: %s%n", x);

}

For more information, see [The try-with-resources Statement](https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html).

Alternatively, you can embed the file I/O methods in a try block and then catch any exceptions in a catch block. If your code has opened any streams or channels, you should close them in a finally block. The previous example would look something like the following using the try-catch-finally approach:

Charset charset = Charset.forName("US-ASCII");

String s = ...;

BufferedWriter writer = null;

try {

writer = Files.newBufferedWriter(file, charset);

writer.write(s, 0, s.length());

} catch (IOException x) {

System.err.format("IOException: %s%n", x);

} finally {

if (writer != null) writer.close();

}

For more information, see [Catching and Handling Exceptions](https://docs.oracle.com/javase/tutorial/essential/exceptions/handling.html).

In addition to IOException, many specific exceptions extend [FileSystemException](http://docs.oracle.com/javase/8/docs/api/java/nio/file/FileSystemException.html). This class has some useful methods that return the file involved [(getFile)](http://docs.oracle.com/javase/8/docs/api/java/nio/file/FileSystemException.html#getFile--), the detailed message string [(getMessage)](http://docs.oracle.com/javase/8/docs/api/java/nio/file/FileSystemException.html#getMessage--), the reason why the file system operation failed [(getReason)](http://docs.oracle.com/javase/8/docs/api/java/nio/file/FileSystemException.html#getReason--), and the "other" file involved, if any [(getOtherFile)](http://docs.oracle.com/javase/8/docs/api/java/nio/file/FileSystemException.html#getOtherFile--).

The following code snippet shows how the getFile method might be used:

try (...) {

...

} catch (NoSuchFileException x) {

System.err.format("%s does not exist\n", x.getFile());

}

For purposes of clarity, the file I/O examples in this lesson may not show exception handling, but your code should always include it.

## Varargs

Several Files methods accept an arbitrary number of arguments when flags are specified. For example, in the following method signature, the ellipses notation after the CopyOption argument indicates that the method accepts a variable number of arguments, or varargs, as they are typically called:

Path Files.move(Path, Path, **CopyOption...**)

When a method accepts a varargs argument, you can pass it a comma-separated list of values or an array (CopyOption[]) of values.

In the move example, the method can be invoked as follows:

import static java.nio.file.StandardCopyOption.\*;

Path source = ...;

Path target = ...;

Files.move(source,

target,

REPLACE\_EXISTING,

ATOMIC\_MOVE);

For more information about varargs syntax, see [Arbitrary Number of Arguments](https://docs.oracle.com/javase/tutorial/java/javaOO/arguments.html#varargs).

## Atomic Operations

Several Files methods, such as move, can perform certain operations atomically in some file systems.

An atomic file operation is an operation that cannot be interrupted or "partially" performed. Either the entire operation is performed or the operation fails. This is important when you have multiple processes operating on the same area of the file system, and you need to guarantee that each process accesses a complete file.

## Method Chaining

Many of the file I/O methods support the concept of method chaining.

You first invoke a method that returns an object. You then immediately invoke a method on that object, which returns yet another object, and so on. Many of the I/O examples use the following technique:

String value = Charset.defaultCharset().decode(buf).toString();

UserPrincipal group =

file.getFileSystem().getUserPrincipalLookupService().

lookupPrincipalByName("me");

This technique produces compact code and enables you to avoid declaring temporary variables that you don't need.

## What Is a Glob?

Two methods in the Files class accept a glob argument, but what is a glob?

You can use glob syntax to specify pattern-matching behavior.

A glob pattern is specified as a string and is matched against other strings, such as directory or file names. Glob syntax follows several simple rules:

* An asterisk, \*, matches any number of characters (including none).
* Two asterisks, \*\*, works like \* but crosses directory boundaries. This syntax is generally used for matching complete paths.
* A question mark, ?, matches exactly one character.
* Braces specify a collection of subpatterns. For example:
  + {sun,moon,stars} matches "sun", "moon", or "stars".
  + {temp\*,tmp\*} matches all strings beginning with "temp" or "tmp".
* Square brackets convey a set of single characters or, when the hyphen character (-) is used, a range of characters. For example:
  + [aeiou] matches any lowercase vowel.
  + [0-9] matches any digit.
  + [A-Z] matches any uppercase letter.
  + [a-z,A-Z] matches any uppercase or lowercase letter.

Within the square brackets, \*, ?, and \ match themselves.

* All other characters match themselves.
* To match \*, ?, or the other special characters, you can escape them by using the backslash character, \. For example: \\ matches a single backslash, and \? matches the question mark.

Here are some examples of glob syntax:

* \*.html – Matches all strings that end in .html
* ??? – Matches all strings with exactly three letters or digits
* \*[0-9]\* – Matches all strings containing a numeric value
* \*.{htm,html,pdf} – Matches any string ending with .htm, .html or .pdf
* a?\*.java – Matches any string beginning with a, followed by at least one letter or digit, and ending with .java
* {foo\*,\*[0-9]\*} – Matches any string beginning with foo or any string containing a numeric value

**Note:** If you are typing the glob pattern at the keyboard and it contains one of the special characters, you must put the pattern in quotes ("\*"), use the backslash (\\*), or use whatever escape mechanism is supported at the command line.

The glob syntax is powerful and easy to use. However, if it is not sufficient for your needs, you can also use a regular expression. For more information, see the [Regular Expressions](https://docs.oracle.com/javase/tutorial/essential/regex/index.html) lesson.

For more information about the glob sytnax, see the API specification for the [getPathMatcher](http://docs.oracle.com/javase/8/docs/api/java/nio/file/FileSystem.html#getPathMatcher-java.lang.String-) method in the FileSystem class.

## Link Awareness

The Files class is "link aware." Every Files method either detects what to do when a symbolic link is encountered, or it provides an option enabling you to configure the behavior when a symbolic link is encountered.

# Checking a File or Directory

You have a Path instance representing a file or directory, but does that file exist on the file system? Is it readable? Writable? Executable?

## Verifying the Existence of a File or Directory

The methods in the Path class are syntactic, meaning that they operate on the Path instance. But eventually you must access the file system to verify that a particular Path exists, or does not exist. You can do so with the [exists(Path, LinkOption...)](http://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html#exists-java.nio.file.Path-java.nio.file.LinkOption...-) and the [notExists(Path, LinkOption...)](http://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html#notExists-java.nio.file.Path-java.nio.file.LinkOption...-) methods. Note that !Files.exists(path) is not equivalent to Files.notExists(path). When you are testing a file's existence, three results are possible:

* The file is verified to exist.
* The file is verified to not exist.
* The file's status is unknown. This result can occur when the program does not have access to the file.

If both exists and notExists return false, the existence of the file cannot be verified.

## Checking File Accessibility

To verify that the program can access a file as needed, you can use the [isReadable(Path)](http://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html#isReadable-java.nio.file.Path-), [isWritable(Path)](http://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html#isWritable-java.nio.file.Path-), and [isExecutable(Path)](http://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html#isExecutable-java.nio.file.Path-) methods.

The following code snippet verifies that a particular file exists and that the program has the ability to execute the file.

Path file = ...;

boolean isRegularExecutableFile = Files.isRegularFile(file) &

Files.isReadable(file) & Files.isExecutable(file);

**Note:** Once any of these methods completes, there is no guarantee that the file can be accessed. A common security flaw in many applications is to perform a check and then access the file. For more information, use your favorite search engine to look up TOCTTOU (pronounced TOCK-too).

## Checking Whether Two Paths Locate the Same File

When you have a file system that uses symbolic links, it is possible to have two different paths that locate the same file. The [isSameFile(Path, Path)](http://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html#isSameFile-java.nio.file.Path-java.nio.file.Path-) method compares two paths to determine if they locate the same file on the file system. For example:

Path p1 = ...;

Path p2 = ...;

if (Files.isSameFile(p1, p2)) {

// Logic when the paths locate the same file

}

# Deleting a File or Directory

You can delete files, directories or links. With symbolic links, the link is deleted and not the target of the link. With directories, the directory must be empty, or the deletion fails.

The Files class provides two deletion methods.

The [delete(Path)](http://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html#delete-java.nio.file.Path-) method deletes the file or throws an exception if the deletion fails. For example, if the file does not exist a NoSuchFileException is thrown. You can catch the exception to determine why the delete failed as follows:

try {

Files.delete(path);

} catch (NoSuchFileException x) {

System.err.format("%s: no such" + " file or directory%n", path);

} catch (DirectoryNotEmptyException x) {

System.err.format("%s not empty%n", path);

} catch (IOException x) {

// File permission problems are caught here.

System.err.println(x);

}

The [deleteIfExists(Path)](http://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html#deleteIfExists-java.nio.file.Path-) method also deletes the file, but if the file does not exist, no exception is thrown. Failing silently is useful when you have multiple threads deleting files and you don't want to throw an exception just because one thread did so first.

# Copying a File or Directory

You can copy a file or directory by using the [copy(Path, Path, CopyOption...)](http://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html#copy-java.nio.file.Path-java.nio.file.Path-java.nio.file.CopyOption...-) method. The copy fails if the target file exists, unless the REPLACE\_EXISTING option is specified.

Directories can be copied. However, files inside the directory are not copied, so the new directory is empty even when the original directory contains files.

When copying a symbolic link, the target of the link is copied. If you want to copy the link itself, and not the contents of the link, specify either the NOFOLLOW\_LINKS or REPLACE\_EXISTING option.

This method takes a varargs argument. The following StandardCopyOption and LinkOption enums are supported:

* REPLACE\_EXISTING – Performs the copy even when the target file already exists. If the target is a symbolic link, the link itself is copied (and not the target of the link). If the target is a non-empty directory, the copy fails with the FileAlreadyExistsException exception.
* COPY\_ATTRIBUTES – Copies the file attributes associated with the file to the target file. The exact file attributes supported are file system and platform dependent, but last-modified-time is supported across platforms and is copied to the target file.
* NOFOLLOW\_LINKS – Indicates that symbolic links should not be followed. If the file to be copied is a symbolic link, the link is copied (and not the target of the link).

If you are not familiar with enums, see [Enum Types](https://docs.oracle.com/javase/tutorial/java/javaOO/enum.html).

The following shows how to use the copy method:

import static java.nio.file.StandardCopyOption.\*;

...

Files.copy(source, target, REPLACE\_EXISTING);

In addition to file copy, the Files class also defines methods that may be used to copy between a file and a stream. The [copy(InputStream, Path, CopyOptions...)](http://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html#copy-java.io.InputStream-java.nio.file.Path-java.nio.file.CopyOption...-) method may be used to copy all bytes from an input stream to a file. The [copy(Path, OutputStream)](http://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html#copy-java.nio.file.Path-java.io.OutputStream-) method may be used to copy all bytes from a file to an output stream.

The [Copy](https://docs.oracle.com/javase/tutorial/essential/io/examples/Copy.java) example uses the copy and Files.walkFileTree methods to support a recursive copy. See [Walking the File Tree](https://docs.oracle.com/javase/tutorial/essential/io/walk.html) for more information.

# Moving a File or Directory

You can move a file or directory by using the [move(Path, Path, CopyOption...)](http://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html#move-java.nio.file.Path-java.nio.file.Path-java.nio.file.CopyOption...-) method. The move fails if the target file exists, unless the REPLACE\_EXISTING option is specified.

Empty directories can be moved. If the directory is not empty, the move is allowed when the directory can be moved without moving the contents of that directory. On UNIX systems, moving a directory within the same partition generally consists of renaming the directory. In that situation, this method works even when the directory contains files.

This method takes a varargs argument – the following StandardCopyOption enums are supported:

* REPLACE\_EXISTING – Performs the move even when the target file already exists. If the target is a symbolic link, the symbolic link is replaced but what it points to is not affected.
* ATOMIC\_MOVE – Performs the move as an atomic file operation. If the file system does not support an atomic move, an exception is thrown. With an ATOMIC\_MOVE you can move a file into a directory and be guaranteed that any process watching the directory accesses a complete file.

The following shows how to use the move method:

import static java.nio.file.StandardCopyOption.\*;

...

Files.move(source, target, REPLACE\_EXISTING);

Though you can implement the move method on a single directory as shown, the method is most often used with the file tree recursion mechanism. For more information, see [Walking the File Tree](https://docs.oracle.com/javase/tutorial/essential/io/walk.html).

# Managing Metadata (File and File Store Attributes)

The definition of metadata is "data about other data." With a file system, the data is contained in its files and directories, and the metadata tracks information about each of these objects: Is it a regular file, a directory, or a link? What is its size, creation date, last modified date, file owner, group owner, and access permissions?

A file system's metadata is typically referred to as its file attributes. The Files class includes methods that can be used to obtain a single attribute of a file, or to set an attribute.

|  |  |
| --- | --- |
| **Methods** | **Comment** |
| [size(Path)](http://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html#size-java.nio.file.Path-) | Returns the size of the specified file in bytes. |
| [isDirectory(Path, LinkOption)](http://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html#isDirectory-java.nio.file.Path-java.nio.file.LinkOption...-) | Returns true if the specified Path locates a file that is a directory. |
| [isRegularFile(Path, LinkOption...)](http://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html#isRegularFile-java.nio.file.Path-java.nio.file.LinkOption...-) | Returns true if the specified Path locates a file that is a regular file. |
| [isSymbolicLink(Path)](http://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html#isSymbolicLink-java.nio.file.Path-) | Returns true if the specified Path locates a file that is a symbolic link. |
| [isHidden(Path)](http://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html#isHidden-java.nio.file.Path-) | Returns true if the specified Path locates a file that is considered hidden by the file system. |
| [getLastModifiedTime(Path, LinkOption...)](http://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html#getLastModifiedTime-java.nio.file.Path-java.nio.file.LinkOption...-) [setLastModifiedTime(Path, FileTime)](http://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html#setLastModifiedTime-java.nio.file.Path-java.nio.file.attribute.FileTime-) | Returns or sets the specified file's last modified time. |
| [getOwner(Path, LinkOption...)](http://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html#getOwner-java.nio.file.Path-java.nio.file.LinkOption...-) [setOwner(Path, UserPrincipal)](http://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html#setOwner-java.nio.file.Path-java.nio.file.attribute.UserPrincipal-) | Returns or sets the owner of the file. |
| [getPosixFilePermissions(Path, LinkOption...)](http://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html#getPosixFilePermissions-java.nio.file.Path-java.nio.file.LinkOption...-) [setPosixFilePermissions(Path, Set<PosixFilePermission>)](http://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html#setPosixFilePermissions-java.nio.file.Path-java.util.Set-) | Returns or sets a file's POSIX file permissions. |
| [getAttribute(Path, String, LinkOption...)](http://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html#getAttribute-java.nio.file.Path-java.lang.String-java.nio.file.LinkOption...-) [setAttribute(Path, String, Object, LinkOption...)](http://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html#setAttribute-java.nio.file.Path-java.lang.String-java.lang.Object-java.nio.file.LinkOption...-) | Returns or sets the value of a file attribute. |

If a program needs multiple file attributes around the same time, it can be inefficient to use methods that retrieve a single attribute. Repeatedly accessing the file system to retrieve a single attribute can adversely affect performance. For this reason, the Files class provides two readAttributes methods to fetch a file's attributes in one bulk operation.

|  |  |
| --- | --- |
| **Method** | **Comment** |
| [readAttributes(Path, String, LinkOption...)](http://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html#readAttributes-java.nio.file.Path-java.lang.String-java.nio.file.LinkOption...-) | Reads a file's attributes as a bulk operation. The String parameter identifies the attributes to be read. |
| [readAttributes(Path, Class<A>, LinkOption...)](http://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html#readAttributes-java.nio.file.Path-java.lang.Class-java.nio.file.LinkOption...-) | Reads a file's attributes as a bulk operation. The Class<A> parameter is the type of attributes requested and the method returns an object of that class. |

Before showing examples of the readAttributes methods, it should be mentioned that different file systems have different notions about which attributes should be tracked. For this reason, related file attributes are grouped together into views. A view maps to a particular file system implementation, such as POSIX or DOS, or to a common functionality, such as file ownership.

The supported views are as follows:

* [BasicFileAttributeView](http://docs.oracle.com/javase/8/docs/api/java/nio/file/attribute/BasicFileAttributeView.html) – Provides a view of basic attributes that are required to be supported by all file system implementations.
* [DosFileAttributeView](http://docs.oracle.com/javase/8/docs/api/java/nio/file/attribute/DosFileAttributeView.html) – Extends the basic attribute view with the standard four bits supported on file systems that support the DOS attributes.
* [PosixFileAttributeView](http://docs.oracle.com/javase/8/docs/api/java/nio/file/attribute/PosixFileAttributeView.html) – Extends the basic attribute view with attributes supported on file systems that support the POSIX family of standards, such as UNIX. These attributes include file owner, group owner, and the nine related access permissions.
* [FileOwnerAttributeView](http://docs.oracle.com/javase/8/docs/api/java/nio/file/attribute/FileOwnerAttributeView.html) – Supported by any file system implementation that supports the concept of a file owner.
* [AclFileAttributeView](http://docs.oracle.com/javase/8/docs/api/java/nio/file/attribute/AclFileAttributeView.html) – Supports reading or updating a file's Access Control Lists (ACL). The NFSv4 ACL model is supported. Any ACL model, such as the Windows ACL model, that has a well-defined mapping to the NFSv4 model might also be supported.
* [UserDefinedFileAttributeView](http://docs.oracle.com/javase/8/docs/api/java/nio/file/attribute/UserDefinedFileAttributeView.html) – Enables support of metadata that is user defined. This view can be mapped to any extension mechanisms that a system supports. In the Solaris OS, for example, you can use this view to store the MIME type of a file.

A specific file system implementation might support only the basic file attribute view, or it may support several of these file attribute views. A file system implementation might support other attribute views not included in this API.

In most instances, you should not have to deal directly with any of the FileAttributeView interfaces. (If you do need to work directly with the FileAttributeView, you can access it via the [getFileAttributeView(Path, Class<V>, LinkOption...)](http://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html#getFileAttributeView-java.nio.file.Path-java.lang.Class-java.nio.file.LinkOption...-) method.)

The readAttributes methods use generics and can be used to read the attributes for any of the file attributes views. The examples in the rest of this page use the readAttributes methods.

