CH1

Java -

enhances object-oriented paradigm

adds integrated support for multithreading

provides library that simplifies Internet access

buzzwords -

simple; secure; portable; object-oriented; multi-threaded; robust (reliable); architecture-neutral; interpreted; high performance; distributed; dynamic;

applets -

downloads automatically; runs on browsers; small; runs locally; runs in special Java environment

servlet -

for servers

platform compatible by bytecode

(Java compiler output not executable code)

runs on Java Virtual Machines

JVMs differ by platform

Just-in-time compiler converts bytecode to executable code in real time

CH2

**Java Overview**

Programs have 2 elements: **code and data (objects)**

Methods are code; variables are data

Programming languages can focus on 1 of 2:

* Process-oriented: code acting on data
* Object-oriented: data controlling access to code

OOP principle:

* Encapsulation – “wraps” code and data and prevent outside from accessing each object of a class
* Inheritance
* Polymorphism – one interface for multiple methods

**Coding basics**

The source code (“compilation unit”) uses *.java* extension.

All codes are within a class

Name of the class that contains the *main()* method must match with the file name (case sensitive)

Classes enclosed by {}

Statements end in semi-colons

Whitespace – at least one whitespace between tokens

Identifiers (names) the following characters are legal:

* uppercase and lowercase letters
* numbers (must not begin with a number)
* \_
* $ (not for general use)

Literal – constants

Keywords – cannot be used as identifiers

CH3

**Strongly typed language – meaning**

1. Every variable or expression has a type
2. Types are strictly defined
3. All assignments are checked for type compatibility

**Primitive Types**

1. Integers
2. Floating point numbers
3. Characters
4. Boolean

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Width** | 1 | 8 | 16 | 32 | 64 |
|  |  |  |  |  |  |
| Integer |  | Byte | Short | Int | Long |
| Float |  |  |  | Float | Double |
| Character |  |  | Char |  |  |
| Boolean | Boolean |  |  |  |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Range** | 8 | 16 | 32 | 64 |
| **integer** | –128 to 127 | –32,768 to 32,767 | –2,147,483,648 to 2,147,483,647 | –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 |
| **float** |  |  | 1.4e-45 to 3.4e38 | 4.9e-324 to 1.8e308 |

float – *single precision* – become imprecise when the value is either very large or very small, or over iterative calculations

double – double precision – can actually be faster on certain processors

character example

*class demo {*

*public static void main(String args[]) {*

*char ch1, ch2;*

*ch1 = 88; //code for X*

*ch2 = ‘Y’;*

*ch1++; //ch1 is now Y*

*}*

*}*

Boolean example

*class demo {*

*public static void main(String args[]) {*

*Boolean a = true;*

*Boolean b = flase;*

*if(a) System.out.println(“is printed”);*

*if(b) System.out.println(“is not printed”);*

*System.out.println(“10 > 9 is “ + (10 > 9));*

*}*

*}*

Integer bases: 10, 8 (octal), 16 (hexadecimal)

* Octal: 0-7
* Hexadecimal: 0-f (a to f represent 10 to 15)

Signify octal constants with leading zero

Signify hexadecimal constants with leading zero-x (0x or 0X)

Signify binary constants with leading zero-b (0b or 0B)

Declare *long* with suffix *L* (e.g. 123456L)

Underscores in integer and floating literals are ignored (e.g. 123\_456\_789 == 123456789)

Floating points default to *double*. To specify a *float* literal, append it with *F* or *f*. To specify a *double* literal, append it with *D* or *d*.

boolean: true != 1, false != 0

String literals

String literals are enclosed in double quotes

Java strings must begin and end on the same line

|  |  |
| --- | --- |
| Escape Sequences | Description |
| \xxx | Octal character for xxx (“0xxx” also works) |
| \uxxxx | Hexadecimal Unicode character for xxxx (“0bxxxx” also works) |
| \’ | single quote |
| \” | double quote |
| \\ | backslash |
| \r | carriage return |
| \n | new line |
| \f | form feed |
| \t | tab |
| \b | backspace |

**Declaring variables**

*type identifier [ = value / identifier];*

*type identifier = Method(parameter); //dynamic initialization*

Scope: variables are created when their scope is entered, and destroyed when their scope is left

**Type conversion**

When assigning value of one type to a variable of another type, *widening conversion* will take place automaticallyif:

* The two types are compatible
  + numeric types are compatible with each other
  + incompatible types: char and boolean; numeric to char or boolean
* The destination type is larger than the source type

**Casting** – explicitly converts between incompatible types. General form:

*(target-type) value*

*//example*

*int a;*

*byte b = (byte) a;*

* If the target type range is too small, the value is reduced modulo to the target type’s range
* When assigning floating point to integer, the fractional component is lost

**Promotion rules**

* mathematical expressions automatically promote byte, short and char to int
* if one operand is long, the whole expression is promoted to long
* if one operand is float, the entire expression is promoted to float
* if one operand is double, the result is double

**Declaring arrays**

First way: *type ArrayName[] = new type[#];*

* *ArrayName* is the array’s name
* # is the number of elements in the array, NOT the number of elements minus 1
* contents of the array default to 0 or false
* Literals or expressions can be used inside array initializers

Second way: *type ArrayName[] = { 1, 2, 3, …}*

Multi-dimensional arrays

In java, multi-dimensional arrays are arrays of arrays

Left index indicates row; right index indicates column

|  |  |  |  |
| --- | --- | --- | --- |
| [0][0] | [0][1] | [0][2] | [0][3] |
| [1][0] | [1][1] | [1][2] | [1][3] |
| [2][0] | [2][1] | [2][2] | [2][3] |

Ways to declare 2D arrays:

1. *type ArrayName[][] = new type[x][y];*
2. *type ArrayName[][] = new type[x][];*

*ArrayName[0] = new type[y0];*

*ArrayName[1] = new type[y1];*

*…*

*ArrayName[x-1] = new type[y];*

* Different rows can have different numbers of elements with this declaration

1. *Type ArrayName[][] = {*

*{1, 2, 3, …},*

*{4, 5, 6, …},*

*…*

*};*

1. *type[][] ArrayName = new type[x][y];*

3-dimensional array: *type ArrayName[][][] = new type[x][y][z]*

Alternate declaration

*type[] ArrayName = new type[x];*

*type[][] ArrayName = new type[x][y];*

useful when declaring multiple arrays: e.g. *int[] array1, array2, array3*

CH4

**Operators**

**Arithmetic operators**

division does not return fraction when acted on int type

*a +=4* means *a = a + 4*

*a%= 2* means *a = a % 2*

general: "*var = var op expression*" => "*var op= expression*"

*a =+ b* means *a = (+b)* (i.e. *a = b*)

compound assignments

*y = ++x* means *x = x + 1; y = x*

*y = x++* means *y = x; x = x + 1*

*a >>=4* means *a = a >> 4*

**Bitwise Operators**

Converting binary number to decimal number: reading from right to left: 20, 21, 22, ... decimal number = sum of the 2n's (n being the places where the binary bit is 1)

e.g.: 00101010 -> 21 + 23 + 25 = 42

Changing signs: invert all bits, then add 1 (two’s complement)

To decode a negative number, first invert all bits, translate to decimal, then add 1

The high-order bit determines the sign of an integer (0 for positive, 1 for negative)

|  |  |
| --- | --- |
| Operator | Explanation |
| ~ | Unary not – inverts all bits |
| & | And – produces 1 if both operands are 1, produces 0 otherwise |
| | | Or— produces 1 if either operands are 1 |
| ^ | Exclusive or (XOR) – produces 1 if exactly one operand is 1 |
| << | Left shift— shifts left a specific number of places (removes left-most bit, adds 0 to the right). Effect: doubles value per place shifted; shifting 1 into high-order position produces negative value |
| >> | Right shift— shifts right, adding the previous top bit's content to the left (sign extension)  Effect: halves value per place shifted and removes remainder |
| >>> | Unsigned right shift – adds 0 on the left (useful when values don't represent numbers) |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| A | B | A | B | A & B | A ^ B | ~A |
| 0 | 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 | 0 |

Examples (Note: 8-bit only for brevity, variables will be promoted to int in practice)

0010 1010

&1010 0111

= 0010 0010

1111 0000 >> 1 = 1111 1000

0011 0010 >> 2 = 0000 1100

0110 0011 << 3 = 0001 1000

1111 1111 >>>4 = 0000 1111

& 0x0f

ANDing with 0x0f (00001111) removes all high order bits

Compound assignments –the following are equivalent:

a >>= 4; a = a >> 4;

a |= b; a = a | b;

**Relational operators**

|  |  |
| --- | --- |
| Operator | Explanation |
| = | Assignment |
| == | Equal to |
| != | Not equal to |
| > |  |
| < |  |
| <= |  |
| >= |  |

outcome is boolean

**Boolean logical operators**

& (and); %= (and assignment)

| (or); |= (or assignment)

^ (exclusive or); ^= (XOR assignment)

! (not)

== (equal to) != (not equal to)

&&, || (short-circuit operators)

* Produces output without evaluating right operand if left operand can determine outcome alone
* Application: can be used if running the right operand would produce an error

?: (three-way if-then-else)

* General form: *A? B: C*
* *A* is anything that evaluates to a boolean value
* If A is true, B is evaluated; if A is false, C is evaluated
* The entire expression is equal to the evaluated expression (B or C)
* B and C must output the same type, which can’t be *void*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **A** | **B** | **A & B** | **A | B** | **A ^ B** | **!A** |
| False | False | False | False | False | True |
| False | True | False | True | True | True |
| True | False | False | True | True | False |
| True | True | True | True | False | False |

standard: use short-circuit AND and OR in cases involving boolean logic, use single-character versions for bitwise operations

**Operator precedence**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Highest** |  |  |  |  |  |
| [] | () | . |  |  |  |
| ++ (postfix) | -- (postfix) |  |  |  |  |
| ++ (prefix) | -- (prefix) | ~ | ! | +/- (unary) | (type-cast) |
| \* | / | % |  |  |  |
| +/- |  |  |  |  |  |
| >> | >>> | << |  |  |  |
| > | >= | < | <= | Instanceof |  |
| == | != |  |  |  |  |
| & |  |  |  |  |  |
| ^ |  |  |  |  |  |
| | |  |  |  |  |  |
| && |  |  |  |  |  |
| || |  |  |  |  |  |
| ?: |  |  |  |  |  |
| = | Op= |  |  |  |  |
| **Lowest** |  |  |  |  |  |

All order of evaluation are from left to right (except assignment, which evaluates right to left)

CH5

**Control Statements – 3 categories: selection, iteration, jump**

**Selection statements: if and switch**

If

* If general structure

*If (condition) a;*

*else b;*

* a and b can be single statements or blocks of statements
* *else* clause is optional
* nested ifs:

*if(condition1) {*

*if(condition2) statement1;*

*if(condition3) statement2;*

*else statement3; //associates with if(condition3)*

*}*

*else statement4;*

* *else* associates with closest *if* (that *if* can’t be already associated with another *else*)
* if-else-if structure

*if (condition1) a;*

*else if (condition2) b;*

*else if (condition3) c;*

…

*else d*

* The first time a condition is true and the associated statement executes, the rest of the ladder is bypassed
* If no conditions are true, the final else statement executes
* if no final else and all conditions are false, no action takes place

Switch structure:

switch (expression) {

case value1:

//sequence;

break;

case value2:

//sequence;

break;

case value3:

case value4:

//sequence;

break; //both case 3 and case 4 will execute the same sequence

…

default:

//sequence;

}

* *expression* can be of type *byte, short, int, char, enumeration,* or *String*
* the value of *expression* is compared with the value of *case;* if match is found, that case is executed; if no match is found, default is executed
* *break* ends a case, and is optional
  + if *break* is not present, both the matching *case* sequence and *default* sequence will be executed
* *default* executes if no case matches. It is optional; if no *default* and no match, no action takes place
* *switch* only tests for equality whereas *if* can test for any boolean expression
* *switches* are more efficient

Nested switch

*Switch (count) {*

*Case1:*

*Switch (target) {*

*Case 1:*

*//sequence;*

*Break;*

*Case 2:*

*//sequence;*

*Break;*

*}*

*Break;*

*Case2:*

*……*

*}*

* *count* and *target* are separate and do not conflict

**iteration statements: for, while, do-while**

**while** structure

*while(condition) {*

*//body}*

* body never executes if condition is always false
* if body is empty, the loop still repeats

**do-while** structure

*do {*

*//body*

*} while(condition);*

* does stuff first, then evaluate condition at the end of each loop
* executes body at least once, even if condition is always false

**for** – 2 types: traditional & for-each

traditional structure

*for(initialization; condition; iteration) {*

*//body}*

* execution order: initialization – condition – (if true) body – iteration – condition (repeat)
* can declare variable in initialization (it only exists within the loop)
* initialization and iteration can have multiple statements – use comma to separate them. e.g.

*for(int a = 1, int b = 4; a<b; a++, b--) {*

*//body }*

* condition can be any boolean statement (does not have to involve loop control variable)
* initialization or iteration can be empty. e.g.

*boolean done = false;*

*for( ; !done; ) {*

*//body }*

* empty initialization, iteration and condition creates an infinite loop. i.e.

*for( ; ; ) {*

*//body}*

for-each loop structure

*for(type IterationVariable: collection) {*

*//body }*

Example

*Int list[] = {1,2,3,4,5 };*

*Int sum = 0;*

*For(int x: list) sum += x;*

* cycles through a collection of objects (e.g. array) in sequential fashion
* iteration variable receives element from collection
* can break the loop early with an if statement in the body of the loop: *if(condition) break;*

Example

*int nums[] = {1, 2, 3, 4, 5, 6, 7, 8};*

*int val = 5;*

*boolean found = false;*

*for(int x: nums) {*

*if(x==val) {*

*found = true;*

*break; }*

*}*

* collection can be multi-dimensional – if collection is N dimensional then iteration variables are N-1 dimensional

Example: cycling through the elements in a 2D array *nums*:

*nums[][] = new int[3][5];*

*for(int x[] : nums) {  
for(int y : x) {*

*//body }*

*}*

**Jump statements: break, continue, return**

break usages

* force immediate termination of a loop, ignoring conditions
* use as a form of *goto*
  + has form *break BlockLabel;*
  + terminates the labelled block and continues immediately after
  + to name a block, use *name: {//body}*
  + the labelled block must enclose the *break* statement

Example

*first: {*

*second: {*

*third: {*

*if(10>9) break second;*

*System.out.println(“does not execute”);*

*}*

*System.out.println(“does not execute”);*

*}*

*System.out.println(“first block still executes”);*

*}*

* note: the if condition is necessary, else an unreachable code error would occur

continue usages

* on *continue*, the current iteration of the loop ends
* *continue label* specifies which loop to continue

Example

*outer: for(int I = 0; I < 10; i++) {*

*System.out.println(“Pass “ + i);*

*for(int j = 0; j < 10; j++) {*

*System.out.print(j + “ “);*

*if(j==5) {*

*System.out.println();*

*continue outer;*

*}*

*}*

return usage

* return to the caller of the method

CH6

**Classes**

* A class defines a new data type
* a class is a template for an object, an object is an instance of a class
* Data/variables defined within a class are **instance variables**
* Code are contained within **methods**
* methods and variables are collectively called **members** of the class

Objects

* Create new objects of custom type by *ClassName ObjectName = new ClassName();*
  + Actually 2 steps: 1) declare variable of class type; 2) assign object to this variable. Functionally identical *to ClassName ObjectName; ObjectName = new ClassName();*
  + Actually, the Object merely refers to the object of the class, rather than is the object
  + ClassName**()** specifies the constructor for the class (often defined within the class)
* Access class variables and methods with dot (.)
* Name the .java file after the class where the main() method is contained
* When you assign one object reference variable to another object reference variable, you are  
  not creating a copy of the object, you are only making a copy of the reference.
  + *box b1 = new box(); box b2 = b1; //b1 and b2 are referring to the same object*

Methods

General form:

*type name(parameters) {//body of method*

*}*

* *type* specifies the type of data returned by the method (*void* if nothing)
* parameters separated by commas
* the value(s) passed to the parameter(s) are called argument(s)
* to return value, use *return value*
  + afterwards, *ClassName.MethodName()* will refer to this returned value
* Can use *ObjectName.method()* to invoke methods.

Constructors

* automatically initializes an object upon creation
* has same name as the class it is in
* default sets everything to 0

form:

*ClassName() {*

*//Sets up object*

*}*

Parameterized constructors form

*ClassName(type parameter1, type parameter2, …) {*

*//Sets up object based on the given parameters*

*}*

* To initialize object, use *new* *ClassName(parameter1, parameter2, …);*

*This* keyword

* Used inside a method to refer to the current object
* Looks like *this.InstanceVariable*
* When local variables and instance variables are named identically, the instance variable is hidden; *this* bypasses this hiding

Garbage collection

* When no reference to an object exists, it goes into the garbage collection
* No need to delete the actual code

*Finalize()* method form

protected void finalize() {

//finalize code}

* Define actions to be taken before an object is added to garbage
* Only executes when an object is about to be added to garbage (else doesn’t)

Stacks

* An archetypal example of encapsulation
* Utilizes first-in, last-out ordering
* Controlled with *push* and *pop*
  + Push puts an item on top of the stack
  + Pop removes an item from the stack

CH7

**Overloading methods**

Java allows multiple methods in the same class to have the same name, as long as they take in different parameters. The methods are called overloaded in this case.

* Valid overloaded methods take in different type and/or number of parameters. Vararg vs regular parameters are also valid overloads.
* Java will choose which method to run based on the parameter(s) given in the call
* Java can automatically convert parameters to a different type if no method takes the original type and the types are compatible. E.g. *method(double x)* defined, *method(int x)* called, *method(int x)* not defined, Java will convert *x* to double in order to use *method(double x)*

**Overloading constructors**

Same as methods, but for constructors. Java will choose which constructor to run based on the parameters given when *new* executes.

Passing object as parameter to constructor

*ClassName(ClassName ob) {*

*//sets up object (with ob.Parameter) }*

Note: ob doesn’t need to exist at this time

**2 ways to pass argument to subroutine:**

1. Call-by-value: the value of the argument is given to the parameter of the subroutine. Changes to this parameter has no effect on the argument.
2. Call-by reference: the argument itself is given to the parameter of the subroutine. Changes to this parameter will reflect on the argument.

Passing primitives to a method uses pass-by-value; passing objects uses pass-by-reference. Example:

*class demo {*

*int a = 1;*

*void value-meth(int i) { //call by value*

*I \*= 2;*

*}*

*void ref-meth(demo o) { //call by reference*

*o.a \*= 2;*

*}*

*}*

* When a variable of a class type is created, it is only a reference to an object

**Methods can return any type, included ones defined by the user**

**Recursion**

* The process of defining something in terms of itself, i.e. allowing a method to call itself. such a method is recursive.
* Must have *if* somewhere in the method in order to stop the recursion at some point

Example: factorial

*class fac {*

*int factorial(int n) {*

*int result;*

*if(n==1) return 1;*

*result = factorial(n-1)\*n;*

*return result;*

*}*

*}*

**Access Control**

Access modifiers: **public, private, protected**

* Public: any code can access
* Private: only members of the class can access
  + Outside of its class, private code cannot be used directly; must be accessed using public methods involving them

example

*class priv {*

*private int c;*

*void set\_c(int i) {c = i;}*

*int get\_c() {return c;}*

*}*

*class demo {*

*public static void main(String args[]) {*

*priv p1 = new priv();*

*//p1.c = 10; //error*

*//System.out.println(p1.c); //error*

*p1.set\_c(10); //ok*

*System.out.println(p1.get\_c()); //ok*

*}*

*}*

* Default is public within the member’s package, but cannot be accessed outside of its package
* Access modifier precedes the rest of a member’s type declaration (e.g. *public int I; private int methodname(int a, double b)* )

**Understanding Static**

* Static members can be used without reference to any object
* Both method and variables can be static
  + *main()* is the most common static member (since it must be called before any object exist)
* All instances of a class share the same static variable. Static variables are not created when new objects of a class are created
* Restrictions to static methods:
  + Can only directly call other static methods
  + Can only call non-static methods through an object
  + Can only directly access static data
  + Cannot use *this* or *super*
* Outside of its class, static methods and variables can be used without any object, using *ClassName.method()* or *ClassName.variable*
* Static blocks can initialize static variables. It is executed exactly once, when the class is first loaded.

Example:

*class static\_demo {*

*static int a, b;*

*static {*

*a = 3;*

*b = a + 1;*

*}*

*static print\_a\_b() {*

*System.out.println(“a, b are “ + a + “ “ + b);*

*}*

*}*

*class demo {*

*public static void main(String args[]) {*

*static\_demo.a = 6;*

*static\_demo.print\_a\_b();*

*}*

*}*

**Final**

* Prevents a field from being modified
* Final fields must be initialized (given a value when declared, or used a constructor)
* Convention: final fields use ALL CAP names
* Final parameters cannot be changed within the method
* Final local variables can only be assigned values once
* Final methods ch-8

**Array again**

* Arrays are implemented as objects
* All arrays have attribute *length* which hold the size of the array
  + Reached by *ArrayName.length*

**Nested classes**

* Nested class has access to members of the class (or any block) in which it is nested (including private ones)
* Enclosing class does not have access to members of the nested class
* Static nested classes must access non-static methods of its enclosing class through an object (seldom used)
* Inner class – a nested class that is non-static
* Code outside the enclosing class cannot instantiate an inner class; instances of inner classes can only be created within the scope of the enclosing class
* Inner classes can be declared within any block

**Strings**

* Strings are objects of type *String*
  + Declare strings with *String StringName = “content of a ” + “string”;*
* String objects cannot be altered once created
* The only operator for Strings is *+.*
* Methods of the String class
  + *equals()* tests two strings for equality. Form:

*Boolean equals(StringB)*

* + *length()* obtains the length of a string. Form:

*int length()*

* + *charAt()* obtains the character at a specific index. Form:

*char charAt(index)*

* Strings can be in arrays. e.g.

*String str[] = {“one”, “two”, “three” };*

**Command Line Arguments**

* Given by the user when executing the program, stated directly following the name of the program
* Stored as Strings in the *args[]* array passed to *main()*

Variable-length Arguments – *varargs*

Introduced in JDK 5

*varargs* allows the number of arguments passed to a method to be variable.

* Specify variable-length argument with three periods (…). General form:

*return-type meth-name(arg-type … v)*

where *v* is an array with zero or more arguments

* + When calling *vaTest*, list the arguments plainly, e.g. *vaTest(1, 2, 3, 4);*

Example

*class demo {*

*static void var\_meth(int … v) {*

*System.out.println(“v contains “);*

*for(int x : v)*

*System.out.print(x + “ “);*

*System.out.println();*

*}*

*public static void main(String args[]) {*

*var\_meth(1);*

*var\_meth(1, 2, 3, 4);*

*var\_meth();*

*}*

*}*

* Methods can have normal parameters as well as variable-length parameters. Example:

*int var\_meth(int a, double b, int … v) {//…*

* + The variable-length parameter must be the last to be declared in the method
  + There can only be one variable-length parameter
  + Overloaded varargs methods can create ambiguity errors, e.g. when the parameter is empty, since empty can be any type

Pre-varargs solutions:

1. Create overloaded methods that take different numbers of arguments.
2. Put the arguments into an array and pass the array.

CH8

**Inheritance**

Inheritance allows hierarchical classifications, by defining a general class with common traits, and inheriting them to other classes. Such a class is called a **superclass**. A class that inherits a superclass is a **subclass**.

Creating a superclass: no special syntax. Creating a subclass:

*class subclass extends superclass {//…*

* There can only be one superclass for each subclass
* Subclasses have access to all *public* members of the superclass
  + These members can be referred to directly in the subclass
* Superclasses still function as independent classes; subclasses can also act as superclasses for other classes
* A reference variable of a superclass can be assigned a reference to any subclass of that superclass
  + The variable will only have access to members defined by the superclass

Example

*class superclass {*

*int a; int b;}*

*class subclass extends superclass {*

*int c;}*

*class demo {*

*public static void main(String args[]) {*

*subclass subob = new subclass();*

*superclass superob = new superclass();*

*subob.a = 1;*

*subob.b = 2;*

*superob = subob;*

*}*

*}*

* It is valid to assign a subclass object to a superclass object
* in this case superob will then have *superob.a = 1* and *superob.b = 2*, but *superob.c* is undefined

Keyword*Super*

A subclass can call its superclass’ constructor with

*super(args-list);*

* args-list corresponds to the list of parameters required by the constructor of the superclass
* *super()* must be the first statement inside the subclass’ constructor
* The superclass can then set its parameters as private

Example:

*subclass(int a, int b, int c) {*

*super(a, b);*

*k = c;*

*}*

* *super()* (if used) must be the first statement in a subclass’ constructor
* If using constructor that takes an object as parameter, the subclass’ constructor may pass an object of subclass type to super()
* Superclass’ constructors run first, then the subclass’

Second use for *super*

* Used as a prefix to refer to the superclass to the current class (similar to *this*), i.e.

*Super.member;*

* + *Super* always refers to the closest superclass
* Multiple variables can have the same name in the subclass and superclass, i.e.

*class superclass {*

*int i;}*

*class subclass extends superclass {*

*int i; //hides superclass.i*

*i = 1;*

*super.i = 2; //subclass.i and superclass.i are separate variables*

*}*

Multilevel Hierarchy

A subclass can be a superclass of another class, which can be a superclass of another class, etc. This chain of hierarchy can go on indefinitely.

*super()* always refers to the closest superclass.

When constructors are called

Constructors are called in the order of derivation, from superclass to subclass.

Method Overriding

* Overriding methods allow Java to support run time polymorphism. A general class can define a general method, and have its subclasses define specific implementations of that method.
  + Superclass and subclasses form a hierarchy of lesser to greater specialization.
* When a method in a subclass has the same name and type signature (parameter type) as a method in a superclass, the subclass method overrides the superclass method
  + If the type signatures are different, the methods are overloaded instead
* When called within the subclass, the method will always refer to the method defined in the subclass
* To call the superclass method in the subclass, use *super.method()*

Dynamic Method Dispatch

The type of object being referenced (not the type of the reference variable) determines which version of overridden method is run

* A superclass reference variable can refer to a subclass object
* Dynamic method dispatch resolves call to a method at run time rather than compile time

example

*class A {*

*void meth() {*

*System.out.println(‘A’);*

*}*

*}*

*class B extends A {*

*void meth() {*

*System.out.println(‘B’);*

*}*

*}*

*Class demo {*

*Public static void main(String args[]) {*

*A a = new A();*

*B b = new B();*

*A r; //r is a reference of type A*

*r = a; //r refers to an A object*

*r.meth(); //outputs “A”*

*r = b; //r refers to a B object*

*r.meth(); //outputs “B”*

*}*

*}*

Abstract Classes

Sometimes a method does not have a meaningful implementation in the superclass. Can either:

1. Report an error message
2. Use *abstract* modifier: an abstract method is always overridden

* An abstract method has no body

General form:

*abstract return-type MethodName(parameter-list);*

Any class that contains abstract methods must also be declared abstract. To declare abstract class:

a*bstract class Name { //…*

* There can be no object in an abstract class
* An abstract class can declare non-abstract (concrete) methods
* An abstract class cannot be instantiated with *new*
  + References to an abstract class can still be created (i.e. *AbstractClass A;*)
    - This means *A* can be used to refer to an object of any class derived from *AbstractClass*
* There are no abstract constructors or abstract static methods
  + (concrete constructors and static methods are fine)
* Subclass of an abstract class must either implement all abstract methods in the superclass, or be abstract itself

*final,* again

*final* has 3 uses:

1. To create a named constant (described previously)
2. To prevent overriding a method
   * Methods declared as final cannot be overridden
   * Specify *final* at the start of a method’s declaration—

*final return-type MethodName(parameter-list)*

* + Final methods allow *inlining* of the compiler, which improves performance
    - Call to final methods can be resolved at compile time rather than dynamically at run time (early binding)

1. To prevent inheritance
   * Precede class declaration with *final* to prevent a class from being inherited
   * Declaring a class final implicitly makes all of its methods *final* too
   * It’s impossible to declare a class both *final* and *abstract*

Object class

* Object is a superclass of all other classes
* A reference variable of type *Object* can refer to any type of object (including arrays)

Object methods

* *Object clone()* –creates an object that is the same as the cloned object
* *boolean equals(Object ob)* –determines if two objects are equal
* *void finalize()* –called before an unused object is recycled
* *class<?> getClass()* –obtains the class of an object at run time
* ­*int hashCode()*
* *void notify()*
* *void notifyAll()*
* *String toString()* –returns a string that describes the object (automatically called in *println()*)
* *void wait()*
* *getClass(), notify(), notifyAll(), wait()* are *final*

CH9

Packages and Interfaces

**Packages** compartmentalize the class name space, so that one can name classes without worrying about duplicating an already existing class name

Both a naming and visibility control mechanism

Creating a package:

*package PackageName;*

(this is the first statement of a java source file)

* any class defined within this file will belong to the package
* without this statement, the classes go into the default package, which has no name
* the .class files of any class within *PackageName* must be stored inside a folder named *PackageName*
* can define a hierarchy of packages: separate each package from the one above it with a period. Example:

*package java.awt.image*;

(this package needs to be stored in java\awt\image)

Telling a program where to find the packages—three ways:

1. execute java in the package’s directory
2. set *CLASSPATH* environmental variable
3. specify the path with the *-classpath* option with *java* and *javac*

The classpath must NOT include the target package itself, but rather the path to the package. For example, if the Windows address to a package is *C:\Java\TargetPackage*, then the classpath to T*argetPackage* is *C:\Java.*

Access protection

Packages act as containers for classes and subordinate packages; classes act as containers for code and data (classes are the smallest unit of abstraction)

Data by category and visibility interaction:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Category | Public | Private | Protected | No modifier |
| Same class | Yes | Yes | Yes | Yes |
| Subclasses in the same package | Yes | No | Yes | Yes |
| Non-subclasses in the same package | Yes | No | Yes | Yes |
| Subclasses in different packages | Yes | No | Yes | No |
| Non-subclasses in different packages | Yes | No | No | No |

Observation:

* Public is always visible
* Private is only visible within own class
* Default is visible within package
* Use protected if subclass access desired

This table **only applies to members of classes**. A non-nested class only has 2 access levels: **public** and **default**. Public makes the class always accessible; default makes the class only accessible to its package.