The remainder of this section covers the following topics:

* [Basic File Attributes](https://docs.oracle.com/javase/tutorial/essential/io/fileAttr.html#basic#basic)
* [Setting Time Stamps](https://docs.oracle.com/javase/tutorial/essential/io/fileAttr.html#time#time)
* [DOS File Attributes](https://docs.oracle.com/javase/tutorial/essential/io/fileAttr.html#dos#dos)
* [POSIX File Permissions](https://docs.oracle.com/javase/tutorial/essential/io/fileAttr.html#posix#posix)
* [Setting a File or Group Owner](https://docs.oracle.com/javase/tutorial/essential/io/fileAttr.html#lookup#lookup)
* [User-Defined File Attributes](https://docs.oracle.com/javase/tutorial/essential/io/fileAttr.html#user#user)
* [File Store Attributes](https://docs.oracle.com/javase/tutorial/essential/io/fileAttr.html#store#store)

## Basic File Attributes

As mentioned previously, to read the basic attributes of a file, you can use one of the Files.readAttributes methods, which reads all the basic attributes in one bulk operation. This is far more efficient than accessing the file system separately to read each individual attribute. The varargs argument currently supports the [LinkOption](http://docs.oracle.com/javase/8/docs/api/java/nio/file/LinkOption.html) enum, NOFOLLOW\_LINKS. Use this option when you do not want symbolic links to be followed.

**A word about time stamps:** The set of basic attributes includes three time stamps: creationTime, lastModifiedTime, and lastAccessTime. Any of these time stamps might not be supported in a particular implementation, in which case the corresponding accessor method returns an implementation-specific value. When supported, the time stamp is returned as an [FileTime](http://docs.oracle.com/javase/8/docs/api/java/nio/file/attribute/FileTime.html) object.

The following code snippet reads and prints the basic file attributes for a given file and uses the methods in the [BasicFileAttributes](http://docs.oracle.com/javase/8/docs/api/java/nio/file/attribute/BasicFileAttributes.html) class.

Path file = ...;

BasicFileAttributes attr = Files.readAttributes(file, BasicFileAttributes.class);

System.out.println("creationTime: " + attr.creationTime());

System.out.println("lastAccessTime: " + attr.lastAccessTime());

System.out.println("lastModifiedTime: " + attr.lastModifiedTime());

System.out.println("isDirectory: " + attr.isDirectory());

System.out.println("isOther: " + attr.isOther());

System.out.println("isRegularFile: " + attr.isRegularFile());

System.out.println("isSymbolicLink: " + attr.isSymbolicLink());

System.out.println("size: " + attr.size());

In addition to the accessor methods shown in this example, there is a fileKey method that returns either an object that uniquely identifies the file or null if no file key is available.

## Setting Time Stamps

The following code snippet sets the last modified time in milliseconds:

Path file = ...;

BasicFileAttributes attr =

Files.readAttributes(file, BasicFileAttributes.class);

long currentTime = System.currentTimeMillis();

FileTime ft = FileTime.fromMillis(currentTime);

Files.setLastModifiedTime(file, ft);

}

## DOS File Attributes

DOS file attributes are also supported on file systems other than DOS, such as Samba. The following snippet uses the methods of the [DosFileAttributes](http://docs.oracle.com/javase/8/docs/api/java/nio/file/attribute/DosFileAttributes.html) class.

Path file = ...;

try {

DosFileAttributes attr =

Files.readAttributes(file, DosFileAttributes.class);

System.out.println("isReadOnly is " + attr.isReadOnly());

System.out.println("isHidden is " + attr.isHidden());

System.out.println("isArchive is " + attr.isArchive());

System.out.println("isSystem is " + attr.isSystem());

} catch (UnsupportedOperationException x) {

System.err.println("DOS file" +

" attributes not supported:" + x);

}

However, you can set a DOS attribute using the [setAttribute(Path, String, Object, LinkOption...)](http://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html#setAttribute-java.nio.file.Path-java.lang.String-java.lang.Object-java.nio.file.LinkOption...-) method, as follows:

Path file = ...;

Files.setAttribute(file, "dos:hidden", true);

## POSIX File Permissions

POSIX is an acronym for Portable Operating System Interface for UNIX and is a set of IEEE and ISO standards designed to ensure interoperability among different flavors of UNIX. If a program conforms to these POSIX standards, it should be easily ported to other POSIX-compliant operating systems.

Besides file owner and group owner, POSIX supports nine file permissions: read, write, and execute permissions for the file owner, members of the same group, and "everyone else."

The following code snippet reads the POSIX file attributes for a given file and prints them to standard output. The code uses the methods in the [PosixFileAttributes](http://docs.oracle.com/javase/8/docs/api/java/nio/file/attribute/PosixFileAttributes.html) class.

Path file = ...;

PosixFileAttributes attr =

Files.readAttributes(file, PosixFileAttributes.class);

System.out.format("%s %s %s%n",

attr.owner().getName(),

attr.group().getName(),

PosixFilePermissions.toString(attr.permissions()));

The [PosixFilePermissions](http://docs.oracle.com/javase/8/docs/api/java/nio/file/attribute/PosixFilePermissions.html) helper class provides several useful methods, as follows:

* The toString method, used in the previous code snippet, converts the file permissions to a string (for example, rw-r--r--).
* The fromString method accepts a string representing the file permissions and constructs a Set of file permissions.
* The asFileAttribute method accepts a Set of file permissions and constructs a file attribute that can be passed to the Path.createFile or Path.createDirectory method.

The following code snippet reads the attributes from one file and creates a new file, assigning the attributes from the original file to the new file:

Path sourceFile = ...;

Path newFile = ...;

PosixFileAttributes attrs =

Files.readAttributes(sourceFile, PosixFileAttributes.class);

FileAttribute<Set<PosixFilePermission>> attr =

PosixFilePermissions.asFileAttribute(attrs.permissions());

Files.createFile(file, attr);

The asFileAttribute method wraps the permissions as a FileAttribute. The code then attempts to create a new file with those permissions. Note that the umask also applies, so the new file might be more secure than the permissions that were requested.

To set a file's permissions to values represented as a hard-coded string, you can use the following code:

Path file = ...;

Set<PosixFilePermission> perms =

PosixFilePermissions.fromString("rw-------");

FileAttribute<Set<PosixFilePermission>> attr =

PosixFilePermissions.asFileAttribute(perms);

Files.setPosixFilePermissions(file, perms);

The [Chmod](https://docs.oracle.com/javase/tutorial/essential/io/examples/Chmod.java) example recursively changes the permissions of files in a manner similar to the chmod utility.

## Setting a File or Group Owner

To translate a name into an object you can store as a file owner or a group owner, you can use the [UserPrincipalLookupService](http://docs.oracle.com/javase/8/docs/api/java/nio/file/attribute/UserPrincipalLookupService.html) service. This service looks up a name or group name as a string and returns a UserPrincipal object representing that string. You can obtain the user principal look-up service for the default file system by using the [FileSystem.getUserPrincipalLookupService](http://docs.oracle.com/javase/8/docs/api/java/nio/file/FileSystem.html#getUserPrincipalLookupService--) method.

The following code snippet shows how to set the file owner by using the setOwner method:

Path file = ...;

UserPrincipal owner = file.GetFileSystem().getUserPrincipalLookupService()

.lookupPrincipalByName("sally");

Files.setOwner(file, owner);

There is no special-purpose method in the Files class for setting a group owner. However, a safe way to do so directly is through the POSIX file attribute view, as follows:

Path file = ...;

GroupPrincipal group =

file.getFileSystem().getUserPrincipalLookupService()

.lookupPrincipalByGroupName("green");

Files.getFileAttributeView(file, PosixFileAttributeView.class)

.setGroup(group);

## User-Defined File Attributes

If the file attributes supported by your file system implementation aren't sufficient for your needs, you can use the UserDefinedAttributeView to create and track your own file attributes.

Some implementations map this concept to features like NTFS Alternative Data Streams and extended attributes on file systems such as ext3 and ZFS. Most implementations impose restrictions on the size of the value, for example, ext3 limits the size to 4 kilobytes.

A file's MIME type can be stored as a user-defined attribute by using this code snippet:

Path file = ...;

UserDefinedFileAttributeView view = Files

.getFileAttributeView(file, UserDefinedFileAttributeView.class);

view.write("user.mimetype",

Charset.defaultCharset().encode("text/html");

To read the MIME type attribute, you would use this code snippet:

Path file = ...;

UserDefinedFileAttributeView view = Files

.getFileAttributeView(file,UserDefinedFileAttributeView.class);

String name = "user.mimetype";

ByteBuffer buf = ByteBuffer.allocate(view.size(name));

view.read(name, buf);

buf.flip();

String value = Charset.defaultCharset().decode(buf).toString();

The [Xdd](https://docs.oracle.com/javase/tutorial/essential/io/examples/Xdd.java) example shows how to get, set, and delete a user-defined attribute.

**Note:** In Linux, you might have to enable extended attributes for user-defined attributes to work. If you receive an UnsupportedOperationException when trying to access the user-defined attribute view, you need to remount the file system. The following command remounts the root partition with extended attributes for the ext3 file system. If this command does not work for your flavor of Linux, consult the documentation.

$ sudo mount -o remount,user\_xattr /

If you want to make the change permanent, add an entry to /etc/fstab.

## File Store Attributes

You can use the [FileStore](http://docs.oracle.com/javase/8/docs/api/java/nio/file/FileStore.html) class to learn information about a file store, such as how much space is available. The [getFileStore(Path)](http://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html#getFileStore-java.nio.file.Path-) method fetches the file store for the specified file.

The following code snippet prints the space usage for the file store where a particular file resides:

Path file = ...;

FileStore store = Files.getFileStore(file);

long total = store.getTotalSpace() / 1024;

long used = (store.getTotalSpace() -

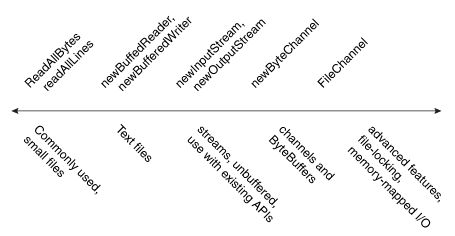
store.getUnallocatedSpace()) / 1024;

long avail = store.getUsableSpace() / 1024;

The [DiskUsage](https://docs.oracle.com/javase/tutorial/essential/io/examples/DiskUsage.java) example uses this API to print disk space information for all the stores in the default file system. This example uses the [getFileStores](http://docs.oracle.com/javase/8/docs/api/java/nio/file/FileSystem.html#getFileStores--) method in the FileSystem class to fetch all the file stores for the the file system.

# Reading, Writing, and Creating Files

This page discusses the details of reading, writing, creating, and opening files. There are a wide array of file I/O methods to choose from. To help make sense of the API, the following diagram arranges the file I/O methods by complexity.



File I/O Methods Arranged from Less Complex to More Complex

On the far left of the diagram are the utility methods readAllBytes, readAllLines, and the write methods, designed for simple, common cases. To the right of those are the methods used to iterate over a stream or lines of text, such as newBufferedReader, newBufferedWriter, then newInputStream and newOutputStream. These methods are interoperable with the java.io package. To the right of those are the methods for dealing with ByteChannels, SeekableByteChannels, and ByteBuffers, such as the newByteChannel method. Finally, on the far right are the methods that use FileChannel for advanced applications needing file locking or memory-mapped I/O.

**Note:** The methods for creating a new file enable you to specify an optional set of initial attributes for the file. For example, on a file system that supports the POSIX set of standards (such as UNIX), you can specify a file owner, group owner, or file permissions at the time the file is created. The [Managing Metadata](https://docs.oracle.com/javase/tutorial/essential/io/fileAttr.html) page explains file attributes, and how to access and set them.

This page has the following topics:

* [The OpenOptions Parameter](https://docs.oracle.com/javase/tutorial/essential/io/file.html#openOptions#openOptions)
* [Commonly Used Methods for Small Files](https://docs.oracle.com/javase/tutorial/essential/io/file.html#common#common)
* [Buffered I/O Methods for Text Files](https://docs.oracle.com/javase/tutorial/essential/io/file.html#textfiles#textfiles)
* [Methods for Unbuffered Streams and Interoperable with java.io APIs](https://docs.oracle.com/javase/tutorial/essential/io/file.html#streams#streams)
* [Methods for Channels and ByteBuffers](https://docs.oracle.com/javase/tutorial/essential/io/file.html#channels#channels)
* [Methods for Creating Regular and Temporary Files](https://docs.oracle.com/javase/tutorial/essential/io/file.html#creating#creating)

## The OpenOptions Parameter

Several of the methods in this section take an optional OpenOptions parameter. This parameter is optional and the API tells you what the default behavior is for the method when none is specified.

The following StandardOpenOptions enums are supported:

* WRITE – Opens the file for write access.
* APPEND – Appends the new data to the end of the file. This option is used with the WRITE or CREATE options.
* TRUNCATE\_EXISTING – Truncates the file to zero bytes. This option is used with the WRITE option.
* CREATE\_NEW – Creates a new file and throws an exception if the file already exists.
* CREATE – Opens the file if it exists or creates a new file if it does not.
* DELETE\_ON\_CLOSE – Deletes the file when the stream is closed. This option is useful for temporary files.
* SPARSE – Hints that a newly created file will be sparse. This advanced option is honored on some file systems, such as NTFS, where large files with data "gaps" can be stored in a more efficient manner where those empty gaps do not consume disk space.
* SYNC – Keeps the file (both content and metadata) synchronized with the underlying storage device.
* DSYNC – Keeps the file content synchronized with the underlying storage device.

## Commonly Used Methods for Small Files

### Reading All Bytes or Lines from a File

If you have a small-ish file and you would like to read its entire contents in one pass, you can use the [readAllBytes(Path)](http://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html#readAllBytes-java.nio.file.Path-) or [readAllLines(Path, Charset)](http://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html#readAllLines-java.nio.file.Path-java.nio.charset.Charset-) method. These methods take care of most of the work for you, such as opening and closing the stream, but are not intended for handling large files. The following code shows how to use the readAllBytes method:

Path file = ...;

byte[] fileArray;

fileArray = Files.readAllBytes(file);

### Writing All Bytes or Lines to a File

You can use one of the write methods to write bytes, or lines, to a file.

* [write(Path, byte[], OpenOption...)](http://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html#write-java.nio.file.Path-byte:A-java.nio.file.OpenOption...-)
* [write(Path, Iterable< extends CharSequence>, Charset, OpenOption...)](http://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html#write-java.nio.file.Path-java.lang.Iterable-java.nio.charset.Charset-java.nio.file.OpenOption...-)

The following code snippet shows how to use a write method.

Path file = ...;

byte[] buf = ...;

Files.write(file, buf);

## Buffered I/O Methods for Text Files

The java.nio.file package supports channel I/O, which moves data in buffers, bypassing some of the layers that can bottleneck stream I/O.

### Reading a File by Using Buffered Stream I/O

The [newBufferedReader(Path, Charset)](http://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html#newBufferedReader-java.nio.file.Path-java.nio.charset.Charset-) method opens a file for reading, returning a BufferedReader that can be used to read text from a file in an efficient manner.

The following code snippet shows how to use the newBufferedReader method to read from a file. The file is encoded in "US-ASCII."

Charset charset = Charset.forName("US-ASCII");

try (BufferedReader reader = Files.newBufferedReader(file, charset)) {

String line = null;

while ((line = reader.readLine()) != null) {

System.out.println(line);

}

} catch (IOException x) {

System.err.format("IOException: %s%n", x);

}

### Writing a File by Using Buffered Stream I/O

You can use the [newBufferedWriter(Path, Charset, OpenOption...)](http://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html#newBufferedWriter-java.nio.file.Path-java.nio.charset.Charset-java.nio.file.OpenOption...-) method to write to a file using a BufferedWriter.

The following code snippet shows how to create a file encoded in "US-ASCII" using this method:

Charset charset = Charset.forName("US-ASCII");

String s = ...;

try (BufferedWriter writer = Files.newBufferedWriter(file, charset)) {

writer.write(s, 0, s.length());

} catch (IOException x) {

System.err.format("IOException: %s%n", x);

}

## Methods for Unbuffered Streams and Interoperable with java.io APIs

### Reading a File by Using Stream I/O

To open a file for reading, you can use the [newInputStream(Path, OpenOption...)](http://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html#newInputStream-java.nio.file.Path-java.nio.file.OpenOption...-) method. This method returns an unbuffered input stream for reading bytes from the file.

Path file = ...;

try (InputStream in = Files.newInputStream(file);

BufferedReader reader =

new BufferedReader(new InputStreamReader(in))) {

String line = null;

while ((line = reader.readLine()) != null) {

System.out.println(line);

}

} catch (IOException x) {

System.err.println(x);

}

### Creating and Writing a File by Using Stream I/O

You can create a file, append to a file, or write to a file by using the [newOutputStream(Path, OpenOption...)](http://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html#newOutputStream-java.nio.file.Path-java.nio.file.OpenOption...-) method. This method opens or creates a file for writing bytes and returns an unbuffered output stream.

The method takes an optional OpenOption parameter. If no open options are specified, and the file does not exist, a new file is created. If the file exists, it is truncated. This option is equivalent to invoking the method with the CREATE and TRUNCATE\_EXISTING options.

The following code snippet opens a log file. If the file does not exist, it is created. If the file exists, it is opened for appending.

import static java.nio.file.StandardOpenOption.\*;

// Convert the string to a

// byte array.

String s = ...;

byte data[] = s.getBytes();

try (OutputStream out = new BufferedOutputStream(

Files.newOutputStream(CREATE, APPEND))) {

...

out.write(data, 0, data.length);

} catch (IOException x) {

System.err.println(x);

}

## Methods for Channels and ByteBuffers

### Reading and Writing Files by Using Channel I/O

While stream I/O reads a character at a time, channel I/O reads a buffer at a time. The [ByteChannel](http://docs.oracle.com/javase/8/docs/api/java/nio/channels/ByteChannel.html) interface provides basic read and write functionality. A [SeekableByteChannel](http://docs.oracle.com/javase/8/docs/api/java/nio/channels/SeekableByteChannel.html) is a ByteChannel that has the capability to maintain a position in the channel and to change that position. A SeekableByteChannel also supports truncating the file associated with the channel and querying the file for its size.

The capability to move to different points in the file and then read from or write to that location makes random access of a file possible. See [Random Access Files](https://docs.oracle.com/javase/tutorial/essential/io/rafs.html) for more information.

There are two methods for reading and writing channel I/O.

* [newByteChannel(Path, OpenOption...)](http://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html#newByteChannel-java.nio.file.Path-java.nio.file.OpenOption...-)
* [newByteChannel(Path, Set<? extends OpenOption>, FileAttribute<?>...)](http://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html#newByteChannel-java.nio.file.Path-java.util.Set-java.nio.file.attribute.FileAttribute...-)

**Note:** The newByteChannel methods return an instance of a SeekableByteChannel. With a default file system, you can cast this seekable byte channel to a [FileChannel](http://docs.oracle.com/javase/8/docs/api/java/nio/channels/FileChannel.html) providing access to more advanced features such mapping a region of the file directly into memory for faster access, locking a region of the file so other processes cannot access it, or reading and writing bytes from an absolute position without affecting the channel's current position.

Both newByteChannel methods enable you to specify a list of OpenOption options. The same [open options](https://docs.oracle.com/javase/tutorial/essential/io/file.html#openOptions#openOptions) used by the newOutputStream methods are supported, in addition to one more option: READ is required because the SeekableByteChannel supports both reading and writing.

Specifying READ opens the channel for reading. Specifying WRITE or APPEND opens the channel for writing. If none of these options is specified, the channel is opened for reading.

The following code snippet reads a file and prints it to standard output:

// Defaults to READ

try (SeekableByteChannel sbc = Files.newByteChannel(file)) {

ByteBuffer buf = ByteBuffer.allocate(10);

// Read the bytes with the proper encoding for this platform. If

// you skip this step, you might see something that looks like

// Chinese characters when you expect Latin-style characters.

String encoding = System.getProperty("file.encoding");

while (sbc.read(buf) > 0) {

buf.rewind();

System.out.print(Charset.forName(encoding).decode(buf));

buf.flip();

}

} catch (IOException x) {

System.out.println("caught exception: " + x);

The following code snippet, written for UNIX and other POSIX file systems, creates a log file with a specific set of file permissions. This code creates a log file or appends to the log file if it already exists. The log file is created with read/write permissions for owner and read only permissions for group.

import static java.nio.file.StandardCopyOption.\*;

// Create the set of options for appending to the file.

Set<OpenOptions> options = new HashSet<OpenOption>();

options.add(APPEND);

options.add(CREATE);

// Create the custom permissions attribute.

Set<PosixFilePermission> perms =

PosixFilePermissions.fromString("rw-r------");

FileAttribute<Set<PosixFilePermission>> attr =

PosixFilePermissions.asFileAttribute(perms);

// Convert the string to a ByteBuffer.

String s = ...;

byte data[] = s.getBytes();

ByteBuffer bb = ByteBuffer.wrap(data);

try (SeekableByteChannel sbc = Files.newByteChannel(file, options, attr)) {

sbc.write(bb);

} catch (IOException x) {

System.out.println("exception thrown: " + x);

}

## Methods for Creating Regular and Temporary Files

### Creating Files

You can create an empty file with an initial set of attributes by using the [createFile(Path, FileAttribute<?>)](http://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html#createFile-java.nio.file.Path-java.nio.file.attribute.FileAttribute...-) method. For example, if, at the time of creation, you want a file to have a particular set of file permissions, use the createFile method to do so. If you do not specify any attributes, the file is created with default attributes. If the file already exists, createFile throws an exception.

In a single atomic operation, the createFile method checks for the existence of the file and creates that file with the specified attributes, which makes the process more secure against malicious code.

The following code snippet creates a file with default attributes:

Path file = ...;

try {

// Create the empty file with default permissions, etc.

Files.createFile(file);

} catch (FileAlreadyExistsException x) {

System.err.format("file named %s" +

" already exists%n", file);

} catch (IOException x) {

// Some other sort of failure, such as permissions.

System.err.format("createFile error: %s%n", x);

}

[POSIX File Permissions](https://docs.oracle.com/javase/tutorial/essential/io/fileAttr.html#posix) has an example that uses createFile(Path, FileAttribute<?>) to create a file with pre-set permissions.

You can also create a new file by using the newOutputStream methods, as described in [Creating and Writing a File using Stream I/O](https://docs.oracle.com/javase/tutorial/essential/io/file.html#createStream). If you open a new output stream and close it immediately, an empty file is created.

### Creating Temporary Files

You can create a temporary file using one of the following createTempFile methods:

* [createTempFile(Path, String, String, FileAttribute<?>)](http://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html#createTempFile-java.nio.file.Path-java.lang.String-java.lang.String-java.nio.file.attribute.FileAttribute...-)
* [createTempFile(String, String, FileAttribute<?>)](http://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html#createTempFile-java.lang.String-java.lang.String-java.nio.file.attribute.FileAttribute...-)

The first method allows the code to specify a directory for the temporary file and the second method creates a new file in the default temporary-file directory. Both methods allow you to specify a suffix for the filename and the first method allows you to also specify a prefix. The following code snippet gives an example of the second method:

try {

Path tempFile = Files.createTempFile(null, ".myapp");

System.out.format("The temporary file" +

" has been created: %s%n", tempFile)

;

} catch (IOException x) {

System.err.format("IOException: %s%n", x);

}

The result of running this file would be something like the following:

The temporary file has been created: /tmp/509668702974537184.myapp

The specific format of the temporary file name is platform specific.

# Random Access Files

Random access files permit nonsequential, or random, access to a file's contents. To access a file randomly, you open the file, seek a particular location, and read from or write to that file.

This functionality is possible with the [SeekableByteChannel](http://docs.oracle.com/javase/8/docs/api/java/nio/channels/SeekableByteChannel.html) interface. The SeekableByteChannel interface extends channel I/O with the notion of a current position. Methods enable you to set or query the position, and you can then read the data from, or write the data to, that location. The API consists of a few, easy to use, methods:

* [position](http://docs.oracle.com/javase/8/docs/api/java/nio/channels/SeekableByteChannel.html#position--) – Returns the channel's current position
* [position(long)](http://docs.oracle.com/javase/8/docs/api/java/nio/channels/SeekableByteChannel.html#position-long-) – Sets the channel's position
* [read(ByteBuffer)](http://docs.oracle.com/javase/8/docs/api/java/nio/channels/SeekableByteChannel.html#read-java.nio.ByteBuffer-) – Reads bytes into the buffer from the channel
* [write(ByteBuffer)](http://docs.oracle.com/javase/8/docs/api/java/nio/channels/SeekableByteChannel.html#write-java.nio.ByteBuffer-) – Writes bytes from the buffer to the channel
* [truncate(long)](http://docs.oracle.com/javase/8/docs/api/java/nio/channels/SeekableByteChannel.html#truncate-long-) – Truncates the file (or other entity) connected to the channel

[Reading and Writing Files With Channel I/O](https://docs.oracle.com/javase/tutorial/essential/io/file.html#channelio) shows that the Path.newByteChannel methods return an instance of a SeekableByteChannel. On the default file system, you can use that channel as is, or you can cast it to a [FileChannel](http://docs.oracle.com/javase/8/docs/api/java/nio/channels/FileChannel.html) giving you access to more advanced features, such as mapping a region of the file directly into memory for faster access, locking a region of the file, or reading and writing bytes from an absolute location without affecting the channel's current position.

The following code snippet opens a file for both reading and writing by using one of the newByteChannel methods. The SeekableByteChannel that is returned is cast to a FileChannel. Then, 12 bytes are read from the beginning of the file, and the string "I was here!" is written at that location. The current position in the file is moved to the end, and the 12 bytes from the beginning are appended. Finally, the string, "I was here!" is appended, and the channel on the file is closed.

String s = "I was here!\n";

byte data[] = s.getBytes();

ByteBuffer out = ByteBuffer.wrap(data);

ByteBuffer copy = ByteBuffer.allocate(12);

try (FileChannel fc = (FileChannel.open(file, READ, WRITE))) {

// Read the first 12

// bytes of the file.

int nread;

do {

nread = fc.read(copy);

} while (nread != -1 && copy.hasRemaining());

// Write "I was here!" at the beginning of the file.

fc.position(0);

while (out.hasRemaining())

fc.write(out);

out.rewind();

// Move to the end of the file. Copy the first 12 bytes to

// the end of the file. Then write "I was here!" again.

long length = fc.size();

fc.position(length-1);

copy.flip();

while (copy.hasRemaining())

fc.write(copy);

while (out.hasRemaining())

fc.write(out);

} catch (IOException x) {

System.out.println("I/O Exception: " + x);

}

# Creating and Reading Directories

Some of the methods previously discussed, such as delete, work on files, links and directories. But how do you list all the directories at the top of a file system? How do you list the contents of a directory or create a directory?

This section covers the following functionality specific to directories:

* [Listing a File System's Root Directories](https://docs.oracle.com/javase/tutorial/essential/io/dirs.html#listall#listall)
* [Creating a Directory](https://docs.oracle.com/javase/tutorial/essential/io/dirs.html#create#create)
* [Creating a Temporary Directory](https://docs.oracle.com/javase/tutorial/essential/io/dirs.html#createTemp#createTemp)
* [Listing a Directory's Contents](https://docs.oracle.com/javase/tutorial/essential/io/dirs.html#listdir#listdir)
* [Filtering a Directory Listing By Using Globbing](https://docs.oracle.com/javase/tutorial/essential/io/dirs.html#glob#glob)
* [Writing Your Own Directory Filter](https://docs.oracle.com/javase/tutorial/essential/io/dirs.html#filter#filter)

## Listing a File System's Root Directories

You can list all the root directories for a file system by using the [FileSystem.getRootDirectories](http://docs.oracle.com/javase/8/docs/api/java/nio/file/FileSystem.html#getRootDirectories--) method. This method returns an Iterable, which enables you to use the [enhanced for](https://docs.oracle.com/javase/tutorial/java/nutsandbolts/for.html) statement to iterate over all the root directories.

The following code snippet prints the root directories for the default file system:

Iterable<Path> dirs = FileSystems.getDefault().getRootDirectories();

for (Path name: dirs) {

System.err.println(name);

}

## Creating a Directory

You can create a new directory by using the [createDirectory(Path, FileAttribute<?>)](http://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html#createDirectory-java.nio.file.Path-java.nio.file.attribute.FileAttribute...-) method. If you don't specify any FileAttributes, the new directory will have default attributes. For example:

Path dir = ...;

Files.createDirectory(path);

The following code snippet creates a new directory on a POSIX file system that has specific permissions:

Set<PosixFilePermission> perms =

PosixFilePermissions.fromString("rwxr-x---");

FileAttribute<Set<PosixFilePermission>> attr =

PosixFilePermissions.asFileAttribute(perms);

Files.createDirectory(file, attr);

To create a directory several levels deep when one or more of the parent directories might not yet exist, you can use the convenience method, [createDirectories(Path, FileAttribute<?>)](http://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html#createDirectories-java.nio.file.Path-java.nio.file.attribute.FileAttribute...-). As with the createDirectory(Path, FileAttribute<?>) method, you can specify an optional set of initial file attributes. The following code snippet uses default attributes:

Files.createDirectories(Paths.get("foo/bar/test"));

The directories are created, as needed, from the top down. In the foo/bar/test example, if the foo directory does not exist, it is created. Next, the bar directory is created, if needed, and, finally, the test directory is created.

It is possible for this method to fail after creating some, but not all, of the parent directories.

## Creating a Temporary Directory

You can create a temporary directory using one of createTempDirectory methods:

* [createTempDirectory(Path, String, FileAttribute<?>...)](http://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html#createTempDirectory-java.nio.file.Path-java.lang.String-java.nio.file.attribute.FileAttribute...-)
* [createTempDirectory(String, FileAttribute<?>...)](http://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html#createTempDirectory-java.lang.String-java.nio.file.attribute.FileAttribute...-)

The first method allows the code to specify a location for the temporary directory and the second method creates a new directory in the default temporary-fle directory.

## Listing a Directory's Contents

You can list all the contents of a directory by using the [newDirectoryStream(Path)](http://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html#newDirectoryStream-java.nio.file.Path-) method. This method returns an object that implements the [DirectoryStream](http://docs.oracle.com/javase/8/docs/api/java/nio/file/DirectoryStream.html) interface. The class that implements the DirectoryStream interface also implements Iterable, so you can iterate through the directory stream, reading all of the objects. This approach scales well to very large directories.

**Remember:** The returned DirectoryStream is a stream. If you are not using a try-with-resources statement, don't forget to close the stream in the finally block. The try-with-resources statement takes care of this for you.

The following code snippet shows how to print the contents of a directory:

Path dir = ...;

try (DirectoryStream<Path> stream = Files.newDirectoryStream(dir)) {

for (Path file: stream) {

System.out.println(file.getFileName());

}

} catch (IOException | DirectoryIteratorException x) {

// IOException can never be thrown by the iteration.

// In this snippet, it can only be thrown by newDirectoryStream.

System.err.println(x);

}

The Path objects returned by the iterator are the names of the entries resolved against the directory. So, if you are listing the contents of the /tmp directory, the entries are returned with the form /tmp/a, /tmp/b, and so on.

This method returns the entire contents of a directory: files, links, subdirectories, and hidden files. If you want to be more selective about the contents that are retrieved, you can use one of the other newDirectoryStream methods, as described later in this page.

Note that if there is an exception during directory iteration then DirectoryIteratorException is thrown with the IOException as the cause. Iterator methods cannot throw exception exceptions.

## Filtering a Directory Listing By Using Globbing

If you want to fetch only files and subdirectories where each name matches a particular pattern, you can do so by using the [newDirectoryStream(Path, String)](http://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html#newDirectoryStream-java.nio.file.Path-java.lang.String-) method, which provides a built-in glob filter. If you are not familiar with glob syntax, see [What Is a Glob?](https://docs.oracle.com/javase/tutorial/essential/io/fileOps.html#glob)

For example, the following code snippet lists files relating to Java: .class, .java, and .jar files.:

Path dir = ...;

try (DirectoryStream<Path> stream =

Files.newDirectoryStream(dir, "\*.{java,class,jar}")) {

for (Path entry: stream) {

System.out.println(entry.getFileName());

}

} catch (IOException x) {

// IOException can never be thrown by the iteration.

// In this snippet, it can // only be thrown by newDirectoryStream.

System.err.println(x);

}

## Writing Your Own Directory Filter

Perhaps you want to filter the contents of a directory based on some condition other than pattern matching. You can create your own filter by implementing the [DirectoryStream.Filter<T>](http://docs.oracle.com/javase/8/docs/api/java/nio/file/DirectoryStream.Filter.html) interface. This interface consists of one method, accept, which determines whether a file fulfills the search requirement.

For example, the following code snippet implements a filter that retrieves only directories:

DirectoryStream.Filter<Path> filter =

newDirectoryStream.Filter<Path>() {

public boolean accept(Path file) throws IOException {

try {

return (Files.isDirectory(path));

} catch (IOException x) {

// Failed to determine if it's a directory.

System.err.println(x);

return false;

}

}

};

Once the filter has been created, it can be invoked by using the [newDirectoryStream(Path, DirectoryStream.Filter<? super Path>)](http://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html#newDirectoryStream-java.nio.file.Path-java.nio.file.DirectoryStream.Filter-) method. The following code snippet uses the isDirectory filter to print only the directory's subdirectories to standard output:

Path dir = ...;

try (DirectoryStream<Path>

stream = Files.newDirectoryStream(dir, filter)) {

for (Path entry: stream) {

System.out.println(entry.getFileName());

}

} catch (IOException x) {

System.err.println(x);

}

This method is used to filter a single directory only. However, if you want to find all the subdirectories in a file tree, you would use the mechanism for [Walking the File Tree](https://docs.oracle.com/javase/tutorial/essential/io/walk.html).

# Links, Symbolic or Otherwise

As mentioned previously, the java.nio.file package, and the Path class in particular, is "link aware." Every Path method either detects what to do when a symbolic link is encountered, or it provides an option enabling you to configure the behavior when a symbolic link is encountered.

The discussion so far has been about [symbolic or soft links](https://docs.oracle.com/javase/tutorial/essential/io/path.html#symlink), but some file systems also support hard links. Hard links are more restrictive than symbolic links, as follows:

* The target of the link must exist.
* Hard links are generally not allowed on directories.
* Hard links are not allowed to cross partitions or volumes. Therefore, they cannot exist across file systems.
* A hard link looks, and behaves, like a regular file, so they can be hard to find.
* A hard link is, for all intents and purposes, the same entity as the original file. They have the same file permissions, time stamps, and so on. All attributes are identical.

Because of these restrictions, hard links are not used as often as symbolic links, but the Path methods work seamlessly with hard links.

Several methods deal specifically with links and are covered in the following sections:

* [Creating a Symbolic Link](https://docs.oracle.com/javase/tutorial/essential/io/links.html#symLink#symLink)
* [Creating a Hard Link](https://docs.oracle.com/javase/tutorial/essential/io/links.html#hardLink#hardLink)
* [Detecting a Symbolic Link](https://docs.oracle.com/javase/tutorial/essential/io/links.html#detect#detect)
* [Finding the Target of a Link](https://docs.oracle.com/javase/tutorial/essential/io/links.html#read#read)

## Creating a Symbolic Link

If your file system supports it, you can create a symbolic link by using the [createSymbolicLink(Path, Path, FileAttribute<?>)](http://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html#createSymbolicLink-java.nio.file.Path-java.nio.file.Path-java.nio.file.attribute.FileAttribute...-) method. The second Path argument represents the target file or directory and might or might not exist. The following code snippet creates a symbolic link with default permissions:

Path newLink = ...;

Path target = ...;

try {

Files.createSymbolicLink(newLink, target);

} catch (IOException x) {

System.err.println(x);

} catch (UnsupportedOperationException x) {

// Some file systems do not support symbolic links.

System.err.println(x);

}

The FileAttributes vararg enables you to specify initial file attributes that are set atomically when the link is created. However, this argument is intended for future use and is not currently implemented.

## Creating a Hard Link

You can create a hard (or regular) link to an existing file by using the [createLink(Path, Path)](http://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html#createLink-java.nio.file.Path-java.nio.file.Path-) method. The second Path argument locates the existing file, and it must exist or a NoSuchFileException is thrown. The following code snippet shows how to create a link:

Path newLink = ...;

Path existingFile = ...;

try {

Files.createLink(newLink, existingFile);

} catch (IOException x) {

System.err.println(x);

} catch (UnsupportedOperationException x) {

// Some file systems do not

// support adding an existing

// file to a directory.

System.err.println(x);

}

## Detecting a Symbolic Link

To determine whether a Path instance is a symbolic link, you can use the [isSymbolicLink(Path)](http://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html#isSymbolicLink-java.nio.file.Path-) method. The following code snippet shows how:

Path file = ...;

boolean isSymbolicLink =

Files.isSymbolicLink(file);

For more information, see [Managing Metadata](https://docs.oracle.com/javase/tutorial/essential/io/fileAttr.html).