* When a class is public, it must be the only public class within its file, and the file name must match with the public class

Importing Packages

There are no packages in the default unnamed package; all built-in Java classes are in some named package. All standard Java classes are stored in the package *java*.

**Import** statements occur immediately after the package statement. Structure:

*Import pkg1.pkg2.(ClassName|\*);*

* *pkg1* is the top level package; *pkg2* is the subordinate package inside *pkg1*
* can either import an explicit class with *ClassName* or the entire package with \*
* there is no limit to the depth of the package hierarchy

Effects of import statement

* Importing a package allows its class to be used directly with its name, without specifying its package
* if there are multiple classes of the same name in different imported packages, one needs to specify the package (making it *fully-qualified*) when using that class, else an error occurs
* importing is optional, one can still use the class without importing its package
* java.lang is implicitly imported in all programs (java is useless without it)

Interfaces

Interfaces are similar to classes, but lacking instance variables and with methods with no body.

* Any number of classes can implement an interface; a class can implement any number of interfaces
* To implement an interface, a class must create the **complete** set of methods defined by the interface (but is free to determine the details)

General form:

a*ccess interface name{*

*ReturnType MethodName1(parameter-list);*

*ReturnType MethodName2(parameter-list);*

*Type FinalVarName1 = value;*

*Type FinalVarName2 = value;*

*}*

* The access modifiers work the same way as in classes
  + Interfaces can only be *public* or *[default]*
* All methods and variables are implicitly *public*
* The variables declared within interfaces are implicitly *static* and *final*

Implementing Interfaces

general form:

*class ClassName [extends Superclass] implements interface1 [,interface2] {//…*

* The methods that implement an interface must be *public* and match the return type with the interface definition
* If implementing more than one interface, the interfaces are comma separated
* The class can declare new members of its own

Interface references

* One can declare variables of the interface type (i.e. *interface1 A; )*
* Any instance of any class that implemented the declared interface can be referred to by such a variable. i.e.

*Inteface var = new Clas();*

* *Inteface* is the name of an interface and *Clas* is a class that implemented *Inteface*
* Variable *var* is declared with an interface type but assigned an instance of a class
* *var* only has knowledge to the methods declared by *Inteface*
* The version of method is determined by the type of object *var* refers to at run time. E.g.

*interface callback{*

*void callback(int a);*

*}*

*class client implements callback {*

*public void callback(int a) {*

*System.out.println(“first callback, ”, + a); }*

*}*

*class client2 implements callback {*

*public void callback(int a) {*

*System.out.println(“second callback, ”, + a); }*

*int call(int a) { //this function isn’t implemented by the interface*

*return a\*2; }*

*}*

*class demo {*

*public static void main(String args[]) {*

*callback Var = new client();*

*callback ob = new client2();*

*Var.callback(2); //this prints “first callback, 2”*

*Var = ob; //var is now a client2() object*

*Var.callback(2); //this now prints “second callback, 2”*

*Var.call(2); //error, objects of type callback have no knowledge of call()*

*}*

*}*

Partial Implementations

If a class uses an interface but does not fully implement all methods defined by the interface, the class must be declared *abstract.* Its subclasses must implement all methods defined by the interface or be abstract themselves.

Nested Interface

An interface can be a member of a class or interface (member interface / nested interface)

* A nested interface can be *public, private,* or *protected*
* When a nested interface is used outside of its enclosing scope, it must be qualified by the name of the class/interface that encloses it

Example:

*class A {*

*public interface NestedIF {*

*int meth(int x);*

*}*

*}*

*class B implements A.NestedIF {*

*public int meth(int x) {return x+1;}*

*}*

Intermission: *Random*

Random is one of the standard classes of java

It contains various methods for producing pseudorandom numbers: e.g. Random.nextDouble() returns a random double in 0.0. to 1.0 range

*Random rand = new Random();*

*rand.nextDouble();*

**Extending interfaces**

An interface can inherit another interface. The syntax is identical to inheriting classes:

*Interface inteface2 extends inteface1 {//…*

* A class that implements an interface that extends another interface must implement all methods defined in both interfaces

CH10

**Exception Handling**

An exception is an abnormal condition that arises at run time (i.e. a run time error).

Lingo: “exceptional condition” = error

* Exceptions can be generated by the Java run-time system, or manually by the code
  + Exceptions generated by Java relate to fundamental errors to the Java language and execution environment
  + Manually generated exceptions typically report some error conditions to the caller of a method
* Exception keywords: *try, catch, throw, throws, finally*
  + A *try* block contains program statements
  + If an error occurs within a try block, it is *thrown*
    - Manually throw with keyword *throw*
  + *catch* handles this exception
  + *finally* block contains code that must be executed after a try block completes
* When an exception is detected, Java runtime system constructs a new exception object and *throws* this exception. This stops the code’s execution
* General form:

*try {*

*//block of code to monitor errors }*

*catch(ExceptionType exOb) {*

*//exception handler for ExceptionType }*

*finally {*

*//block of code to execute after try block ends }*

**Exception Types**

All exception types are subclasses of the built-in class *Throwable*. Only instances of *Throwable* or its subclass are thrown by the JVM or can be thrown with a *throw* statement. Only *Throwable* or its subclass can be the argument type in a *catch* clause.

*Throwable* has two subclasses:

1. *Exception* – exceptions that the user program should catch
   * used when creating custom exception types
   * has the subclass *RuntimeException* – automatically defined exceptions
2. *Error* – exceptions that aren’t expected to be caught under normal circumstances by the program
   * Used to indicate errors with the runtime environment
   * Typically created in response to catastrophic errors that can’t be handled by the program

**Uncaught Exceptions**

When no exception handlers are provided, exceptions are caught by the default handler

The default handler displays a string that pinpoints where the exception occurred. E.g.

*Java.lang.ArithmeticException: / by zero*

*At exec.main (exec.java:4)*

* First line shows the type of exception
* Second line shows which method, what the file name, and which line caused the exception

**Manually catching exceptions**

Enclose the code to monitor inside a *try* block; immediately following, include a *catch* clause that specifies the type of error to catch.

A catch block’s scope is the try block immediately preceding it.

When an exception is found in the try block, the code immediately jumps to the catch block, without executing the rest of the try block. Once the catch block has executed, the program returns to following the try/catch blocks. E.g.

c*lass demo{*

*public static void main(String args[]) {*

*try {*

*int d = 0;*

*int a = 4 / d; //jumps to catch block, bypassing remaining code in try block*

*System.out.println(“this will not be displayed”);*

*}*

*catch (ArithmeticException e) {*

*System.out.println(“attempted division by zero”);*

*}*

*System.out.println(“after the catch block”);*

*}*

*}*

Displaying a description of an exception by passing the exception as an argument to a *println()* statement. i.e.

*System.out.println(“Exception: ” + e);*

Displays: Exception: java.lang.ArithmeticException: / by zero

Multiple catch blocks can be setup for a single try block.

* The catch blocks would be inspected in order. The first time a catch block catches an exception and executes, the rest of the try/catch block is bypassed.
* When multiple catch statements are used, exception subclasses **must come before any of their superclasses**, else would create an unreachable code error. Example:

*try {*

*//try block*

*}*

*catch (Exception e) {*

*//generic exception catch*

*}*

*catch (ArithmeticException e) {*

*/\* this cannot be reached as the prior catch block would catch any ArithmeticException exceptions (ArithmeticException is a subclass of Exception) \*/*

*}*

Nested try blocks

* When an exception is found in a nested try block, Java first inspects the nearest catch block(s), then the enclosing catch block(s)
* If the inner try block does not have a matching catch handler, the exception is passed to the outer try block’s catch handler
* Nested try blocks can also occur if the try block is within a method and another block calls it. Then the method block is the inner block and the calling block is the outer block. .e.g.

*static void TryMethod(int a) {*

*try {//implement method }*

*}*

*public static void main(String args[]) {*

*try{*

*int a = args.length;*

*TryMethod(a);*

*} catch(ArithmeticException e) {*

*//implement catch}*

*}*

*throw* – allows the program to throw an exception explicitly. Statement form:

t*hrow ThrowableInstance;*

*ThrowableInstance* must be an object of type *Throwable* or subclass of *Throwable*.

* Primitive types or non-throwable types (such as String and Object) cannot be used as exceptions
* To obtain a Throwable object:
  1. Use a parameter in a catch clause
  2. Create one with the *new*operator
     + Many of java’s built-in runtime exceptions have at least two constructors: one with no parameter and one that takes a string parameter
     + In the second case, the argument specifies a string that describes the exception. It will be displayed when the object is called in *print()* or *println()*

Throwimmediately stops the flow of execution, bypassing the subsequent statements. The nearest try block is then inspected.

Example of the two ways to throw:

s*tatic void demoproc() {*

*try {*

*throw new NullPointerException(“demo”);*

*System.out.println(“this won’t be executed”);*

*} catch (NullPointerException e) {*

*System.out.println(“Caught exception.”);*

*throw e; //rethrow the exception*

*}*

*}*

*public static void main(String args[]) {*

*try {*

*demoproc();*

*} catch (NullPointerException e) {*

*System.out.println(“Caught again: ” + e);*

*//outputs “Caught again: java.lang.NullPointerException: demo”*

*}*

*}*

*throws* – if a method can cause an exception that it does not handle, it must specify this behavior with a *throws* clause in its declaration, i.e.

*Type MethodName(Parameter-list) throws exception-list {*

*//body of method }*

* The throws clause lists all types of exceptions the method might throw, except those of type *Error* or *RuntimeException* or their subclasses.
* The exceptions are comma-separated
* Inside the method, the exceptions are thrown in *throw* clauses
* The block that calls this method would need to setup a try/catch block
* Failure to include exception(s) in the *throws* clause will cause the method to not compile

*finally*– creates a block to be executed after a try/catch block has completed and before the code following the try/catch block

* Will execute whether or not an exception is thrown
* Useful when some code needs to be executed even if exceptions are found before it (e.g. closing a file)
* Each *try* statement requires at least one *catch* or *finally* clause

**Exceptions built into java.lang**

Most general built-in exceptions are subclasses of *RuntimeException*.

* These exceptions need not be included in any method’s *throws* list.
* They are called **unchecked exceptions** as the compiler does not check to see if a method handles or throws these exceptions.

List:

|  |  |
| --- | --- |
| **Exception** | **Explanation** |
| *ArithmeticException* |  |
| *ArrayIndexOutOfBountsException* |  |
| *ArrayStoreException* | Assignment to an array element of an incompatible type. |
| *ClassCastException* | Invalid cast. |
| *EnumConstantNotPresentException* | Attempt to use undefined enumeration value. |
| *IllegalArgumentException* | Illegal argument when invoking a method. |
| *IllegalMonitorStateException* | Illegal monitor operation. |
| *IllegalStateException* | Environment or application in incorrect state. |
| *IllegalThreadStateException* |  |
| *IndexOutofBountsException* |  |
| *NegativeArraySizeException* |  |
| *NullPointerException* | Invalid use of a null reference. |
| *NumberFormatException* | Invalid conversion of a string to a numeric format. |
| *SecurityException* |  |
| *StringIndexOutOfBounds* |  |
| *TypeNotPresentException* |  |
| *UnsupportedOperationException* |  |

The following exceptions defined in *java.lang* are **checked.**

* They must be included in the *throws* list of a method if that method can generate them and does not handle them itself.

List:

|  |  |
| --- | --- |
| **Exception** | **Explanation** |
| *ClassNotFoundException* |  |
| *CloneNotSupportedException* | Attempt to clone an object that does not implement the *Cloneable*interface. |
| *IllegalAccessException* | Access denied. |
| *InstantiationException* | Attempt to create an object of an abstract class or interface. |
| *InterruptedException* | One thread has been interrupted by another thread. |
| *NoSuchFieldException* |  |
| *NoSuchMethodException* |  |
| *ReflectiveOperationException* | Superclass of reflection-related exceptions |

**Creating exception subclasses**

Simply define a subclass of *Exception*(which is a subclass of *Throwable*)

* The subclass doesn’t need to actually implement anything; its existence allows them to be used as exception types.
* *Exception*class does not define any method, but *Throwable*does (so *Exception* inherits them)

List of methods:

|  |  |
| --- | --- |
| **Method** | **Description** |
| *Final void addSuppressed(Throwable exc)* | Adds *exc* to the list of suppressed exceptions associated with the invoking exception. (Suppressed exceptions are primarily generated by the *try-with-resources* statement) |
| *Throwable fillInStackTrace()* | Returns a *Throwable* object that contains a completed stack trace. |
| *Throwable getCause()* | (Chained exception) Returns the exception that underlies the current exception. *null* is returned if there is no underlying exception. |
| *String getLocalizedMessage()* | Returns a localized description of the exception. |
| *String getMessage()* | Returns a description of the exception. |
| *StacktraceElement[] getStackTrace()* | Returns an array that contains the stack trace, one element at a time, as an array of *StackTraceELement*. |
| *Final Throwable[] getSuppressed()* | Obtains the suppressed exceptions associated with the invoking exception and returns an array that contains the result. |
| *Throwable initCause(Throwable causeExc)* | (Chained exception) Associates *causeExc* with the invoking exception as a cause of the invoking exception. Returns a reference to the exception. The cause exception can only be set once per exception, either by *Initcause()* or by a constructor. |
| *Void printStackTrace()* |  |
| *Void printStackTrace(PrintStream/PrintWriter stream)* | Sends the stack trace to the specified stream. |
| *Void setStackTrace(StackTraceElement elements[])* | Sets the stack trace to the element passed in *elements*. (not for general use) |
| *String toString()* | Returns a String object containing the name of the exception followed by a description of the exception. (Called by *println()* when outputting a *Throwable* object) |

*Exception* has four constructors:

* *Exception()*
* *Exception(String msg)*

creates an exception that with or without description.

Using this constructor results in displaying “name of the exception: description”; one can override the *toString()* method to make it return something else. Example:

*class MyException extends Exception {*

*private int detail;*

*MyException(int a) {*

*detail = a; }*

*public String toString() { //overrides toString() to display the value of detail*

*return “MyException[” + detail + “]”;*

*}*

*}*

*Class ExceptionDemo {*

*static void compute(int a) throws MyException {*

*System.out.println(“Called compute(“ + a + “)”);*

*if(a > 10)*

*throw new MyException(a);*

*System.out.println(“normal exit”);*

*}*

*public static void main(String args[]) {*

*try {*

*compute(1); //legal value, does not get caught*

*compute(20);*

*} catch (MyException e) {*

*System.out.println(“Caught “ + e); //returns “Caught MyException[20]”*

*}*

*}*

*}*

The other two constructors have to do with **Chained Exceptions** – associating another exception with an exception (the second exception describes the cause of the first). The chain of exceptions can go on indefinitely.

These constructors are:

* *Exception(Throwable causeExc)*
* *Exception(String msg, Throwable causeExc)*

where *causeExc* is the exception that causes the current exception.

These constructors also work with the *Error*, *Exception*and*RuntimeException*classes.

Example of chained exceptions

*class demo {*

*public static void main(String args[]) {*

*try {*

*NullPointerException e = new NullPointerException(“e descriptor”); //create exception e*

*e.initCause(new ArithmeticException(“cause descriptor”));*

*throw e;*

*} catch(NullPointerException e) {*

*System.out.println(“Caught: “ + e);*

*//displays “Caught: java.lang.NullPointerException: e descriptor”*

*System.out.println(“Original cause caught: “ + e.getCause());*

*//displays “Original cause caught: java.lang.ArithmeticException: cause descriptor*

*}*

*}*

*}*

**JDK 7 New Exception Features**

1. *try-with-resources*: automates the process of releasing a resource that is no longer needed (ch13)
2. *multi*-*catch:* allows more than one exception (of any type) to be caught by the same catch clause

To use multi-catch, separate each exception type in the catch clause with the *OR* operator |.

* Each parameter is implicitly *final*

example:

*class multicatch {*

*public static void main(String args[]) {*

*int a = 10, b = 0;*

*int vals[] = {1, 2, 3 };*

*try {*

*int result = a/b;*

*vals[10] = 10;*

*} catch(ArithmeticException | ArrayIndexOutOfBoundsException e) {*

*System.out.println(“Caught: “ + e);*

*}*

*}*

*}*

1. *final rethrow* or *more precise rethrow*: restricts the type of exceptions that can be rethrown to only those checked exceptions that the associated *try* block throws, that are not handled by a preceding *catch* clause, and that are a subtype or supertype of the parameter.

* The catch parameter must be effectively *final* (meaning it must not be assigned a new value inside the catch block), or explicitly *final*

CH11

**Multithreaded Programming**

Java has built-in multithreaded programming support.

A multithreaded program has multiple parts (**threads**) that can run concurrently, with separate paths of execution.

There are two types of multitasking: **process-based** and **thread-based**.

Process-based: the program (process) is the smallest unit of dispatchable code

* Individual processes are heavy weight; switching between processes also costly

Thread-based: the thread is the smallest unit of dispatchable code, meaning a single program can perform multiple tasks simultaneously

* Threads collectively share a heavyweight process

Java maximizes processing power efficiency by minimizing idle time.

* In single thread processing often CPU processing power cannot be fully utilized because another part of the system is slower: e.g. disk read/write speed, network speed, user input

Single threaded operation utilizes **event loop with polling** – a single thread of control runs an infinite loop, once polling mechanism returns a signal for what to do next, the event loop dispatches control to the appropriate handler, then wait until the handler returns. Nothing can happen in the program in the meantime. If a thread blocks, the entire program stops working.

Thread states:

1. Running
2. Ready to run (once it gets CPU time)
3. Suspended
4. Resumed (from suspension)
5. Blocked (waiting for resource)
6. Terminated (cannot be resumed)

Thread priorities – integers used to decide when to switch from one thread to the next (a *context switch*). Rules:

1. A thread can be preempted by a higher priority thread
2. A thread can voluntarily relinquish control
3. In the case of equal priority threads, sometimes the system splits processing time, sometimes one thread must voluntarily give up control in order for them to run

Synchronization – with the **monitor**

* A box that can hold only one thread. Once a thread enters a monitor, all other threads must wait until that thread exits the monitor
* A monitor protects a shared asset from being manipulated by more than a thread at a time
* In Java, each object has its own implicit monitor that is entered by calling the object’s synchronized methods (this prevents other objects from calling any of this object’s synchronized methods)

Messaging – how threads communicate with each other – via calls to predefined methods that all objects have

**The *Thread* class and *Runnable* interface**

To create a new thread, the program can either extend *Thread* or implement the *Runnable* interface

Some of *Thread* class’ methods:

|  |  |
| --- | --- |
| **Method** | **Meaning** |
| *final String getName()* | Obtains a thread’s name. |
| *final void setName(String ThreadName)* |  |
| *final int getPriority()* | Obtains a thread’s priority. |
| *final void setPriority(int priority)* |  |
| *final boolean isAlive()* | Determines whether a thread is running. |
| *final void join() throws InterruptedException* | Waits for a thread to terminate. Can give parameter (in ms) to specify wait time. |
| *void run()* | Entry point for thread. |
| *static void sleep(long milliseconds)* | Suspends a thread for *milliseconds* ms. |
| *void start()* | Starts a thread by calling its run method. |
| *static Thread currentThread()* |  |
| *static int enumerate(Thread threads[])* | Puts copies of all *Thread* objects in the current thread’s group into *threads* and returns the number of threads. |
| *static void yield()* |  |
| *Thread.State getState()* |  |
| *boolean isInterrupted()* |  |

Thread constructors:

* *Thread()*
* *Thread(Runnable threadOb)*
* *Thread(Runnable threadOb, String threadName)*
* *Thread(String threadName)*
* *Thread(ThreadGroup groupOb, Runnable threadOb)*
* *Thread(ThreadGroup groupOb, Runnable threadOb, String threadName)*
* *Thread(ThreadGroup groupOb, String threadName)*

where *threadOb* is an instance of a class that implements *Runnable* and defines where execution of the thread begins. If thread name is not provided, the JVM will create one. *groupOb* specifies which thread group the new thread will belong.

**The Main Thread**

The one thread that begins running as soon as a Java program starts

* It is the thread from which other threads will be spawned
* It often is the last thread to finish execution because it performs various shutdown actions

The main thread can be controlled with a Thread object. Obtain reference to one with the *currentThread()* method, a public static member of *Thread* that returns a reference to the thread in which it is called.

When a thread object is used as argument to *println()*, it displays: the name of the thread, its priority (default is 5), the name of its group. (A thread group is a data structure that controls the state of a collection of threads.) Example:

*class threadDemo {*

*public static void main(String args[]) {*

*Thread t = Thread.currentThread();*

*System.out.println(t); //displays “Thread[main, 5, main]”*

*t.setName(“New Thread”);*

*System.out.println(t); //displays “Thread[New Thread, 5, main]”*

*try{*

*Thread.sleep(1000); }*

*catch (InterruptedException e) {*

*System.out.println(“Main thread interrupted”);*

*}*

*}*

*}*

*sleep(long)*causes the thread to suspend for the specified amount of time in milliseconds. This method may throw an *InterruptedException*.

**Creating a Thread** – to do so, create a class that:

* **Implements *Runnable* interface**

To implement *Runnable*, a class need only implement a single method *run()*, declared as

*public void run()*

Inside *run()*, define the code that constitutes the new thread. Thread ends when *run()* returns.

Then instantiate an object of type *Thread* from within the class, using *Thread* constructor

­*Thread(Runnable threadOb, String “ThreadName”)*

where *threadOb* is a class that implements *Runnable*, this is where the thread begins execution; the name of the thread given by *ThreadName*.

The new thread will only start running when its *start()* method is called ( *void start()* ), declared within *Thread*. (*start()* executes a call to *run()*)

Example

*class NewThread implements Runnable {*

*Thread t;*

*NewThread() {*

*t = new Thread(this, “Demo Thread”); //keyword “this”*

*System.out.println(“Second thread: “ + t);*

*t.start(); //this runs run()*

*}*

*public void run() {*

*try {*

*for(int j = 5; j > 0; j--) { //prints j every 500ms*

*System.out.println(“Second Thread: “ + j);*

*Thread.sleep(500) ;*

*}*

*} catch (InterruptedException e) {*

*System.out.println(“Thread interrupted.”);*

*}*

*System.out.println(“Exiting thread”);*

*}*

*}*

*class demo {*

*public static void main(String args[]) {*

*new NewThread(); //creates Demo thread, also starts for loop in run()*

*try {*

*for(int j = 5; j > 0; j--) {*

*System.out.println(“Main thread: “ + j);*

*Thread.sleep(1000); //this for loop operates twice as slowly as the other one*

*}*

*} catch (InterruptedException e) {*

*System.out.println(“Main Thread interrupted.”);*

*}*

*System.out.println(“Exiting main thread”);*

*}*

*}*

In this example, the 2 for loops exist in separate threads with the same priority, thus they alternate between execution in a single core system. The *main()* loop executes half as frequently as the *run()* loop.

Note: in some (especially old) programs, the main thread often must be the last thread to finish running.

* **Extends *Thread* class**

The extending class must override the *run()* method, and call *start()* to begin execution of the new thread. Example:

*class NewThread extends Thread {*

*NewThread() {*

*super(“Demo”); //invokes the Thread(String threadName) constructor*

*start();*

*}*

*public void run() {*

*//implement run() }*

*}*

*class ExtendThread {*

*public static void main(String args[]) {*

*new NewThread(); //creates a new thread*

*}*

**Creating Multiple Threads**

To create multiple threads, simply instantiate the class that implemented Runnable / extended Thread multiple times, e.g.

*class NewThread implements Runnable {*

*//implement class }*

*class Demo {*

*public static void main(String args[]) {*

*new NewThread(“one”);*

*new NewThread(“two”);*

*}*

*}*

**Using *isAlive()* and *join()***

*isAlive()* determines whether a thread has finished. General form:

*final boolean isAlive()*

*isAlive()* returns true if the thread is still running, false if it has finished running.

*join()* is more commonly used; it waits until the thread on which it is called terminates. The user can also specify a maximum amount of wait time. General form:

*final void join() throws InterruptedException*

Example showing *isAlive()* and *join()*:

*class NewThread implements Runnable {*

*//implement class }*

*class demo {*

*public static void main(String args[]) {*

*NewThread ob1 = new NewThread(“one”);*

*NewThread ob2 = new NewThread(“two”);*

*NewThread ob3 = new NewThread(“three”);*

*System.out.println(“Thread one is alive: “ + ob1.t.isAlive());*

*System.out.println(“Thread two is alive: “ + ob2.t.isAlive());*

*System.out.println(“Thread three is alive: “ + ob3.t.isAlive()); //all three return “true”*

*try {*

*ob1.t.join();*

*ob2.t.join();*

*ob3.t.join(); //returns when ob3 stops executing*

*} catch(InterruptedException e) {*

*System.out.println(“Main thread interrupted”);*

*}*

*System.out.println(“Thread one is alive: “ + ob1.t.isAlive());*

*System.out.println(“Thread two is alive: “ + ob2.t.isAlive());*

*System.out.println(“Thread three is alive: “ + ob3.t.isAlive()); //all three return “false”*

*System.out.println(“Main thread exiting);*

*}*

*}*

**Thread Priorities**

*setPriority()* (a member of *Thread*) sets a threads priority. General form:

*final void setPriority(int level);*

*level* must be within the range of *MIN\_PRIORITY* and ­*MAX\_PRIORITY* (1 to 10). Using *NORM\_PRIORITY* returns a thread to default priority (5).

Priorities are *static final* variables within *Thread*.

*getPriority()* obtains the current priority setting. General form:

*final int getPriority()*

It is good practice to make threads yield control once in a while rather than rely on preemptive behavior.

**Synchronization**

When multiple threads share a resource, synchronization ensures the resource is used by only one thread at a time.

Only one thread can **own** a monitor at a time. When a thread acquires a monitor, they are said to have **entered** the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread *exits* (they are **waiting** for the monitor).

There are two ways to synchronize, both involving the *synchronized* keyword:

* Using synchronized methods
* Using synchronized statements

**Synchronized Methods**

* All objects have implicit monitors associated with them
* To enter an object’s monitor, call a method that has been modified with the *synchronized* keyword
* While a thread is inside a synchronized method, all other threads trying to call any *synchronized* method on the same instance have to wait
* Returning the *synchronized* method exits the monitor
* Multiple threads calling the same method on the same object at the same time is called a *race condition* (the threads race to finish completing the method first). To avoid this, one needs to *serialize* access to the method
* Synchronized method syntax

*synchronized type MethodName(parameter-list) {//body}*

**Synchronized Statement**

If synchronizing methods is not possible (for example because one does not have access to the source code of the method), the alternative is to put calls to the methods inside a *synchronized* block.

Syntax:

*Synchronized(obj) {*

*//statements to be synchronized }*

A synchronized block ensures calls to a method that is a member of *obj* only occurs after the current thread has entered *obj*’s monitor.

The following are equivalent:

*synchronized type MethodName(parameter-list) {*

*//body}*

*obj.Methodname(param-list);*

and

*type MethodName(parameter-list) {*

*//body}*

*synchronized(obj) {*

*obj.Methodname(param-list);*

*}*

**Interthread Communication**

Polling – a loop that repeatedly checks some condition and take action when condition is met. Wastes CPU time

To avoid polling, Java uses the *wait(), notify(), notifyAll()* methods.