## Finding the Target of a Link

You can obtain the target of a symbolic link by using the [readSymbolicLink(Path)](http://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html#readSymbolicLink-java.nio.file.Path-) method, as follows:

Path link = ...;

try {

System.out.format("Target of link" +

" '%s' is '%s'%n", link,

Files.readSymbolicLink(link));

} catch (IOException x) {

System.err.println(x);

}

If the Path is not a symbolic link, this method throws a NotLinkException.

# Walking the File Tree

Do you need to create an application that will recursively visit all the files in a file tree? Perhaps you need to delete every .class file in a tree, or find every file that hasn't been accessed in the last year. You can do so with the [FileVisitor](http://docs.oracle.com/javase/8/docs/api/java/nio/file/FileVisitor.html) interface.

This section covers the following:

* [The FileVisitor Interface](https://docs.oracle.com/javase/tutorial/essential/io/walk.html#filevisitor#filevisitor)
* [Kickstarting the Process](https://docs.oracle.com/javase/tutorial/essential/io/walk.html#invoke#invoke)
* [Considerations When Creating a FileVisitor](https://docs.oracle.com/javase/tutorial/essential/io/walk.html#order#order)
* [Controlling the Flow](https://docs.oracle.com/javase/tutorial/essential/io/walk.html#return#return)
* [Examples](https://docs.oracle.com/javase/tutorial/essential/io/walk.html#ex#ex)

## The FileVisitor Interface

To walk a file tree, you first need to implement a FileVisitor. A FileVisitor specifies the required behavior at key points in the traversal process: when a file is visited, before a directory is accessed, after a directory is accessed, or when a failure occurs. The interface has four methods that correspond to these situations:

* [preVisitDirectory](http://docs.oracle.com/javase/8/docs/api/java/nio/file/FileVisitor.html#preVisitDirectory-T-java.nio.file.attribute.BasicFileAttributes-) – Invoked before a directory's entries are visited.
* [postVisitDirectory](http://docs.oracle.com/javase/8/docs/api/java/nio/file/FileVisitor.html#postVisitDirectory-T-java.io.IOException-) – Invoked after all the entries in a directory are visited. If any errors are encountered, the specific exception is passed to the method.
* [visitFile](http://docs.oracle.com/javase/8/docs/api/java/nio/file/FileVisitor.html#visitFile-T-java.nio.file.attribute.BasicFileAttributes-) – Invoked on the file being visited. The file's BasicFileAttributes is passed to the method, or you can use the [file attributes](https://docs.oracle.com/javase/tutorial/essential/io/fileAttr.html) package to read a specific set of attributes. For example, you can choose to read the file's DosFileAttributeView to determine if the file has the "hidden" bit set.
* [visitFileFailed](http://docs.oracle.com/javase/8/docs/api/java/nio/file/FileVisitor.html#visitFileFailedy-T-java.io.IOException-) – Invoked when the file cannot be accessed. The specific exception is passed to the method. You can choose whether to throw the exception, print it to the console or a log file, and so on.

If you don't need to implement all four of the FileVisitor methods, instead of implementing the FileVisitor interface, you can extend the [SimpleFileVisitor](http://docs.oracle.com/javase/8/docs/api/java/nio/file/SimpleFileVisitor.html) class. This class, which implements the FileVisitor interface, visits all files in a tree and throws an IOError when an error is encountered. You can extend this class and override only the methods that you require.

Here is an example that extends SimpleFileVisitor to print all entries in a file tree. It prints the entry whether the entry is a regular file, a symbolic link, a directory, or some other "unspecified" type of file. It also prints the size, in bytes, of each file. Any exception that is encountered is printed to the console.

The FileVisitor methods are shown in bold:

import static java.nio.file.FileVisitResult.\*;

public static class PrintFiles

extends SimpleFileVisitor<Path> {

// Print information about

// each type of file.

@Override

**public FileVisitResult visitFile(Path file,**

**BasicFileAttributes attr)** {

if (attr.isSymbolicLink()) {

System.out.format("Symbolic link: %s ", file);

} else if (attr.isRegularFile()) {

System.out.format("Regular file: %s ", file);

} else {

System.out.format("Other: %s ", file);

}

System.out.println("(" + attr.size() + "bytes)");

return CONTINUE;

}

// Print each directory visited.

@Override

**public FileVisitResult postVisitDirectory(Path dir,**

**IOException exc)** {

System.out.format("Directory: %s%n", dir);

return CONTINUE;

}

// If there is some error accessing

// the file, let the user know.

// If you don't override this method

// and an error occurs, an IOException

// is thrown.

@Override

**public FileVisitResult visitFileFailed(Path file,**

**IOException exc)** {

System.err.println(exc);

return CONTINUE;

}

}

## Kickstarting the Process

Once you have implemented your FileVisitor, how do you initiate the file walk? There are two walkFileTree methods in the Files class.

* [walkFileTree(Path, FileVisitor)](http://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html#walkFileTree-java.nio.file.Path-java.nio.file.FileVisitor-)
* [walkFileTree(Path, Set<FileVisitOption>, int, FileVisitor)](http://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html#walkFileTree-java.nio.file.Path-java.util.Set-int-java.nio.file.FileVisitor-)

The first method requires only a starting point and an instance of your FileVisitor. You can invoke the PrintFiles file visitor as follows:

Path startingDir = ...;

PrintFiles pf = new PrintFiles();

Files.walkFileTree(startingDir, pf);

The second walkFileTree method enables you to additionally specify a limit on the number of levels visited and a set of [FileVisitOption](http://docs.oracle.com/javase/8/docs/api/java/nio/file/FileVisitOption.html) enums. If you want to ensure that this method walks the entire file tree, you can specify Integer.MAX\_VALUE for the maximum depth argument.

You can specify the FileVisitOption enum, FOLLOW\_LINKS, which indicates that symbolic links should be followed.

This code snippet shows how the four-argument method can be invoked:

import static java.nio.file.FileVisitResult.\*;

Path startingDir = ...;

EnumSet<FileVisitOption> opts = EnumSet.of(FOLLOW\_LINKS);

Finder finder = new Finder(pattern);

Files.walkFileTree(startingDir, opts, Integer.MAX\_VALUE, finder);

## Considerations When Creating a FileVisitor

A file tree is walked depth first, but you cannot make any assumptions about the iteration order that subdirectories are visited.

If your program will be changing the file system, you need to carefully consider how you implement your FileVisitor.

For example, if you are writing a recursive delete, you first delete the files in a directory before deleting the directory itself. In this case, you delete the directory in postVisitDirectory.

If you are writing a recursive copy, you create the new directory in preVisitDirectory before attempting to copy the files to it (in visitFiles). If you want to preserve the attributes of the source directory (similar to the UNIX cp -p command), you need to do that after the files have been copied, in postVisitDirectory. The [Copy](https://docs.oracle.com/javase/tutorial/essential/io/examples/Copy.java) example shows how to do this.

If you are writing a file search, you perform the comparison in the visitFile method. This method finds all the files that match your criteria, but it does not find the directories. If you want to find both files and directories, you must also perform the comparison in either the preVisitDirectory or postVisitDirectory method. The [Find](https://docs.oracle.com/javase/tutorial/essential/io/examples/Find.java) example shows how to do this.

You need to decide whether you want symbolic links to be followed. If you are deleting files, for example, following symbolic links might not be advisable. If you are copying a file tree, you might want to allow it. By default, walkFileTree does not follow symbolic links.

The visitFile method is invoked for files. If you have specified the FOLLOW\_LINKS option and your file tree has a circular link to a parent directory, the looping directory is reported in the visitFileFailed method with the FileSystemLoopException. The following code snippet shows how to catch a circular link and is from the [Copy](https://docs.oracle.com/javase/tutorial/essential/io/examples/Copy.java) example:

@Override

public FileVisitResult

visitFileFailed(Path file,

IOException exc) {

if (exc instanceof FileSystemLoopException) {

System.err.println("cycle detected: " + file);

} else {

System.err.format("Unable to copy:" + " %s: %s%n", file, exc);

}

return CONTINUE;

}

This case can occur only when the program is following symbolic links.

## Controlling the Flow

Perhaps you want to walk the file tree looking for a particular directory and, when found, you want the process to terminate. Perhaps you want to skip specific directories.

The FileVisitor methods return a [FileVisitResult](http://docs.oracle.com/javase/8/docs/api/java/nio/file/FileVisitResult.html) value. You can abort the file walking process or control whether a directory is visited by the values you return in the FileVisitor methods:

* CONTINUE – Indicates that the file walking should continue. If the preVisitDirectory method returns CONTINUE, the directory is visited.
* TERMINATE – Immediately aborts the file walking. No further file walking methods are invoked after this value is returned.
* SKIP\_SUBTREE – When preVisitDirectory returns this value, the specified directory and its subdirectories are skipped. This branch is "pruned out" of the tree.
* SKIP\_SIBLINGS – When preVisitDirectory returns this value, the specified directory is not visited, postVisitDirectory is not invoked, and no further unvisited siblings are visited. If returned from the postVisitDirectory method, no further siblings are visited. Essentially, nothing further happens in the specified directory.

In this code snippet, any directory named SCCS is skipped:

import static java.nio.file.FileVisitResult.\*;

public FileVisitResult

preVisitDirectory(Path dir,

BasicFileAttributes attrs) {

(if (dir.getFileName().toString().equals("SCCS")) {

return SKIP\_SUBTREE;

}

return CONTINUE;

}

In this code snippet, as soon as a particular file is located, the file name is printed to standard output, and the file walking terminates:

import static java.nio.file.FileVisitResult.\*;

// The file we are looking for.

Path lookingFor = ...;

public FileVisitResult

visitFile(Path file,

BasicFileAttributes attr) {

if (file.getFileName().equals(lookingFor)) {

System.out.println("Located file: " + file);

return TERMINATE;

}

return CONTINUE;

}

## Examples

The following examples demonstrate the file walking mechanism:

* [Find](https://docs.oracle.com/javase/tutorial/essential/io/examples/Find.java) – Recurses a file tree looking for files and directories that match a particular glob pattern. This example is discussed in [Finding Files](https://docs.oracle.com/javase/tutorial/essential/io/find.html).
* [Chmod](https://docs.oracle.com/javase/tutorial/essential/io/examples/Chmod.java) – Recursively changes permissions on a file tree (for POSIX systems only).
* [Copy](https://docs.oracle.com/javase/tutorial/essential/io/examples/Copy.java) – Recursively copies a file tree.
* [WatchDir](https://docs.oracle.com/javase/tutorial/essential/io/examples/WatchDir.java) – Demonstrates the mechanism that watches a directory for files that have been created, deleted or modified. Calling this program with the -r option watches an entire tree for changes. For more information about the file notification service, see [Watching a Directory for Changes](https://docs.oracle.com/javase/tutorial/essential/io/notification.html).

# Finding Files

If you have ever used a shell script, you have most likely used pattern matching to locate files. In fact, you have probably used it extensively. If you haven't used it, pattern matching uses special characters to create a pattern and then file names can be compared against that pattern. For example, in most shell scripts, the asterisk, \*, matches any number of characters. For example, the following command lists all the files in the current directory that end in .html:

% ls \*.html

The java.nio.file package provides programmatic support for this useful feature. Each file system implementation provides a [PathMatcher](http://docs.oracle.com/javase/8/docs/api/java/nio/file/PathMatcher.html). You can retrieve a file system's PathMatcher by using the [getPathMatcher(String)](http://docs.oracle.com/javase/8/docs/api/java/nio/file/FileSystem.html#getPathMatcher-java.lang.String-) method in the FileSystem class. The following code snippet fetches the path matcher for the default file system:

String pattern = ...;

PathMatcher matcher =

FileSystems.getDefault().getPathMatcher("glob:" + pattern);

The string argument passed to getPathMatcher specifies the syntax flavor and the pattern to be matched. This example specifies glob syntax. If you are unfamiliar with glob syntax, see [What is a Glob](https://docs.oracle.com/javase/tutorial/essential/io/fileOps.html#glob).

Glob syntax is easy to use and flexible but, if you prefer, you can also use regular expressions, or regex, syntax. For further information about regex, see the [Regular Expressions](https://docs.oracle.com/javase/tutorial/essential/regex/index.html) lesson. Some file system implementations might support other syntaxes.

If you want to use some other form of string-based pattern matching, you can create your own PathMatcher class. The examples in this page use glob syntax.

Once you have created your PathMatcher instance, you are ready to match files against it. The PathMatcher interface has a single method, [matches](http://docs.oracle.com/javase/8/docs/api/java/nio/file/PathMatcher.html#matches-java.nio.file.Path-), that takes a Path argument and returns a boolean: It either matches the pattern, or it does not. The following code snippet looks for files that end in .java or .class and prints those files to standard output:

PathMatcher matcher =

FileSystems.getDefault().getPathMatcher("glob:\*.{java,class}");

Path filename = ...;

if (matcher.matches(filename)) {

System.out.println(filename);

}

## Recursive Pattern Matching

Searching for files that match a particular pattern goes hand-in-hand with walking a file tree. How many times do you know a file is somewhere on the file system, but where? Or perhaps you need to find all files in a file tree that have a particular file extension.

The [Find](https://docs.oracle.com/javase/tutorial/essential/io/examples/Find.java) example does precisely that. Find is similar to the UNIX find utility, but has pared down functionally. You can extend this example to include other functionality. For example, the find utility supports the -prune flag to exclude an entire subtree from the search. You could implement that functionality by returning SKIP\_SUBTREE in the preVisitDirectory method. To implement the -L option, which follows symbolic links, you could use the four-argument walkFileTree method and pass in the FOLLOW\_LINKS enum (but make sure that you test for circular links in the visitFile method).

To run the Find application, use the following format:

% java Find <path> -name "<glob\_pattern>"

The pattern is placed inside quotation marks so any wildcards are not interpreted by the shell. For example:

% java Find . -name "\*.html"

Here is the source code for the Find example:

/\*\*

\* Sample code that finds files that match the specified glob pattern.

\* For more information on what constitutes a glob pattern, see

\* http://docs.oracle.com/javase/tutorial/essential/io/fileOps.html#glob

\*

\* The file or directories that match the pattern are printed to

\* standard out. The number of matches is also printed.

\*

\* When executing this application, you must put the glob pattern

\* in quotes, so the shell will not expand any wild cards:

\* java Find . -name "\*.java"

\*/

import java.io.\*;

import java.nio.file.\*;

import java.nio.file.attribute.\*;

import static java.nio.file.FileVisitResult.\*;

import static java.nio.file.FileVisitOption.\*;

import java.util.\*;

public class Find {

public static class Finder

extends SimpleFileVisitor<Path> {

private final PathMatcher matcher;

private int numMatches = 0;

Finder(String pattern) {

matcher = FileSystems.getDefault()

.getPathMatcher("glob:" + pattern);

}

// Compares the glob pattern against

// the file or directory name.

void find(Path file) {

Path name = file.getFileName();

if (name != null && matcher.matches(name)) {

numMatches++;

System.out.println(file);

}

}

// Prints the total number of

// matches to standard out.

void done() {

System.out.println("Matched: "

+ numMatches);

}

// Invoke the pattern matching

// method on each file.

@Override

public FileVisitResult visitFile(Path file,

BasicFileAttributes attrs) {

find(file);

return CONTINUE;

}

// Invoke the pattern matching

// method on each directory.

@Override

public FileVisitResult preVisitDirectory(Path dir,

BasicFileAttributes attrs) {

find(dir);

return CONTINUE;

}

@Override

public FileVisitResult visitFileFailed(Path file,

IOException exc) {

System.err.println(exc);

return CONTINUE;

}

}

static void usage() {

System.err.println("java Find <path>" +

" -name \"<glob\_pattern>\"");

System.exit(-1);

}

public static void main(String[] args)

throws IOException {

if (args.length < 3 || !args[1].equals("-name"))

usage();

Path startingDir = Paths.get(args[0]);

String pattern = args[2];

Finder finder = new Finder(pattern);

Files.walkFileTree(startingDir, finder);

finder.done();

}

}

Recursively walking a file tree is covered in [Walking the File Tree](https://docs.oracle.com/javase/tutorial/essential/io/walk.html).

# Lesson: Introduction to Collections

A *collection* — sometimes called a container — is simply an object that groups multiple elements into a single unit. Collections are used to store, retrieve, manipulate, and communicate aggregate data. Typically, they represent data items that form a natural group, such as a poker hand (a collection of cards), a mail folder (a collection of letters), or a telephone directory (a mapping of names to phone numbers). If you have used the Java programming language — or just about any other programming language — you are already familiar with collections.

## What Is a Collections Framework?

A *collections framework* is a unified architecture for representing and manipulating collections. All collections frameworks contain the following:

* **Interfaces:** These are abstract data types that represent collections. Interfaces allow collections to be manipulated independently of the details of their representation. In object-oriented languages, interfaces generally form a hierarchy.
* **Implementations:** These are the concrete implementations of the collection interfaces. In essence, they are reusable data structures.
* **Algorithms:** These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces. The algorithms are said to be *polymorphic*: that is, the same method can be used on many different implementations of the appropriate collection interface. In essence, algorithms are reusable functionality.

Apart from the Java Collections Framework, the best-known examples of collections frameworks are the C++ Standard Template Library (STL) and Smalltalk's collection hierarchy. Historically, collections frameworks have been quite complex, which gave them a reputation for having a steep learning curve. We believe that the Java Collections Framework breaks with this tradition, as you will learn for yourself in this chapter.

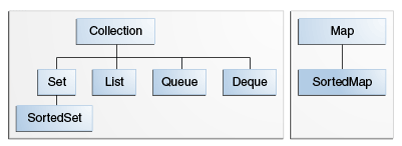
## Benefits of the Java Collections Framework

The Java Collections Framework provides the following benefits:

* **Reduces programming effort:** By providing useful data structures and algorithms, the Collections Framework frees you to concentrate on the important parts of your program rather than on the low-level "plumbing" required to make it work. By facilitating interoperability among unrelated APIs, the Java Collections Framework frees you from writing adapter objects or conversion code to connect APIs.
* **Increases program speed and quality:** This Collections Framework provides high-performance, high-quality implementations of useful data structures and algorithms. The various implementations of each interface are interchangeable, so programs can be easily tuned by switching collection implementations. Because you're freed from the drudgery of writing your own data structures, you'll have more time to devote to improving programs' quality and performance.
* **Allows interoperability among unrelated APIs:** The collection interfaces are the vernacular by which APIs pass collections back and forth. If my network administration API furnishes a collection of node names and if your GUI toolkit expects a collection of column headings, our APIs will interoperate seamlessly, even though they were written independently.
* **Reduces effort to learn and to use new APIs:** Many APIs naturally take collections on input and furnish them as output. In the past, each such API had a small sub-API devoted to manipulating its collections. There was little consistency among these ad hoc collections sub-APIs, so you had to learn each one from scratch, and it was easy to make mistakes when using them. With the advent of standard collection interfaces, the problem went away.
* **Reduces effort to design new APIs:** This is the flip side of the previous advantage. Designers and implementers don't have to reinvent the wheel each time they create an API that relies on collections; instead, they can use standard collection interfaces.
* **Fosters software reuse:** New data structures that conform to the standard collection interfaces are by nature reusable. The same goes for new algorithms that operate on objects that implement these interfaces.

# Lesson: Interfaces

The *core collection interfaces* encapsulate different types of collections, which are shown in the figure below. These interfaces allow collections to be manipulated independently of the details of their representation. Core collection interfaces are the foundation of the Java Collections Framework. As you can see in the following figure, the core collection interfaces form a hierarchy.



The core collection interfaces.

A Set is a special kind of Collection, a SortedSet is a special kind of Set, and so forth. Note also that the hierarchy consists of two distinct trees — a Map is not a true Collection.

Note that all the core collection interfaces are generic. For example, this is the declaration of the Collection interface.

public interface Collection<E>...

The <E> syntax tells you that the interface is generic. When you declare a Collection instance you can *and should* specify the type of object contained in the collection. Specifying the type allows the compiler to verify (at compile-time) that the type of object you put into the collection is correct, thus reducing errors at runtime. For information on generic types, see the [Generics (Updated)](https://docs.oracle.com/javase/tutorial/java/generics/index.html) lesson.

When you understand how to use these interfaces, you will know most of what there is to know about the Java Collections Framework. This chapter discusses general guidelines for effective use of the interfaces, including when to use which interface. You'll also learn programming idioms for each interface to help you get the most out of it.

To keep the number of core collection interfaces manageable, the Java platform doesn't provide separate interfaces for each variant of each collection type. (Such variants might include immutable, fixed-size, and append-only.) Instead, the modification operations in each interface are designated *optional* — a given implementation may elect not to support all operations. If an unsupported operation is invoked, a collection throws an [UnsupportedOperationException](http://docs.oracle.com/javase/8/docs/api/java/lang/UnsupportedOperationException.html). Implementations are responsible for documenting which of the optional operations they support. All of the Java platform's general-purpose implementations support all of the optional operations.

The following list describes the core collection interfaces:

* Collection — the root of the collection hierarchy. A collection represents a group of objects known as its *elements*. The Collection interface is the least common denominator that all collections implement and is used to pass collections around and to manipulate them when maximum generality is desired. Some types of collections allow duplicate elements, and others do not. Some are ordered and others are unordered. The Java platform doesn't provide any direct implementations of this interface but provides implementations of more specific subinterfaces, such as Set and List. Also see [The Collection Interface](https://docs.oracle.com/javase/tutorial/collections/interfaces/collection.html) section.
* Set — a collection that cannot contain duplicate elements. This interface models the mathematical set abstraction and is used to represent sets, such as the cards comprising a poker hand, the courses making up a student's schedule, or the processes running on a machine. See also [The Set Interface](https://docs.oracle.com/javase/tutorial/collections/interfaces/set.html) section.
* List — an ordered collection (sometimes called a *sequence*). Lists can contain duplicate elements. The user of a List generally has precise control over where in the list each element is inserted and can access elements by their integer index (position). If you've used Vector, you're familiar with the general flavor of List. Also see [The List Interface](https://docs.oracle.com/javase/tutorial/collections/interfaces/list.html) section.
* Queue — a collection used to hold multiple elements prior to processing. Besides basic Collection operations, a Queue provides additional insertion, extraction, and inspection operations.

Queues typically, but do not necessarily, order elements in a FIFO (first-in, first-out) manner. Among the exceptions are priority queues, which order elements according to a supplied comparator or the elements' natural ordering. Whatever the ordering used, the head of the queue is the element that would be removed by a call to remove or poll. In a FIFO queue, all new elements are inserted at the tail of the queue. Other kinds of queues may use different placement rules. Every Queue implementation must specify its ordering properties. Also see [The Queue Interface](https://docs.oracle.com/javase/tutorial/collections/interfaces/queue.html) section.

* Deque — a collection used to hold multiple elements prior to processing. Besides basic Collection operations, a Deque provides additional insertion, extraction, and inspection operations.

Deques can be used both as FIFO (first-in, first-out) and LIFO (last-in, first-out). In a deque all new elements can be inserted, retrieved and removed at both ends. Also see [The Deque Interface](https://docs.oracle.com/javase/tutorial/collections/interfaces/deque.html) section.

* Map — an object that maps keys to values. A Map cannot contain duplicate keys; each key can map to at most one value. If you've used Hashtable, you're already familiar with the basics of Map. Also see [The Map Interface](https://docs.oracle.com/javase/tutorial/collections/interfaces/map.html) section.

The last two core collection interfaces are merely sorted versions of Set and Map:

* SortedSet — a Set that maintains its elements in ascending order. Several additional operations are provided to take advantage of the ordering. Sorted sets are used for naturally ordered sets, such as word lists and membership rolls. Also see [The SortedSet Interface](https://docs.oracle.com/javase/tutorial/collections/interfaces/sorted-set.html) section.
* SortedMap — a Map that maintains its mappings in ascending key order. This is the Map analog of SortedSet. Sorted maps are used for naturally ordered collections of key/value pairs, such as dictionaries and telephone directories. Also see [The SortedMap Interface](https://docs.oracle.com/javase/tutorial/collections/interfaces/sorted-map.html) section.

To understand how the sorted interfaces maintain the order of their elements, see the [Object Ordering](https://docs.oracle.com/javase/tutorial/collections/interfaces/order.html) section.

# The Collection Interface

A [Collection](http://docs.oracle.com/javase/8/docs/api/java/util/Collection.html) represents a group of objects known as its elements. The Collection interface is used to pass around collections of objects where maximum generality is desired. For example, by convention all general-purpose collection implementations have a constructor that takes a Collection argument. This constructor, known as a *conversion constructor*, initializes the new collection to contain all of the elements in the specified collection, whatever the given collection's subinterface or implementation type. In other words, it allows you to *convert* the collection's type.

Suppose, for example, that you have a Collection<String> c, which may be a List, a Set, or another kind of Collection. This idiom creates a new ArrayList (an implementation of the List interface), initially containing all the elements in c.

List<String> list = new ArrayList<String>(c);

Or — if you are using JDK 7 or later — you can use the diamond operator:

List<String> list = new ArrayList<>(c);

The Collection interface contains methods that perform basic operations, such as int size(), boolean isEmpty(), boolean contains(Object element), boolean add(E element), boolean remove(Object element), and Iterator<E> iterator().

It also contains methods that operate on entire collections, such as boolean containsAll(Collection<?> c), boolean addAll(Collection<? extends E> c), boolean removeAll(Collection<?> c), boolean retainAll(Collection<?> c), and void clear().

Additional methods for array operations (such as Object[] toArray() and <T> T[] toArray(T[] a) exist as well.

In JDK 8 and later, the Collection interface also exposes methods Stream<E> stream() and Stream<E> parallelStream(), for obtaining sequential or parallel streams from the underlying collection. (See the lesson entitled [Aggregate Operations](https://docs.oracle.com/javase/tutorial/collections/streams/index.html) for more information about using streams.)

The Collection interface does about what you'd expect given that a Collection represents a group of objects. It has methods that tell you how many elements are in the collection (size, isEmpty), methods that check whether a given object is in the collection (contains), methods that add and remove an element from the collection (add, remove), and methods that provide an iterator over the collection (iterator).

The add method is defined generally enough so that it makes sense for collections that allow duplicates as well as those that don't. It guarantees that the Collection will contain the specified element after the call completes, and returns true if the Collection changes as a result of the call. Similarly, the remove method is designed to remove a single instance of the specified element from the Collection, assuming that it contains the element to start with, and to return true if the Collection was modified as a result.

## Traversing Collections

There are three ways to traverse collections: (1) using aggregate operations (2) with the for-each construct and (3) by using Iterators.

### Aggregate Operations

In JDK 8 and later, the preferred method of iterating over a collection is to obtain a stream and perform aggregate operations on it. Aggregate operations are often used in conjunction with lambda expressions to make programming more expressive, using less lines of code. The following code sequentially iterates through a collection of shapes and prints out the red objects:

myShapesCollection.stream()

.filter(e -> e.getColor() == Color.RED)

.forEach(e -> System.out.println(e.getName()));

Likewise, you could easily request a parallel stream, which might make sense if the collection is large enough and your computer has enough cores:

myShapesCollection.parallelStream()

.filter(e -> e.getColor() == Color.RED)

.forEach(e -> System.out.println(e.getName()));

There are many different ways to collect data with this API. For example, you might want to convert the elements of a Collection to String objects, then join them, separated by commas:

String joined = elements.stream()

.map(Object::toString)

.collect(Collectors.joining(", "));

Or perhaps sum the salaries of all employees:

int total = employees.stream()

.collect(Collectors.summingInt(Employee::getSalary)));

These are but a few examples of what you can do with streams and aggregate operations. For more information and examples, see the lesson entitled [Aggregate Operations](https://docs.oracle.com/javase/tutorial/collections/streams/index.html).

The Collections framework has always provided a number of so-called "bulk operations" as part of its API. These include methods that operate on entire collections, such as containsAll, addAll, removeAll, etc. Do not confuse those methods with the aggregate operations that were introduced in JDK 8. The key difference between the new aggregate operations and the existing bulk operations (containsAll, addAll, etc.) is that the old versions are all mutative, meaning that they all modify the underlying collection. In contrast, the new aggregate operations do not modify the underlying collection. When using the new aggregate operations and lambda expressions, you must take care to avoid mutation so as not to introduce problems in the future, should your code be run later from a parallel stream.

### for-each Construct

The for-each construct allows you to concisely traverse a collection or array using a for loop — see [The for Statement](https://docs.oracle.com/javase/tutorial/java/nutsandbolts/for.html). The following code uses the for-each construct to print out each element of a collection on a separate line.

for (Object o : collection)

System.out.println(o);

### Iterators

An [Iterator](http://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html) is an object that enables you to traverse through a collection and to remove elements from the collection selectively, if desired. You get an Iterator for a collection by calling its iterator method. The following is the Iterator interface.

public interface Iterator<E> {

boolean hasNext();

E next();

void remove(); //optional

}

The hasNext method returns true if the iteration has more elements, and the next method returns the next element in the iteration. The remove method removes the last element that was returned by next from the underlying Collection. The remove method may be called only once per call to next and throws an exception if this rule is violated.

Note that Iterator.remove is the *only* safe way to modify a collection during iteration; the behavior is unspecified if the underlying collection is modified in any other way while the iteration is in progress.

Use Iterator instead of the for-each construct when you need to:

* Remove the current element. The for-each construct hides the iterator, so you cannot call remove. Therefore, the for-each construct is not usable for filtering.
* Iterate over multiple collections in parallel.

The following method shows you how to use an Iterator to filter an arbitrary Collection — that is, traverse the collection removing specific elements.

static void filter(Collection<?> c) {

for (Iterator<?> it = c.iterator(); it.hasNext(); )

if (!cond(it.next()))

it.remove();

}

This simple piece of code is polymorphic, which means that it works for *any* Collection regardless of implementation. This example demonstrates how easy it is to write a polymorphic algorithm using the Java Collections Framework.

## Collection Interface Bulk Operations

*Bulk operations* perform an operation on an entire Collection. You could implement these shorthand operations using the basic operations, though in most cases such implementations would be less efficient. The following are the bulk operations:

* containsAll — returns true if the target Collection contains all of the elements in the specified Collection.
* addAll — adds all of the elements in the specified Collection to the target Collection.
* removeAll — removes from the target Collection all of its elements that are also contained in the specified Collection.
* retainAll — removes from the target Collection all its elements that are *not* also contained in the specified Collection. That is, it retains only those elements in the target Collection that are also contained in the specified Collection.
* clear — removes all elements from the Collection.

The addAll, removeAll, and retainAll methods all return true if the target Collection was modified in the process of executing the operation.

As a simple example of the power of bulk operations, consider the following idiom to remove *all* instances of a specified element, e, from a Collection, c.

c.removeAll(Collections.singleton(e));

More specifically, suppose you want to remove all of the null elements from a Collection.

c.removeAll(Collections.singleton(null));

This idiom uses Collections.singleton, which is a static factory method that returns an immutable Set containing only the specified element.

## Collection Interface Array Operations

The toArray methods are provided as a bridge between collections and older APIs that expect arrays on input. The array operations allow the contents of a Collection to be translated into an array. The simple form with no arguments creates a new array of Object. The more complex form allows the caller to provide an array or to choose the runtime type of the output array.

For example, suppose that c is a Collection. The following snippet dumps the contents of c into a newly allocated array of Object whose length is identical to the number of elements in c.

Object[] a = c.toArray();

Suppose that c is known to contain only strings (perhaps because c is of type Collection<String>). The following snippet dumps the contents of c into a newly allocated array of String whose length is identical to the number of elements in c.

String[] a = c.toArray(new String[0]);

# The Set Interface

A [Set](http://docs.oracle.com/javase/8/docs/api/java/util/Set.html) is a [Collection](http://docs.oracle.com/javase/8/docs/api/java/util/Collection.html) that cannot contain duplicate elements. It models the mathematical set abstraction. The Set interface contains *only* methods inherited from Collection and adds the restriction that duplicate elements are prohibited. Set also adds a stronger contract on the behavior of the equals and hashCode operations, allowing Set instances to be compared meaningfully even if their implementation types differ. Two Set instances are equal if they contain the same elements.

The Java platform contains three general-purpose Set implementations: HashSet, TreeSet, and LinkedHashSet. [HashSet](http://docs.oracle.com/javase/8/docs/api/java/util/HashSet.html), which stores its elements in a hash table, is the best-performing implementation; however it makes no guarantees concerning the order of iteration. [TreeSet](http://docs.oracle.com/javase/8/docs/api/java/util/TreeSet.html), which stores its elements in a red-black tree, orders its elements based on their values; it is substantially slower than HashSet. [LinkedHashSet](http://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashSet.html), which is implemented as a hash table with a linked list running through it, orders its elements based on the order in which they were inserted into the set (insertion-order). LinkedHashSet spares its clients from the unspecified, generally chaotic ordering provided by HashSet at a cost that is only slightly higher.

Here's a simple but useful Set idiom. Suppose you have a Collection, c, and you want to create another Collection containing the same elements but with all duplicates eliminated. The following one-liner does the trick.

Collection<Type> noDups = new HashSet<Type>(c);

It works by creating a Set (which, by definition, cannot contain duplicates), initially containing all the elements in c. It uses the standard conversion constructor described in the [The Collection Interface](https://docs.oracle.com/javase/tutorial/collections/interfaces/collection.html) section.

Or, if using JDK 8 or later, you could easily collect into a Set using aggregate operations:

c.stream()

.collect(Collectors.toSet()); // no duplicates

Here's a slightly longer example that accumulates a Collection of names into a TreeSet:

Set<String> set = people.stream()

.map(Person::getName)

.collect(Collectors.toCollection(TreeSet::new));

And the following is a minor variant of the first idiom that preserves the order of the original collection while removing duplicate elements:

Collection<Type> noDups = new LinkedHashSet<Type>(c);

The following is a generic method that encapsulates the preceding idiom, returning a Set of the same generic type as the one passed.

public static <E> Set<E> removeDups(Collection<E> c) {

return new LinkedHashSet<E>(c);

}

## Set Interface Basic Operations

The size operation returns the number of elements in the Set (its *cardinality*). The isEmpty method does exactly what you think it would. The add method adds the specified element to the Set if it is not already present and returns a boolean indicating whether the element was added. Similarly, the remove method removes the specified element from the Set if it is present and returns a boolean indicating whether the element was present. The iterator method returns an Iterator over the Set.

The following [program](https://docs.oracle.com/javase/tutorial/collections/interfaces/examples/FindDups.java) prints out all distinct words in its argument list. Two versions of this program are provided. The first uses JDK 8 aggregate operations. The second uses the for-each construct.

Using JDK 8 Aggregate Operations:

import java.util.\*;

import java.util.stream.\*;

public class FindDups {

public static void main(String[] args) {

Set<String> distinctWords = Arrays.asList(args).stream()

.collect(Collectors.toSet());

System.out.println(distinctWords.size()+

" distinct words: " +

distinctWords);

}

}

Using the for-each Construct:

import java.util.\*;

public class FindDups {

public static void main(String[] args) {

Set<String> s = new HashSet<String>();

for (String a : args)

s.add(a);

System.out.println(s.size() + " distinct words: " + s);

}

}

Now run either version of the program.

java FindDups i came i saw i left

The following output is produced:

4 distinct words: [left, came, saw, i]

Note that the code always refers to the Collection by its interface type (Set) rather than by its implementation type. This is a *strongly* recommended programming practice because it gives you the flexibility to change implementations merely by changing the constructor. If either of the variables used to store a collection or the parameters used to pass it around are declared to be of the Collection's implementation type rather than its interface type, *all* such variables and parameters must be changed in order to change its implementation type.

Furthermore, there's no guarantee that the resulting program will work. If the program uses any nonstandard operations present in the original implementation type but not in the new one, the program will fail. Referring to collections only by their interface prevents you from using any nonstandard operations.

The implementation type of the Set in the preceding example is HashSet, which makes no guarantees as to the order of the elements in the Set. If you want the program to print the word list in alphabetical order, merely change the Set's implementation type from HashSet to TreeSet. Making this trivial one-line change causes the command line in the previous example to generate the following output.

java FindDups i came i saw i left

4 distinct words: [came, i, left, saw]

## Set Interface Bulk Operations

Bulk operations are particularly well suited to Sets; when applied, they perform standard set-algebraic operations. Suppose s1 and s2 are sets. Here's what bulk operations do:

* s1.containsAll(s2) — returns true if s2 is a **subset** of s1. (s2 is a subset of s1 if set s1 contains all of the elements in s2.)
* s1.addAll(s2) — transforms s1 into the **union** of s1 and s2. (The union of two sets is the set containing all of the elements contained in either set.)
* s1.retainAll(s2) — transforms s1 into the intersection of s1 and s2. (The intersection of two sets is the set containing only the elements common to both sets.)
* s1.removeAll(s2) — transforms s1 into the (asymmetric) set difference of s1 and s2. (For example, the set difference of s1 minus s2 is the set containing all of the elements found in s1 but not in s2.)