* They are implemented as *final* in *Object*
* They can only be called in a *synchronized* context
* *wait()* tells the calling thread to give up the monitor and sleep until some other thread enters the monitor and calls *notify()*. Can also specify wait period.
  + Remote possibility of waiting threads waking up for no apparent reason. Recommended to *wait()* in a loop.
* *notify()* wakes up a thread that is sleeping after calling *wait()* on the same object
* *notifyAll()* wakes up all threads that called *wait()* on the same object
* forms:

*final void wait() throws InterruptedException;*

*final void notify();*

*final void notifyAll();*

Example — *get()* and *put()* alternates between operation. *put()* produces a number, then *get()* displays that number:

*class Q {*

*int n;*

*boolean valueSet = false;*

*synchronized int get(); {*

*while(!valueSet)*

*try {*

*wait(); //sleeps get() if valueSet is false*

*} catch(InterruptedException e) {*

*System.out.println(“Interruption caught.”);*

*}*

*System.out.println(“Got: ” + n);*

*valueSet = false; //prevents get() from running again*

*notify();*

*return n;*

*}*

*synchronized void put(int n); {*

*while(valueSet)*

*try {*

*wait(); //sleeps put() if valueSet is true*

*} catch(InterruptedException e) {*

*System.out.println(“Interruption caught.”);*

*}*

*this.n = n;*

*valueSet = true; //prevents put() from running again*

*System.out.println(“Put: ” + n);*

*notify();*

*}*

*}*

If the *wait()* and *notify()* methods were not in *put()* and *get()*, either might run multiple times in a row, resulting in *get()* displaying the same number multiple times or skipping numbers.

**Deadlock**

Deadlock is a special type of error where 2 (or more) monitors block each other – two objects enter the monitors X and Y; X tries to access a synchronized method in Y and is blocked; Y tries to access a synchronized method in X and is blocked. Both wait for the other to leave their monitor indefinitely.

Difficult to debug:

1. Rare and requires timing
2. Can involve multiple objects and monitors.

Only solution is to quit the program with ctrl-c.

**Suspending, Stopping, Resuming Threads**

Java 1.1 and earlier – using *suspend(), resume()* and *stop()*:

*final void suspend()*

*final void resume()*

*final void stop()*

These methods are deprecated by Java 2 and no longer work.

Modern Java – different

reason: *suspend()* can cause system failure if the suspended thread does not relinquish resources; *stop()* can corrupt resources if a thread is stopped while modifying a resource(*stop()* causes a thread to release resources).

Instead: the *run()* method must periodically check whether to continue, suspend, resume, or stop execution of a thread (typically done by establishing a flag variable to indicate what *run()* should do)

Example:

*class NewThread implements Runnable {*

*String name;*

*Thread t;*

*boolean suspendFlag;*

*NewThread(String ThreadName) {*

*name = Threadname;*

*t = new Thread(this, name);*

*System.out.println(“created thread: “ + t);*

*suspendFlag = false;*

*t.start();*

*}*

*public void run() {*

*try {*

*for(int j = 0; j < 15; j++) {*

*System.out.println(“tick: “ + j);*

*Thread.sleep(200);*

*synchronized(this) {*

*while(suspendFlag) {*

*wait();*

*}*

*}*

*}*

*} catch(InterruptedException e) {*

*System.out.println(“Thread interrupted.”);*

*}*

*System.out.println(“Thread exiting.”);*

*}*

*synchronized void SuspendThread() {*

*suspendFlag = true;*

*}*

*synchronized void ResumeThread() {*

*suspendFlag = false;*

*notify();*

*}*

*}*

**Obtaining a Thread’s State**

*getState()* method determines the state of a thread. Form:

*Thread.State getState() //return type is Thread.State*

*Thread.State* values:

|  |  |
| --- | --- |
| Value | Explanation |
| NEW | The thread had not begun execution. |
| BLOCKED | The thread has suspended execution while waiting to acquire a lock. |
| RUNNABLE | The thread is either currently running or will run once it gets CPU access. |
| TERMINATED |  |
| TIMED\_WAITING | The thread is suspended for a specified amount of time. Also entered when a timed version of *wait()* or *join()* is called. |
| WAITING | The thread is suspended while waiting for something to occur. Also entered when a non-timed version of *wait()* or *join()* is called. |

CH12

**Enumerations** – list of named constants

Only after JDK 5

In Java, enumerations are implemented as a class, and thus have methods, constructors, instance variables, interfaces, etc.

**Basics**

The *enum* keyword creates an enumeration. Example:

*enum Apple {*

*Cortland, Jonathan, GoldenDel, Gala }*

The members of an enumeration are implicitly *public, static, final*. Their type is the name of enumeration, i.e. Apple (they are “self-typed”).

To initiate an enumeration, declare them like primitive types, rather than with *new*. E.g.

*Apple AppleInstance;*

the only values that *AppleInstance* can take are the ones defined in *Apple*. To assign it the value Gala, use

*AppleInstance = Apple.Gala;*

Instances of enumeration and enumeration constants can be compared using the equality (==) operator and used in *switch* expressions. E.g.

*switch(AppleInstance) {*

*case(Cortland): //note: no need to specify type*

*//…*

*case(Jonathan):*

*//…*

*System.out.println(Apple.Cortland);* displays “Cortland”.

Methods – all enumerations automatically contains two predefined methods: *values()* and *valueOf()*:

*public static enum-type[] values();*

*public static enum-type valueOf(String Str);*

*values()* returns an array containing the enumeration constants. *valueOf()* returns the enumeration constant corresponding to *Str*. Example:

*Apple AllApples[] = Apple.values();*

*for(Apple a: AllApples) //for(Apple a: Apple.values()) also works*

*System.out.print(a + “ “);*

*CL = Apple.valueOf(“Cortland”);*

Enumerations are class types – each enumeration constant is an object of its enumeration type.

When a constructor for an enumeration is created, it is used on every enumeration constant. Example:

*enum Apple {*

*Cortland(15), Jonathan(20), GoldenDel(20), Gala(10);*

*private int price;*

*Apple(int p) {price = p; }*

*int getPrice() {return price; } //called with e.g. Apple.Jonathan.getPrice();*

*}*

Enumerations as classes have two restrictions:

1. Enumerations cannot inherit another class (except 1)
2. Enumerations cannot be a superclass

Enumerations automatically inherit one class upon creation: *java.lang.Enum*, which has several methods.

* An enumeration constant’s position in the list of constants, its *ordinal value*, is retrieved with the *ordinal()* method:

*final int ordinal();*

* + Ordinal values begin with zero
* *compareTo(enum-type e)* compares two enumeration constants’ ordinal values:

*final int compareTo(enum-type e);*

* + The invoking constant and the parameter must be from the same enumeration.
  + If the invoking constant has a lower ordinal value than the parameter, *compareTo()* returns negative; if the invoking constant has a higher ordinal value than the parameter, *compareTo()* returns positive; if the two are equal, *compareTo()* returns zero.
* *equals()* overrides *Object.equals().* It returns true only if two objects are the same constant from the same enumeration.

**Type Wrappers**

Primitive types are not objects and do not have object-specific functions. For example, one can’t pass a primitive type by reference to a method. Type wrappers are classes that encapsulate a primitive type within an object, integrating primitive types to the object hierarchy.

**Character** – wrapper around a *char*. Its constructor is

*Character(char ch)*

where *ch* is the char being wrapped by the Character object.

*char charValue()*

displays the *char* value contained in a *Character* object.

**Boolean** – wrapper around a boolean. Has 2 constructors:

*Boolean(boolean booValue)*

*Boolean(String booString)*

where *booValue* can be true or false; if *booString* contains the string “true” (upper or lower case) then the Boolean will be true, else it will be false.

*boolean booleanValue()*

returns the boolean value equivalent to the invoking object.

**Numeric type wrappers – Byte, Short, Integer, Long, Float, Double** (capital!) **– inherit the abstract class *Number*. It has the constructors**

*Integer(int num)*

*Integer(String str) throws NumberFormatException*

And likewise for all other types.

**Edit:** constructors deprecated by JDK 9. Now recommended to use *valueOf()* to convert a primitive type to a wrapper object:

*static Character valueOf(char ch)*

*static Boolean valueOf(boolean bool) //or String parameter*

*static Integer valueOf(int val)*

*static Integer valueOf(String valStr) throws NumberFormatException*

*//etc.*

example:

*Integer iOb = Integer.valueOf(10);*

*Number* has the methods

*byte byteValue()*

*short shortValue()*

*int intValue()*

*long longValue()*

*float floatValue()*

*double doubleValue()*

All type wrappers override *toString()*. Example:

*Integer intOb = new Integer(100);*

*int i = intOb.intValue();*

*System.out.println(intOb); //displays “100”*

*System.out.println(i) //displays “100”*

The process of wrapping a primitive type inside a wrapper is called **boxing**. The process of extracting a value from a type wrapper is called **unboxing**.

**Autoboxing**/**auto-unboxing – added in JDK 5**

Autoboxing automatically encapsulates a primitive type in a type wrapper whenever an object of that type is needed; auto-unboxing automatically extracts the value from a type wrapper whenever its value is needed (no need to call *intValue()* etc.)

To autobox, simply assign a primitive type to a type wrapper reference, e.g.

*Integer intOb = 100;*

To auto-unbox, assign a type wrapper object to a primitive type, e.g.

*int i = intOb;*

In addition, autoboxing and auto-unboxing automatically occurs whenever an object/primitive type is needed. Example:

*class Demo{*

*public static void main(String args[]) {*

*static int unbox(Integer v) {*

*return v; }*

*Integer intOb = unbox(100);*

*//100 is autoboxed; then v is auto-unboxed; then unbox(100) is autoboxed;*

*System.out.println(intOb);*

*}*

*}*

This applies to expressions. Example:

*class Demo{*

*public static void main(String args[]) {*

*Integer iOb, iOb2;*

*int i;*

*Double dOb = 50.2;*

*iOb = 10;*

*iOb++; //iOb is unboxed, the expression is evaluated, then the result is reboxed*

*iOb2 = (iOb + 10)/2; //iOb is unboxed for the expression, then the result is reboxed*

*i = iOb + iOb2;*

*//iOb and iOb2 are unboxed for the expression, but the result is not reboxed*

*dOb = dOb + iOb;*

*//autoboxing allows the user to mix different numeric types in expressions*

*//auto-unboxing allows Integer objects to control switch statements*

*Integer iOb3 = 1;*

*switch(iOb3) { //iOb3 auto-unboxes when the switch statement executes*

*case 1: //…*

*break;*

*case 2: //…*

*break;*

*default: //..*

*}*

*}*

Characters and Booleans can also be autoboxed/auto-unboxed. Example:

*class Demo{*

*public static void main(String args[]) {*

*Boolean b = true;*

*if(b) System.out.println(“b is true”) //auto-unboxes b*

*char ch = “content”;*

*Character chOb = “old content”; //autoboxes “old content”*

*chOb = ch; //autoboxes ch*

*}*

*}*

Note: autoboxing and auto-unboxing adds overhead and degrades performance. They should only be used when an object/primitive type is required. They are not supposed to eliminate primitive types.

**Annotations/Metadata**

Annotations are added in JDK 5 and allow users to embed supplemental information into a source file. They do not change the actions of a program.

Declaring annotations – annotation creations are based on the interfaces. To declare an annotation, use *@interface.* Example:

*@interface Anno {*

*String strMeth();*

*int intMeth();*

*}*

* Like interfaces, annotations consist solely of method declarations. Also like interfaces, there are no bodies to these methods.
* Annotations cannot have an *extends* clause. However, all annotations types extend the *Annotation* interface automatically.
* *Annotation* overrides the *hashcode()*, *equals()*, *toString()* methods, and defines *annotationType()*, which returns a class object representing the invoking annotation.

Once declared, annotations can be used to annotate a declaration. This can be any declaration: classes, objects, methods, *enum* constants, annotations, etc. The annotation always precedes the rest of the declaration.

Applying an annotation – syntax:

*@AnnotationName(//initialize members) //only members’ names are used, no ()*

*//declaration*

Example – *Anno* being applied to a method declaration:

*@Anno(strMeth = “initial string”, intMeth = 10)*

*public void AnnoMethod() {//… }*

**Retention Policy**

A retention policy determines when an annotation is discarded. Java defines 3 policies, encapsulated within the *java.lang.annotation.RetentionPolicy* enumeration:

1. *SOURCE* – an annotation with retention policy *SOURCE* is retained only in source file and discarded during compilation
2. *CLASS* – an annotation with retention policy *CLASS* is stored in the *.class* file during compilation but not available in the JVM at run time
3. *RUNTIME* – an annotation with retention policy *RUNTIME* is stored in the *.class* file and available in the JVM

A retention policy is specified for an annotation using the built-in annotation *@Retention*, with general form

*@Retention(retention-policy) //default to CLASS if nothing is specified*

Example:

*@Retention(RetentionPolicy.RUNTIME)*

*@interface Anno {*

*String str();*

*int val();*

*}*

Obtaining annotations at run time

If an annotation’s retention policy is *RUNTIME*, then it can be queried at run time through *reflection*, which allows information about a class to be obtained at run time.

*reflection* is contained in the *java.lang.reflect* package.

To use *reflection*, first need a class object that represents the class whose annotation is wanted. One of the ways to do so is with *getClass()*:

*final class<?> getClass()*

Using class object methods to obtain information

* *getMethod()*, *getField()*, *getConstructor()* return information about a method/field/constructor. The return types are *Method/Field/Constructor*. *getMethod()* has general form

*Method getMethod(String MethodName, class<?> … paramTypes)*

where *paramTypes* are class objects representing the arguments the method takes. Note that it is a varargs parameter.

* *getAnnotation()* obtains an annotation associated with a *Class, Method, Field* or *Constructor* object. It has general form

*<A extends Annotation> getAnnotation(Class<A> annoType)*

where *annoType* is a *Class* object representing the annotation in question. The method returns a reference to the annotation, which includes the values of the annotation’s members. Returns *null* if the annotation is not found, e.g. when it does not have *RUNTIME* retention.

Example – using *reflection* to obtain an annotation:

*import java.lang.annotation.\*;*

*import java.lang.reflect.\*;*

*@Retention(RetentionPolicy.RUNTIME)*

*@interface Anno {*

*String str();*

*int val();*

*}*

*class demo {*

*@Anno(str = “Example text”, val = 100)*

*public static void Meth() {*

*demo ob = new demo();*

*try {*

*Class<?> c = ob.getClass(); //get Class object representing this class*

*Method m = c.getMethod(“Meth”); //get Method object representing this method*

*Anno a = m.getAnnotation(Anno.class); //get annotation*

*System.out.println(a.str() + “ “ + a.val() ); //uses method call syntax*

*} catch(NoSuchMethodException exc) {*

*System.out.println(“Method not found.”);*

*}*

*}*

*public static void main(String args[]) {*

*Meth(); //outputs “Example text 100”*

*}*

*}*

class literal

*Anno a = m.getAnnotation(Anno.class);*

the *Anno* type *Class* object is called a class literal. This type of expression can be used whenever a *Class* object of a known class is needed (need class name). Class literals can be obtained for classes, interfaces, primitive types, and arrays.

Example – method with parameters – must specify class objects representing the types of parameters as arguments to *getMethod()*:

*import java.lang.annotation.\*;*

*import java.lang.reflect.\*;*

*@Retention(RetentionPolicy.RUNTIME)*

*@interface Anno {*

*String str();*

*int val();*

*}*

*class demo {*

*@Anno(str = “Example text”, val = 100)*

*public static void Meth(String str, int i) {*

*demo ob = new demo();*

*try {*

*Class<?> c = ob.getClass();*

*Method m = c.getMethod(“Meth”, String.class, int.class);*

*//additional arguments of Class objects representing String and int*

*Anno a = m.getAnnotation(Anno.class);*

*System.out.println(a.str() + “ “ + a.val() );*

*} catch(NoSuchMethodException e) {*

*System.out.println(“Method not found.”);*

*}*

*}*

*public static void main(String args[]) {*

*Meth(“test”, 10); //outputs “Example text 100”*

*}*

*}*

Obtaining all annotations

Obtain all annotations with *RUNTIME* retention with

*Annotation[] getAnnotations()*

(returns an array of annotations) (“s”!)

Example:

*@Retention(RetentionPolicy.RUNTIME)*

*@interface Anno1 {*

*String str();*

*int val();*

*}*

*@Retention(RetentionPolicy.RUNTIME)*

*@interface Anno2 {*

*String des();*

*}*

*@Anno1(str = “class text1”, val = 100)*

*@Anno2(des = “class text2”)*

*class demo {*

*@Anno1(str = “method text1”, val = 99)*

*@Anno2(des = “method text2”)*

*public static void Meth() {*

*demo ob = new demo();*

*try {*

*Annotation annos[] = ob.getClass().getAnnotations();*

*//all annotations for demo*

*System.out.println(“Annotations for demo: “);*

*for(int i: annos)*

*System.out.println(i); //displays*

*/\* “Annotations for demo:*

*@Anno1(str = “class text1”, val = 100)*

*@Anno2(des = “class text2”) “ \*/*

*//Annotation type overrides the toString() method, so that println() displays a //string of description for the annotation*

*System.out.println();*

*Method m = ob.getclass().getMethod(“Meth”);*

*Annotation annos2[] = m.getAnnotations();*

*//all annotations for Meth*

*System.out.println(“Annotations for Meth: “);*

*for(int i: annos2)*

*System.out.println(i); //displays*

*/\* “Annotations for Meth:*

*@Anno1(str = “method text1”, val = 99)*

*@Anno2(des = “method text2”) “ \*/*

*} catch(NoSuchMethodException e){*

*System.out.println(“No such method.”);*

*}*

*}*

*public static void main(String args[]) {*

*Meth();*

*}*

*}*

The *AnnotatedElement* Interface – defined in *java.lang.reflect*. It defines several methods, including *getAnnotation()* and *getAnnotations()*, as well as

*Annotation[] getdeclaredAnnotations()*

which returns all annotations declared (not inherited) in the invoking object, and

*boolean isAnnotationPresent(Class<? extends Annotation> annoType)*

which returns true if *annoType* is associated with the invoking object, false otherwise.

**Default Values**

Annotation members can have default values, with general form

*type member() default val;*

*val* must be compatible with *type*. The default will be used if no value is given when the annotation is used, or the declaration can provide a value. Example:

*@Retention(RetentionPolicy.RUNTIME)*

*@interface Anno {*

*String str() default “example”;*

*String val() default 100;*

*}*

**Marker Annotations** – contain no members. Their sole purpose is to mark a declaration.Example of marker annotation and the use of *isAnnotationPresent()*:

*import java.lang.annotation.\*;*

*import java.lang.reflect.\*;*

*@Retention(RetentionPolicy.RUNTIME)*

*@interface Marker { }*

*class demo {*

*@marker //no () needed!*

*public static void Meth(String str, int i) {*

*demo ob = new demo();*

*try {*

*Method m = ob.getClass().getMethod(“Meth”);*

*if(m.isAnnotationPresent(Marker.class) )*

*System.out.println(“Marker is present.”);*

*} catch(NoSuchMethodException e) {*

*System.out.println(“no such method.”);*

*}*

*}*

*public static void main(String args[]) {*

*Meth();*

*}*

*}*

**Single member annotation**

If an annotation only has one member, **and it’s named “value”,**one can specify the member’s value without its name when the annotation is applied. Example:

*@Retention(RetentionPolicy.RUNTIME)*

*@interface Anno {*

*int value();*

*}*

*class demo {*

*@Anno(100) //equivalent to @Anno(value = 100)*

*public static void Meth() {//…}*

*//…*

single value syntax can also be used if all other members besides value has default values (and are applied). Example:

*@Retention(RetentionPolicy.RUNTIME)*

*@interface Anno {*

*int value();*

*int def() default 10;*

*}*

*class demo {*

*@Anno(100)*

*public static void Meth() {//…}*

*//…*

Built-in annotations

Most are specialized, but 8 are general purpose:

* imported from *java.lang.Annotation*: *@Retention, @Documented, @Target, @Inherited*
* imported from *java.lang*: *@Override, @Deprecated, @SafeVarargs, @SuppressWarnings*

*@Retention*: specifies the retention policy. Only to be used as annotation to an annotation.

*@Documented*: marker annotation, tells a tool that an annotation is to be documented. Only to be used as annotation to an annotation.

*@Target*: specifies the types of declaration to which an annotation can be applied. Only to be used as annotation to an annotation.

Takes one argument, which must be from the *ElementType* enumeration, which can be:

1. ANNOTATION\_TYPE
2. CONSTRUCTOR
3. FIELD
4. METHOD
5. LOCAL\_VARIABLE
6. PACKAGE
7. PARAMETER
8. TYPE (class, interface, or enumeration)

Can specify more than one of them by putting them into a list enclosed with braces, e.g.

*@Target( {ElementType.FIELD, ElementType.PACKAGE} )*

*@Inherited*: marker annotation only to be used as annotation to an annotation, and only affects annotations used on class declarations. Causes the annotation for a superclass to be inherited by the subclass (if a superclass has an annotation with *@Inherited,* its subclasses will return that annotation when asked even if they don’t have it).

*@Override*: marker annotation only to be used on methods. A method annotated with *@Override* **must** override (not overload) a method (else an error would occur).

*@Deprecated*: marker annotation. Indicates the declaration is obsolete.

*@SafeVarargs*: marker annotation, can be applied on varargs *static* or *final* method and constructor declarations. Suppresses warning related to varargs parameters.

*@SuppressWarnings*: the specified warning (in String form) from the compiler will be suppressed. Can be applied to any declaration.

**Annotation Restrictions**

1. no annotation can inherit another
2. methods declared by an annotation cannot have parameters
3. annotations can only return one of:
   1. primitive type;
   2. *Class* or *String* object;
   3. *enum* type;
   4. Annotation type;
   5. array of the preceding
4. annotations cannot be generic (cannot take type parameters)
5. annotation methods cannot have a *throws* clause

CH13

**I/O, Applet, etc.**

I/O Streams

A stream is an abstraction that either produces or consumes information, and linked to a physical device by the Java I/O system.

All streams behave in the same manner. The I/O classes and methods apply to any device.

Stream implementations are located in the *java.io* package.

There are two types of streams: **byte** and **character**, for handling bytes and characters respectively.

**Byte Stream**

Has two abstract superclasses at the top of the class hierarchy: *InputStream* and *OutputStream*, and several concrete subclasses*.* *InputStream* and *OutputStream* define several key methods, including *read()* and *write()*. Each has abstract forms and must be overridden by the subclasses.

The byte stream has the following classes:

|  |  |
| --- | --- |
| **Class** | **Meaning** |
| *BufferedInputStream* |  |
| *BufferedOutputStream* |  |
| *ByteArrayInputStream* | Input stream that reads from a byte array |
| *ByteArrayOutputStream* | Output stream that writes to a byte array |
| *DataInputStream* | Input stream containing methods for reading Java standard data types |
| *DataOutputStream* | Output stream containing methods for writing Java standard data types |
| *FileInputStream* | Input stream that reads from a file |
| *FileOutputStream* | Output stream that writes to a file |
| *FilterInputStream* | Implements *InputStream* |
| *FilterOutputStream* | Implements *OutputStream* |
| *InputStream* |  |
| *ObjectInputStream* | Input stream for objects |
| *ObjectOutputStream* | Output stream for objects |
| *OutputStream* |  |
| *PipedInputStream* |  |
| *PipedOutputStream* |  |
| *PushbackInputStream* | Input stream that supports one-byte “unget”, which returns a byte to the input stream |
| *SequenceInputStream* | Input Stream that is a combination of multiple input streams that will be read one after the other |

**Character Stream**

The top of the character stream are two abstract classes: *Reader* and *Writer*. They define several key methods, including *read()* and *write()*. They each have several concrete subclasses.

These classes are:

|  |  |
| --- | --- |
| **Class** | **Meaning** |
| *BufferedReader* | Buffered input character stream |
| *BufferedWriter* |  |
| *CharArrayReader* |  |
| *CharArrayWriter* |  |
| *FileReader* |  |
| *FileWriter* |  |
| *InputStreamReader* | Input stream for translating bytes to characters |
| *LineNumberReader* | Input stream that counts lines |
| *OutputStreamWriter* | Output stream for translating characters to bytes |
| *PipedReader* | Input pipe |
| *PipedWriter* |  |
| *PrintWriter* | Output stream containing *print()* and *println()* |
| *PushbackReader* | Input stream that allows characters to be returned to the input stream |
| *Reader* |  |
| *StringReader* | Reads from a string |
| *StringWriter* |  |
| *Writer* |  |

**Predefined Streams**

*java.lang* defines a class *System*, which contains 3 predefined stream variables: *in, out, err.* They are declared as *public, static, final*, so that they can be used without reference to a *System* object.

*System.out* is the standard output stream (the console). *System.in* is the standard input (the keyboard). *System.err* is the standard error system (also the console). These streams can be redirected to any compatible I/O device.

*System.in* is an object of type *InputStream*; *System.out* and *System.err* are objects of type *PrintStream*. These are byte streams but can read and write characters.

**Reading Console Input**

Console input is accomplished by reading from *System.in*. To obtain a character-based stream attached to the console, wrap *System.in* in a *BufferedReader* object. A commonly used constructor is

*BufferedReader(Reader InputReader)*

where *Reader* is an abstract class, *InputReader* is the stream that is being linked to instance of *BufferedReader* being created.

One of *Reader*’s subclasses is *InputStreamReader*, which converts bytes to characters. It has the constructor

*InputStreamReader(InputStream IpStream)*

Because *System.in* refers to an object of *InputStream* type, it can be used as the argument for this constructor. Thus an example for creating a *BufferedReader* that is connected to the keyboard is

*BufferedReader br = new BufferedReader(new InputStreamReader(System.in) );*

**Reading Characters**

To read a character from a *BufferedReader*, use *read()*. The version to be used is

*int read() throws IOException*

*read()* reads a character from the input stream and returns it as an integer. When the end of stream is encountered, it returns -1.

Example: the BufferedReader reads *char* from the console and displays them, quitting the program when “q” is encountered.

*import java.io.\*;*

*class demo {*

*public static void main(String args[]) throws IOException {*

*char c;*

*BufferedReader br = new BufferedReader(new InputStreamReader(System.in) );*

*System.out.println(“Enter characters, q to quit.”);*

*do {*

*c = (char) br.read();*

*System.out.println(c);*

*} while(c != ‘q’);*

*}*

*}*

**Reading Strings**

*readLine()*, a member of the *BufferedReader* class, reads a string. It has general form

*String readLine() throws IOException*

Example: the BufferedReader reads *Strings* from the console and displays them, quitting the program when “stop” is encountered.

*import java.io.\*;*

*class demo {*

*public static void main(String args[]) throws IOException {*

*String str;*

*BufferedReader br = new BufferedReader(new InputStreamReader(System.in) );*

*System.out.println(“Enter lines, enter “stop” to quit.”);*

*do {*

*str = br.readLine();*

*System.out.println(str);*

*} while(!str.equals(‘stop’) );*

*}*

*}*

Example: creates an array of String objects to store user input in console, up to 100 lines. “stop” quits the program.

*import java.io.\*;*

*class demo {*

*public static void main(String args[]) throws IOException {*

*String str[] = new String[100];*

*BufferedReader br = new BufferedReader(new InputStreamReader(System.in) );*

*System.out.println(“Enter lines, enter “stop” to quit.”);*

*for(int i = 0, i < 100, i++) {*

*str[i] = br.readLine();*

*if(str[i].equals(“stop”) ) break;*

*}*

*System.out.println(“Displaying the lines:”);*

*for(int I = 0, i < 100, i++) {*

*System.out.println(str[i]);*

*if(str[i].equals(“stop”) ) break;*

*}*

*}*

*}*

**Writing Console Output**

Most typically done with *print()* and *println().* They are defined by *PrintStream*. Because *PrintStream* is derived from *OutputStream*, it also implements the low level methods *write()*, which has form

*void write(int byteval)*

and writes the **low-order 8 bits** of *byteval. write()* is also referenced by *System.out.write()*.

**The *PrintWriter* Class**

*System.out* is most useful for debugging or demonstration purposes. For real world programs, the *PrintWriter* class is recommended. This makes the program easier to internationalize.

*PrintWriter* is a character-based class. *PrintWriter* has several constructors, the one used will be

*PrintWriter(OutputStream OpStream, boolean flushOnNewLine)*

where *flushOnNewLine* controls whether to flush the output stream automatically every time *println()* is called. Specifying *System.out* as *OpStream* and *true* for *flushOnNewLine* makes *PrintWriter* write to console. i.e.

*PrintWriter(System.out, true)*

*PrintWriter* supports *print()* and *println()*.

Example:

*import java.io.\*;*

*public class demo {*

*public static void main(String args[]) {*

*PrintWriter pw = new PrintWriter(System.out, true);*

*int i = 7;*

*double d = 7.01;*

*pw.println(‘Some String’);*

*pw.println(i);*

*pw.println(d);*

*}*

*}*

**Reading and Writing Files**

Two most used classes are *FileInputStream* and *FileOutputStream* which create byte streams linked to files. To open a file, create an object of one of the classes and use the name of the file as argument to the constructor. The following constructors will be used:

*FileInputStream(String Filename) throws FileNotFoundException*

*FileOutputStream(String Filename) throws FileNotFoundException*

The input stream creation throws a *FileNotFoundException* if the file does not exist; the output stream creation throws a *FileNotFoundException* if the file cannot be opened or created. When an output file is opened, any old file of the same name is destroyed. *FileNotFoundException* is a subclass of *IOException*.