To calculate the union, intersection, or set difference of two sets *nondestructively* (without modifying either set), the caller must copy one set before calling the appropriate bulk operation. The following are the resulting idioms.

Set<Type> union = new HashSet<Type>(s1);

union.addAll(s2);

Set<Type> intersection = new HashSet<Type>(s1);

intersection.retainAll(s2);

Set<Type> difference = new HashSet<Type>(s1);

difference.removeAll(s2);

The implementation type of the result Set in the preceding idioms is HashSet, which is, as already mentioned, the best all-around Set implementation in the Java platform. However, any general-purpose Set implementation could be substituted.

Let's revisit the FindDups program. Suppose you want to know which words in the argument list occur only once and which occur more than once, but you do not want any duplicates printed out repeatedly. This effect can be achieved by generating two sets — one containing every word in the argument list and the other containing only the duplicates. The words that occur only once are the set difference of these two sets, which we know how to compute. Here's how [the resulting program](https://docs.oracle.com/javase/tutorial/collections/interfaces/examples/FindDups2.java) looks.

import java.util.\*;

public class FindDups2 {

public static void main(String[] args) {

Set<String> uniques = new HashSet<String>();

Set<String> dups = new HashSet<String>();

for (String a : args)

if (!uniques.add(a))

dups.add(a);

// Destructive set-difference

uniques.removeAll(dups);

System.out.println("Unique words: " + uniques);

System.out.println("Duplicate words: " + dups);

}

}

When run with the same argument list used earlier (i came i saw i left), the program yields the following output.

Unique words: [left, saw, came]

Duplicate words: [i]

A less common set-algebraic operation is the *symmetric set difference* — the set of elements contained in either of two specified sets but not in both. The following code calculates the symmetric set difference of two sets nondestructively.

Set<Type> symmetricDiff = new HashSet<Type>(s1);

symmetricDiff.addAll(s2);

Set<Type> tmp = new HashSet<Type>(s1);

tmp.retainAll(s2);

symmetricDiff.removeAll(tmp);

## Set Interface Array Operations

The array operations don't do anything special for Sets beyond what they do for any other Collection. These operations are described in [The Collection Interface](https://docs.oracle.com/javase/tutorial/collections/interfaces/collection.html) section.

# The List Interface

A [List](http://docs.oracle.com/javase/8/docs/api/java/util/List.html) is an ordered [Collection](http://docs.oracle.com/javase/8/docs/api/java/util/Collection.html) (sometimes called a *sequence*). Lists may contain duplicate elements. In addition to the operations inherited from Collection, the List interface includes operations for the following:

* Positional access — manipulates elements based on their numerical position in the list. This includes methods such as get, set, add, addAll, and remove.
* Search — searches for a specified object in the list and returns its numerical position. Search methods include indexOf and lastIndexOf.
* Iteration — extends Iterator semantics to take advantage of the list's sequential nature. The listIterator methods provide this behavior.
* Range-view — The sublist method performs arbitrary *range operations* on the list.

The Java platform contains two general-purpose List implementations. [ArrayList](http://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html), which is usually the better-performing implementation, and [LinkedList](http://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html) which offers better performance under certain circumstances.

## Collection Operations

The operations inherited from Collection all do about what you'd expect them to do, assuming you're already familiar with them. If you're not familiar with them from Collection, now would be a good time to read [The Collection Interface](https://docs.oracle.com/javase/tutorial/collections/interfaces/collection.html) section. The remove operation always removes *the first* occurrence of the specified element from the list. The add and addAll operations always append the new element(s) to the *end* of the list. Thus, the following idiom concatenates one list to another.

list1.addAll(list2);

Here's a nondestructive form of this idiom, which produces a third List consisting of the second list appended to the first.

List<Type> list3 = new ArrayList<Type>(list1);

list3.addAll(list2);

Note that the idiom, in its nondestructive form, takes advantage of ArrayList's standard conversion constructor.

And here's an example (JDK 8 and later) that aggregates some names into a List:

List<String> list = people.stream()

.map(Person::getName)

.collect(Collectors.toList());

Like the [Set](http://docs.oracle.com/javase/8/docs/api/java/util/Set.html) interface, List strengthens the requirements on the equals and hashCode methods so that two List objects can be compared for logical equality without regard to their implementation classes. Two List objects are equal if they contain the same elements in the same order.

## Positional Access and Search Operations

The basic positional access operations are get, set, add and remove. (The set and remove operations return the old value that is being overwritten or removed.) Other operations (indexOf and lastIndexOf) return the first or last index of the specified element in the list.

The addAll operation inserts all the elements of the specified Collection starting at the specified position. The elements are inserted in the order they are returned by the specified Collection's iterator. This call is the positional access analog of Collection's addAll operation.

Here's a little method to swap two indexed values in a List.

public static <E> void swap(List<E> a, int i, int j) {

E tmp = a.get(i);

a.set(i, a.get(j));

a.set(j, tmp);

}

Of course, there's one big difference. This is a polymorphic algorithm: It swaps two elements in any List, regardless of its implementation type. Here's another polymorphic algorithm that uses the preceding swap method.

public static void shuffle(List<?> list, Random rnd) {

for (int i = list.size(); i > 1; i--)

swap(list, i - 1, rnd.nextInt(i));

}

This algorithm, which is included in the Java platform's [Collections](http://docs.oracle.com/javase/8/docs/api/java/util/Collections.html) class, randomly permutes the specified list using the specified source of randomness. It's a bit subtle: It runs up the list from the bottom, repeatedly swapping a randomly selected element into the current position. Unlike most naive attempts at shuffling, it's *fair* (all permutations occur with equal likelihood, assuming an unbiased source of randomness) and *fast* (requiring exactly list.size()-1 swaps). The following program uses this algorithm to print the words in its argument list in random order.

import java.util.\*;

public class Shuffle {

public static void main(String[] args) {

List<String> list = new ArrayList<String>();

for (String a : args)

list.add(a);

Collections.shuffle(list, new Random());

System.out.println(list);

}

}

In fact, this program can be made even shorter and faster. The [Arrays](http://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html) class has a static factory method called asList, which allows an array to be viewed as a List. This method does not copy the array. Changes in the List write through to the array and vice versa. The resulting List is not a general-purpose List implementation, because it doesn't implement the (optional) add and remove operations: Arrays are not resizable. Taking advantage of Arrays.asList and calling the library version of shuffle, which uses a default source of randomness, you get the following [tiny program](https://docs.oracle.com/javase/tutorial/collections/interfaces/examples/Shuffle.java) whose behavior is identical to the previous program.

import java.util.\*;

public class Shuffle {

public static void main(String[] args) {

List<String> list = Arrays.asList(args);

Collections.shuffle(list);

System.out.println(list);

}

}

## Iterators

As you'd expect, the Iterator returned by List's iterator operation returns the elements of the list in proper sequence. List also provides a richer iterator, called a ListIterator, which allows you to traverse the list in either direction, modify the list during iteration, and obtain the current position of the iterator.

The three methods that ListIterator inherits from Iterator (hasNext, next, and remove) do exactly the same thing in both interfaces. The hasPrevious and the previous operations are exact analogues of hasNext and next. The former operations refer to the element before the (implicit) cursor, whereas the latter refer to the element after the cursor. The previous operation moves the cursor backward, whereas next moves it forward.

Here's the standard idiom for iterating backward through a list.

for (ListIterator<Type> it = list.listIterator(list.size()); it.hasPrevious(); ) {

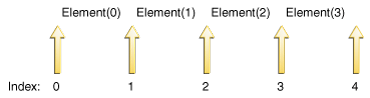
Type t = it.previous();

...

}

Note the argument to listIterator in the preceding idiom. The List interface has two forms of the listIterator method. The form with no arguments returns a ListIterator positioned at the beginning of the list; the form with an int argument returns a ListIterator positioned at the specified index. The index refers to the element that would be returned by an initial call to next. An initial call to previous would return the element whose index was index-1. In a list of length n, there are n+1 valid values for index, from 0 to n, inclusive.

Intuitively speaking, the cursor is always between two elements — the one that would be returned by a call to previous and the one that would be returned by a call to next. The n+1 valid index values correspond to the n+1 gaps between elements, from the gap before the first element to the gap after the last one. The following figure shows the five possible cursor positions in a list containing four elements.



The five possible cursor positions.

Calls to next and previous can be intermixed, but you have to be a bit careful. The first call to previous returns the same element as the last call to next. Similarly, the first call to next after a sequence of calls to previous returns the same element as the last call to previous.

It should come as no surprise that the nextIndex method returns the index of the element that would be returned by a subsequent call to next, and previousIndex returns the index of the element that would be returned by a subsequent call to previous. These calls are typically used either to report the position where something was found or to record the position of the ListIterator so that another ListIterator with identical position can be created.

It should also come as no surprise that the number returned by nextIndex is always one greater than the number returned by previousIndex. This implies the behavior of the two boundary cases: (1) a call to previousIndex when the cursor is before the initial element returns -1 and (2) a call to nextIndex when the cursor is after the final element returns list.size(). To make all this concrete, the following is a possible implementation of List.indexOf.

public int indexOf(E e) {

for (ListIterator<E> it = listIterator(); it.hasNext(); )

if (e == null ? it.next() == null : e.equals(it.next()))

return it.previousIndex();

// Element not found

return -1;

}

Note that the indexOf method returns it.previousIndex() even though it is traversing the list in the forward direction. The reason is that it.nextIndex() would return the index of the element we are about to examine, and we want to return the index of the element we just examined.

The Iterator interface provides the remove operation to remove the last element returned by next from the Collection. For ListIterator, this operation removes the last element returned by next or previous. The ListIterator interface provides two additional operations to modify the list — set and add. The set method overwrites the last element returned by next or previous with the specified element. The following polymorphic algorithm uses set to replace all occurrences of one specified value with another.

public static <E> void replace(List<E> list, E val, E newVal) {

for (ListIterator<E> it = list.listIterator(); it.hasNext(); )

if (val == null ? it.next() == null : val.equals(it.next()))

it.set(newVal);

}

The only bit of trickiness in this example is the equality test between val and it.next. You need to special-case a val value of null to prevent a NullPointerException.

The add method inserts a new element into the list immediately before the current cursor position. This method is illustrated in the following polymorphic algorithm to replace all occurrences of a specified value with the sequence of values contained in the specified list.

public static <E>

void replace(List<E> list, E val, List<? extends E> newVals) {

for (ListIterator<E> it = list.listIterator(); it.hasNext(); ){

if (val == null ? it.next() == null : val.equals(it.next())) {

it.remove();

for (E e : newVals)

it.add(e);

}

}

}

## Range-View Operation

The range-view operation, subList(int fromIndex, int toIndex), returns a List view of the portion of this list whose indices range from fromIndex, inclusive, to toIndex, exclusive. This *half-open range* mirrors the typical for loop.

for (int i = fromIndex; i < toIndex; i++) {

...

}

As the term *view* implies, the returned List is backed up by the List on which subList was called, so changes in the former are reflected in the latter.

This method eliminates the need for explicit range operations (of the sort that commonly exist for arrays). Any operation that expects a List can be used as a range operation by passing a subList view instead of a whole List. For example, the following idiom removes a range of elements from a List.

list.subList(fromIndex, toIndex).clear();

Similar idioms can be constructed to search for an element in a range.

int i = list.subList(fromIndex, toIndex).indexOf(o);

int j = list.subList(fromIndex, toIndex).lastIndexOf(o);

Note that the preceding idioms return the index of the found element in the subList, not the index in the backing List.

Any polymorphic algorithm that operates on a List, such as the replace and shuffle examples, works with the List returned by subList.

Here's a polymorphic algorithm whose implementation uses subList to deal a hand from a deck. That is, it returns a new List (the "hand") containing the specified number of elements taken from the end of the specified List (the "deck"). The elements returned in the hand are removed from the deck.

public static <E> List<E> dealHand(List<E> deck, int n) {

int deckSize = deck.size();

List<E> handView = deck.subList(deckSize - n, deckSize);

List<E> hand = new ArrayList<E>(handView);

handView.clear();

return hand;

}

Note that this algorithm removes the hand from the *end* of the deck. For many common List implementations, such as ArrayList, the performance of removing elements from the end of the list is substantially better than that of removing elements from the beginning.

The following is [a program](https://docs.oracle.com/javase/tutorial/collections/interfaces/examples/Deal.java) that uses the dealHand method in combination with Collections.shuffle to generate hands from a normal 52-card deck. The program takes two command-line arguments: (1) the number of hands to deal and (2) the number of cards in each hand.

import java.util.\*;

public class Deal {

public static void main(String[] args) {

if (args.length < 2) {

System.out.println("Usage: Deal hands cards");

return;

}

int numHands = Integer.parseInt(args[0]);

int cardsPerHand = Integer.parseInt(args[1]);

// Make a normal 52-card deck.

String[] suit = new String[] {

"spades", "hearts",

"diamonds", "clubs"

};

String[] rank = new String[] {

"ace", "2", "3", "4",

"5", "6", "7", "8", "9", "10",

"jack", "queen", "king"

};

List<String> deck = new ArrayList<String>();

for (int i = 0; i < suit.length; i++)

for (int j = 0; j < rank.length; j++)

deck.add(rank[j] + " of " + suit[i]);

// Shuffle the deck.

Collections.shuffle(deck);

if (numHands \* cardsPerHand > deck.size()) {

System.out.println("Not enough cards.");

return;

}

for (int i = 0; i < numHands; i++)

System.out.println(dealHand(deck, cardsPerHand));

}

public static <E> List<E> dealHand(List<E> deck, int n) {

int deckSize = deck.size();

List<E> handView = deck.subList(deckSize - n, deckSize);

List<E> hand = new ArrayList<E>(handView);

handView.clear();

return hand;

}

}

Running the program produces output like the following.

% java Deal 4 5

[8 of hearts, jack of spades, 3 of spades, 4 of spades,

king of diamonds]

[4 of diamonds, ace of clubs, 6 of clubs, jack of hearts,

queen of hearts]

[7 of spades, 5 of spades, 2 of diamonds, queen of diamonds,

9 of clubs]

[8 of spades, 6 of diamonds, ace of spades, 3 of hearts,

ace of hearts]

Although the subList operation is extremely powerful, some care must be exercised when using it. The semantics of the List returned by subList become undefined if elements are added to or removed from the backing List in any way other than via the returned List. Thus, it's highly recommended that you use the List returned by subList only as a transient object — to perform one or a sequence of range operations on the backing List. The longer you use the subList instance, the greater the probability that you'll compromise it by modifying the backing List directly or through another subList object. Note that it is legal to modify a sublist of a sublist and to continue using the original sublist (though not concurrently).

## List Algorithms

Most polymorphic algorithms in the Collections class apply specifically to List. Having all these algorithms at your disposal makes it very easy to manipulate lists. Here's a summary of these algorithms, which are described in more detail in the [Algorithms](https://docs.oracle.com/javase/tutorial/collections/algorithms/index.html) section.

* sort — sorts a List using a merge sort algorithm, which provides a fast, stable sort. (A *stable sort* is one that does not reorder equal elements.)
* shuffle — randomly permutes the elements in a List.
* reverse — reverses the order of the elements in a List.
* rotate — rotates all the elements in a List by a specified distance.
* swap — swaps the elements at specified positions in a List.
* replaceAll — replaces all occurrences of one specified value with another.
* fill — overwrites every element in a List with the specified value.
* copy — copies the source List into the destination List.
* binarySearch — searches for an element in an ordered List using the binary search algorithm.
* indexOfSubList — returns the index of the first sublist of one List that is equal to another.
* lastIndexOfSubList — returns the index of the last sublist of one List that is equal to another.

# The Queue Interface

A [Queue](http://docs.oracle.com/javase/8/docs/api/java/util/Queue.html) is a collection for holding elements prior to processing. Besides basic Collection operations, queues provide additional insertion, removal, and inspection operations. The Queue interface follows.

public interface Queue<E> extends Collection<E> {

E element();

boolean offer(E e);

E peek();

E poll();

E remove();

}

Each Queue method exists in two forms: (1) one throws an exception if the operation fails, and (2) the other returns a special value if the operation fails (either null or false, depending on the operation). The regular structure of the interface is illustrated in the following table.

|  |  |  |
| --- | --- | --- |
| **Queue Interface Structure** | | |
| **Type of Operation** | **Throws exception** | **Returns special value** |
| Insert | add(e) | offer(e) |
| Remove | remove() | poll() |
| Examine | element() | peek() |

Queues typically, but not necessarily, order elements in a FIFO (first-in-first-out) manner. Among the exceptions are priority queues, which order elements according to their values — see the [Object Ordering](https://docs.oracle.com/javase/tutorial/collections/interfaces/order.html) section for details). Whatever ordering is used, the head of the queue is the element that would be removed by a call to remove or poll. In a FIFO queue, all new elements are inserted at the tail of the queue. Other kinds of queues may use different placement rules. Every Queue implementation must specify its ordering properties.

It is possible for a Queue implementation to restrict the number of elements that it holds; such queues are known as *bounded*. Some Queue implementations in java.util.concurrent are bounded, but the implementations in java.util are not.

The add method, which Queue inherits from Collection, inserts an element unless it would violate the queue's capacity restrictions, in which case it throws IllegalStateException. The offer method, which is intended solely for use on bounded queues, differs from add only in that it indicates failure to insert an element by returning false.

The remove and poll methods both remove and return the head of the queue. Exactly which element gets removed is a function of the queue's ordering policy. The remove and poll methods differ in their behavior only when the queue is empty. Under these circumstances, remove throws NoSuchElementException, while poll returns null.

The element and peek methods return, but do not remove, the head of the queue. They differ from one another in precisely the same fashion as remove and poll: If the queue is empty, element throws NoSuchElementException, while peek returns null.

Queue implementations generally do not allow insertion of null elements. The LinkedList implementation, which was retrofitted to implement Queue, is an exception. For historical reasons, it permits null elements, but you should refrain from taking advantage of this, because null is used as a special return value by the poll and peek methods.

Queue implementations generally do not define element-based versions of the equals and hashCode methods but instead inherit the identity-based versions from Object.

The Queue interface does not define the blocking queue methods, which are common in concurrent programming. These methods, which wait for elements to appear or for space to become available, are defined in the interface [java.util.concurrent.BlockingQueue](http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingQueue.html), which extends Queue.