When the program is done using a file, it must be closed with *close()*, implemented by both *FileInputStream* and *FileOutputStream*:

*void close() throws IOException*

failure to close a file can result in memory leak as unused resources remain allocated.

The above is the old way of closing a file (pre-JDK 7). The second way is with the *try-with-resources* statement added in JDK 7, which automatically closes a file when it is no longer needed. In this approach, to read a file, use the *read()* method defined in *FileInputStream*. The version used is

*int read() throws IOException*

Each call reads a single byte from the file and returns as an integer. -1 is returned at the end of the file.

note: to turn the integers to character, use *(char) i;*

Example: opens the text file specified in the command line. (*java demo FileName.txt)*

*import java.io.\*;*

*class demo {*

*public static void main(String args[]) {*

*int j;*

*FileInputStream fin = null;*

*if(args.length != 1) { //confirms a file name has been input*

*System.out.println(“input file name”);*

*return;*

*}*

*try {*

*fin = new FileInputStream(args[0]); //opens the file*

*} catch(FileNotFoundException e) {*

*System.out.println(“File not found.”);*

*return;*

*}*

*try { //read the file until the end of file*

*do {*

*j = fin.read();*

*if(j != -1) System.out.print((char) j);*

*} while(j != -1);*

*} catch(IOException e) {*

*System.out.println(“Error reading file.”);*

*}*

*finally { // good practice to put close() within a finally block.*

*try { //close the file*

*if(fin != null) fin.close(); //only closes if a file has been opened*

*} catch(IOException e) {*

*System.out.println(“Error closing file.”);*

*}*

*}*

*}*

*}*

Because *FileNotFoundException* is a subclass of *IOException*, the open file and read file blocks could have been put in the same *try* block, i.e.

*try {*

*fin = new FileInputStream(args[0]); //opens the file*

*do { //read the file until the end of file*

*j = fin.read();*

*if(j != -1) System.out.print((char) j);*

*} while(j != -1);*

*} catch(IOException e) {*

*System.out.println(“I/O error: ” + e);*

*}*

To write to a file, use *write()* defined by *FileOutputStream*. It has the form

*void write(int byteval) throws IOException*

which writes the byte specified by *byteval* (only the low-order 8 bits).

Example: copying a file with the command line *java demo CopiedFile.txt NewFile.txt*

*import java.io.\*;*

*class demo {*

*public static void main(String args[]) {*

*int j;*

*FileInputStream fin = null;*

*FileOutputStream fout = null;*

*if(args.length != 2) { //checks that the names of the files are given*

*System.out.println(‘input the names of original file and copy.’);*

*return;*

*}*

*try {*

*fin = new FileInputStream(args[0]);*

*fout = new FileOutputStream(args[1]); //read the files*

*do {*

*j = fin.read();*

*if(j != -1) fout.write(j);*

} while(j != -1);

} catch(IOException e) {

*System.out.println(‘I/O error: ’ + e);*

*}*

*finally {*

*try {*

*if(fin != null) fin.close();*

*} catch(IOException e2) {*

*System.out.println(‘error closing input file.’);*

*}*

*try {*

*//having separate try blocks makes it so that fout still closes if fin throws an exception*

*if(fout != null) fout.close();*

*} catch(IOException e2) {*

*System.out.println(‘error closing output file.’);*

*}*

*}*

*}*

*}*

**Automatically Closing A File** (JDK 7 and later)

The feature is called **automatic resource management/ARM**. ARM is based on the *try-with-resources* statement, with general form:

*try(resource specialization) {*

*//use the resource*

*}*

where *resource specialization* is a statement that declares and initializes a resource. The resource is released when the try block ends.

*try-with-resources* can only be used with resources that implement the *AutoCloseable* interface, defined in *java.lang*. *AutoCloseable* is inherited by the *Closeable* interface in *java.io*, so both interfaces can be used when working with streams.

Example:

*import java.io.\*;*

*class demo {*

*public static void main(String args[]) {*

*int j;*

*if(args.length != 1) { //confirms a file name has been input*

*System.out.println(“usage: demo filename”);*

*return;*

*}*

*try(FileInputStream fin = new FileInputStream(args[0]) ) {*

*//fin is local to the try block. It is also implicitly final.*

*do {*

*j = fin.read();*

*if(j != -1) System.out.print((char) j);*

*} while(j != -1);*

*} catch(IOException e) {*

*System.out.println(“Error reading file.”);*

*} catch(FileNotFoundException e2) {*

*System.out.println(“File not found.”);*

*}*

*}*

*}*

Multiple resources can be initialized with a single try statement. To do so, separate the resources with a semicolon, e.g.

*try(FileInputStream fin = new FileInputStream(args[0]);*

*FileoutputStream fout = new FileoutputStream(args[1]) ) {//…}*

Sidenote: double exception

In the above examples there are 2 possible exceptions: 1 in the try block and 1 in the finally block. With the “normal” try block, if both exceptions occur the second (within finally block) will preempt the first (within try block), which is lost. With the try-with-resources block, if both exceptions occur the second is *suppressed*. Suppressed methods can be obtained with *getSuppressed()* defined in *Throwable*.

**Applet Fundamentals PS: deprecated since JDK 9**

Applets – small applications accessed on the Internet, automatically installed and run as part of a web document. It has limited access to resources on the user’s computer.

Example: a simple applet

*import java.awt.\*;*

*import java.applet.\*;*

*public class SimpleApplet extends Applet {*

*public void paint(Graphics g) {*

*g.drawString(“A simple applet”, 20, 20);*

*}*

*}*

* The first import statement imports the *Abstract Window Toolkit (AWT)* classes. Applets interact with the user through the AWT, not with the console based I/O classes. The AWT contains support for a window-based, graphical user interface.
* The second import statement imports *Applet* class. Every applet created must be a subclass (directly or indirectly) of *Applet*.
* the class declared must be *public*, since it will be accessed by code outside the program.
* Inside the class, *paint()* is called. The AWT defines this method and applets must override it. Every time the applet redisplays its output, *paint()* is called.
* *paint()* has one parameter of type *Graphics*
* Inside *paint()* is a call to *drawString()*, a member of the *Graphics* class that outputs a String located at the specified X, Y coordinate. It has general form

*void drawstring(String str, int x, int y);*

* + (0,0) is the upper-left corner
* Applets do not begin execution at *main()*. Most applets do not even have a *main()*. Instead, an applet executes when the name of its class is passed to an applet viewer or network browser.
* Applets compile in the same way as normal. However, applets run differently. There are 2 ways:
  + execute the applet within a web browser
  + using an applet viewer, such as *appletviewer*.

These ways are examined next.

note: the use of local applets is discouraged, they are recommended to be executed through a web browser.

One way to execute an applet in a web browser is to write a short HTML text file that contains a tag that loads the applet. Oracle recommends using the APPLET tag for this purpose, with the following:

*<applet code=”AimpleApplet” width=200 height=60>*

*</applet>*

where *width* and *height* specify the dimensions of the display area of the applet.

If the preceding file is called *RunApp.html*, then the following will run *SimpleApplet*:

*C:\> appletviewer RunApp.html*

One can also include a comment of the above <applet> lines at the head of the java source code, after the import statements and before the class declaration. Then the applet can be compiled with the source file. In this method, applet development has 3 steps:

1. Edit a java source file
2. Compile the program
3. Execute the applet viewer, specifying the source file’s name. The viewer will encounter the APPLET tag within the comment and execute the applet

**The transient and volatile Modifiers**

*transient* and v*olatile* are type modifiers for specialized situations.

When an instance variable is declared as *transient*, its value need not persist when an object is stored. E.g.

*class T {*

*transient int a;*

*int b;*

*}*

If an object of type T is written to a persistent storage, the value of *b* will be saved but the value of *a* will not.

*volatile* tells the compiler that the modified variable can be changed unexpectedly by other parts of the program, e.g. by other threads.

* When multithreading each thread keeps its own copy of a shared variable, and there is also a real/master copy of the variable which is updated at various times such as when a *synchronized* method is entered.
* To increase efficiency, or to make the variable always up to date, specify the variable as *volatile* and the compiler will always use the master copy of the variable.

**Using *instanceof***

Run-time operator *instanceof* allows the program to obtain run time type information about an object.

One example of its usefulness is to catch invalid casting at run time:

* invalid casts create a run time error
* casts involving class hierarchies can produce casting errors that can only be caught at run time
  + E.g., superclass A has subclasses B and C. Thus casting an object of type B or C into type A is valid, but casting an object of type B into type C is invalid.
  + However an object of type A can refer to type B or C, so knowing which is important when attempting to cast to C.

*instanceof* has the syntax

*objref instanceof type*

where *objref* refers to an instance of a class, *type* is a class type. If *objref* is the specified *type* or can be cast into it, *instanceof* returns true; otherwise false.

Example:

*class A {*

*int I;*

*int j;*

*}*

*class B {*

*int I;*

*int j;*

*}*

*class C extends A {*

*int k;*

*}*

*class D extends A {*

*int k;*

*}*

*class demo {*

*public static void main(String args[]) {*

*A a = new A();*

*B b = new B();*

*C c = new C();*

*D d = new D();*

*if(a instanceof A) System.out.println(“a is instance of A”);*

*if(b instanceof B) System.out.println(“b is instance of B”);*

*if(c instanceof A) System.out.println(“c can be cast to A”);*

*if(a instanceof C) System.out.println(“a cannot be cast to C”);*

*if(d instanceof C) System.out.println(“d cannot be cast to C”);*

*A ob;*

*ob = d; //A reference to D*

*if(ob instanceof D) System.out.println(“ob is an instance of D”);*

*ob = c; //A reference to C*

*if(ob instanceof D) System.out.println(“ob is not instance of D”);*

*if(ob instanceof A) System.out.println(“ob can be cast to A”);*

*if(a instanceof object) System.out.println(“All objects can be cast to Object”);*

*if(b instanceof object) System.out.println(“All objects can be cast to Object”);*

*if(c instanceof object) System.out.println(“All objects can be cast to Object”);*

*}*

*}*

***strictfp***

Modifying a class/method/interface with *strictfp* causes Java to revert to an older and stricter floating-point computation model.

Rarely needed.

**Native Methods**

Native code is written in a language native to the CPU. Native methods allows Java access to existing bases of library, and speeds up execution.

The *native* keyword declares a method as native method. Once declared, native methods can be called just like normal methods.

Syntax:

*public native type MethodName();*

Most native methods are written in C. The Java Native Interface (JNI) allows integration of C code into a Java program. Example:

*public class demo {*

*int i;*

*public native void test(); //no body*

*static {*

*System.loadLibrary(“demo”);*

*//load dynamic link library (DLL) containing static method*

*}*

*public static void main(String args[]) {*

*demo ob = new demo();*

*ob.i = 0;*

*System.out.println(ob.i);*

*ob.test();*

*System.out.println(ob.i);*

*}*

*}*

A *static* block only executes once at the beginning of the program’s execution. It contains *loadLibrary()* which loads the DLL containing the native implementation of *test()*. *loadLibrary()* is part of the *System* class, with general form

*static void loadLibrary(String filename)*

where *filename* is the name of the file that holds the library. In Windows, it is assumed to have the .DLL extension.

Compile this file to create demo.class. Next, use javah.exe to produce demo.h. To do so, use the command

*javah -jni demo*

this produces the .h file which must be included in the C file that implements *test()*.

[…]

Problems with Native Methods

* potential security risk – native method can gain access to any part of the system; therefore applets cannot use native methods
* loss of portability – native methods can only run on systems with the DLL file present, and that file must be compatible with the CPU of the system

***assert***

The keyword *assert* creates an assertion, which is a condition that should be true during execution of a program. If an assertion is true, no action takes place; if it is false, an *AssertionError* is thrown. Assertions are usually used during testing, but not in released code.

*assert* has 2 forms:

*assert condition;*

where condition evaluates to a boolean; and

*assert condition: expr;*

where *expr* is a value or String that is passed to the *AssertionError* constructor to be displayed as a String.

To enable assertion checking at run time, the command must include “*-ea*”, e.g.

*java -ea AssertionDemo*

One can also disable assertion with “*-da*”.

Normally, released codes will run with assertion disabled. As such, it is important not to use it for any action required by the program.

Example – enabling assertion will cause the loop to end prematurely:

*class demo {*

*public static void main(String args[]) {*

*int n = 3;*

*for(int i=0; i<10; i++) {*

*n--;*

*assert n > 0;*

*System.out.println(n);*

*}*

*}*

*}*

One can enable or disable all assertions in a package or file. The commands are

*-ea: mypack…*

to enable assertions in the package *mypack*,

*-da: mypack…*

to disable assertions in *mypack*,

*-ea: demo*

to enable assertions in demo.java.

Static import

Following *import* with *static* causes the import statement to import the static members of a class or interface. When using static import, one can refer to the static members directly, without the names of their classes. Example – calculating a hypotenuse:

*import static java.lang.Math.sqrt;*

*import static java.lang.Math.pow;*

*class demo {*

*public static void main(String args[]) {*

*double side1, side2, hypo;*

*side1 = 3.0;*

*side2 = 4.0;*

*hypo = sqrt(pow(side1, 2) + pow(side2, 2) );*

*//equivalent to Math.sqrt(Math.pow(side1, 2) + Math.pow(side2, 2) )*

*System.out.println(“the hypotenuse is “ + hypo);*

*}*

*}*

*Math.pow()* raises a value to a specific power.

Another example is the *println()*:

*import static java.lang.System.out;*

*//…*

*out.println(“printing using out directly”;*

There are 2 forms of static import:

*import static pkg.type-name.static-member-name;*

*import static pkg.type-name.\*;*

Risk of static import: it adds more names into the global namespace, adding more chances of name conflicts. Static import should only be used if a static member will be used repeatedly.

**Invoking Overloaded Constructors Through *This()***

With overloaded constructors, sometimes it’s useful for one constructor to invoke another. This is done with

*this(arg-list)*

when *this()* executes, the constructor whose parameter list matches *arg-list* is executed first. *this()* must be the first statement inside the constructor. Thus *super()* and *this()* cannot be used in the same constructor, as both needs to be first.

Example:

*class demo {*

*int a, b;*

*demo(int i, int j) {*

*a = i;*

*b = j;*

*}*

*demo(int i) {*

*this(i, i);*

*}*

*/\* equivalent to*

*demo(int i) {*

*a = i;*

*b = i;*

*} \*/*

*demo( ) {*

*this(0);*

*}*

*/\* equivalent to*

*demo( ) {*

*a = 0;*

*b = 0;*

*} \*/*

*}*

Constructors that call *this()* will run a bit slower. If a class will be used to create a large amount (1000s) of objects, this impact can be meaningful.

CH14

**Generics**

Generics (added in JDK 5) changed java in 2 ways:

1. it added a new syntactical element to the language
2. it changed many classes and methods in the core API

Generics allows the creation of classes, interfaces and methods that work with various types of data.

The Collections Framework is significantly affected by generics. A collection is a group of objects. The collection classes have been able to work with any type of object before, but generics allows collection classes to be used with complete type safety.

**Generics Fundamentals**

*Generics* at its core means *parameterized types*, which allows the creation of classes, interfaces and methods in which the type of data upon which they operate is specified as a parameter. Such a class/interface/method is called a *generic class/interface/method*.

* Generalized class/interface/methods can be created before by operating through references of type *Object* (which is the superclass of all classes). But they did not have type safety.
* Generics added the type safety. They also streamlined the process by removing the need for explicit casts to translate between *Object* and other types. With generics, casts are automatic and implicit.

Example—*T* is a type parameter that will be replaced by a real type when an object of type *gen* is created:

*class gen<T> {*

*T ob; //declare object of type T*

*gen(T o) {*

*ob = o;*

*}*

*T getob() {*

*return ob;*

*}*

*void showType() {*

*System.out.println(“type of T is “ + ob.getClass().getName() );*

*}*

*class demo {*

*public static void main(String args[]) {*

*Gen<Integer> iOb; //creates a gen reference for Integers*

*iOb = new Gen<Integer>(88); //autoboxing of 88 into an Integer*

*iOb.showType(); //returns “type of T is java.lang.Integer”*

*int v = iOb.getob() //auto-unboxing of Integer to int*

*System.out.println(“v is “ + v); //returns “v is 88”*

*Gen<String> strOb = new Gen<String> (“Test String”); //creates a gen object for Strings*

*strOb.showType(); //returns “type of T is java.lang.String*

*String s = strOb.getOb();*

*System.out.println(“s contains “ + “\”” + s + “\””) //returns “s contains “Test String””*

*}*

*}*

Here, *T* is the name of a *type parameter*—a placeholder for the actual type that will be passed to *Gen* when an object is created. Angled brackets <> denote the declaration of a type parameter. Because *Gen* uses a type parameter, it is a generic class, which is also called a *parameterized type*.

When an object of type T is created, its actual type will be determined by the type passed to it. T can also be used as the return type of a method.

* *getClass()* is defined by *Object* and thus a member of all classes. It returns the calling object’s Class object.
* *getName()* is defined by *Class*, it returns a string representing the class name.

In *Gen<Integer> iOb;*

*Integer* is a *type argument* passed to Gen’s type parameter, *T*. This creates a version of *Gen* where all references to *T* are translated into references to *Integer*. *iOb* must refer to object of type *Gen<Integer>*, attempting to do otherwise will prompt a compile-time error, e.g.

*iOb = new Gen<Double> (88.0); //error*

Note: the Java compiler does not actually create different versions of generic classes (though it is helpful to think this way). Instead, the compiler removes all generic type information and substitute them for the right types. Thus there is really only one version of the generic class in the program. The process of removing generic type information is called *erasure*.

**Generics work only with Objects**

When declaring an instance of a generic type, the type parameter must be a class type, not a primitive type. For example, the following is illegal:

*Gen<int> intOb = new Gen<int>(52);*

Instead of passing primitives, pass its type wrapper instead.

Generic types differ based on type arguments

Generic types with different type arguments are not compatible. For instance, the following is in error:

*iOb = strOb;*

Although both are of type *Gen<T>*, having different *T*’s make them incompatible. This gives generics type safety.

**The benefits of Generics**

Below is a non-generic class functionally equivalent to the generic example above.

*class NonGen {*

*Object ob;*

*NonGen(Object o) {*

*ob = o;*

*}*

*Object getob() {*

*return ob;*

*}*

*void showType() {*

*System.out.println(“Type of ob is “ + ob.getClass().getName() );*

*}*

The generic version has the advantage that *Gen<T>* already knows what type of object *ob* is; *NonGen* does not. This means

1. *NonGen* needs to explicitly cast *ob* when using it, i.e.

*int v = (Integer) iOb.getob(); //casts Object to Integer*

1. *NonGen* instances are always compatible; *Gen<T>* instances are not compatible if *T*’s differ, i.e.

*iOb = strOb; //works in NonGen, not in Gen*

This can then lead to trying to assign the wrong type to *ob*.

In essence, generics convert run-time errors to compile-time errors.

**Generic Class with multiple Type Parameters**

To specify more than one parameter in a generic type, use a comma-separated list. Example:

*class TwoGen<T, V> {*

*T ob1;*

*V ob2;*

*TwoGen(T o1, V o2) {*

*ob1 = o1;*

*ob2 = o2;*

*void showTypes() {*

*System.out.println(“Type of T is “ + ob1.getClass().getName() );*

*System.out.println(“Type of V is “ + ob2.getClass().getName() );*

*}*

*T getob1() {*

*return ob1;*

*}*

*V getob2() {*

*return ob2;*

*}*

*}*

*class demo {*

*public static void main(string args[]) {*

*TwoGen<Integer, String> tgOb = new TwoGen<Integer, String>(88, “Test String”);*

*}*

The parameters can be of any Object type, including 2 of the same, e.g.

*Twogen<String, String> tgOb = new TwoGen<String, String>(“String1”, “String2”);*

General form of a generic class

*class ClassName<type-param-list> {//…}*

*ClassName<type-arg-list> InstanceName = new ClassName<type-arg-list>(init-arg-list);*

**Bounded Types**

Limits the types that can be passed to a type parameter, by declaring a superclass from which all type arguments must derive. This is done with *extends*, i.e.

*class Gen<T extends superclass>*

~~Makes it so that any object of type~~ *~~T~~* ~~must be of type~~ *~~superclass~~* ~~or one of its subclasses; any array of type~~ *~~T[]~~* ~~must contain objects of type~~ *~~superclass~~* ~~or one of its subclasses.~~

Makes it so that any object stored in a *Gen* object must be of type *superclass* or one of its subclasses; any array stored in a *Gen* object must contain objects of type *superclass* or one of its subclasses.

Example—limits the type parameters to numbers (all numeric classes, e.g. Integer, Double are subclasses of *Number*)

*class Stats<T extends Number> {*

*T[] nums; //array of type T*

*Stats(T[] o) {*

*nums = o;*

*}*

*double average() {*

*double sum = 0.0;*

*for(int i=0; i < num.length; i++) sum += nums[i].doubleValue();*

*\*/ without the extends clause, this will prompt an error about doubleValue() not being defined in nums \*/*

*return sum / nums.length;*

*}*

*}*

*class demo {*

*public static void main(String args[]) {*

*Integer inums[] = { 1, 2, 3, 4, 5 };*

*Stats<Integer> iOb = new Stats<Integer>(inums);*

*double avg = iOb.average();*

*System.out.println(“iOb average is “ + avg);*

*// String strs[] = {“1”, “2”, “3” }; //this does not compile*

*// Stats<String> strOb = new Stats<String>(strs);*

*}*

*}*

This program can now use *doubleValue()* as a member of *nums* because type *T* is bounded by *Number*. The bounding of *T* also prevents nonnumeric *Stats* objects from being created.

In addition to classes, interfaces can also be used as a bound.

In addition, there can be a class and multiple interfaces as bounds; in these cases, the class must be stated first, i.e.

*class Gen<T extends MyClass & MyInterface1 & MyInterface2> {//… }*

here, *T* must be a subclass of *MyClass* and implement *MyInterface1* and *MyInterface2*.

**Wildcard Arguments**

The wildcard argument is specified by <*?>*, and represents an unknown type. Wildcard will match any validgeneric class object; it does not affect what types of object can be created.

Example—*sameAvg()* compares whether two *Stats<T>* objects have the same average *nums*.

*boolean sameAvg(Stats<?> ob) {*

*if(average() == ob.average() )*

*return true;*

*return false;*

*}*

without wildcard, this method will only work when the 2 *Stats<T>* objects have the same *T*’s. I.e.

*//this only works when T’s are the same*

*boolean sameAvg(Stats<T> ob) {*

*if(average() == ob.average() )*

*return true;*

*return false;*

*}*

**Bounded Wildcards**

Wildcards can be bounded in the same way as type parameters. A bounded wildcard has either an upper bound or lower bound, which restricts the types of objects upon which a method will operate.

The upper bound is more commonly used, and is created using an *extends* clause:

*type meth-name(Gen<? extends superclass> c) {//… }*

this method will only work with *Gen<T>* where *T* is an object of *superclass* or its subclass.

Example: 3 classes with 2, 3 and 4 dimensions, and 2 method that shows the XY and XYZ coordinates—

*class twoD {*

*int x,y;*

*twoD(int a, int b) {*

*x = a;*

*y = b;*

*}*

*}*

*class threeD extends twoD {*

*int z;*

*threeD(int a, int b, int c) {*

*super(a, b);*

*z = c;*

*}*

*}*

*class fourD extends threeD {*

*int t;*

*fourD(int a, int b, int c, int d) {*

*super(a, b, c);*

*t = d;*

*}*

*}*

*class CoordHolder<T extends twoD> {*

*T[] coords;*

*CoordHolder( T[] o) {*

*coords = o;*

*}*

*}*

*class demo {*

*static void showXY(CoordHolder<?> c) {*

*System.out.println(“x, y coordinates are: “);*

*for(int i=0; i < c.coords.length; i++)*

*System.out.println(c.coords[i].x + “ “ + c.coords[i].y);*

*System.out.println();*

*}*

*static void showXYZ(CoordHolder<? extends threeD> c) {*

*System.out.println(“x, y, z coordinates are: “);*

*for (int i=0; i<c.coords.length; i++)*

*System.out.println(c.coords[i].x + “ “ + c.coords[i].y + “ “ + c.coords[i].z);*

*System.out.println();*

*}*

*twoD ex2d[] = {*

*new twoD(0, 0),*

*new twoD(1,1),*

*new twoD(2,2)*

*};*

*CoordHolder<twoD> ex2dhd = new CoordHolder<twoD>(ex2d);*

*threeD ex3d[] = {*

*new threeD(0, 0, 0),*

*new threeD(1, 1, 1),*

*new threeD(2, 2, 2)*

*};*

*CoordHolder<threeD> ex3dhd = new CoordHolder<threeD>(ex3d);*

*public static void main(String args[]) {*

*showXY(ex2dhd);*

*showXY(ex3dhd);*

*//showXYZ(ex2dhd); //error*

*showXYZ(ex3dhd);*

*}*

*}*

The lower bound has the general form

*<? super subclass>*

in which case only classes that are superclasses of *subclass* are acceptable arguments (this does not include *subclass* itself).

**Creating a generic method**

Methods inside a generic class can make use of the class’ type parameter and are automatically generic relative to the type parameter.

Moreover, it’s possible to declare a generic method that uses its own type parameter(s), and also do so within a non-generic class.

Example: generic method inside a non-generic class. The method determines if an object is a member of an array.

*class demo {*

*static<T, V extends T> boolean IsInArray(T x, V[] y) {*

*for(int i=0; i<y.length; i++)*

*if(x.equals(y[i])) return true;*

*return false;*

*}*

*public static void main(String args[]) {*

*Integer nums[] = {1, 2, 3, 4, 5};*

*if(IsInArray(2, nums))*

*System.out.println(“2 is in nums”);*

*}*

note:

1. the type parameters are declared before the return type;
2. the *V extends T* clause ensures the 2 arguments are compatible
3. no restriction on whether generic methods are static or not
4. when calling *IsInArray()*, T and V are adjusted automatically, e.g. with *IsinArray(2, nums)*

General syntax for a generic method:

*<type-parameter-list> return-type method-name(parameter-list) {//…}*

where the type parameters are comma-separated.

**Generic Constructors**

Generic constructors can exist in non generic classes. Example:

*class GenCon {*

*private double val;*

*<T extends Number> GenCon(T arg) {*

*val = arg.doubleValue();*

*}*

*void showval() {*

*System.out.println(“val: “ + val);*

*}*

*class demo {*

*public static void main(String args[]) {*

*GenCon gc = new GenCon(100);*

*GenCon gc2 = new GenCon(123.5F);*

*gc.showval();*

*gc2.showval();*

*}*

*}*

**Generic Interface**

Generic interfaces are specified just like generic classes.

General syntax for a generic interface and class implementation:

*interface interface-name<type-parameter-list> {//…*

*class class-name<type-parameter-list> implements interface-name<type-arg-list> {//…*

where the type parameters are comma-separated.

Example:

*interface minmax<T extends Comparable<T>> {*

*T min();*

*T max();*

*}*

*class MyClass<T extends Comparable<T>> implements minmax<T> {*

*//no need to declare “T extends Comparable<T>” again*

*T[] vals;*

*demo(T[] o) {vals = o;}*

*public T min() {*

*T v = vals[0];*

*for(int i=1; I < vals.length; i++)*

*if(vals[i].compareTo(V) < 0) v = vals[i];*

*return v;*

*}*

*public T max() {*

*T v = vals[0];*

*for(int i=1; I < vals.length; i++)*

*if(vals[i].compareTo(v) > 0) v = vals[i];*

*return v;*

*}*

*}*

*class demo {*

*public static void main(String args[]) {*

*Integer inums[] = {3, 5, 6, 2, 6 };*

*Character chs[] = {‘b’, ‘r’, ‘q’, ‘a’ };*

*MyClass<Integer> iob = new MyClass<Integer>(inums);*

*MyClass<Character> cob = new MyClass<Character>(chs);*

*System.out.println(“max value in inums is “ + iob.max());*

*System.out.println(“min value in inums is “ + iob.min());*

*System.out.println(“max value in chs is “ + chs.max());*

*System.out.println(“min value in chs is “ + chs.min());*

*}*

in *<T extends Comparable<T>>*, the type parameter is *T*, its upper bound is *Comparable*. *Comparable* is an interface defined by *java.lang*; classes that implement it defines objects that can be ordered (e.g. numbers and letters). *Comparable* is also generic.

In general, if a class implements a generic interface, then that class must also be generic. E.g. the following is wrong:

*class MyClass implements minmax<T> {//error*

However if a class implements a **specific type** of a generic interface, e.g.

*class MyClass implements minmax<Integer> {//OK*

then the class needs not be generic.

Benefits of generic interfaces:

1. it can be implemented for different types of data
2. it allows constraints (bounds) on the types of data that can be implemented

**Raw Types and Legacy Code**

Pre JDK 5 generics did not exist. To ensure legacy code works with generics, Java allows a generic class to be used without type arguments. This creates a **raw type**. This effectively means its type parameter is *Object* and thus removes the type safety that generics provides.

Example:

*class Gen<T> {*

*T ob;*

*Gen(T o) {*

*o = ob;*

*}*

*T getob() {*

*return ob;*

*}*

*}*

*class demo {*

*public static void main(String args[]) {*

*Gen<Integer> iob = new Gen<Integer>(88);*

*Gen<String> strob = new Gen<String>(“test string”);*

*Gen raw = new Gen(new Double(98.6));*

*double d = (Double) raw.getob(); //cast is necessary*

*//Integer I = (Integer) raw.getob(); //error*

*//raw types can be assigned to a variable of any specific generic type, and vice versa*

*//this may lead to errors*

*strob = raw;*

*raw = iob;*

*}*

*}*

**Generic Class Hierarchy**

In generic class hierarchy, any type arguments needed by a generic superclass must be passed up the hierarchy by all subclasses. e.g.

*class Gen2<T> extends Gen<T> {//…*

Here, *Gen2* passes T to *Gen*. Any type argument passed to Gen2 is also passed to Gen.

Even if *Gen2* isn’t generic, it must still follow this syntax.

However, if a subclass extends a specific type of a superclass, it does not need *<?>*, e.g.

*class Gen2 extends Gen<String> {//…*

A subclass can also add its own parameter, e.g.

*class Gen2<T, V> extends Gen<T> {//…*

A non-generic superclass can have generic subclasses. Example:

*class nonGen {*

*int num;*

*nonGen(int i) {*

*num = I;*

*}*

*}*

*class Gen<T> extends nonGen {*

*T ob;*

*Gen(T o, int i) {*

*super(i);*

*ob = o;*

*}*

*}*

*instanceof* and Generic Hierarchy

*instanceof* can be applied to objects of generic classes. Example:

*class Gen<T> { //same generic class as before*

*T ob;*

*Gen(T o) {*

*o = ob;*

*}*

*T getob() {*

*return ob;*

*}*

*}*

*class Gen2<T> extends Gen<T> {*

*Gen2(T o) {*

*super(o);*

*}*

*}*

*class demo {*

*public static void main(String args[]) {*

*Gen<Integer> iob = new Gen<Integer>(88);*

*Gen<Integer> iob2 = new Gen2<Integer>(99);*

*if(iob2 instanceof Gen2<?>)*

*System.out.println(“iob2 is an instance of Gen2”);*

*if(iob2 instanceof Gen<?>)*

*System.out.println(“iob2 is an instance of Gen”);*

*if(iob instanceof Gen2<?>)*

*System.out.println(“iob is not an instance of Gen2”);*

*//cannot be compiled: generic type info does not exist at run time*

*//if(iob instanceof Gen<Integer>)*

*//System.out.println(“won’t be printed”);*

*}*

*}*

Casting

One instance of a generic class can only be cast to another if they are compatible and their type arguments are the same. Example (using previous program):

*(Gen<Integer>) iob2 //fine*

*(Gen<Long>) iob2 //illegal*

Overriding methods in a generic class

A method in a generic class can be overridden like any other method – no special requirement or syntax.

Type Inference with Generics

To instantiate the following class:

*class demo<T, V> {*

*T ob1;*

*V ob2;*

*demo(T a, V b) {*

*ob1 = a;*

*ob2 = b;*

*}*

*}*

pre-JDK7:

*demo<Integer, String> dob = new demo<Integer, String>(98, “Example”);*

new syntax:

*demo<Integer, String> dob = new demo<>(98, “Example”);*

*<*> is called the diamond operator, which asks the compiler to infer the type arguments. General instantiation syntax:

*class-name<type-arg-list> var-name = new class-name<>(constructor-arg-list);*

Type inference in parameter passing

*<>* can be used when initiating a generic class object as a method parameter. For example, if the generic class *demo* has defined the method *meth-name(demo<T, V>)*, it can be called with

*dob.meth-name(new demo<>(1, “ex”));*

limits to type inference: cannot be used when assigning a subclass object to a superclass reference. Example:

*class A<T, V> {//…*

*class B<T, V> extends A<T, V> {//…*

*//demo<A<Integer, Long>, String> mcob = new demo<>(new B<Integer, Long>(), “error”);*

Erasure

Erasure allows compatibility between generics code and older non-generics code.

How erasure works: when code is compiled, all generic type information is erased and replaced with their bound type (Object is no bound provided) and then cast to the proper type. This means no type parameters exist at run time (only raw types).

Example

*Gen<Integer> iob = new Gen<Integer>(9);*

*int x = iob.get\_ob();*

is compiled as

*Gen iob = new Gen(9);*

*int x = (Integer) iob.get\_ob();*

Erasure also means *iob.getClass().getName()* returns *Gen*.

Bridge methods

A bridge method is used when an overriding method has 2 different type erasures in a superclass and subclass. In this case, a method is generated that uses the type erasure of the superclass, and that method calls the method that has the type erasure of the subclass method.

Ambiguity Errors

Ambiguity errors occur when erasure causes two distinct generic declarations to resolve to the same type.

Example:

*class Gen<T, V> {*

*T ob1;*

*V ob2;*

*//these overloaded methods raise an ambiguity error*

*void meth(T o) {*

*ob1 = o;*

*}*

*void meth(V o) {*

*ob2 = o;*

*}*

*}*

There are two ambiguity problems here:

1. it’s possible for *T* and *V* to be of the same type, thus making the 2 *meth()* identical.
2. erasure reduces both methods to be

*void meth(Object o)*

There is no easy fix to ambiguity error – the best solution is to avoid raising one in the first place, in this case by not overloading *meth()*.

Some Generic Restrictions

1. Type parameters cannot be instantiated, i.e. the following is illegal:

*new T();*

since the type of *T* is not resolved.

1. static members cannot use a type parameter, i.e. the following is illegal:

*static T ob;*

*static T getob() {return ob;}*

1. Generic array restriction: cannot instantiate an array whose element type is a type parameter. (but referencing an existing array is ok)
2. Generic array restriction: cannot create an array of type-specific generic references. Example:

*class gen<T extends numbers> {*

*T ob;*

*T vals[]; //ok*

*Gen(T o, T[] nums) {*

*ob = o;*

*//vals = new T[10]; //cannot create array of T*

*vals = nums; //ok to assign reference to existing array*

*}*

*}*

*class demo {*

*public static void main(String args[]) {*

*//Gen<Integer> gens[] = new Gen<Integer>[10]*

*//cannot create array of type-specific references*

*Gen<?> gens[] = new Gen<?>[10]*

*//ok to create array of wildcard references*

*}*

*}*

1. A generic class cannot extend *Throwable* – cannot create generic exception classes.

CH15

**Lambda Expressions**

Lambda expression – added in JDK8. Enhances java for 2 reasons:

1. adds new syntax elements and streamline the way certain constructs are implemented
2. new capabilities incorporated into the API library

Introduction

Lambda expression includes two constructs: lambda expression itself, and functional interface.

**Lambda Expression** – an unnamed method, used to implement a method defined by a functional interface rather than executed on its own. Also called a **closure**.

**Functional interface** – an interface that contains one and only one abstract method (it can also implement any public method defined by *Object* such as *equals()* ). Example: *Runnable* only defines *run()*. Usually represents a single action. A functional interface defines the *target type* of a lambda expression – a lambda expression can be used only in a context in which its target type is specified. Also called a *SAM (Single Abstract Method) type*.

Lambda expression fundamentals

The *Lambda/arrow operator*: *->* (pronounced “becomes” or “goes to”) divides the lambda expression. To the left are the parameters required by the lambda expression (can be empty); to the right is the lambda body, which specifies the actions. The body can be either a single expression or a block.

Example 1:

*() -> 12*

this is similar to the following method (but the lambda expression doesn’t have a name)

*int meth() {return 12;}*

Example 2:

*() -> Math.random() \* 100*

this return a random value from *Math.random()* and multiplies it by 100 (equivalent to

*int meth() {return Math.random()\*100;}*

Example 3:

*(n) -> (n & 2)==0*

this returns true if *n* is even. (equivalent to

*boolean meth(n) {return n&2 == 0;}*

The type of a parameter can be explicitly given or inferred. There is no limit on the number of parameters.

Functional interfaces

Prior to JDK 8, all interface methods are implicitly abstract; beginning with JDK 8, it is possible to specifiy default behavior for a method declared in an interface – a *default method*. Now an interface method is only abstract if it does not have a default implementation.

A lambda expression forms the implementation of the abstract method of a functional interface that specifies its target type. Therefore it can be specified only in a context in which a target type is defined. One way to create such a context is by assigning a lambda expression to a functional interface reference. Other target type contexts include variable initialization, *return* statements, and method arguments.

Example:

*interface MyNumber { //define a functional interface*

*double getValue();*

*}*

*class demo {*

*public static void main(String args[]) {*

*MyNumber mynum;*

*mynum = () -> 1.2; //assign a lambda expression to an interface reference*

*System.out.println(mynum.getValue() ); //displays 1.2*

*//mynum = () -> “1.2”; //illegal: String is not compatible with getValue()*

*}*

*}*

When a lambda expression occurs in a target type context, an instance of a class is automatically created that implements the functional interface, with the lambda expression defining the behavior of the functional interface’s abstract method. The lambda expression is executed when that method is called through the target.

In order for a lambda expression to work in a target type context, its type and the type of the abstract method must be compatible, and any exceptions thrown by the lambda expression must be acceptable to the method.

Example: lambda expression with a parameter

*interface NumericTest {*

*boolean test(int n);*

*}*

*class demo {*

*public static void main(String args[]) {*

*NumericTest eventest = (n) -> (n & 2)==0; //tests for if a number is even*

*//type of n is inferred as int*

*//this is also valid: (int n) -> (n & 2)==0;*

*if(eventest.test(10)) System.out.println(“10 is even”);*

*if(eventest.test(9)) System.out.println(“9 is not even”);*

*NumericTest negativetest = (n) -> n >= 0; //tests for if a number is non-negative*

*if(negativetest.test(10) System.out.println(“10 is positive”);*

*if(negativetest.test(-2)) System.out.println(“-2 is negative”);*

*}*

*}*

note: when a lambda expression has only one parameter, it does not need to be surrounded with parantheses on the left side of *->*, i.e. this is valid:

*n -> (n & 2)==0*

Example: lambda expression with 2 parameters

*interface NumericTest {*

*boolean test(int n, int d);*

*}*

*class demo {*

*public static void main(String args[]) {*

*NumericTest factortest = (n, d) -> (n % d)==0; //tests for if d is a factor of n*

*if(factortest.test(10, 2)) System.out.println(“2 is a factor of 10”);*

*}*

*}*

if the parameters’ types are to be declared, they must all be declared. Do this:

*(int n, int d) -> (n & d)==0;*

not this:

*(int n, d) -> (n % d)==0;*

Block Lambda Expressions

Lambda expressions with a single expression are sometimes called “expression lambdas”; the expression is called an “expression body”.

Lambda expressions with a block of expressions are called “block lambdas”; the expression block is called a “block body”.

Note that to return a value in a block body, the *return* keyword must be used. *return* causes a return from the lambda, not the method.

Example: computing the factorial of an int –

*interface NumericFn {*

*int fn(int n);*

*}*

*class demo {*

*public static void main(String args[]) {*

*NumericFn factorial = (n) -> {*

*int result = 1;*

*for(int i=1; i<=n; i++)*

*result = i \* result;*

*return result;*

*};*

*System.out.println(“13! = “ + factorial.fn(13));*

*}*

*}*

Example: reversing the characters in a String –

*interface StringFn {*

*String fn(String str);*

*}*

*class demo {*

*public static void main(String args[]) {*

*StringFn reverse = (n) -> { /\*the parameter in the lambda expression does not need to have the same name as the parameter in the abstract method\*/*

*String result = “”;*

*int i;*

*for(i = n.length() – 1; i >= 0; i--)*

*result += n.charAt(i);*

*return result;*

*};*

*System.out.println(reverse.fn(“Lambda”);*

*}*

*}*

Generic Functional Interface

A lambda expression itself cannot be generic, but functional interfaces can. In that case, the target type of the lambda expression is determined in part by the type argument(s) specified by the functional interface.

Example: in the last section, *NumericFn*’s and *StringFn*’s methods differ only in return type. A generic interface can handle both circumstances.

*interface SomeFn<T> {*

*T fn(T t);*

*}*

*class demo {*

*public static void main(String args[]) {*

*SomeFn<String> reverse = (n) -> {*

*String result = “”;*

*int i;*

*for(i = n.length() – 1; i >= 0; i--)*

*result += n.charAt(i);*

*return result;*

*};*

*System.out.println(reverse.fn(“Lambda”);*

*SomeFn<Integer> factorial = (n) -> {*

*int result = 1;*

*for(int i=1; i<=n; i++)*

*result = i \* result;*

*return result;*

*};*

*System.out.println(“13! = “ + factorial.fn(13));*

*}*

*}*

Passing Lambda Expressions as Arguments

A lambda expression can be passed in any context that provides a target type. One of these is when a lambda expression is passed as an argument. This is a common practice and gives a way to pass executable code as an argument to a method.

Requirement: the parameter receiving the lambda expression must be of a functional interface type compatible with the lambda.

Example:

*interface StringFn {*

*String fn(String n);*

*}*

*class demo {*

*static String stringop(StringFn sf, String s) {*

*return sf.fn(s);*

*}*

*/\* sf is of type StringFn, thus it can be assigned to a reference to any instance of StringFn, including the instance created by a lambda expression. \*/*

*public static void main(String args[]) {*

*String str1 = “example string”;*

*String str2;*

*//passing a single expression lambda*

*//this makes a string uppercase*

*str2 = stringop((str) -> str.toUpperCase(), str1);*

*System.out.println(str2);*

*//passing a block lambda*

*//this removes spaces*

*str2 = stringop((str) -> {*

*String result = “”;*

*int i;*

*for(i = 0; i < str.length(); i++)*

*if(str.charAt(i) != “ “*

*result +=str.charAt(i);*

*return result;*

*}, str1);*

*System.out.println(str2);*

*//can also create a lambda expression and pass its interface instance*

*StringFn reverse = (str) -> {*

*String result = “”;*

*int i;*

*for(i = n.length() – 1; i >= 0; i--)*

*result += n.charAt(i);*

*return result;*

*};*

*str2 = stringop(reverse, str1);*

*System.out.println(str2);*

*}*

*}*

When a lambda expression is passed as an argument, an instance of the functional interface is created and a reference to that object is passed to the method.

In addition to variable initialization, assignment and argument passing, the following also constitute target type contexts:

1. casts
2. the ? operator
3. array initializers
4. *return* statements
5. lambda expressions

Lambda Expressions and Exceptions

Lambda expressions can throw exceptions. However if the exception is checked, the abstract method in the functional interface must list it in its *throws* clause.

Example: computes the average of an array. Throws the custom exception *EmptyArrayException*.

*interface ArrayFn {*

*double avg(double[] n) throws EmptyArrayException;*

*}*

*class EmptyArrayException extends Exception {*

*EmptyArrayException() {*

*super(“Array is empty”);*

*}*

*}*

*class demo {*

*public static void main(String args[]) throws EmptyArrayException {*

*double[] values = {1.0, 2.0, 3.0 };*

*double[] zeros = new double[0];*

*ArrayFn average = (n) -> {*

*double sum = 0;*

*for(int i=0; i < n.length; i++)*

*sum += n[i];*

*return sum / n.length;*

*};*

*System.out.println(“The average is “ + average.avg(values));*

*System.out.println(“The average is “ + average.avg(zeros));*

*}*

*}*

Note: the parameter in the lambda expression is *n* rather than *n[]*, because the target context is *double[]* and the lambda expression infers from it. Alternatively declare it as *double[] n*.

Lambda Expressions and Variable Capture BMark

Lambda expressions have access to variables defined by the enclosing scope, including instances and *static* variables. They also have access to *this* (explicitly and implicitly), which refers to the invoking instance of the lambda expression’s enclosing class.

When a lambda expression uses a local variable from its enclosing scope, a special situation called *variable capture* is created. In this case, a lambda expression may only use local variables that are *effectively final*—one whose value does not change after being first assigned (they need not be *final*). The *this* parameter of an enclosing scope is automatically effectively final, and lambda expressions do not have a *this* of their own.

Important: a local variable of the enclosing scope cannot be modified by the lambda expression, else it would no longer be effectively final and no longer eligible for capture.

Note: a lambda expression can use and modify instance variables from its invoking class.

Example

*interface MyFn {*

*int fn(int n);*

*}*

*class demo {*

*public static void main(String args[]) {*

*int num = 10;*

*MyFn myfn1 = (n) -> {*

*int v = num + n; //this does not modify num and is ok*

*//num++; //this modifies num and is illegal*

*return v;*

*};*

*//num = 9; //this makes num no longer effectively final and would cause an error*

*}*

*}*

Method References

Method reference provides a way to refer to a method without executing it. It also requires a target type context consisting of a compatible functional interface. When evaluated, a method reference also creates an instance of the functional interface.

Method references to static methods

To create a *static* method reference, use the general syntax

*ClassName::MethodName*

The :: separator is added in JDK 8 expressively for this purpose.

Example:

*interface StringFn {*

*String fn(String n);*

*}*

*class StringOps {*

*static String StrReverse(String str) { //this reverses a String*

*String result = “”;*

*for (int i = str.length() – 1; i >= 0; i--)*

*result += str.charAt(i);*

*return result;*

*}*

*}*

*class demo {*

*static String SfOps(StringFn sf, String s) {*

*return sf.fn(s);*

*}*

*public static void main(String args[]) {*

*String str1 = “example string”;*

*String str2;*

*str2 = SfOps(StringOps::StrReverse, str1);*

*System.out.println(str2);*

*}*

*}*

where in *StringOps::StrReverse* evaluates to a reference to an object in which *StrReverse* provides the implementation of *fn()* in *StringFn*, since *StrReverse* is compatible with the *StringFn* interface.

Method references to instance methods //bookmark

To pass a reference to an instance method on a specific object, use the syntax

*objRef::MethodName*

Example: the previous example with this approach

*interface StringFn {*

*String fn(String n);*

*}*

*class StringOps {*

*static String StrReverse(String str) { //this reverses a String*

*String result = “”;*

*for (int i = str.length() – 1; i >= 0; i--)*

*result += str.charAt(i);*

*return result;*

*}*

*}*

*class demo {*

*static String SfOps(StringFn sf, String s) {*

*return sf.fn(s);*

*}*

*public static void main(String args[]) {*

*String str1 = “example string”;*

*String str2;*

*StringOps strops = new StringOps();*

*str2 = SfOps(strops::StrReverse, str1);*

*System.out.println(str2);*

*}*

*}*

One can also specify an instance method that can be used with any object of a given class, by using the syntax

*ClassName::instanceMethodName*

With this form, the first parameter of the functional interface matches to the invoking object and the second parameter matches the parameter specified by the method.

Example: *counter()* counts the number of objects in an array that satisfy the condition defined by *fn()*, in this case it counts instances of the *HighTemp* class.

*interface MyFn<T> {*

*boolean fn(T v1, T v2);*

*}*

*class HighTemp {*

*private int hTemp;*

*HighTemp(int ht) {hTemp = ht;}*

*//sameTemp() and lessTemp() are instance methods of HighTemp*

*boolean sameTemp(HighTemp ht2) {*

*//determines if two days have the same highest temperature*

*return hTemp == ht2.hTemp;*

*}*

*boolean lessTemp(HighTemp ht2) {*

*//determines if the invoking day has a lower highest temperature*

*return hTemp < ht2.hTemp;*

*}*

*}*

*class demo {*

*//returns the number of occurrences of an object in which fn() returns true*

*static <T> int counter(T[] vals, MyFn<T> f, T v) {*

*int count = 0;*

*for(int i=0; i<vals.length; i++)*

*if(f.fn(vals[i], v)) count++;*

*return count;*

*}*

*public static void main(String args[]) {*

*int count;*

*HighTemp[] WeekDayHighs = {new HighTemp(89), new HighTemp(82), new HighTemp(90), new HighTemp(80), new HighTemp(82), new HighTemp(84), new HighTemp(83) };*

*count = counter(WeekDayHighs, HighTemp::sameTemp, new HighTemp(89));*

*System.out.println(count + “ days had a high of 89”);*

*count = counter(WeekDayHighs, HighTemp::lessTemp, new HighTemp(89));*

*System.out.println(count + “ days had a high less than 89”);*

*}*

*}*

Both *sameTemp()* and *lessTemp()* are compatible with *MyFn* since they both return a boolean type and takes a *HighTemp* type parameter.

The invoking object type can be mapped to the first parameter of *fn()* and the argument mapped to the second parameter. i.e. in *HighTemp::sameTemp*, an instance of the functional interface *MyFn* is created in which the parameter type of the first parameter is *HighTemp* (the type of the invoking object) and that of the second parameter is *HighTemp* (the type of *sameTemp()*’s parameter).

Aside: one can refer to the superclass version of a method with *super*, as in

*super::method-name*

Method references with generics

When a generic method is specified as a method reference, and the method belongs in a non-generic class, use the syntax

*class-name::<type>method-name*

it’s not required to explicitly provide a type parameter, it could be inferred.

If the class is generic, use the syntax

*class-name<type1>::<type2>method-name*

example: finding the largest element in a collection. Can use *max()* defined by the *Collections* class: in this version of *max()*, it needs a reference to the collection and an instance of an object that implements the *Comparator<T>* interface. *Comparator* implements *compare()*, which takes 2 arguments – the 2 objects being compared.

Note: *ArrayList* is a class in the *java.util* package. It creates a resizable array. It has the methods *add()* adds an element to the array; *get(index)* returns the element at the given index; *set(index, content)* sets the element at the given index; *remove(index)* removes an element; *size()* returns the array size.

*import java.util.\*;*

*class MyClass {*

*private int val;*

*MyClass(int v) {val = v;}*

*int getVal() {return val;}*

*}*

*class demo {*

*static int compareMC(MyClass a, MyClass b) {*

*return a.getVal() – b.getVal();*

*}*

*/\* this is compatible with the compare() method defined by comparator<T>. \*/*

*public static void main(String args[]) {*

*ArrayList<MyClass> al = new ArrayList<MyClass>();*

*al.add(new MyClass(1));*

*al.add(new MyClass(2));*

*al.add(new MyClass(4));*

*al.add(new MyClass(10));*

*al.add(new MyClass(7));*

*MyClass maxVal = Collections.max(al, demo::compareMC);*

*System.out.println(“max value is: “ + maxVal.getVal());*

*}*

*}*

Pre-JDK 8, this example requires creating a class that implements *Comparator<T>* and then create an instance, and pass that instance to *max()*. But here *MyClass* neither defines any comparison method, nor does it implement *Comparator*.

Constructor References

References to constructors can be created with the general syntax

*class-name::new*

This reference can be assigned to any functional interface reference that defines a method compatible with the constructor.

Example:

*interface MyFn {*

*MyClass fn(int n);*

*}*

*class MyClass {*

*private int val;*

*MyClass(int v) {val = v; }*

*MyClass() {val = 0; }*

*int getVal() {return val; }*

*}*

*class demo {*

*public static void main(String args[]) {*

*MyFn MCcons = MyClass::new;*

*//because fn() take one parameter, new refers to the parameterized constructor in MyClass*

*MyClass mc = MCcons.func(100);*

*System.out.println(“val in mc is “ + mc.getVal());*

*}*

*}*

*MCcons.func(100)* creates an instance of *MyClass*. In essence, *MCcons* becomes another way to call *MyClass(int v)*.

Constructor references to generic classes are created in the same fashion:

*class-name<type>::new*

Constructor references to arrays have the syntax

*type[]::new*

Example (using code from previous example)

*interface ArrayCreator<T> {*

*T fn(int n);*

*}*

*//…*

*ArrayCreator<MyClass[]> ArrayCons = MyClass[]::new;*

*MyClass[] a = ArrayCons.fn(2);*

*a[0] = new MyClass(1);*

*a[1] = new MyClass(2);*

*fn(2)* causes a two element array to be created, and the following gives each element an initial value.

Predefined Functional Interfaces

JDK 8 adds a new package *java.util.function* that provides several predefined functional interfaces. Examples:

|  |  |
| --- | --- |
| Interface | Explanation |
| UnaryOperator<T> | Apply a unary operation to a type *T* object and return a type *T* result. Method is *apply()*. |
| BinaryOperator<T> | Apply an operation to 2 type *T* objects and return a type *T* result. Method is *apply()*. |
| Consumer<T> | Apply an operation on a type *T* object. Method is *accept()*. |
| Supplier<T> | Return a type *T* object. Method is *get()*. |
| Function<T, R> | Apply an operation to an object of type *T* and return a type *R* result. Method is *apply()*. |
| Predicate<T> | determine if a type *T* object fulfills some constraint, returns a *boolean*. Method is *test()*. |

Example: using the *Function* interface to compute factorials

*import java.util.function.Function;*

*class demo {*

*public static void main(String args[]) {*

*Function<Integer, Integer> factorial = (n) -> {*

*int result = 1;*

*for(int i = 1; i <= n; i++)*

*result = i \* result;*

*return result;*

*};*

*System.out.println(“The factorial of 13 is “ + factorial.apply(13));*

*}*

*}*

CH16

**Modules**

Added in JDK 9, modules give a way to describe the relationships and dependencies of codes, as well as control which parts of a module are accessible to other modules.

Generally speaking modules are most helpful to large applications by reducing the management complexity.

The Java API library has also been organized into modules, making it possible to specify which parts of the API are required by the program (useful for reducing the run-time footprint).

Module Basics

A *module* is a grouping of packages and resources. A module declaration specifies the name of a module and defines the relationship a module and its packages have to other modules. The module keywords are

1. *exports; module; open; opens; provides; requires; to; transitive; uses; with*

These are only keywords in the context of module declarations. (formally these are called *restricted keywords*)

A module declaration is contained in a file called *module-info.java*. It is then compiled by *javac* into a class file known as its *module descriptor*. *module-info.java* can only contain a module definition, not other declarations.

Module declaration syntax:

*module module-name {*

*//module definition }*

The definition can be empty, which results in a declaration that simply names the module.

Two Key Features of Modules

1. a module’s ability to specify that it requires another module, by use of a *requires* statement. This is checked at both compile and run time.
2. a module’s ability to control which, if any, of its packages are accessible by another module, by use of an *exports* statement. *public* and *protected* types within a package are accessible to other modules if (and only if) they are explicitly exported.

**Example application** – contains 2 modules: *appstart* contains a package called *appstart.appdemo* containing the application’s *main()* method; 2nd module *appfn* contains a package called *appfn.simplefn* containing static math methods. The entire application is contained in a directory tree beginning with *modapp*.

note1: prefixing the module’s name is not required.

note2: (to make sure names are unique) suggested naming scheme: reverse domain name (e.g. a project associated with *goggle.com* would use *com.goggle* as the module prefix and package names).

Creating a module in Eclipse:

1. create a project with the module name (Java Project)
   * modules that depend on each other uses the same package
   * go to *Project layout > Create separate folders for sources and class files > configure default… (Preferences (Filtered) ) >*
     + *Folders > Source folder name: src; Output folder name: modules/module-name(example) > apply and close*
   * Next*:* check *Create module-info.java file* > finish
2. create the main class
3. create *module-info.java*
   * right click on project and go to **configure > create module-info.java**
   * by default, all packages in a module are exported
4. to import another module, need to add its project in build-path. Go to Property of project > Java Build Path > Project tab > Modulepath > add the desired module

Source file for *SimpleFunction*  (located in *src\appfn\appfn\simplefn*):

**package** modul\_function.SimpleFunction;

**public** **class** SimpleFunction {

**public** **static** **boolean** isFactor(**int** a, **int** b) {

//returns true if the first parameter is a factor of the second

**if**((b%a) == 0) **return** **true**;

**return** **false**;

}

**public** **static** **int** minCommonFactor(**int** a, **int** b) {

a = Math.*abs*(a); //abs() returns the absoluate value of its parameter

b = Math.*abs*(b);

**int** min = a<b ? a : b; //min is the smaller one of a or b

**for**(**int** i = 2; i<= min/2; i++) {

//stops at min/2 to avoid running unnecessary loops

**if**(*isFactor*(i, a) && *isFactor*(i, b))

**return** i;

}

**return** 1;

}

**public** **static** **int** maxCommonFactor(**int** a, **int** b) {

a = Math.*abs*(a);

b = Math.*abs*(b);

**int** min = a<b ? a:b;

**for**(**int** i = min/2; i >= 2; i--) {

**if**(*isFactor*(i,a) && *isFactor*(i,b))

**return** i;

}

**return** 1;

}

}

Source file for *Start*

**package** modul.start;

**import** modul\_function.SimpleFunction;

**public** **class** Start {

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

**if**(SimpleMathFn.*isFactor*(2, 10))

System.***out***.println("2 is a factor of 10");

System.***out***.println("smallest common factor of 35 and 105 is " + SimpleMathFn.*minCommonFactor*(35, 105));

System.***out***.println("greatest common factor of 35 and 105 is " + SimpleMathFn.*maxCommonFactor*(35, 105));

}

}

Next add *module-info.java* for each module. These are located one above the package directory.

for *modul.function*:

**module** modul.function {

**exports** modul\_function.SimpleFunction;

}

for *appstart*

**module** modul.start {

**requires** modul.function;

}

Compiling and running module-based programs

* Use *javac*
* need to explicitly specify a module path.
  + A module path tells the compiler where the compiled files will be located
* execute *javac* at the top level directory
* specify where to put the output *.class* file with *-d*, i.e.