In the following example program, a queue is used to implement a countdown timer. The queue is preloaded with all the integer values from a number specified on the command line to zero, in descending order. Then, the values are removed from the queue and printed at one-second intervals. The program is artificial in that it would be more natural to do the same thing without using a queue, but it illustrates the use of a queue to store elements prior to subsequent processing.

import java.util.\*;

public class Countdown {

public static void main(String[] args) throws InterruptedException {

int time = Integer.parseInt(args[0]);

Queue<Integer> queue = new LinkedList<Integer>();

for (int i = time; i >= 0; i--)

queue.add(i);

while (!queue.isEmpty()) {

System.out.println(queue.remove());

Thread.sleep(1000);

}

}

}

In the following example, a priority queue is used to sort a collection of elements. Again this program is artificial in that there is no reason to use it in favor of the sort method provided in Collections, but it illustrates the behavior of priority queues.

static <E> List<E> heapSort(Collection<E> c) {

Queue<E> queue = new PriorityQueue<E>(c);

List<E> result = new ArrayList<E>();

while (!queue.isEmpty())

result.add(queue.remove());

return result;

}

# The Deque Interface

Usually pronounced as deck, a deque is a double-ended-queue. A double-ended-queue is a linear collection of elements that supports the insertion and removal of elements at both end points. The Deque interface is a richer abstract data type than both Stack and Queue because it implements both stacks and queues at the same time. The [Deque](http://docs.oracle.com/javase/8/docs/api/java/util/Deque.html) interface, defines methods to access the elements at both ends of the Deque instance. Methods are provided to insert, remove, and examine the elements. Predefined classes like [ArrayDeque](http://docs.oracle.com/javase/8/docs/api/java/util/ArrayDeque.html) and [LinkedList](http://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html) implement the Deque interface.

Note that the Deque interface can be used both as last-in-first-out stacks and first-in-first-out queues. The methods given in the Deque interface are divided into three parts:

## Insert

The addfirst and offerFirst methods insert elements at the beginning of the Deque instance. The methods addLast and offerLast insert elements at the end of the Deque instance. When the capacity of the Deque instance is restricted, the preferred methods are offerFirst and offerLast because addFirst might fail to throw an exception if it is full.

## Remove

The removeFirst and pollFirst methods remove elements from the beginning of the Deque instance. The removeLast and pollLast methods remove elements from the end. The methods pollFirst and pollLast return null if the Deque is empty whereas the methods removeFirst and removeLast throw an exception if the Deque instance is empty.

## Retrieve

The methods getFirst and peekFirst retrieve the first element of the Deque instance. These methods dont remove the value from the Deque instance. Similarly, the methods getLast and peekLast retrieve the last element. The methods getFirst and getLast throw an exception if the deque instance is empty whereas the methods peekFirst and peekLast return NULL.

The 12 methods for insertion, removal and retieval of Deque elements are summarized in the following table:

|  |  |  |
| --- | --- | --- |
| **Deque Methods** | | |
| **Type of Operation** | **First Element (Beginning of the Deque instance)** | **Last Element (End of the Deque instance)** |
| **Insert** | addFirst(e) offerFirst(e) | addLast(e) offerLast(e) |
| **Remove** | removeFirst() pollFirst() | removeLast() pollLast() |
| **Examine** | getFirst() peekFirst() | getLast() peekLast() |

In addition to these basic methods to insert,remove and examine a Deque instance, the Deque interface also has some more predefined methods. One of these is removeFirstOccurence, this method removes the first occurence of the specified element if it exists in the Deque instance. If the element does not exist then the Deque instance remains unchanged. Another similar method is removeLastOccurence; this method removes the last occurence of the specified element in the Deque instance. The return type of these methods is boolean, and they return true if the element exists in the Deque instance.

# The Map Interface

A [Map](http://docs.oracle.com/javase/8/docs/api/java/util/Map.html) is an object that maps keys to values. A map cannot contain duplicate keys: Each key can map to at most one value. It models the mathematical *function* abstraction. The Map interface includes methods for basic operations (such as put, get, remove, containsKey, containsValue, size, and empty), bulk operations (such as putAll and clear), and collection views (such as keySet, entrySet, and values).

The Java platform contains three general-purpose Map implementations: [HashMap](http://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html), [TreeMap](http://docs.oracle.com/javase/8/docs/api/java/util/TreeMap.html), and [LinkedHashMap](http://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashMap.html). Their behavior and performance are precisely analogous to HashSet, TreeSet, and LinkedHashSet, as described in [The Set Interface](https://docs.oracle.com/javase/tutorial/collections/interfaces/set.html) section.

The remainder of this page discusses the Map interface in detail. But first, here are some more examples of collecting to Maps using JDK 8 aggregate operations. Modeling real-world objects is a common task in object-oriented programming, so it is reasonable to think that some programs might, for example, group employees by department:

// Group employees by department

Map<Department, List<Employee>> byDept = employees.stream()

.collect(Collectors.groupingBy(Employee::getDepartment));

Or compute the sum of all salaries by department:

// Compute sum of salaries by department

Map<Department, Integer> totalByDept = employees.stream()

.collect(Collectors.groupingBy(Employee::getDepartment,

Collectors.summingInt(Employee::getSalary)));

Or perhaps group students by passing or failing grades:

// Partition students into passing and failing

Map<Boolean, List<Student>> passingFailing = students.stream()

.collect(Collectors.partitioningBy(s -> s.getGrade()>= PASS\_THRESHOLD));

You could also group people by city:

// Classify Person objects by city

Map<String, List<Person>> peopleByCity

= personStream.collect(Collectors.groupingBy(Person::getCity));

Or even cascade two collectors to classify people by state and city:

// Cascade Collectors

Map<String, Map<String, List<Person>>> peopleByStateAndCity

= personStream.collect(Collectors.groupingBy(Person::getState,

Collectors.groupingBy(Person::getCity)))

Again, these are but a few examples of how to use the new JDK 8 APIs. For in-depth coverage of lambda expressions and aggregate operations see the lesson entitled [Aggregate Operations](https://docs.oracle.com/javase/tutorial/collections/streams/index.html).

## Map Interface Basic Operations

The basic operations of Map (put, get, containsKey, containsValue, size, and isEmpty) behave exactly like their counterparts in Hashtable. The [following program](https://docs.oracle.com/javase/tutorial/collections/interfaces/examples/Freq.java) generates a frequency table of the words found in its argument list. The frequency table maps each word to the number of times it occurs in the argument list.

import java.util.\*;

public class Freq {

public static void main(String[] args) {

Map<String, Integer> m = new HashMap<String, Integer>();

// Initialize frequency table from command line

for (String a : args) {

Integer freq = m.get(a);

m.put(a, (freq == null) ? 1 : freq + 1);

}

System.out.println(m.size() + " distinct words:");

System.out.println(m);

}

}

The only tricky thing about this program is the second argument of the put statement. That argument is a conditional expression that has the effect of setting the frequency to one if the word has never been seen before or one more than its current value if the word has already been seen. Try running this program with the command:

java Freq if it is to be it is up to me to delegate

The program yields the following output.

8 distinct words:

{to=3, delegate=1, be=1, it=2, up=1, if=1, me=1, is=2}

Suppose you'd prefer to see the frequency table in alphabetical order. All you have to do is change the implementation type of the Map from HashMap to TreeMap. Making this four-character change causes the program to generate the following output from the same command line.

8 distinct words:

{be=1, delegate=1, if=1, is=2, it=2, me=1, to=3, up=1}

Similarly, you could make the program print the frequency table in the order the words first appear on the command line simply by changing the implementation type of the map to LinkedHashMap. Doing so results in the following output.

8 distinct words:

{if=1, it=2, is=2, to=3, be=1, up=1, me=1, delegate=1}

This flexibility provides a potent illustration of the power of an interface-based framework.

Like the [Set](http://docs.oracle.com/javase/8/docs/api/java/util/Set.html)and [List](http://docs.oracle.com/javase/8/docs/api/java/util/List.html)interfaces, Map strengthens the requirements on the equals and hashCode methods so that two Map objects can be compared for logical equality without regard to their implementation types. Two Map instances are equal if they represent the same key-value mappings.

By convention, all general-purpose Map implementations provide constructors that take a Map object and initialize the new Map to contain all the key-value mappings in the specified Map. This standard Map conversion constructor is entirely analogous to the standard Collection constructor: It allows the caller to create a Map of a desired implementation type that initially contains all of the mappings in another Map, regardless of the other Map's implementation type. For example, suppose you have a Map, named m. The following one-liner creates a new HashMap initially containing all of the same key-value mappings as m.

Map<K, V> copy = new HashMap<K, V>(m);

## Map Interface Bulk Operations

The clear operation does exactly what you would think it could do: It removes all the mappings from the Map. The putAll operation is the Map analogue of the Collection interface's addAll operation. In addition to its obvious use of dumping one Map into another, it has a second, more subtle use. Suppose a Map is used to represent a collection of attribute-value pairs; the putAll operation, in combination with the Map conversion constructor, provides a neat way to implement attribute map creation with default values. The following is a static factory method that demonstrates this technique.

static <K, V> Map<K, V> newAttributeMap(Map<K, V>defaults, Map<K, V> overrides) {

Map<K, V> result = new HashMap<K, V>(defaults);

result.putAll(overrides);

return result;

}

## Collection Views

The Collection view methods allow a Map to be viewed as a Collection in these three ways:

* keySet — the Set of keys contained in the Map.
* values — The Collection of values contained in the Map. This Collection is not a Set, because multiple keys can map to the same value.
* entrySet — the Set of key-value pairs contained in the Map. The Map interface provides a small nested interface called Map.Entry, the type of the elements in this Set.

The Collection views provide the *only* means to iterate over a Map. This example illustrates the standard idiom for iterating over the keys in a Map with a for-each construct:

for (KeyType key : m.keySet())

System.out.println(key);

and with an iterator:

// Filter a map based on some

// property of its keys.

for (Iterator<Type> it = m.keySet().iterator(); it.hasNext(); )

if (it.next().isBogus())

it.remove();

The idiom for iterating over values is analogous. Following is the idiom for iterating over key-value pairs.

for (Map.Entry<KeyType, ValType> e : m.entrySet())

System.out.println(e.getKey() + ": " + e.getValue());

At first, many people worry that these idioms may be slow because the Map has to create a new Collection instance each time a Collection view operation is called. Rest easy: There's no reason that a Map cannot always return the same object each time it is asked for a given Collection view. This is precisely what all the Map implementations in java.util do.

With all three Collection views, calling an Iterator's remove operation removes the associated entry from the backing Map, assuming that the backing Map supports element removal to begin with. This is illustrated by the preceding filtering idiom.

With the entrySet view, it is also possible to change the value associated with a key by calling a Map.Entry's setValue method during iteration (again, assuming the Map supports value modification to begin with). Note that these are the *only* safe ways to modify a Map during iteration; the behavior is unspecified if the underlying Map is modified in any other way while the iteration is in progress.

The Collection views support element removal in all its many forms — remove, removeAll, retainAll, and clear operations, as well as the Iterator.remove operation. (Yet again, this assumes that the backing Map supports element removal.)

The Collection views *do not* support element addition under any circumstances. It would make no sense for the keySet and values views, and it's unnecessary for the entrySet view, because the backing Map's put and putAll methods provide the same functionality.

## Fancy Uses of Collection Views: Map Algebra

When applied to the Collection views, bulk operations (containsAll, removeAll, and retainAll) are surprisingly potent tools. For starters, suppose you want to know whether one Map is a submap of another — that is, whether the first Map contains all the key-value mappings in the second. The following idiom does the trick.

if (m1.entrySet().containsAll(m2.entrySet())) {

...

}

Along similar lines, suppose you want to know whether two Map objects contain mappings for all of the same keys.

if (m1.keySet().equals(m2.keySet())) {

...

}

Suppose you have a Map that represents a collection of attribute-value pairs, and two Sets representing required attributes and permissible attributes. (The permissible attributes include the required attributes.) The following snippet determines whether the attribute map conforms to these constraints and prints a detailed error message if it doesn't.

static <K, V> boolean validate(Map<K, V> attrMap, Set<K> requiredAttrs, Set<K>permittedAttrs) {

boolean valid = true;

Set<K> attrs = attrMap.keySet();

if (! attrs.containsAll(requiredAttrs)) {

Set<K> missing = new HashSet<K>(requiredAttrs);

missing.removeAll(attrs);

System.out.println("Missing attributes: " + missing);

valid = false;

}

if (! permittedAttrs.containsAll(attrs)) {

Set<K> illegal = new HashSet<K>(attrs);

illegal.removeAll(permittedAttrs);

System.out.println("Illegal attributes: " + illegal);

valid = false;

}

return valid;

}

Suppose you want to know all the keys common to two Map objects.

Set<KeyType>commonKeys = new HashSet<KeyType>(m1.keySet());

commonKeys.retainAll(m2.keySet());

A similar idiom gets you the common values.

All the idioms presented thus far have been nondestructive; that is, they don't modify the backing Map. Here are a few that do. Suppose you want to remove all of the key-value pairs that one Map has in common with another.

m1.entrySet().removeAll(m2.entrySet());

Suppose you want to remove from one Map all of the keys that have mappings in another.

m1.keySet().removeAll(m2.keySet());

What happens when you start mixing keys and values in the same bulk operation? Suppose you have a Map, managers, that maps each employee in a company to the employee's manager. We'll be deliberately vague about the types of the key and the value objects. It doesn't matter, as long as they're the same. Now suppose you want to know who all the "individual contributors" (or nonmanagers) are. The following snippet tells you exactly what you want to know.

Set<Employee> individualContributors = new HashSet<Employee>(managers.keySet());

individualContributors.removeAll(managers.values());

Suppose you want to fire all the employees who report directly to some manager, Simon.

Employee simon = ... ;

managers.values().removeAll(Collections.singleton(simon));

Note that this idiom makes use of Collections.singleton, a static factory method that returns an immutable Set with the single, specified element.

Once you've done this, you may have a bunch of employees whose managers no longer work for the company (if any of Simon's direct-reports were themselves managers). The following code will tell you which employees have managers who no longer works for the company.

Map<Employee, Employee> m = new HashMap<Employee, Employee>(managers);

m.values().removeAll(managers.keySet());

Set<Employee> slackers = m.keySet();

This example is a bit tricky. First, it makes a temporary copy of the Map, and it removes from the temporary copy all entries whose (manager) value is a key in the original Map. Remember that the original Map has an entry for each employee. Thus, the remaining entries in the temporary Map comprise all the entries from the original Map whose (manager) values are no longer employees. The keys in the temporary copy, then, represent precisely the employees that we're looking for.

There are many more idioms like the ones contained in this section, but it would be impractical and tedious to list them all. Once you get the hang of it, it's not that difficult to come up with the right one when you need it.

## Multimaps

A *multimap* is like a Map but it can map each key to multiple values. The Java Collections Framework doesn't include an interface for multimaps because they aren't used all that commonly. It's a fairly simple matter to use a Map whose values are List instances as a multimap. This technique is demonstrated in the next code example, which reads a word list containing one word per line (all lowercase) and prints out all the anagram groups that meet a size criterion. An *anagram group* is a bunch of words, all of which contain exactly the same letters but in a different order. The program takes two arguments on the command line: (1) the name of the dictionary file and (2) the minimum size of anagram group to print out. Anagram groups containing fewer words than the specified minimum are not printed.

There is a standard trick for finding anagram groups: For each word in the dictionary, alphabetize the letters in the word (that is, reorder the word's letters into alphabetical order) and put an entry into a multimap, mapping the alphabetized word to the original word. For example, the word bad causes an entry mapping abd into bad to be put into the multimap. A moment's reflection will show that all the words to which any given key maps form an anagram group. It's a simple matter to iterate over the keys in the multimap, printing out each anagram group that meets the size constraint.

[The following program](https://docs.oracle.com/javase/tutorial/collections/interfaces/examples/Anagrams.java) is a straightforward implementation of this technique.

import java.util.\*;

import java.io.\*;

public class Anagrams {

public static void main(String[] args) {

int minGroupSize = Integer.parseInt(args[1]);

// Read words from file and put into a simulated multimap

Map<String, List<String>> m = new HashMap<String, List<String>>();

try {

Scanner s = new Scanner(new File(args[0]));

while (s.hasNext()) {

String word = s.next();

String alpha = alphabetize(word);

List<String> l = m.get(alpha);

if (l == null)

m.put(alpha, l=new ArrayList<String>());

l.add(word);

}

} catch (IOException e) {

System.err.println(e);

System.exit(1);

}

// Print all permutation groups above size threshold

for (List<String> l : m.values())

if (l.size() >= minGroupSize)

System.out.println(l.size() + ": " + l);

}

private static String alphabetize(String s) {

char[] a = s.toCharArray();

Arrays.sort(a);

return new String(a);

}

}

Running this program on a 173,000-word dictionary file with a minimum anagram group size of eight produces the following output.

9: [estrin, inerts, insert, inters, niters, nitres, sinter,

triens, trines]

8: [lapse, leaps, pales, peals, pleas, salep, sepal, spale]

8: [aspers, parses, passer, prases, repass, spares, sparse,

spears]

10: [least, setal, slate, stale, steal, stela, taels, tales,

teals, tesla]

8: [enters, nester, renest, rentes, resent, tenser, ternes,

treens]

8: [arles, earls, lares, laser, lears, rales, reals, seral]

8: [earings, erasing, gainers, reagins, regains, reginas,

searing, seringa]

8: [peris, piers, pries, prise, ripes, speir, spier, spire]

12: [apers, apres, asper, pares, parse, pears, prase, presa,

rapes, reaps, spare, spear]

11: [alerts, alters, artels, estral, laster, ratels, salter,

slater, staler, stelar, talers]

9: [capers, crapes, escarp, pacers, parsec, recaps, scrape,

secpar, spacer]

9: [palest, palets, pastel, petals, plates, pleats, septal,

staple, tepals]

9: [anestri, antsier, nastier, ratines, retains, retinas,

retsina, stainer, stearin]

8: [ates, east, eats, etas, sate, seat, seta, teas]

8: [carets, cartes, caster, caters, crates, reacts, recast,

traces]

Many of these words seem a bit bogus, but that's not the program's fault; they're in the dictionary file. Here's the [dictionary file](https://docs.oracle.com/javase/tutorial/collections/interfaces/examples/dictionary.txt) we used. It was derived from the Public Domain ENABLE benchmark reference word list.

# Object Ordering

A List l may be sorted as follows.