*javac -d [output location] [full path to the .java file to compile]*

* + can compile multiple files at once:

*javac -d [output location] [full path to first .java] [full path to second .java]*

usually compilethe source file and *module-info.java* at once thie way

* for modules that depend on another module: need to specify module path. Do so with *--module-path [module-name]*
* run the app with *java*:

*java --module-path [module-name] -m [full path to .java file]*

* *-m* specifies the name of the main class

Closer look at *requires* and *exports*

*requires* has the general form

**requires** module-name;

* The module with the *requires* statement is said to *read* the module required by it.
* When multiple modules are required, they must be specified in separate *requires* statements.

*exports* has the general form

**exports** package-name;

* Exporting makes all public and protected types and members in the package accessible to other modules.
* Without exporting, a package is inaccessible to other modules, including *public* types and members.
* When multiple packages are exported, each needs its own *exports* statement.

*requires* and *exports* work together: both must be present in order to work

*requires* and *exports* can only occur within a *module* statement, and moreover a *module* statement must occur in a file called *module-info.java*.

*java.base* and the Platform Modules

The API modules are referred to as *platform modules*, and their names all begin with *java*. Of the platform modules, the most important is *java.base*, which includes and exports fundamental packages such as *java.lang* and *java.io*. *java.base* is automatically accessible to all modules, and all modules automatically require *java.base*.

By modularizing the API, it becomes possible to deploy an application with only the packages that it requires, rather than the entire JRE.

Legacy Code and the Unnamed Module

Backward compatibility of legacy code is achieved with 2 features:

1. The *unnamed module*.

When one uses code that is not part of a named module, it automatically becomes part of the unnamed module.

All packages in the unnamed module are automatically exported.

The unnamed module can access any other modules.

1. Automatic use of the class path (rather than module path).

Exporting to a Specific Module

Adding a *to* clause to the *exports* statement makes a package accessible to only a specific set of modules. This is known as a *qualified export*. General form:

**exports** package-name **to** module-names;

where the module names are comma separated.

If a qualified export is used, compiling the *.java* requires a *--module-source-path* clause:

*javac -d [destination] –module-source-path [name of folder containint the module (src)] [full path to .java]*

which automatically compiles the files under the directory

This way of using *javac* is called *multi-module mode* because it allows more than one module to be compiled at once. This can be useful for avoiding errors since the exporting and requiring modules are compiled at the same time.

requires transitive

Given 3 modules A, B, C, with the dependencies

* A requires B
* B requires C

with the *module-info.java* files for A and B

**module** A {

**requires** B;

}

**module** B {

**exports** package-name;

**requires** C;

}

This is fine if A does not need access to a type defined in C. But if it does, there are 2 solutions:

1. add a **requires** C statement to A’s file
2. add the *transitive* keyword after *requires*, i.e.

**module** B {

**exports** package-name;

**requires transitive** C;

}

**transitive** makes any module that depends on B automatically depend on C.

This is helpful if many modules requires B as option 1 would have to be repeated many times.

Note: **transitive** will be interpreted as an identifier if a separator immediately follows **transitive**.

Services

In programming there is the **what** and the **how**. For example, the interface specifies the **what**, whereas the implementing class specifies the **how**. Another way Java does so is with *plug-ins*, through the use of *services* and *service providers*.

Note: the applications that use services and service providers are typically fairly sophisticated, and service-based features are often not needed.

Basics

In Java, a *service* is a program unit whose functionality is defined by an interface or abstract class. A concreate implementation of a service is supplied by a *service provider*. A service defines the form of some action, the service provider supplies that action.

Service providers are supported by the *ServiceLoader* class, packaged in *java.util*. It is declared with

*class service Loader<S>*

where *S* specifies the service type. The *load()* method loads service providers. The form used here is

*public static <S> ServiceLoader<S> load(Class <S> service-type)*

where *service-type* is a *Class* object for the desired service type. Obtain it with *class-name.class*.

Calling *load()* returns a *ServiceLoader* instance.

Service-based Keywords

The service based keywords are **provides**, **uses**, **with**. A module specifies that it provides a service with a **provides** statement, that it requires a service with a **uses** statement. The specific type of service provider is declared with **with**.

The general form of **provides**:

**provides** service-type **with** implementation-types;

The service type is often an interface or abstract class. The implementation types are comma separated.

The general form of **uses**:

**uses** service-type;

Service Example

Using the previous example, adding 2 additional modules under *src*: *user.functions* and *user.function.implementation*. The service will supply functions that take two *int* arguments.

The service interfaces

Two service related interfaces are needed: one specifies the form of an action, one specifies the form of the action provider. Both are in the *user-functions.binary-functions* package.

The first, *BinaryFunctions*, declares the form of a binary function:

**package** user.binary.functions;

**public** **interface** BinaryFunctions {

**public** String get\_name(); //obtains the name of the function

**public** **int** func(**int** a, **int** b); //specifies the function to perform

}

The second, *FunctionProvider*, declares the form of the service provider.

**package** user.binary.functions;

**import** user.binary.functions.BinaryFunctions;

**public** **interface** FunctionProvider {

**public** BinaryFunctions get();

}

The Implementation Classes

There are two implementations for *BinaryFunctions*: *AbsPlus* and *AbsMinus*, which returns the sum or difference of the absolute values of the arguments.

The code for *AbsPlus*:

**package** user.binary.functions;

**import** user.binary.functions.BinaryFunctions;

**public** **class** AbsPlus **implements** BinaryFunctions {

@Override

**public** String get\_name() {

**return** "AbsPlus";

}

@Override

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

**return** Math.*abs*(a)+Math.*abs*(b);

}

}

The code for *AbsMinus*:

**package** user.binary.functions;

**import** user.binary.functions.BinaryFunctions;

**public** **class** AbsMinus **implements** BinaryFunctions {

@Override

**public** String get\_name() {

**return** "AbsMinus";

}

@Override

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

**return** Math.*abs*(a) - Math.*abs*(b);

}

}

*AbsPlusProvider* provides an instance of *AbsPlus*:

**package** user.binary.functions;

**public** **class** AbsPlusProvider **implements** FunctionProvider {

@Override

**public** BinaryFunctions get() {

**return** **new** AbsPlus();

}

}

*AbsMinusProvider* provides an instance of *AbsMinus*:

**package** user.binary.functions;

**public** **class** AbsMinusProvider **implements** FunctionProvider {

@Override

**public** BinaryFunctions get() {

**return** **new** AbsMinus();

}

}

The module definitions

The *user.functions* module has the definition

**module** user.functions {

**exports** user.binary.functions;

}

The *user.function.implementation* module has the definition

**module** user.function.implementation {

**requires** user.functions;

**provides** user.binary.functions.FunctionProvider **with** user.function.implementation.AbsPlusProvider, user.function.implementation.AbsMinusProvider;

}

*user.functions* is required because it contains *BinaryFunctions* and *FunctionProvider*.

Updated main() file

**package** modul.start;

**import** modul\_function.SimpleFunction.\*;

**import** java.util.ServiceLoader;

**import** user.binary.functions.\*;

**public** **class** Start {

**public** **static** **void** main(String[] args) {

**if**(SimpleFunction.isFactor(2, 10))

System.out.println("2 is a factor of 10");

System.out.println("smallest common factor of 35 and 105 is " + SimpleFunction.minCommonFactor(35, 105));

System.out.println("greatest common factor of 35 and 105 is " + SimpleFunction.maxCommonFactor(35, 105));

ServiceLoader<FunctionProvider> prov = ServiceLoader.load(FunctionProvider.**class**);

//all classes that implement FunctionProvider will be found

BinaryFunction bf = **null**;

//find the provider for AbsPlus and obtain the function

**for**(FunctionProvider fp : prov) {

**if**(fp.get().getName().equals("AbsPlus")) {

bf = fp.get();

**break**;

}

}

**if**(bf != **null**)

System.out.println(bf.func(12,-4));

**else**

System.out.println("AbsPlus not found");

bf = **null**;

//find the provider for AbsMinus and obtain the function

**for**(FunctionProvider fp: prov) {

**if**(fp.get().getName().equals("AbsMinus")) {

bf = fp.get();

**break**;

}

}

**if**(bf != **null**)

System.out.println(bf.func(12,-4));

**else**

System.out.println("AbsMinus not found");

}

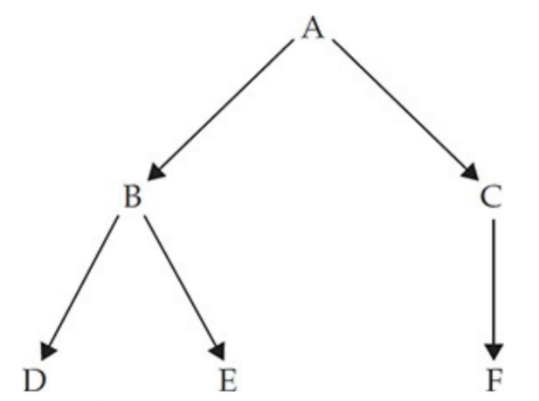
}

furthermore, the module-info file for *Start* also needs *requires user.functions* and *uses binary.user.function.FunctionProvider* added to it.

Module Graphs

Module graphs are created by the compiler to resolve the dependence relationships between modules.

Example: 6 modules A, B, C, D, E, F. A requires B and C, B requires D and E, C requires F. The module graph is



The arrows point from the dependent module to the required module.

Three Specialized Module Features

1. The *open* module

Normally, the types in a module’s packages are only accessible if explicitly exported. However in an *open* module, all packages are accessible at run time; only explicitly exported packages are accessible at compile time. This allows the packages in the module to be accessed through reflection.

*open* module is declared with

**open** **module** moduleName {

//module definition

}

1. The *opens* statement

A module can open a specific package to run-time access with an *opens* statement:

**opens** packageName;

Additionally, there can be a *to* clause to specify which modules to open the package(s) to.

Like the *open* module, *opens* statement also only gives run-time access but not compile time. It is possible to both export and open a module.

1. *requires static*

Normally, *requires* specifies a dependence that is enforced both at compile and run time. It is possible to relax that requirement to not require a module at run time, by using the *static* modifier in a *requires* statement. General form:

**requires** **static** moduleName;

*jlink* and Module JAR Files

The command *jlink* creates a run time image from unarchived modules.

JAR stands for Java ARchive, and is a file format typically used for application deployment.

P2

CH17

**String Handling**

String objects cannot be edited; however there are 2 options for modifiable strings: *StringBuffer* and *StringBuilder*. These 3 classes are defined in *java.lang*, are *final*, and implement the *CharSequence* interface.

String Constructors

* String()
* String(**char** chars[])

Note: *String(‘abc’)* equivalent to *char chars[] = {‘a’, ‘b’, ‘c’ }; String(chars)*

* String(**char** chars[], **int** StartIndex, **int** NumChars)

where *StartIndex* specifies the index at which the String begins, and *NumChars* specifies the number of characters to use. Example:

**char** chars[] = {'a','b','c','d','e'};

String s = **new** String(chars, 2, 3);

initializes the String with “cde”.

* String(String str)

clones a String object.

* String(**byte** asciichars[])
* String(**byte** asciichars[], **int** StartIndex, **int** NumChars)

initialize a String from 8-bit bytes constructed from the ASCII character set. (Strings on the Internet typically use this)

* String(StringBuffer strbf)
* String(StringBuilder strbd)

construct a String from a *StringBuffer* or *StringBuilder*.

* String(**int** CodePoints[], **int** StartIndex, **int** NumChars)

construct a String from an array of Unicode code points.

* Additionally, there are constructors that specify a *Charset*.

Special String Operations

* String literals – example:

String str = "abc";

**int** i = "abc".length();

* String concatenation – example:

String str = "a" + "b";

* Concatenation with other types

String concatenation can be done with other data types. When it happens that data is automatically converted into String. Example:

**int** a = 2;

String str = "a" + "b" + a;

String operator “+” has precedence over math operator “+”. Thus, String str2 = "two" + 2 + 2; produces “two22”.

* String Conversion and *toString()*

During String concatenation, types are converted to String by calling *valueOf()* defined by *String*. For primitive types, *valueOf()* returns a string representation of the type; for objects, *valueOf()* calls *toString()* on the object. *toString()* has the general form

String toString();

Character Extraction

* *char charAt(int index)*

extracts a character from a String at the given index.

* *void getChars(int source\_start, int source\_end, char target[], int target\_start)*

allows more than one character to be extracted. *source\_start* specifies the beginning of the substring, *source\_end* – 1 specifies the end of the substring. *target* will receive the characters (it must be big enough to hold the characters). *target\_*start is the index of *target* that will begin receiving the characters. Example:

String str = "example String for getChars method";

**int** start = 19;

**int** end = 27;

**char** target[] = **new** **char**[end - start];

str.getChars(start, end, target, 0);

System.***out***.println(target);

(returns “getChars”)

* *byte[] getBytes()*

this is an alternative to *getChars()* that stores the characters in an array of bytes, using the default character-to-byte conversion of the platform. Useful when exporting a String value to an environment that does not support 16-bit Unicode characters.

* *char[] toCharArray()*

converts all characters in a String object into a character array. This returns an array of characters for the entire string.

String Comparison

* *boolean equals(Object str)*

Compares *str* with the invoking object. Case sensitive.

* *boolean equalsIgnoreCase(String str)*

Ignores cases.

* *boolean regionMatches(int startIndex, String str, int strStartIndex, int numChars)*
* *boolean regionMatches(boolean ignoreCase, int startIndex, String str, int strStartIndex, int numChars)*

*regionMatches()* compares a specific region in side a string with another specific region in another string. *startIndex* specifies the index at which the region begins within the invoking object; *strStartIndex* specifies the index at which the region begins within the target string; *numChars* specifies the length of the substring being compared.

* *boolean startsWith(String str)*
* *boolean endsWith(String str)*
* *boolean startsWith(String str, int startIndex)*

determine if the invoking String begins/ends with *str*. Second form of *startsWith()* specifies the index at which point the comparison begins.

* *equals()* versus ==

*equals()* compares the characters within 2 Strings; == compares 2 object references to see whether they refer to the same instance.*­* Example:

String str = "example";

String str2 = **new** String(str);

**boolean** b = str.equals(str2);

**boolean** b2 = str ==str2;

System.***out***.println(b); //outputs true

System.***out***.println(b2); //outputs false

* *int compareTo(String str)*
* *int compareToIgnoreCase(String str)*

Determines if the start of a string is less than, equal to, or greater than the start of another. (A string is less than another if it comes before the other in dictionary order.) *compareTo()* outputs negative if the invoking string is less than *str*. *compareToIgnoreCase()* ignores cases.

Example: sorting an array of strings in ascending order

String list[] = {"babble", "azzzz", "demonstration", "Zed", "example", "c", "zed" };

**for**(**int** j = 0; j < list.length; j++) {

**for**(**int** i = j + 1; i < list.length; i++) {

**if**(list[i].compareTo(list[j]) < 0) {

String t = list[j];

list[j] = list[i];

list[i] = t;

}

}

System.***out***.println(list[j] + ", ");

//outputs "Zed, azzzz, babble, c, demonstration, example, zed,"

}

Searching Strings

The String class has two methods for searching a string for a specified character or substring:

* *int indexOf(int ch)*
* *int indexOf(String str)*

searches the first occurrence of a character or substring.

* *int lastIndexOf(int ch)*
* *int lastIndexOf(String str)*

searches the first occurrence of a character or substring.

*ch* or *str* is the character/string being sought. Both methods return the index at which the character or substring was found, or -1 if not found.

* *int indexOf(int ch, int startIndex)*
* *int lastIndexOf(int ch, int startIndex)*

*startIndex* specifies the index at which point the search begins. for *indexOf()*, the search goes from *startIndex* to end of string; for *lastIndexOf()*, the search goes from *startIndex* to zero.

Modifying a String

To modify a String, one can copy it into a *StringBuffer* or *StringBuilder*, or use one of the String methods that construct a modified copy of the string.

* *String substring(int startIndex)*
* *String substring(int startIndex, int endIndex)*

*substring()* extracts a substring. *startIndex* specifies the index at which the substring will begin, and the optional *endIndex* specifies the index where the substring ends (without it, the substring runs to the end of the invoking string).

Example: replacing every “is” with “was” within a string

String str = "this is a test";

String search = "is";

String sub = "was";

String result = "";

**int** i;

**do** {

System.***out***.println(str);

i = str.indexOf(search);

**if**(i != -1) {

result = str.substring(0,i);

result = result + sub;

result = result + str.substring(i + search.length());

str = result;

}

} **while**(i != -1);

* *String concat(String str)*

creates a new string of the invoking string with *str* appended to the end. Functionally identical to +.

* *String replace(char original, char replacement)*

replaces all occurrences of *original* with *replacement*.

* *String replace(charSequence original, charSequence replacement)*

replaces one character sequence with another.

* *String trim()*

returns a copy of the invoking string with leading and trailing white spaces removed. (spaces are characters with a value of 32 or less, includes space, tab, carriage return, line feed, etc.)

* *String stripLeading()*
* *String striptrailing()*

delete white spaces at the beginning/end of the invoking string. (added in JDK 11)

* String strip()

deletes all white spaces from the invoking string. (added in JDK11)

Data Conversion Using *valueOf()*

*valueOf()* converts data from internal format to a human readable form (a String). It is a static method that is overloaded for all built-in types, so its parameter can be any primitive type.

For most arrays, *valueOf()* returns something unintelligible, but for arrays of *char*, a String object is created that contains the characters in the array. A special version of *valueOf()* allows the specification of a subset of a *char* array, with the general form

*static String valueOf(char chars[], int startIndex, int numChars)*

Changing the Case of Characters Within a String

* *String toLowerCase()*
* *String toUpperCase()*

Convert all characters in a string to lowercase/uppercase.

Joining Strings *non-existent?*

JDK added a new method to String called *join()*, which concatenates 2 or more strings and separating them with a delimiter. It has 2 forms:

* *static String join(CharSequence delim, CharSequence strs)*

where delim specifies the delimiter, *strs* (which is one or more parameters) is the list of character sequences. Since String implements the *CharSequence* interface, *strs* can be a list of strings.

* the second form joins a list of strings obtained from an *Iterable* interface reference.

Additional String Methods

|  |  |
| --- | --- |
| Method | Description |
| *int codePointAt(int i)* | Returns the Unicode code point at the location specified by *i*. |
| *int codePointBefore(int i)* | Returns the Unicode code point at the location that precedes *i*. |
| *int codePointCount(int start, int end)* | Returns the number of code points in the invoking String between *start* and *end*-1. |
| *boolean contains(CharSequence str)* | Returns true if the invoking object contains the content of *str*. |
| *boolean contentEquals(CharSequence str)* | Returns true if the invoking string contains the same string as *str*. |
| *boolean contentEquals(StringBuffer str)* |  |
| *static String format(String fmtstr, Object …args)* | Returns a string formatted as specified by *fmtstr*. |
| *static String format(Locale loc, String fmtstr, Object …args)* |  |
| *boolean isEmpty()* |  |
| *Stream<String> lines()* | Decomposes a string into individual lines, and returns a *Stream* containing the lines (JDK11). |
| *boolean matches(String regExp)* | Returns true if the invoking string matches the regular expression passed in *regExp*. |
| *int offsetByCodePoints(int start, int num)* | Returns the index within the invoking string that is *num* code points beyond the starting index specified by *start*. |
| *String replaceFirst(String regExp, String newStr)* | Returns a string in which the first substring that matches the regular expression specified by *regExp* is replaced by *newStr*. |
| *String replaceAll(String regExp, String newStr)* | Replaces all regular expressions matching *regExp* with *newStr*. |
| *String[] split(String regExp)* | Decomposes the invoking string into parts and returns it as an array. *regExp* delimits the parts. |
| *String[] split(String regExp, int max)* | *max* specifies the number of parts. If negative, the string is fully decomposed; if zero, the string is fully decomposed but no trailing empty strings will be included. |
| *CharSequence subsequence(int startIndex, int stopIndex)* | Returns a substring of the invoking string, starting at *startIndex* and ending at *stopIndex*. |

*StringBuffer*

*StringBuffer* supports a modifiable string.

Constructors

* *StringBuffer()*
* *StringBuffer(int size)*
* *StringBuffer(String str)*
* *StringBuffer(CharSequence chars)*

The default constructor reserves room for 16 characters without reallocation (this is done as reallocation is a costly process, and frequent reallocations can fragment memory). The 3rd and 4th constructors initialize the *StringBuffer* and add room for 16 more characters.

*length()* and *capacity()*

*length()* returns the current length of a *StringBuffer*, whereas *capacity()* returns the total allocated capacity.

*ensureCapacity()*

Preallocates room for a certain amount of characters. Has general form

*void ensureCapacity(int minCapacity)*

*setLength()*

Sets the length of the string within a *StringBuffer* object. When size is increased, null characters may be added to the end, when size is decreased, characters at the end of the string may be lost. Has general form

*void setLength(int len)*

*charAt()* and *setCharAt()*

* *void charAt(int where)*
* *void setCharAt(int where, char ch)*

obtain the value of a single character at index *where*, or replace it with *ch*.

*getChars()*

* *void getChars(int sourceStart, int sourceEnd, char target[], int targetStart)*

copies a substring of a *StringBuffer* into an array. The substring contains the characters from *sourceStart* to *sourceEnd-1*. *target* will receive the characters (it must be large enough to hold the characters), starting at index *targetStart*.

*append()*

* *StringBuffer append(Object obj)*

Returns the stringbuffer with the appended type. Accepts any type of data and appends it to the invoking *StringBuffer* object.

can call multiple times to append multiple objects, e.g.

*StringBuffer sb = new StringBuffer(40);*

*int a = 3;*

*String s = sb.append(“4 = ”).append(a).append(“ + 1”).toString();*

*insert()*

* *StringBuffer insert(int index, Object obj)*

inserts a string into another and returns the result. Accepts all primitive types plus *String*, *Object,* and *CharSequence*.

*reverse()*

* *StringBuffer reverse()*

returns the reverse of the calling object.

*delete()* and *deleteCharAt()*

* *StringBuffer delete(int startIndex, int endIndex)*
* *StringBuffer deleteAt(int loc)*

deletes the characters from *startIndex* to *endIndex—1*, or the character at index *loc*.

*replace()*

* *StringBuffer replace(int startIndex, int endIndex, String str)*

replaces the substring from *startIndex* through *endIndex-1* with *str*.

*substring()*

* *String substring(int startIndex)*
* *String substring(int startIndex, int endIndex)*

returns the substring from *startIndex* to the end of invoking *StringBuffer* or to *endIndex-1*.

Additional methods

|  |  |
| --- | --- |
| Method | Description |
| *StringBuffer appendCodePoint(int ch)* | Appends a Unicode code point to the end of the invoking object and returns it. |
| *int codePointAt(int i)* | Returns the Unicode code point at the location specified by *i*. |
| *int codePointBefore(int i)* |  |
| *int codePointCount(int start, int end)* | Returns the number of code points between *start* and *end-1*. |
| *int indexOf(String str)* | Searches for the first occurrence of *str*. |
| *int indexOf(String str, int startIndex)* | Search for the first occurrence of *str* starting from *startIndex*. |
| *int lastIndexOf(String str)* | Searches for the last occurrence of *str*. |
| *int lastIndexOf(String str, int startIndex)* | Searches for the last occurrence of *str* starting from *startindex*. |
| *int offsetByCodePoints(int start, int num)* |  |
| *CharSequence subSequence(int startIndex, int stopIndex)* | Returns a substring of the invoking string, beginning at *startIndex* and stopping at *stopIndex*. |
| *void trimToSize()* | Reduces the StringBuffer’s size to better fit the current contents. |

*StringBuilder*

*StringBuilder* is similar to *StringBuffer* except for one important difference: it is not synchronized. Its advantage is faster performance.

CH18

**Exploring *java.lang***

*java.lang* is automatically imported into all programs, and contains classes and interfaces fundamental to Java programming. Beginning with JDK 9 , *java.lang* is part of the *java.base* module.

Primitive Type Wrappers

Some methods (many not listed here)

*byteValue()* returns the value of the invoking object as a byte. (similar method for other primitive wrapper types)

|  |  |
| --- | --- |
| **Methods Defined by *String*** | **Description** |
| *static boolean isFinite(double num)* | returns true if *num* is not NaN or infinite. |
| *boolean isInfinite()* | returns true if the invoking object contains an infinite value. |
| *static boolean isInfinite(double num)* |  |
| *boolean is NaN()* | returns true if the invoking object contains an value that is not a number (example: 0/0.) |
| *static boolean isNaN(double num)* |  |
| *static double longBitsToDouble(long num)* |  |
| *static StringtoHexString(double num)* |  |

|  |  |
| --- | --- |
| **Methods Defined by *Byte* and *Short*** | **Description** |
| *static int compare(byte num1, byte num2)* | returns *num1 – num2*. |
| *int compareTo(Byte b)* | returns ­*invokingObject – b*. |
| *static int compareUnsigned(byte num1, byte num2)* |  |
| *static Byte decode(String str) throws NumberFormatException* |  |
| *int hashCode()* | returns the hash code for the invoking object. |
| *static int hashCode(byte num)* | returns the hash code for *num*. |
| *static byte parseByte(String str) throws NumberFormatException* |  |
| *static byte parseByte(String str, int radix) throws NumberFormatException* |  |
| *static int toUnsignedInt(byte val)* |  |

|  |  |
| --- | --- |
| **Methods Defined by *Integer* and *Long* (in addition to ones above) (replace *int* with *long* for *long* version)** | **Description** |
| *static int divideUnsigned(int dividend, int divisor)* | returns *dividend/divisor* as an unsigned value. |
| *static Integer getInteger(String propertyName)* | returns the value associated with the environmental property specified by *propertyName*. Returns null on failure. |
| *static Integer getInteger(String propertyName, int default)* |  |
| *static int highestOneBit(int num)* | Returns the position of the highest order bit in *num*. |
| *static int lowestOneBit(int num)* |  |
| *static int max(int val, int val2)* | returns the largest of *val* and *val2*. |
| *static int min(int val, int val2)* |  |
| *static int numberOfLeadingZeros(int num)* |  |
| *static int numberOfTrailingZeros(int num)* |  |
| *static int parseInt(CharSequence chars, int startIndex, int stopIndex, int radix) throws NumberFormatException* | returns the integer equivalent of the numbers contained in *chars* between *startIndex* and *stopIndex-1*, using *radix*. |
| *static int parseInt(CharSequence chars) throws NumberFormatException* |  |
| *static int parseUnsignedInt(CharSequence chars, int startIndex, int stopIndex, int radix) throws NumberFormatException* |  |
| *static int remainderUnsigned(int dividend, int divisor)* |  |
| *static int reverse(int num)* | reverses the order of the bits in *num* and return the result. |
| *static int reverseBytes(int num)* | reverses the order of the bytes in *num* and return the result. |
| *static int rotateLeft(int num, int n)* | returns the result of rotating *num* left by *n* positions. |
| *static int rotateRight(int num, int n)* |  |
| *static int signum(int num)* | returns -1 if *num* is negative, 0 if 0, 1 if positive. |
| *static int sum(int val, int val2)* |  |
| *static String toBinaryString(int num)* | returns a binary equivalent of *num* in a string. |
| *static String to HexString(int num)* |  |
| *static String toOctalString(int num)* |  |
| *static long toUnsignedLong(int val)* |  |
| *static String toUnsignedString(int val)* |  |

Converting Numbers to and from Strings

*parseByte()*, *parseShort()*, *parseInt()*, *parseLong(), parseFloat(), parseDouble()* convert a string (the caller) to a *byte/short/int/long*/float/double.

*toBinaryString()*, *toHexString()*, *toOctalString()* convert a value into a binary/hexadecimal/octal string.

Example:

String str = "example";

**int** i = Integer.*parseInt*(str);

**int** j = 3000;

System.***out***.println(Integer.*toBinaryString*(j));

System.***out***.println(Integer.*toHexString*(j));

System.***out***.println(Integer.*toOctalString*(j));

Notable methods for *Character*

* *static char forDigit(int num, int radix)*
* *static int digit(char digit, int radix)*

converts between integer values and the digits they represent.

* *int compareTo(Character c)*

returns zero if the invoking object and *c* have the same value, negative if the invoking object has a lower value.

* *getDirectionality()*

determines the direction of a character (niche).

Unicode Code Point Support

Beginning with JDK 5, the *Character*­ class includes support for 32 bit Unicode characters (16 bit before). Characters with values greater than *FFFF* (maximum under 16 bit) are called supplemental characters. The basic multilingual plane (BMP) are the characters between *0* and *FFFF*.

A supplemental character has a value greater than a *char* can hold. Java handles this in two ways:

1. Java uses 2 *chars* to represent a supplemental characters – the first called the high surrogate, the second called the low surrogate. Methods such as *codePointAt()* translate between code points and supplemental characters.
2. Java overloaded several pre-existing methods in the *Character* class to use *int* rather than *char*. Several methods were added to provide additional support, such as –
   * *static int charCount(int cp)*

returns 1 if *cp* can be represented with a single *char*, 2 if 2 *chars* are needed.

* + *static int codePointAt(CharSequence chars, int loc)*

returns the code point at location specified by *loc*. Can take *chars* can be a *CharSequence* or *Char* array.

* + *static boolean isBmpCodePoint(int cp)*
  + *static boolean isHighSurrogate(char ch)*
  + *static boolean isLowSurrogate(char ch)*
  + *static boolean isSupplementaryCodePoint(int cp)*
  + *static boolean isSurrogatePair(char highCh, char lowCh)*
  + *static boolean isValidCodePoint(int cp)*
  + *static char[] to Chars(int cp)*

*Boolean*

*Boolean* contains the constants *TRUE* and *FALSE*, and defines the *TYPE* field, which is the *Class* object for boolean.

|  |  |
| --- | --- |
| **Methods Defined by *Boolean*** | **Description** |
| *static Boolean getBoolean(String propertyName)* | returns true if the system property specified by *propertyName* is true. |
| *static boolean logicalAnd(boolean op1, boolean op2)* |  |
| *static boolean logicalOr(boolean op1, boolean op2)* |  |
| *static boolean logicalXor(boolean op1, boolean op2)* |  |

*Void*

*Void* has one field, *TYPE*, which holds a reference to the *Class* object for type *void*.

*Process*

The abstract *Process* class encapsulates a *process*. It is used as as superclass for the type of objects created by *exec()* in the *Runtime* class, or by *start()* in the *ProcessBuilder* class. A handle to the process can be obtained in the form of a *ProcessHandle* instance (JDK 9), with *ProcessHandle.Info* instances providing information about the process.

|  |  |
| --- | --- |
| **Methods Defined by *Process*** | **Description** |
| *boolean isAlive()* |  |
| *ProcessHandle.Info info()* |  |
| *InputStream getErrorStream()* | returns an input stream that reads input from the process’ *err* output stream. |
| *InputStream getInputStream()* |  |
| *OutputStream getOutputStream()* |  |
| *int exitValue()* |  |
| *void destroy()* | terminates the process. |
| *Process destroyForcibly()* | terminates the invoking process and returns a reference to it. |
| *Stream<ProcessHandle> children()* | returns a stream containing *ProcessHandle* objects that represent the immediate children of the invoking process |
| *Stream<ProcessHandle> descendants* | returns *ProcessHandle* objects that represent all children of the invoking process and their children. |
| *CompletableFuture<Process> onExit()* |  |
| *long pid()* | returns the process ID. |
| *boolean supportsNormalTermination* | returns true if *destroy()* will result in normal termination (as opposed to forced) |
| *ProcessHandle toHandle()* | returns the handle of the invoking process |
| *int waitFor() throws InterruptedException* | returns the exit code of the process when the caller terminates. |
| *boolean waitFor(long waitTime, TimeUnit timeUnit) throws InterruptedException* | returns true if the process has ended by the end of wait time. The amount of wait time is specified by *waitTime* in the unit of *timeUnit*. |

*Runtime*

*Runtime* cannot be instantiated. However, a reference to the current *Runtime* object can be obtained by calling the static *Runtime.getRuntime()*. Untrusted code will typically raise a *SecurityException* when calling *Runtime* methods.

Notable methods of *RunTime*

* *Process exec(String programName) throws IOException*

executes the program specified as a separate process. Additional form specifies a command line instead of program name

* *void exit(int exitCode)*
* *long freeMemory()*

returns the free memory available to the runtime system, in unit of bytes.

* *long totalMemory()*
* *void gc()*

initiates the garbage collection.

* *void runFinalization()*
* *void load(String libraryFileName) //complete path*
* *void loadLibrary(String libraryName)*

load the specified dynamic library.

* *void halt(int code)*

immediately terminates the JVM. *code* is returned to the invoking progress.

Uses for the *Runtime* class:

1. memory management

Although Java runs garbage collection periodically, one can also manually do so by calling *gc()* (*gc()* only suggests the JVM to run garbage collection, it does not instantly cause it).

*totalMemory()* and *freeMemory()* are useful for getting an idea of how much memory the system has. (Example utilities are to check for code efficiency, or to check how many more objects can be instantiated)

1. executing other programs

*exec()* allows Java to execute other programs. It returns a *Process* object. *exec()* is environment-dependent. The *Process* object returned can be manipulated by *Process’* methods. Example methods:

* *destroy()*
* *waitFor() //causes the program to wait until the subprocess finishes*
* *exitValue()* *//returns the value returned by the subprocess when it finishes*
* *getOutputStream()*
* *getInputStream()* *//write to and read from the subprocess’ standard input and output*

Example:

Runtime r = Runtime.*getRuntime*();

Process p = **null**;

**try** {

p = r.exec("notepad");

p.waitFor();

} **catch**(Exception e) {

System.***out***.println("error");

}

System.***out***.println(p.exitValue());

*Runtime.Version*

(added JDK 9) *Runtime.Version* encapsulates version information pertaining to the Java environment. Obtain an instance of *Runtime.Version* by calling *Runtime.version()*.

Modern Java version number contains 4 elements, in order: feature release counter, interim release counter, update release counter, patch release counter. Additional elements may follow.

JDK 10 adds *version()* which returns a list of the version numbers;

*int feature(); int interim(); int update(); int patch()*, which return the various version numbers;

*build()* which returns the build number;

*pre()* returns pre-release information.

*ProcessBuilder*

*ProcessBuilder* is another way to start and manage processes. It has the constructors

*ProcessBuilder(List<String> args)*

*ProcessBuilder(String …args)*

where *args* is a list of arguments that specify the name of the program to be executed with its command-line arguments.

Notable methods:

* *Process start() throws IOException* – begins the process specified by the invoking object
* *List<String> command()* – returns the name of the program and its arguments. Changes to the list affect the invoking object.
* *ProcessBuilder command(List<String> args)* – sets the name and arguments of the program to *args*.
* *File directory()* -- returns the current working directory of the invoking object (*null* if same as current Java program).
* *ProcessBuilder directory(File dir)* – changes the working directory of the invoking object.
* *ProcessBuilder.Redirect redirectError()*
* *ProcessBuilder redirectError(File f)*

The abstract *ProcessBuilder.Redirect* class encapsulates an I/O source or target linked to a subprocess. It provides methods for redirecting the source or target of I/O operations, e.g. *to()* and *from()*, *appendTo()*, *redirectInput()*, *redirectOutput()*, *type()* (returns an object of enumeration type *ProcessBuilder.Redirect.Type*)

To create a process using *ProcessBuilder*, create an instance of *ProcessBuilder* and specify the name of the program and arguments. Execute the program by calling *start()*.

Example: open the txt file “filename.txt”.

**try** {

ProcessBuilder pb = **new** ProcessBuilder("notepad.exe", "filename");

pb.start();

} **catch**(Exception e) {

System.***out***.println("error");

}

*System*

Holds a collection of static methods and variables.

Notable methods (often throws *SecurityException* if operation is not permitted):

* *static void arraycopy(Object source, int sourceStart, Object target, int targetStart, int size)*

copies the array *source*, starting at index *sourceStart*, to *target*, starting at index *targetStart*. Copies *size* number of elements.

* + notable use: shifts arrays (faster than long hand methods), e.g.

**byte** a[] = {1, 2, 3, 4, 5 };

**byte** b[] = **new** **byte**[a.length];

**byte** c[] = **new** **byte**[a.length];

System.*arraycopy*(a, 0, b, 1, a.length-1); //shifts a right by 1

System.*arraycopy*(a, 1, c, 0, a.length-1); //shifts a left by 1

*static String getenv(String which)*

returns the environmental variable associated with *which* and its value.

* ­*static Properties getProperties()*

returns the perperties associated with the JVM.

* *static String getProperty(String which)*

returns the property associated with *which*.

* *static void runFinalization()*

initiates calls to *finalize()* on unused objects.

* *static long currentTimeMillis()*

returns the current time in milliseconds since Jan 1 1970.

* *static long nanoTime()*

obtains time passed since arbitrary time in nanoseconds.

* + notable use: time a section of a program with *currentTimeMillis()* or *nanoTime()*, e.g.

**long** start, end, elapsed\_time;

start = System.*currentTimeMillis*();

//perform some tasks

end = System.*currentTimeMillis*();

elapsed\_time = end - start;

* *static System.Logger getLogger(String logName)*

returns a program log. *System.Logger* is the interface to the logger.

*Object*

notable methods

* *boolean equals(Object obj)*
* *void finalize() throws Throwable*
* *final Class<?> getClass()*
* *int hashCode()*
* *final void notify()*
* *final void notifyAll()*
* *String toString()*
* *final void wait(long milliseconds) throws InterruptedException*
* *Object clone() throws CloneNotSupportedException*

Only classes that implement the *Cloneable* interface can be cloned. Cloning can cause unintended side effects: if a cloned object contains a reference variable, changes to that variable will reflect on both original and clone objects. Example: cloned object opens an I/O stream; both objects will be able to write to that stream; if one object closes the stream, the other will not be able to write to it.

*clone()* is declared *protected*, so it can only be called by a class that implements *Cloneable*, or it can be overridden to be *public*.

Example: class implements *Cloneable* with a method that clones the object

**class** klone **implements** Cloneable {

**int** a;

**public** Object clone() {

**try** {

**return** (klone) **super**.clone();

} **catch**(CloneNotSupportedException e) {

System.***out***.println("clone not allowed");

**return** **this**;

}

}

}

*Class*

*Class* objects are created automatically when classes are loaded. *Class* object cannot be explicitly declared. Obtain *Class* objects by calling *getClass()*.

*Class* defines numerous methods that return members of the class based on their types, e.g. *getAnnotations()*, *getConstructors()*, *getMethods()*, *getInterface()*, *getModule()*, etc.

Beginning with JDK 11, *Class* provides 3 methods relating to a *nest* – a group of classes and/or interfaces nested within an outer class or interface. It is not a source code mechanism, and exists on a JVM level.

* *getNestHost()* – obtains a nest’s top-level class/interface
* *isNestMateOf()* – tests if a class/interface is of the same nest as another
* *getNestMembers()*

*Math*

Contains static functions used for geometry and trigonometry, as well as some general purpose methods. Defines two *double* constants: *E* and *PI*.

Trigonometric Functions

* take a double parameter in radians and return a double in radians.

Exponential Functions

* *static double cbrt(double args)* – cube root
* *static double exp(double arg)* – earg
* *static double expm1(double arg)* – earg-1
* *static double log(double arg)* – loge(arg)
* *static double log10(double arg)*
* *static double log1p(double arg) –* loge(arg+1)
* *static double pow(double x, double y) –* xy
* *static double scalb(double arg, int factor) –* arg \* 2factor
  + has overloaded form that returns float
* *static double sqrt(double arg)*

Rounding Functions

* *static int abs(int arg) – overloaded forms return long, float, double*
* *static double ceil(double arg) –* returns the smallest whole number greater than/equal to arg.
* *static double floor(double arg) –* returns the smallest whole number smaller than/equal to arg.
* *static int floorDiv(int dividend, int dividor) –* returns the floor of dividend/divisor.
* *static int floorMod(int dividend, int divisor) –* returns the floor of remainder of dividend/divisor.
* *static int max(int x, int y) –* returns the max of x and y
* *static int min(int x, int y)*
* *static int round(float arg) –* returns arg rounded up to the nearest int.
* *static double nextDown(double val) –* returns the next value lower than *val*.
* *static double nextUp(double arg)*
* *static double nextAfter(double arg, double toward) –* returns the next value from *val*, in the direction of *toward*.
* *static double ulp(double arg) –* *unit of last place* – returns the smallest possible increment between values in *val*’s accuracy.

Misc. Methods

*Exact* suffix: throws an exception if overflow occurs.

* *static int addExact(int arg1, int arg2)*
* *static int subtractExact(int arg1, int arg2)*
* *static int decrementExact(int arg)*
* *static int incrementExact(int arg)*
* *static int multiplyExact(int arg1, int arg2)*
* *static int negateExact(int arg)*
* *static int toIntExact(double arg)*
* *static float copySign(float arg, float target)*
* *static double fma(double arg1, double arg2, double arg3) –* returns arg1\*arg2+arg3.
* *static int getExponent(double arg) –* returns the base-2 exponent of *arg* in binary.
* *static double hypot(double side1, double side2)* – returns the hypotenuse of a triangle with sides *side1* and *side2*.
* *static double IEEEremainder(double dividend, double divisor) – returns the remainder of dividend/divisor.*
* *static double random()* – returns random value between 0 and 1.
* *static float signum(double arg)* – returns 0 if arg is 0, 1 if arg is positive, -1 if arg is negative.
* *static double toDegrees(double angle)*
* *static double toRadians(double angle)*

*StrictMath*

The *StrictMath* class has the same methods as *Math*, but return more precise results (*Math* reduces precision to improve performance).

*Runnable* Interface, *Thread* and *ThreadGroup* classes

*Runnable* and *Thread*: see Ch11.

*ThreadGroup* has two constructors:

* *ThreadGroup(String groupName)*
* *ThreadGroup(ThreadGroup parentOb, String groupName)*

notable methods:

* *int activeCount()*
* *int activeGroupCount()* – returns the approximate number of active groups for which the invoking thread is a parent.
* *final void checkAccess()* – verify that the invoking thread may access/change the called group
* *final void destroy()* – destroys the thread group and its child groups
* *final int getMaxPriority()*
* *final ThreadGroup getParent()*
* *final void interrupt()* – invokes interrupt() on all threads in the group and subgroups.
* *void list()* – displays information about the group.

Example: creating 2 thread groups each with 2 threads, suspend group A and resume it, and let the threads finish running.

**class** NewThread **extends** Thread {

**boolean** suspendFlag;

NewThread(String threadName, ThreadGroup groupName) {

**super**(groupName, threadName);

suspendFlag = **false**;

}

**public** **void** run() {

**try** {

**for**(**int** i=5; i>0; i--) {

System.***out***.println("Thread: " + getName() + "; tick: " + i);

Thread.*sleep*(1000);

**synchronized**(**this**) {

**while**(suspendFlag) {

wait();

}

}

}

} **catch**(Exception e) {

System.***out***.println("Exception in " + getName());

}

System.***out***.println("thread " + getName() + " exiting.");

}

**synchronized** **void** suspendThread() {

suspendFlag = **true**;

}

**synchronized** **void** resumeThread() {

suspendFlag = **false**;

notify();

}

}

**public** **class** demo {

**public** **static** **void** main(String args[]) {

ThreadGroup groupA = **new** ThreadGroup("group A");

ThreadGroup groupB = **new** ThreadGroup("group B");

NewThread thread1 = **new** NewThread("one", groupA);

NewThread thread2 = **new** NewThread("two", groupA);

NewThread thread3 = **new** NewThread("three", groupB);

NewThread thread4 = **new** NewThread("four", groupB);

thread1.start();

thread2.start();

thread3.start();

thread4.start();

System.***out***.println("suspending group A");

Thread ga[] = **new** Thread[groupA.activeCount()];

groupA.enumerate(ga); //lists the threads in group A

**for**(**int** i=0; i<ga.length; i++) {

((NewThread)ga[i]).suspendThread();

}

**try** {

Thread.*sleep*(4000);

} **catch**(InterruptedException e) {

System.***out***.println("main thread interrupted.");

}

System.***out***.println("resuming group A");

**for**(**int** i=0; i<ga.length; i++) {

((NewThread)ga[i]).resumeThread();

}

//wait for threads to finish

**try** {

thread1.join();

thread2.join();

thread3.join();

thread4.join();

} **catch**(Exception e) {

System.***out***.println("main thread exception.");

}

System.***out***.println("main thread exiting.");

}

}

*java.lang* defines two additional thread related classes:

* *ThreadLocal* – used to create thread local variables. Each thread gets its own thread ocal variables.
* *InheritableThreadLocal* – used to create thread local variables that may be inherited.

Package

*Package* encapsulates information about packages. Notable methods

* *<A extends Annotation> A getAnnotation(Class<A> annoType)*
* *Annotations[] getAnnotations()*
* *Annotations[] getDeclaredAnnotations()* – only returns declared annotations (not inherited)
* *String getName()* – returns the name of the invoking package
* *String getImplementationTitle()* – returns the title of the invoking package
* *String getImplementationVendor()* – returns the name of the implementor of invoking package.
* *static Package[] getPackages()*
* *boolean isAnnotationPresent(Class<? extends Annotation> anno)* – returns true if *anno* is associated with the invoking object.

*Module*

*Module* instances can manage access rights to a module and obtain information about a module. Notable methods:

* *String getName()*
* *String getPackages()*
* *String getDescriptor()*
* *void addExports() –* exports a package to a specified module.
* *void addOpens()*
* *void addReads()* – reads another module.
* *void addUses()*
* *boolean canRead()*
* *boolean canUse()*
* *boolean isExported()* – determines if a package is exported by the invoking module.
* *boolean isOpen()*

to get a *Module* instance: call *getModule()* on a *Class* instance.

*StackTraceElement*

The *StackTraceElement* class describes a stack frame – an element in a stack trace. It represents an execution point.

Various methods return arrays of *StackTraceElement*s.

notable methods:

* *String getClassName()*
* *String getClassLoaderName()*
* *String getFileName()*

*StackWalker* and *StackWalker.StackFrame*

These 2 interfaces are added in JDK 9. Obtain a *StackWalker* instance by calling the static *getInstance()*. Initiate stack walking by calling *walk()* defined by *StackWalker*.

*Enum*

All enumerations automatically inherit *Enum*. *Enum* is declared with

*class Enum<E extends Enum<E>>*

Notable methods

* *protected final Object clone() throws CloneNotSupportedException*
* *static <T extends Enum<T>> T valueOf(Class<T> enum-type, String name)* – returns the constant associated with *name* in the numeration type specified by *enum-type*.

*ClassValue*

A generic class used to associate a value with a type. Not for normal programming.

*CharSequence ­*Interface

This interface defines methods that grant read-only access to a sequence of characters. Implemented by *String*, *StringBuffer*, *StringBuilder*, etc..

*Comparable* Interface

Generic. Classes that implement *Comparable* contain objects that can be compared & ordered. Declared with

*interface Comparable<T>*

where *T* is the type of objects being compared. The main method is

*int compareTo(T obj)*

which returns 0 if the 2 objects are equal, negative value if the invoking object is smaller.

*Appendable* Interface

Classes that implement *Appendable* has objects that can be appended with character(s). Its main method is

*Appendable append() throws IOException*

which takes *char* or *CharSequence* for argument.

*Iterable* Interface

Must be implemented by classes whose objects will be used by the for-each version for loop.

*Readable* Interface

Indicates that an object can be used as a source for characters.

*AutoCloseable* Interface

Only objects of classes that implement *AutoCloseable* can be used with try-with-resources. Implemented by all I/O classes that open a stream that can be closed.

*Thread.UncaughtExceptionHandler* Interface

Implemented by classes that handle uncaught exceptions.

*java.lang* subpackages

1. *java.lang.annotation*
2. *java.lang.instrument* – defines features that can be used to add instrumentation
3. *java.lang.invoke* – supports dynamic language features
4. *java*.*lang.management*
5. *java.lang.module*
6. *java.lang.ref* – control the garbage collection process
7. *java.lang.reflect*

*Reflection* is the ability of a program to analyze code at run time. *java.lang.reflect* provides the ability to obtain information about fields, constructors, methods, and modifiers.

CH19

***java.util* Part 1 – Collections Framework**

The collections framework is a hierarchy of interfaces and classes for managing groups of objects. It standardizes the way groups of objects are handled. It was released in J2SE 1.2.

The collections framework has several goals:

1. improve performance
2. interoperability between types of collections
3. simplify extending/adopting a collection
4. integration of standard arrays into the collections framework

**Algorithms** are static methods within the *Collections* class, and provide a standard means of manipulating collections.

The *Iterator* interface offers a general purpose, standardized way of accessing the elements within a collections, one at a time – an iterator provides a means of enumerating the contents of a collection.

The *Spliterator*, added in JDK 8, is a type of iterator that provides support for parallel iteration.

The framework also defines several map interfaces & classes, which store key/value pairs.

**The Collection Interfaces**

The collections framework defines several key interfaces.

**Summary**

|  |  |
| --- | --- |
| Interface | Description |
| *Collection* | Root interface in the *collection hierarchy*. |
| *Deque* | Extends *Queue* to handle double-ended queues. |
| *List* | Extends *Collection* to handle lists of objects. |
| *NavigableSet* | Extends *SortedSet* to handle closest-match searches. |
| *Queue* | Extends *Collection* to handle queues, where elements can only be removed from the head. |
| *Set* | Extends *Collection* to handle sets, which must contain unique elements. |
| *SortedSet* | Extends *Set* to handle sorted sets. |
| **below not described in book** | |
| *BlockingQueue* | Extends *Queue* to support operations that wait for the queue to become non-empty when retrieving an element, and wait for space to become available when storing an element. |
| *Blocking Deque* | Extends *BlockingQueue* to handle deques. |

The *Collection ­*Interface

*Collection* extends the *Iterable* interface. *Collection* must be implemented by any class that defines a collection. It forms the foundation of the collections framework.

It is declared with

*interface Collection<E>*

where *E* is the type of objects that the collection will hold.

Notable methods:

|  |  |
| --- | --- |
| **Method** | **Description** |
| *boolean add(E obj)* | adds *obj* to the invoking collection (*obj* must be compatible with the type of data expected). Returns false if *obj* is already an member and the collection does not allow duplicates. |
| *boolean addAll(Collection<? extends E> c)* |  |
| *void clear()* | Removes all elements from the invoking collection. |
| *boolean contains(Object obj)* | Returns true if *obj* is an element of the invoking collection. |
| *boolean containsAll(Collection<?> c)* |  |
| *boolean equals(Object obj)* |  |
| *int hashCode()* |  |
| *boolean isEmpty()* |  |
| *Iterator<E> iterator()* | returns an iterator for the invoking collection. |
| *default Stream<E> stream()* | returns a stream that uses the invoking collection as source for elements. |
| *default Stream<E> parallelStream()* | returns a stream that uses the invoking collection as source for elements. Supports parallel operations if possible. |
| *boolean remove(Object obj)* | returns false if *obj* is removed. |
| *boolean removeAll(Collection<?> c)* |  |
| *default boolean removeIf(Predicate <? super E> condition)* |  |
| *boolean retainAll(Collection<?> c)* | removes all elements except those in *c*. Returns true if elements were changed. |
| *int size()* |  |
| *default Spliterator<E> spliterator()* | returns a spliterator for the invoking collection. |
| *Object[] toArray()* | returns an array of the elements from the invoking collection. |
| *<T> T[] toArray(T array[])* | If the size of *array* equals the number of elements in the invoking collection, they are stored in *array*. If *array* size is too small, a new array of the right size is created. If *array* size is too big, the last elements are set to null. |

The *List* Interface

*List* extends *Collection* and declares the behavior of a collection that stores a sequence of elements. A list may contain duplicate elements. *List* is generic and declared with

*interface List<E>*

where *E* is the type of objects the list will hold.

Notable methods:

|  |  |
| --- | --- |
| **Method** | **Description** |
| *boolean add(int index, E obj)* | adds *obj* to the invoking list at the index specified. Preexisting elements are shifted up. If *index* is not provided, *obj* is added to the end of the list. |
| *boolean addAll(int index, Collection<? extends E> c)* |  |
| *static<E> List<E> copyOf(Collection<? extends E> from)* | returns a list containing the same elements as *from* that cannot be modified. |
| *E get(int index)* |  |
| *E set(int index, E obj)* |  |
| *int indexOf(Object obj)* |  |
| *int lastIndexOf(Object obj)* |  |
| *ListIterator<E> listIterator()* | returns an iterator to the start of the invoking list. |
| *ListIterator<E> listIterator(int index)* |  |
| *static<E> List<E> of(parameter-list)* | creates an unmodifiable list containing the elements specified in *parameter-list*. Null elements are not allowed. |
| *E remove(int index)* |  |
| *default void replaceAll(UnaryOperator<E> func)* | updates all elements in the list with the value obtained from the *func* function. |
| *default void sort(Comparator<? super E> comp)* | sort the list using the comparator *comp*. |
| *List<E> subList(int start, int end)* | returns a list including elements from *start* to *end-1*. |

The *Set* Interface

*Set* extends *Collection* and defines a set, which does not allow duplicate elements. Set is generic and has the declaration

*interface Set<E>*

where *E* is the type of objects that the set will hold.

*Set* only defines 2 methods:

* *static<E> Set<E> of(param-list)* – which is just like the *List* version
* *static<E> Set<E> copyOf(Collection<? extends E> from)* – also like the *List* version

The *SortedSet* Interface

*SortedSet* extends *Set* and declares the behavior of a set sorted in ascending natural order. It is generic and declared with

*interface SortedSet<E>*

Notable methods:

|  |  |
| --- | --- |
| **Method** | **Description** |
| *Comparator<? super E> comparator()* | returns the comparator of the invoking sorted set. |
| *E first()* | returns the first element in the invoking sorted set. |
| *E last()* |  |
| *SortedSet<E> headset(E end)* | returns a *SortedSet* containing the elements less than *end* in the invoking sorted set. |
| *SortedSet<E> tailSet(E start)* | returns a *SortedSet* containing the elements greater than or equal to *start* in the invoking sorted set. |
| *SortedSet<E> subset(E start, E end)* | returns a *SortedSet* containing elements from *start* to *end-1*. |

The *NavigableSet* Interface

*NavigableSet* extends *SortedSet* and declares a collection that supports the retrieval of elements based on the closest match to a given value/values.

Notable methods:

|  |  |
| --- | --- |
| **Method** | **Description** |
| *E ceiling(E obj)* | returns the smallest element that is >= *obj*. |
| *E floor(E obj)* | returns the largest element that is <= *obj*. |
| *E higher(E obj)* | returns the smallest element that is > obj. |
| *E lower(E obj)* | returns the largest element that is < *obj*. |
| *Iterator<E> iterator()* | returns an iterator in ascending order. |
| *Iterator<E> descendingIterator()* | returns an iterator in descending order. |
| *NavigableSet<E> descendingSet()* | returns the reverse of the invoking set. |
| *E pollfirst()* | returns and removes the first (smallest) element. |
| *E pollLast()* |  |
| *NavigableSet<E> subSet(E lowerbound, boolean IncudelLower, E upperbound, boolean IncludeUpper)* | returns a portion of the set whose elements range from *lowerbound* to *upperbound*.  default *IncludeLower* = true, *IncludeUpper* = false |
| *SortedSet<E> subSet(E lowerbound, E upperbound)* |  |
| *NavigableSet<E> tailSet(E lowerbound, boolean include)* | returns a portion of the set whose elements are greater than (or equal to) *lowerbound*. |
| *SortedSet<E> tailSet(E lowerbound)* |  |

The *Queue* Interface

*Queue* extends *Collection* and defines a queue, which is a first-in, first-out list. Elements can only be removed from the head of a queue.

Notable methods:

|  |  |
| --- | --- |
| **Method** | **Description** |
| *E element()* | returns the element at the head of the queue. |
| *boolean offer(E obj)* | attempts to add *obj* to the queue. (some queues have a fixed length and can be full) |
| *E peek()* | returns the element at the head of the queue. returns null if empty. |
| *E poll()* | returns and removes the element at the head of the queue. returns null if empty. |

each of these methods has two forms, which differ on what happens when the operation fails.

|  |  |  |
| --- | --- | --- |
|  | throws exception | returns null or false |
| **Insert** | *add(e)* | *offer(e)* |
| **Remove** | *remove()* | *poll()* |
| **Examine** | *element()* | *peek()* |

The *Deque* Interface

*Deque* extends *Queue* and defines a double-ended queue, which can be either first-in, first-out or last-in, first-out. *Deque* does not support indexed access to elements.