Collections.sort(l);

If the List consists of String elements, it will be sorted into alphabetical order. If it consists of Date elements, it will be sorted into chronological order. How does this happen? String and Date both implement the [Comparable](http://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html) interface. Comparable implementations provide a *natural ordering* for a class, which allows objects of that class to be sorted automatically. The following table summarizes some of the more important Java platform classes that implement Comparable.

|  |  |
| --- | --- |
| **Classes Implementing Comparable** | |
| **Class** | **Natural Ordering** |
| Byte | Signed numerical |
| Character | Unsigned numerical |
| Long | Signed numerical |
| Integer | Signed numerical |
| Short | Signed numerical |
| Double | Signed numerical |
| Float | Signed numerical |
| BigInteger | Signed numerical |
| BigDecimal | Signed numerical |
| Boolean | Boolean.FALSE < Boolean.TRUE |
| File | System-dependent lexicographic on path name |
| String | Lexicographic |
| Date | Chronological |
| CollationKey | Locale-specific lexicographic |

If you try to sort a list, the elements of which do not implement Comparable, Collections.sort(list) will throw a [ClassCastException](http://docs.oracle.com/javase/8/docs/api/java/lang/ClassCastException.html). Similarly, Collections.sort(list, comparator) will throw a ClassCastException if you try to sort a list whose elements cannot be compared to one another using the comparator. Elements that can be compared to one another are called *mutually comparable*. Although elements of different types may be mutually comparable, none of the classes listed here permit interclass comparison.

This is all you really need to know about the Comparable interface if you just want to sort lists of comparable elements or to create sorted collections of them. The next section will be of interest to you if you want to implement your own Comparable type.

## Writing Your Own Comparable Types

The Comparable interface consists of the following method.

public interface Comparable<T> {

public int compareTo(T o);

}

The compareTo method compares the receiving object with the specified object and returns a negative integer, 0, or a positive integer depending on whether the receiving object is less than, equal to, or greater than the specified object. If the specified object cannot be compared to the receiving object, the method throws a ClassCastException.

The [following class representing a person's name](https://docs.oracle.com/javase/tutorial/collections/interfaces/examples/Name.java) implements Comparable.

import java.util.\*;

public class Name implements Comparable<Name> {

private final String firstName, lastName;

public Name(String firstName, String lastName) {

if (firstName == null || lastName == null)

throw new NullPointerException();

this.firstName = firstName;

this.lastName = lastName;

}

public String firstName() { return firstName; }

public String lastName() { return lastName; }

public boolean equals(Object o) {

if (!(o instanceof Name))

return false;

Name n = (Name) o;

return n.firstName.equals(firstName) && n.lastName.equals(lastName);

}

public int hashCode() {

return 31\*firstName.hashCode() + lastName.hashCode();

}

public String toString() {

return firstName + " " + lastName;

}

public int compareTo(Name n) {

int lastCmp = lastName.compareTo(n.lastName);

return (lastCmp != 0 ? lastCmp : firstName.compareTo(n.firstName));

}

}

To keep the preceding example short, the class is somewhat limited: It doesn't support middle names, it demands both a first and a last name, and it is not internationalized in any way. Nonetheless, it illustrates the following important points:

* Name objects are *immutable*. All other things being equal, immutable types are the way to go, especially for objects that will be used as elements in Sets or as keys in Maps. These collections will break if you modify their elements or keys while they're in the collection.
* The constructor checks its arguments for null. This ensures that all Name objects are well formed so that none of the other methods will ever throw a NullPointerException.
* The hashCode method is redefined. This is essential for any class that redefines the equals method. (Equal objects must have equal hash codes.)
* The equals method returns false if the specified object is null or of an inappropriate type. The compareTo method throws a runtime exception under these circumstances. Both of these behaviors are required by the general contracts of the respective methods.
* The toString method has been redefined so it prints the Name in human-readable form. This is always a good idea, especially for objects that are going to get put into collections. The various collection types' toString methods depend on the toString methods of their elements, keys, and values.

Since this section is about element ordering, let's talk a bit more about Name's compareTo method. It implements the standard name-ordering algorithm, where last names take precedence over first names. This is exactly what you want in a natural ordering. It would be very confusing indeed if the natural ordering were unnatural!

Take a look at how compareTo is implemented, because it's quite typical. First, you compare the most significant part of the object (in this case, the last name). Often, you can just use the natural ordering of the part's type. In this case, the part is a String and the natural (lexicographic) ordering is exactly what's called for. If the comparison results in anything other than zero, which represents equality, you're done: You just return the result. If the most significant parts are equal, you go on to compare the next most-significant parts. In this case, there are only two parts — first name and last name. If there were more parts, you'd proceed in the obvious fashion, comparing parts until you found two that weren't equal or you were comparing the least-significant parts, at which point you'd return the result of the comparison.

Just to show that it all works, here's [a program that builds a list of names and sorts them](https://docs.oracle.com/javase/tutorial/collections/interfaces/examples/NameSort.java).

import java.util.\*;

public class NameSort {

public static void main(String[] args) {

Name nameArray[] = {

new Name("John", "Smith"),

new Name("Karl", "Ng"),

new Name("Jeff", "Smith"),

new Name("Tom", "Rich")

};

List<Name> names = Arrays.asList(nameArray);

Collections.sort(names);

System.out.println(names);

}

}

If you run this program, here's what it prints.

[Karl Ng, Tom Rich, Jeff Smith, John Smith]

There are four restrictions on the behavior of the compareTo method, which we won't go into now because they're fairly technical and boring and are better left in the API documentation. It's really important that all classes that implement Comparable obey these restrictions, so read the documentation for Comparable if you're writing a class that implements it. Attempting to sort a list of objects that violate the restrictions has undefined behavior. Technically speaking, these restrictions ensure that the natural ordering is a *total order* on the objects of a class that implements it; this is necessary to ensure that sorting is well defined.

## Comparators

What if you want to sort some objects in an order other than their natural ordering? Or what if you want to sort some objects that don't implement Comparable? To do either of these things, you'll need to provide a [Comparator](http://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html) — an object that encapsulates an ordering. Like the Comparable interface, the Comparator interface consists of a single method.

public interface Comparator<T> {

int compare(T o1, T o2);

}

The compare method compares its two arguments, returning a negative integer, 0, or a positive integer depending on whether the first argument is less than, equal to, or greater than the second. If either of the arguments has an inappropriate type for the Comparator, the compare method throws a ClassCastException.

Much of what was said about Comparable applies to Comparator as well. Writing a compare method is nearly identical to writing a compareTo method, except that the former gets both objects passed in as arguments. The compare method has to obey the same four technical restrictions as Comparable's compareTo method for the same reason — a Comparator must induce a total order on the objects it compares.

Suppose you have a class called Employee, as follows.

public class Employee implements Comparable<Employee> {

public Name name() { ... }

public int number() { ... }

public Date hireDate() { ... }

...

}

Let's assume that the natural ordering of Employee instances is Name ordering (as defined in the previous example) on employee name. Unfortunately, the boss has asked for a list of employees in order of seniority. This means we have to do some work, but not much. The following program will produce the required list.

import java.util.\*;

public class EmpSort {

static final Comparator<Employee> SENIORITY\_ORDER =

new Comparator<Employee>() {

public int compare(Employee e1, Employee e2) {

return e2.hireDate().compareTo(e1.hireDate());

}

};

// Employee database

static final Collection<Employee> employees = ... ;

public static void main(String[] args) {

List<Employee> e = new ArrayList<Employee>(employees);

Collections.sort(e, SENIORITY\_ORDER);

System.out.println(e);

}

}

The Comparator in the program is reasonably straightforward. It relies on the natural ordering of Date applied to the values returned by the hireDate accessor method. Note that the Comparator passes the hire date of its second argument to its first rather than vice versa. The reason is that the employee who was hired most recently is the least senior; sorting in the order of hire date would put the list in reverse seniority order. Another technique people sometimes use to achieve this effect is to maintain the argument order but to negate the result of the comparison.

// Don't do this!!

return -r1.hireDate().compareTo(r2.hireDate());

You should always use the former technique in favor of the latter because the latter is not guaranteed to work. The reason for this is that the compareTo method can return any negative int if its argument is less than the object on which it is invoked. There is one negative int that remains negative when negated, strange as it may seem.

-Integer.MIN\_VALUE == Integer.MIN\_VALUE

The Comparator in the preceding program works fine for sorting a List, but it does have one deficiency: It cannot be used to order a sorted collection, such as TreeSet, because it generates an ordering that is *not compatible with* equals. This means that this Comparator equates objects that the equals method does not. In particular, any two employees who were hired on the same date will compare as equal. When you're sorting a List, this doesn't matter; but when you're using the Comparator to order a sorted collection, it's fatal. If you use this Comparator to insert multiple employees hired on the same date into a TreeSet, only the first one will be added to the set; the second will be seen as a duplicate element and will be ignored.

To fix this problem, simply tweak the Comparator so that it produces an ordering that *is compatible with* equals. In other words, tweak it so that the only elements seen as equal when using compare are those that are also seen as equal when compared using equals. The way to do this is to perform a two-part comparison (as for Name), where the first part is the one we're interested in — in this case, the hire date — and the second part is an attribute that uniquely identifies the object. Here the employee number is the obvious attribute. This is the Comparator that results.

static final Comparator<Employee> SENIORITY\_ORDER =

new Comparator<Employee>() {

public int compare(Employee e1, Employee e2) {

int dateCmp = e2.hireDate().compareTo(e1.hireDate());

if (dateCmp != 0)

return dateCmp;

return (e1.number() < e2.number() ? -1 :

(e1.number() == e2.number() ? 0 : 1));

}

};

One last note: You might be tempted to replace the final return statement in the Comparator with the simpler:

return e1.number() - e2.number();

Don't do it unless you're *absolutely sure* no one will ever have a negative employee number! This trick does not work in general because the signed integer type is not big enough to represent the difference of two arbitrary signed integers. If i is a large positive integer and j is a large negative integer, i - j will overflow and will return a negative integer. The resulting comparator violates one of the four technical restrictions we keep talking about (transitivity) and produces horrible, subtle bugs. This is not a purely theoretical concern; people get burned by it.

# The SortedSet Interface

A [SortedSet](http://docs.oracle.com/javase/8/docs/api/java/util/SortedSet.html) is a [Set](http://docs.oracle.com/javase/8/docs/api/java/util/Set.html) that maintains its elements in ascending order, sorted according to the elements' natural ordering or according to a Comparator provided at SortedSet creation time. In addition to the normal Set operations, the SortedSet interface provides operations for the following:

* Range view — allows arbitrary range operations on the sorted set
* Endpoints — returns the first or last element in the sorted set
* Comparator access — returns the Comparator, if any, used to sort the set

The code for the SortedSet interface follows.

public interface SortedSet<E> extends Set<E> {

// Range-view

SortedSet<E> subSet(E fromElement, E toElement);

SortedSet<E> headSet(E toElement);

SortedSet<E> tailSet(E fromElement);

// Endpoints

E first();

E last();

// Comparator access

Comparator<? super E> comparator();

}

## Set Operations

The operations that SortedSet inherits from Set behave identically on sorted sets and normal sets with two exceptions:

* The Iterator returned by the iterator operation traverses the sorted set in order.
* The array returned by toArray contains the sorted set's elements in order.

Although the interface doesn't guarantee it, the toString method of the Java platform's SortedSet implementations returns a string containing all the elements of the sorted set, in order.

## Standard Constructors

By convention, all general-purpose Collection implementations provide a standard conversion constructor that takes a Collection; SortedSet implementations are no exception. In TreeSet, this constructor creates an instance that sorts its elements according to their natural ordering. This was probably a mistake. It would have been better to check dynamically to see whether the specified collection was a SortedSet instance and, if so, to sort the new TreeSet according to the same criterion (comparator or natural ordering). Because TreeSet took the approach that it did, it also provides a constructor that takes a SortedSet and returns a new TreeSet containing the same elements sorted according to the same criterion. Note that it is the compile-time type of the argument, not its runtime type, that determines which of these two constructors is invoked (and whether the sorting criterion is preserved).

SortedSet implementations also provide, by convention, a constructor that takes a Comparator and returns an empty set sorted according to the specified Comparator. If null is passed to this constructor, it returns a set that sorts its elements according to their natural ordering.

## Range-view Operations

The range-view operations are somewhat analogous to those provided by the List interface, but there is one big difference. Range views of a sorted set remain valid even if the backing sorted set is modified directly. This is feasible because the endpoints of a range view of a sorted set are absolute points in the element space rather than specific elements in the backing collection, as is the case for lists. A range-view of a sorted set is really just a window onto whatever portion of the set lies in the designated part of the element space. Changes to the range-view write back to the backing sorted set and vice versa. Thus, it's okay to use range-views on sorted sets for long periods of time, unlike range-views on lists.

Sorted sets provide three range-view operations. The first, subSet, takes two endpoints, like subList. Rather than indices, the endpoints are objects and must be comparable to the elements in the sorted set, using the Set's Comparator or the natural ordering of its elements, whichever the Set uses to order itself. Like subList, the range is half open, including its low endpoint but excluding the high one.

Thus, the following line of code tells you how many words between "doorbell" and "pickle", including "doorbell" but excluding "pickle", are contained in a SortedSet of strings called dictionary:

int count = dictionary.subSet("doorbell", "pickle").size();

In like manner, the following one-liner removes all the elements beginning with the letter f.

dictionary.subSet("f", "g").clear();

A similar trick can be used to print a table telling you how many words begin with each letter.

for (char ch = 'a'; ch <= 'z'; ) {

String from = String.valueOf(ch++);

String to = String.valueOf(ch);

System.out.println(from + ": " + dictionary.subSet(from, to).size());

}

Suppose you want to view a *closed interval*, which contains both of its endpoints, instead of an open interval. If the element type allows for the calculation of the successor of a given value in the element space, merely request the subSet from lowEndpoint to successor(highEndpoint). Although it isn't entirely obvious, the successor of a string s in String's natural ordering is s + "\0" — that is, s with a null character appended.

Thus, the following one-liner tells you how many words between "doorbell" and "pickle", including doorbell *and* pickle, are contained in the dictionary.

count = dictionary.subSet("doorbell", "pickle\0").size();

A similar technique can be used to view an *open interval*, which contains neither endpoint. The open-interval view from lowEndpoint to highEndpoint is the half-open interval from successor(lowEndpoint) to highEndpoint. Use the following to calculate the number of words between "doorbell" and "pickle", excluding both.

count = dictionary.subSet("doorbell\0", "pickle").size();

The SortedSet interface contains two more range-view operations — headSet and tailSet, both of which take a single Object argument. The former returns a view of the initial portion of the backing SortedSet, up to but not including the specified object. The latter returns a view of the final portion of the backing SortedSet, beginning with the specified object and continuing to the end of the backing SortedSet. Thus, the following code allows you to view the dictionary as two disjoint volumes (a-m and n-z).

SortedSet<String> volume1 = dictionary.headSet("n");

SortedSet<String> volume2 = dictionary.tailSet("n");

## Endpoint Operations

The SortedSet interface contains operations to return the first and last elements in the sorted set, not surprisingly called first and last. In addition to their obvious uses, last allows a workaround for a deficiency in the SortedSet interface. One thing you'd like to do with a SortedSet is to go into the interior of the Set and iterate forward or backward. It's easy enough to go forward from the interior: Just get a tailSet and iterate over it. Unfortunately, there's no easy way to go backward.

The following idiom obtains the first element that is less than a specified object o in the element space.

Object predecessor = ss.headSet(o).last();

This is a fine way to go one element backward from a point in the interior of a sorted set. It could be applied repeatedly to iterate backward, but this is very inefficient, requiring a lookup for each element returned.

## Comparator Accessor

# The SortedSet interface contains an accessor method called comparator that returns the Comparator used to sort the set, or null if the set is sorted according to the *natural ordering* of its elements. This method is provided The SortedMap Interface

A [SortedMap](http://docs.oracle.com/javase/8/docs/api/java/util/SortedMap.html) is a [Map](http://docs.oracle.com/javase/8/docs/api/java/util/Map.html) that maintains its entries in ascending order, sorted according to the keys' natural ordering, or according to a Comparator provided at the time of the SortedMap creation. Natural ordering and Comparators are discussed in the [Object Ordering](https://docs.oracle.com/javase/tutorial/collections/interfaces/order.html) section. The SortedMap interface provides operations for normal Map operations and for the following:

* Range view — performs arbitrary range operations on the sorted map
* Endpoints — returns the first or the last key in the sorted map
* Comparator access — returns the Comparator, if any, used to sort the map

The following interface is the Map analog of [SortedSet](http://docs.oracle.com/javase/8/docs/api/java/util/SortedSet.html).

public interface SortedMap<K, V> extends Map<K, V>{

Comparator<? super K> comparator();

SortedMap<K, V> subMap(K fromKey, K toKey);

SortedMap<K, V> headMap(K toKey);

SortedMap<K, V> tailMap(K fromKey);

K firstKey();

K lastKey();

}

## Map Operations

The operations SortedMap inherits from Map behave identically on sorted maps and normal maps with two exceptions:

* The Iterator returned by the iterator operation on any of the sorted map's Collection views traverse the collections in order.
* The arrays returned by the Collection views' toArray operations contain the keys, values, or entries in order.

Although it isn't guaranteed by the interface, the toString method of the Collection views in all the Java platform's SortedMap implementations returns a string containing all the elements of the view, in order.

## Standard Constructors

By convention, all general-purpose Map implementations provide a standard conversion constructor that takes a Map; SortedMap implementations are no exception. In TreeMap, this constructor creates an instance that orders its entries according to their keys' natural ordering. This was probably a mistake. It would have been better to check dynamically to see whether the specified Map instance was a SortedMap and, if so, to sort the new map according to the same criterion (comparator or natural ordering). Because TreeMap took the approach it did, it also provides a constructor that takes a SortedMap and returns a new TreeMap containing the same mappings as the given SortedMap, sorted according to the same criterion. Note that it is the compile-time type of the argument, not its runtime type, that determines whether the SortedMap constructor is invoked in preference to the ordinary map constructor.

SortedMap implementations also provide, by convention, a constructor that takes a Comparator and returns an empty map sorted according to the specified Comparator. If null is passed to this constructor, it returns a Map that sorts its mappings according to their keys' natural ordering.

## Comparison to SortedSet

Because this interface is a precise Map analog of SortedSet, all the idioms and code examples in [The SortedSet Interface](https://docs.oracle.com/javase/tutorial/collections/interfaces/sorted-set.html) section apply to SortedMap with only trivial modifications.