Notable methods:

|  |  |
| --- | --- |
| **Method** | **Description** |
| *void addFirst(E obj)* | adds *obj* to the head of the deque. |
| *void addLast(E obj)* | adds *obj* to the tail of the deque. |
| *Iterator<E> descending Iterator()* |  |
| *E getFirst()* |  |
| *E getLast()* |  |
| *boolean offerFirst(E obj)* |  |
| *boolean offerLast(E obj)* |  |
| *E peekFirst()* |  |
| *E peekLast()* |  |
| *E pollFirst()* |  |
| *E pollLast()* |  |
| *E pop()* | returns and removes the head element. |
| *void push(E obj)* | adds *obj* to the head of the deque. |
| *E removeFirst()* |  |
| *E removeLast()* |  |
| *boolean removeFirstOccurrence(Object obj)* | the only methods to be able to remove interior elements. |
| *boolean removeLastOccurrence(Object obj)* |

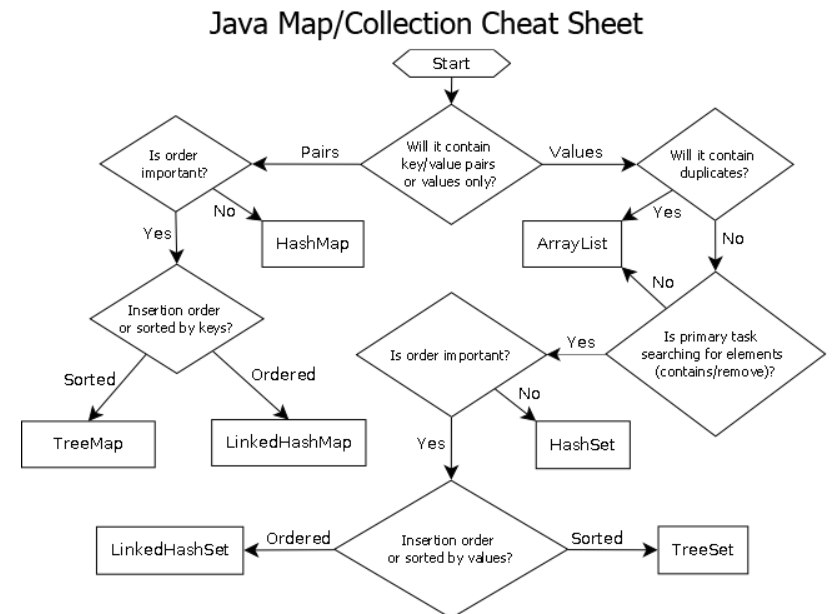
Some of these methods have two forms, which differ on what happens when the operation fails.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **First element** | | **Last element** | |
|  | throws exception | returns null/false | throws exception | returns null/false |
| **insert** | *addFirst(e)* | *offerFirst(e)* | *addLast(e)* | *offerLast(e)* |
| **remove** | *removeFirst(e)* | *pollFirst(e)* | *removeLast()* | *pollLast()* |
| **examine** | *getFirst()* | *peekFirst()* | *getLast()* | *peekLast()* |

**The Collection Classes**

**Summary**

|  |  |
| --- | --- |
| Class | Description |
| *AbstractCollection* | Provide a skeletal implementation of *Collection/List/Queue/…* to reduce effort required to implement it. Only use is to act as superclass for custom collections. |
| *AbstractList* |
| *AbstractQueue* |
| *AbstractSequentialList* |
| *AbstractSet* |
| *LinkedList* | Doubly-linked list. Implements *List* and *Deque*. |
| *ArrayList* | Resizable array implementation of *List*. |
| *ArrayDeque* | Resizable array implementation of *Deque*. |
| *EnumSet* | Specialized *Set* implementation for use with enumerations. Efficient. |
| *HashSet* | *Set* implementation backed by a hash table. Iteration order can change. |
| *LinkedHashSet* | Doubly-linked *HashSet*, ensuring predictable iteration order. |
| *PriorityQueue* | *Queue* ordered based on natural ordering or provided *Comparator*. |
| *TreeSet* | *NavigableSet* ordered based on natural ordering or provided *Comparator*. |



The *ArrayList* Class

*ArrayList* is generic and declared with

*class ArrayList<E>*

*ArrayList* supports dynamic arrays with variable sizes (standard arrays are of fixed lengths). Array lists are created with an initial size. When this size is exceeded, the array list is automatically enlarged; when objects are removed, the array list can be shrunk.

*ArrayList* has the following constructors:

1. *ArrayList()* – creates an empty list
2. *ArrayList(Collection<? extends E> c)* – initializes the array list with elements from collection *c*
3. *ArrayList(int capacity)*

Example demonstrating arraylist’s variable size:

**import** java.util.\*;

**class** demo {

**public** **static** **void** main(String args[]) {

ArrayList<String> al = **new** ArrayList<String>(); //creates an empty array list

System.***out***.println(al.size()); //returns "0"

al.add("B"); //adds to end of list

al.add("C");

al.add("D");

al.add(0, "A"); //specifies index to add

System.***out***.println("contents of al: " + al);

System.***out***.println(al.size()); //returns"4"

al.remove(2); //specifies index to remove

al.remove("A");

System.***out***.println(al.size()); //returns "2"

}

}

One can manually increase the capacity of an *ArrayList* object by calling *ensureCapacity()*. It has the form

*void ensureCapacity(int cap)*

where *cap* is the new capacity of the collection.

This reduces the number of reallocations if multiple elements are added beyond the array list’s original size.

Conversely, one can manually decrease the capacity of an *ArrayList* object by calling *trimtoSize()*.

Obtaining an array from an *ArrayList*: call *toArray()*

this has several advantages:

1. faster processing time
2. to pass an array to a method that does not accept a collection
3. for legacy codes

*toArray()* has 3 forms and can return *Object[]* or *T[]*:

* *Object[] toArray()*
* *<T> T[] toArray(T array[])*
* *default <T> T[] toArray(function<T[]> arrayGen)*

example: convert *al* to an array (following last example)

//...

Integer[] a = **new** Integer[al.size()];

a = al.toArray(a);

The *LinkedList* Class

*LinkedList* extends *AbstractSequentialList* and implements *List*, *Deque*, *Queue*. It is declared with

*class LinkedList<E>*

it has the constructors

1. *LinkedList()* – creates an empty linked list
2. *LinkedList(Collection<? extends E> c)* – initializes a linked list with elements of *c*

Example:

**import** java.util.\*;

**class** demo {

**public** **static** **void** main(String args[]) {

LinkedList<String> ll = **new** LinkedList<String>();

ll.add("C"); //various ways to add elements

ll.add("D");

ll.add("E");

ll.addFirst("A");

ll.add(1, "B");

ll.addLast("F");

ll.set(2, "changes C"); //changes an element

String str = ll.get(2); //obtains an elements

ll.remove(2); //various ways to remove elements

ll.removeFirst();

ll.removeLast();

ll.remove("F");

System.***out***.println("contents of ll: " + ll);

}

}

The *HashSet* Class

*HashSet* extends *AbstractSet* and implements *Set*. It creates a storage that uses a hash table. It has the constructors

1. *HashSet()*
2. *HashSet(Collection<? extends E> c)*
3. *HashSet(int capacity)* – default capacity is 16
4. *HashSet(int capacity, float fillRatio)* – the fill ratio must be between 0 and 1 and determines how full the hash set can be before being resized (0.75 by default)

The advantage of hash set is that it executes *add()*, *contains()*, *remove()* at the same speed even on large sets.

In hashing, the information are automatically transformed into a unique value (its hash code) which is used as the index. Hence *HashSet* cannot guarantee the order of its elements.

*HashSet* defines no methods.

The *LinkedHashSet* Class

*LinkedHashSet* extends *HashSet* and adds no member. It maintains the order of its elements, to the order in which they were inserted.

example:

hs.add("a");

hs.add("b");

hs.add("c");

hs.add("d");

If *hs* were a *HashSet*, abcd will likely not be in order; if *hs* were a *LinkedHashSet*, abcd always will be in order.

The *TreeSet* Class

*TreeSet* extends *AbstractSet* and implements *NavigableSet*. It creates a collection using a tree for storage. The elements are automatically put in sorted order (ascending by default). It has the constructors

1. *TreeSet()*
2. *TreeSet(Collection<? extends E> c)*
3. *TreeSet(Comparator<? super E> comp)*
4. *TreeSet(SortedSet<E> ss)*

*TreeSet* is excellent for storing large amounts of sorted information that must be found quickly.

The *PriorityQueue* Class

*PriorityQueue* extends *AbstractQueue* and implements *Queue*. It creates a queue that prioritizes based on the queue’s comparator. They are dynamic and grow as necessary. It has the constructors

1. *PriorityQueue()* –default capacity is 11
2. *PriorityQueue(int capacity)*
3. *PriorityQueue(Comparator<? super E> comp)*
4. *PriorityQueue(int capacity, Comparator<? super E> comp)*
5. *PriorityQueue(Collection<? extends E> c)* – last 3 initialize the queues with elements in *c*
6. *PriorityQueue(PriorityQueue<? extends E> c)*
7. *PriorityQueue(SortedSet<? extends E> c)*

The *ArrayDeque* Class

*ArrayDeque* extends *AbstractCollection* and implements *Deque*, and defines no methods. It creates a dynamic array with no capacity restrictions (*Deque* supports but doesn’t require limited capacity). It has the constructors

1. *ArrayDeque()* – default capacity is 16
2. *ArrayDeque(int size)*
3. *ArrayDeque(Collection<? extends E> c)*

Example: creating an ArrayDeque, fill it with elements, then pop them.

**import** java.util.\*;

**class** Nonempty {

**public** **static** **void** main(String args[]) {

ArrayDeque<String> ad = **new** ArrayDeque<String>();

ad.push("A");

ad.push("B");

ad.push("C");

**while**(ad.peek() != **null**)

System.***out***.print(ad.pop() + " ");

}

}

The *EnumSet* Class

*EnumSet* extends *AbstractSet* and implements *Set*. It creates sets with elements of *enum* types. It is declared with

*class EnumSet<E extends Enum<E>>*

*EnumSet* has no constructors. Use methods to create objects.

|  |  |
| --- | --- |
| **Method** | **Description** |
| *static <E extends Enum<E> EnumSet<E> allOf(Class<E> t)* | creates an *EnumSet* containing the elements of *t.* |
| *static <E extends Enum<E> EnumSet<E> complementOf(EnumSet<E> e)* | creates an *EnumSet* containing the elements not in *e*. |
| *static <E extends Enum<E> EnumSet<E> copyOf(EnumSet<E> c)* | creates an *EnumSet* from the elements stored in *c*. |
| *static <E extends Enum<E> EnumSet<E> noneOf(Class<E> t)* | creates an *EnumSet* from the elements not in *t*. |
| *static <E extends Enum<E> EnumSet<E> of(E v, E …varargs)* | creates an *EnumSet* from *v* and any additional values specified. |
| *static <E extends Enum<E> EnumSet<E> range(E start, E end)* | creates an *EnumSet* containing the elements from *start* to *end*. |

**Accessing a Collection via an Iterator**

An iterator is an object that either implements *Iterator* or *ListIterator*. It allows cycling through a collection and obtaining or removing elements. *ListIterator* enables bidirectional cycling and modifying elements. *Iterator* and *ListIterator* are declared with

*interface Iterator<E>*

*interface ListIterator<E>*

*Iterator* defines the methods

|  |  |
| --- | --- |
| **Method** | **Description** |
| *default void forEachRemaining(Consumer<? super E> action)* | the action *action* is executed on each unprocessed element of the collection. |
| *boolean hasNext()* |  |
| *E next()* | returns the next element. |
| *default void remove()* |  |

*ListIterator* defines the methods

|  |  |
| --- | --- |
| **Method** | **Description** |
| *void add(E obj)* |  |
| *default void forEachRemaining(Consumer<? super E> action* |  |
| *boolean hasNext()* |  |
| *boolean hasPrevious()* |  |
| *E next()* |  |
| *E previous()* |  |
| *int nextIndex()* |  |
| *int previousIndex()* |  |
| *void remove()* |  |
| *void set(E obj)* | assigns *obj* to the element last returned by *next()* or *previous()* |

Starting with JDK 8, *spliterator* can also be used to cycle through a collection.

To use an iterator, first obtain one by calling the *iterator()* method, provided by every collection class which returns an iterator to the start of the collection. In general, follow these step:

1. call *iterator()* to obtain an iterator to the start of the collection. (collections that implement *List* can call *listIterator()* instead)
2. create a loop that operates as long as *hasNext()* returns true.
3. obtain each element within the loop by calling *next()*.

Example: create an *ArrayList*, modify each element with *ListIterator*, then display the list twice, backwards and forwards.

**import** java.util.\*;

**class** Nonempty {

**public** **static** **void** main(String args[]) {

ArrayList<String> al = **new** ArrayList<String>();

al.add("A");

al.add("B");

al.add("C");

al.add("D");

ListIterator<String> lit = al.listIterator();

**while**(lit.hasNext()) { //adds + to each element

String elem = lit.next();

lit.set(elem + "+");

}

**while**(lit.hasPrevious()) { //displays the list backwards

System.***out***.print(lit.previous() + " "); //returns "D+ C+ B+ A+"

}

lit = al.listIterator();

**while**(lit.hasNext()) { //displays the list normally

System.***out***.print(lit.next() + " ");

}

}

}

The For-each Alternative to Iterator

As all collection classes implement the *Iterable* interface, they can all be cycled through a for loop. The for-each loop is more convenient than using an *iterator*, but it cannot modify elements or cycle backwards.

Example: use a for loop to calculate the sum of the elements of an *ArrayList*.

**import** java.util.\*;

**class** Nonempty {

**public** **static** **void** main(String args[]) {

ArrayList<Integer> al = **new** ArrayList<Integer>();

al.add(1);

al.add(2);

al.add(3);

al.add(4);

**int** sum = 0;

**for**(**int** v: al)

sum += v;

System.***out***.println("sum is " + sum);

}

}

**Spliterators**

The *Spliterator* interface was added in JDK 8. It cycles through a sequence of elements, but with more functionality than *Iterator* or *ListIterator*. Notably, it supports parallel iteration (ch28). It also streamlines operation by combining *hasNext()* and *next()* into one method.

Notable methods:

|  |  |
| --- | --- |
| **Method** | **Description** |
| *int characteristics()* | returns the characteristics of the invoking spliterator. |
| *long estimateSize()* | estimates the number of elements left to iterate. |
| *default void forEachRemaining(Consumer<? super T> action)* | applies *action* to each unprocessed element. |
| *default Comparator<? super T> getComparator()* |  |
| *default long getExactSizeIFknown()* |  |
| *default boolean hasCharacteristics(int val)* | returns true if the invoking spliterator has the characteristics described by *val*. |
| *boolean tryAdvance(Consumer<? super T> action)* | executes *action* on the next element. returns false if no elements remain. |
| *Spliterator<T> trySplit()* | attempt to split the invoking spliterator. If successful, the 2 spliterators iterate over different portions of the sequence. |

* *Comsumer* is a functional interface that applies an action on an object (ch20). It defines *accept()* with the form

*void accept(T objRef)*

* For basic iterating, call *tryAdvance()* until it returns false.
* *tryAdvance()* passes the next element to *objRef*.
* *forEachRemaining()* is a streamlined approach to applying some action to all elements of the sequence.

example: create an array list and display its content; then create another array list containing the square roots of the first list, and display its content.

**import** java.util.\*;

**class** Nonempty {

**public** **static** **void** main(String args[]) {

ArrayList<Double> al = **new** ArrayList<>();

al.add(1.);

al.add(2.);

al.add(3.);

al.add(4.);

System.***out***.println("displaying first list: ");

Spliterator<Double> splitit = al.spliterator();

**while**(splitit.tryAdvance((n) -> System.***out***.print(n + " ")));

System.***out***.println();

System.***out***.println();

splitit = al.spliterator();

ArrayList<Double> al2 = **new** ArrayList<>();

//create new array list containing the square roots of al's elements

**while**(splitit.tryAdvance(n -> al2.add(Math.*sqrt*(n))));

System.***out***.println("displaying second list:");

splitit = al2.spliterator();

**while**(splitit.tryAdvance(n -> System.***out***.print(n + " ")));

}

}

*Spliterator* has several subinterfaces for use with primitive types double, int, long: *Spliterator.OfPrimitive; Spliterator.OfDouble; Spliterator.OfInt; Spliterator.OfLong*.

**Storing User Defined Classes in Collections**

Collections can store any type of object, including user created types. Doing so is usually good for compacting the program, and should be done whenever possible.

Example:

**import** java.util.\*;

**class** price {

**private** String name;

**private** **int** price;

price(String s, **int** i) {

name = s;

price = i;

}

**public** String toString() {

**return** name + ": $" + price;

}

}

**class** Nonempty {

**public** **static** **void** main(String args[]) {

LinkedList<price> ll = **new** LinkedList<>();

ll.add(**new** price("apple", 4));

ll.add(**new** price("pear", 3));

ll.add(**new** price("orange", 2));

**for**(price p : ll)

System.***out***.println(p);

}

}

**The *RandomAccess* Interface**

*RandomAccess* contains no members. By implementing this interface, a collection signals that it supports efficient random access to elements. One can use *instanceof()* to check if a class is suitable for certain random access operations.

*ArrayList* implements *RandomAccess*.

**Working with Maps**

A map or key/value pair is an object that stores association between keys and values. A key is an object that is used to retrieve a value. The keys must be unique, but the values can be duplicates.

Maps don’t implement the *Iterable* interface, so they cannot be cycled in a for-each loop, nor can they obtain an iterator.

**The Map Interfaces**

**Summary**

|  |  |
| --- | --- |
| **Interface** | **Description** |
| *Map* | maps unique keys to values. |
| *Map.Entry* | describes an element in a map. Inner class of *Map*. |
| *SortedMap* | maintains keys in ascending order. Extends *Map*. |
| *NavigableMap* | has closest-match search feature for retrieving entries. Extends *SortedMap*. |

* *Map*

*Map* is declared with

*interface Map<K, V>*

where *K* specifies the type of keys, *V* specifies the type of values.

Methods:

|  |  |
| --- | --- |
| **Method** | **Description** |
| *void clear()* | removes all key/value pairs from the invoking map. |
| *default V compute(K k, BiFunction<? super K, ? super V, ? extends V> func)* | calls *func* to construct a new value. If it returns null, any pre-existing pairing is removed. If not, a new key/value pair is added to the map. |
| *default V computeIfAbsent(K k, Function<? super K, ? extends V> func)* | returns the value associated with the key *k*. If absent, a value is constructed with *func* and the two paired. |
| *default V computeIfPresent(K k, BiFunction<? super K, ? super V, ? extends V> func)* | If *k* is in the map, *func* is used to construct a new value for it and replace the old one. |
| *boolean containsKey(Object k)* |  |
| *boolean vontainsValue(Object v)* |  |
| *static<K, V> Map<K, V> copyOf(Map<? extends K, ? extends V> from)* | returns an unmodifiable map containing the key/value pairs specified by *from*. |
| *static <K,V> Map.Entry<K, V> entry(K k, V v)* | returns an unmodifiable map containing the given key and value. |
| *Set<Map.Entry<K, V>> entrySet()* | returns a set containing the entries in the map. |
| *boolean equals(Object obj)* | returns true if *obj* is a map and contains the same entries. |
| *default void forEach(BiConsumer<? super K, ? super V> action)* | executes *action* on each element in the invoking map. |
| *V get(Object k)* | returns the value associated with key *k*. |
| *default V getOrDefault(Object k, V defVal)* | returns *defVal* if *k* is not in the map. |
| *int hashCode()* |  |
| *boolean isEmpty()* |  |
| *Set<K> keySet()* | returns a *Set* containing the keys in the invoking map. |
| *default V merge(K k, V v, BiFunction<? super V, ? super V, ? extends V> func)* | add the pair *k,v* to the map if *k* is not in the map. If it is, *func* creates a new value and pairs it with *k*. |
| *static <K, V> Map<K, V> of(parameter-list)* | creates an unmodifiable map containing the entries of *parameter-list*. |
| *static <K, V> Map<K, V> ofEntries(Map.Entry<? extends K, ? extends V> …entries)* | returns an unmodifiable map containing the key/value mappings described by *entries*. |
| *V put(K k, V v)* | puts an entry in the invoking map, overwriting the previous value associated with *k.* Returns null if the key did not exist. |
| *void putAll(Map<? extends K, ? extends V> m)* | puts all entries from *m* into this map. |
| *default V putIfAbsent(K k, V v)* |  |
| *V remove(Object k)* | remove the entry whose key equals *k*. |
| *default boolean remove(Object k, Object v)* | if *k/v* is a pair in the invoking map, true is returned and they are removed. |
| *default boolean replace(K k, V oldV, V newV)* |  |
| *default V replace(K k, V v)* |  |
| *default void replaceAll(BiFunction<? super K, ? super V, ? extends V> func)* | replace each element of the invoking map with the result of *func*. |
| *int size()* |  |
| *Collection<V> values()* | returns a collection containing the values in the map. |

The two basic methods, *get()* and *put()*, obtains a value when given the key or puts a value into a map.

While they are part of the collections framework, maps are not collections as they do not implement the *Collection* interface. To obtain a collection-view of a map, call *entrySet()* which returns a *Set* containing the elements in the map. To obtain a collection-view of the keys, use *keySet()*. To obtain a collection-view of the values, use *values()*. Changing the collection affects the map and vice versa.

* *SortedMap*

*SortedMap* extends *Map* and ensures that entries are maintained in ascending order of the keys. It is declared with

*interface SortedMap<K, V>*

|  |  |
| --- | --- |
| **Method** | **Description** |
| *Comparator<? super K> comparator()* |  |
| *K firstKey()* |  |
| *K lastKey()* |  |
| *SortedMap<K, V> headMap(K end)* | returns a sorted map for the map entries whose keys are less than *end*. |
| *SortedMap<K, V> tailMap(K start)* | returns a sorted map for the map entries whose keys are greater than or equal to *start*. |
| *SortedMap<K, V> submap(K start, K end)* |  |

* *NavigableMap*

*NavigableMap* extends *SortedMap* and declares the behavior of a map that supports retrieval of entries based on closest match to a given key/keys.

|  |  |
| --- | --- |
| **Method** | **Description** |
| *Map.Entry<K, V> ceilingEntry(K obj)* | search for the smallest key *k* such that *k >= obj*. Returns its entry. |
| *K ceilingKey(K obj)* | search for the smallest key *k* such that *k >= obj*. Returns *k*. |
| *Map.Entry<K, V> higherEntry(K obj)* | […] *k > obj* […] |
| *K higherKey(K obj)* | […] *k > obj* […] |
| *Map.Entry<K, V> floorEntry(K obj)* | search for the largest key *k* such that *k <= obj* |
| *K floorKey(K obj)* | […] *k <= obj* […] |
| *Map.Entry<K, V> lowerEntry(K obj)* | […] *k < obj* […] |
| *K lowerKey(K obj)* | […] *k < obj* […] |
| *NavigableSet<K> descendingKeySet()* | returns a *NavigableSet* containing the keys in the invoking map in descending order. |
| *NavigableMap<K, V> descendingMap()* |  |
| *Map.Entry<K,V> firstEntry()* | returns the first entry in the map (the one with the least key). |
| *Map.Entry<K, V> lastEntry()* |  |
| *Map.Entry<K, V> pollFirstEntry()* | returns and removes the first entry. |
| *Map.Entry<K, V> pollLastEntry()* |  |
| *NavigableMap<K,V> headMap(K upperBound, boolean include)* | returns a *NavigableMap* containing all entries from the invoking map with keys < or <= *upperBound*. |
| *NavigableMap<K, V> tailMap(K lowerBound, boolean include)* |  |
| *NavigableMap<K, V> subMap(K lowerBound, boolean lowinclude, K upperBound, boolean highinclude)* |  |
| *NavigableSet<K> navigableKeySet()* | returns a *NavigableSet* containing the keys in the invoking map. |

* ­*Map.Entry*

*Map.Entry* allows working with map entries. A map entry is an element of the *Set* returned by *entrySet()* defined by *Map*.

|  |  |
| --- | --- |
| **Method** | **Description** |
| *static Comparator comparingByKey()* |  |
| *static Comparatot comparingByValue()* |  |
| *boolean equals(Object obj)* | returns true if *obj* is a *Map.entry* whose key and value are equal to the invoking object’s. |
| *K getKey()* |  |
| *V getValue()* |  |
| *int hashCode()* |  |
| *V setValue(V v)* |  |

**The Map Classes**

|  |  |
| --- | --- |
| **Class** | **Note** |
| *AbstractMap* | implements most of the *Map* interface. |
| *EnumMap* | extends *AbstractMap*. |
| *HashMap* | extends *AbstractMap*. |
| *TreeMap* | extends *AbstractMap*. |
| *WeakHashMap* | extends *HashMap*. Uses weak keys, which allow unused elements to be garbage-collected. |
| *IdentityHashMap* | extends *AbstractMap*. |

* *HashMap*

*HashMap* extends *AbstractMap* and implements *Map*. Hashing allows the execution time of *get()* and *put()* to remain constant even for large sets. *HashMap* does not guarantee the order of its elements.

*HashMap* is declared with

*class HashMap<K, V>*

It has the following constructors:

1. *HashMap()* (default capacity is 16)
2. *HashMap(Map<? extends K, ? extends V> m)*
3. *HashMap(int capacity)*
4. *HashMap(int capacity, float fillRatio)* (default fill ratio is 0.75)

Example

**import** java.util.\*;

**class** Nonempty {

**public** **static** **void** main(String args[]) {

HashMap<String, Double> hm = **new** HashMap<String, Double>();

//create a hash map and put elements into it

hm.put("A", -1.);

hm.put("B", 2.);

hm.put("C", 3.);

hm.put("D", 4.);

Set<Map.Entry<String, Double>> hmSet = hm.entrySet();

//obtain a set of the entries

**for**(Map.Entry<String, Double> me : hmSet) { //display the set

System.***out***.print(me.getKey() + ": ");

System.***out***.println(me.getValue());

}

hm.put("A", hm.get("A") + 2.); //add 2 to A's value

System.***out***.println("A's new value: " + hm.get("A"));

}

}

Note: the *put()* method automatically replaces any pre-existing value associated with the specified key.

* *TreeMap*

*TreeMap* extends *AbstractMap* and implements *NavigableMap*. It creates maps stored in a tree structure, which is an efficient means of storing key/value pairs in sorted order and allows rapid retrieval. *TreeMap* always stores its elements in ascending **key** order.

It has the constructors

1. *TreeMap()*
2. *TreeMap(Comparator<? super K> comp)* – creates an empty map that will be sorted using *comp*
3. *TreeMap(Map<? extends K, ? extends V> m)*
4. *TreeMap(SortedMap<K, ? extends V> sm)*

*TreeMap* defines no method of its own.

* *LinkedHashMap*

*LinkedHashMap* extends *HashMap* and maintains the order of entries by the order they were inserted. Alternatively, it can be sorted by the order the elements were last accessed.

It has the constructors

1. *LinkedHashMap()*
2. *LinkedHashMap(Map<? extends K, ? extends V> m)*
3. *LinkedHashMap(int capacity)*
4. *LinkedHashMap(int capacity, float fillRatio)*
5. *LinkedHashMap(int capacity, float fillRatio, boolean order)* – *order*==true: access order; *order*==false: insertion order.

The only method defined by *LinkedHashMap* is *removeEldestEntry()*:

*protected boolean removeEldestEntry(Map.Entry<K, V> e)*

by default this returns false and does nothing. To make it remove the oldest entry, override it and make it return true.

* *IdentityHashMap*

*IdentityHashMap* extends *AbstractMap* and implements *Map*. It is not for general use.

* *EnumMap*

*EnumMap* extends *AbstractMap* and implements *Map*. It is for use with keys of an *enum* type. It has the declaration

*class EnumMap<K extends enum<K>, V>*

It has the constructors

1. *EnumMap(Class<K> kType)* – creates an empty *EnumMap* of type *kType*
2. *EnumMap(Map<K, ? extends V> m)*
3. *EnumMap(EnumMap<K, ? extends V> em)* – creates an *EnumMap* initialized with the values of *em*

and defines no methods.

*Comparators*

*TreeSet* and *TreeMap* store elements in sorted order. *Comparator* defines what this means. By default, java uses “natural orering”; alternatively, specify a different *Comparator* when constructing the set or map.

*Comparator* is declared with

*interface Comparator<T>*

where *T* is the type of object being compared.

Methods:

These are the only 2 methods defined prior to JDK 8.

|  |  |
| --- | --- |
| **Method** | **Description** |
| *int compare(T obj1, T obj2)* | must be overridden by implementing class. Used to alter the way objects are ordered. |
| *boolean equals(Object obj)* | simple comparators won’t need to override *equals()* |
| *default Comparator<T> reversed()* | reverses the ordering of the calling comparator. |
| *static <T extends Comparable<? super T>> Comparator<T> reverseOrder()* | returns a comparator with reverse natural ordering. |
| *static <T extends Comparable<? super T>> Comparator<T> naturalOrder()* | returns a comparator with natural ordering. |
| *static <T> Comparator<T> nullsFirst(Comparator<? super T> comp)* | returns a comparator that views null as lesser/greater than other values. It compares non-null values as *comp* does. |
| *static <T> Comparator<T> nullsLast(Comparator<? super T> comp)* |  |
| *default Comparator<T> thenComparing(Comparator<? super T> thenByComp)* | returns a comparator that performs a second comparison when the first comparison indicates the objects are equal (“compare by X then compare by Y”) |
| *default <U extends Comparable<? super U>> Comparator<T> thenComparing(Function<? super T, ? extends U> getKey)* | these 2 versions specify the functional interface *Function*. (U is the type of the key) *getKey* is the function that obtains the second comparison. |
| *default <U> Comparator<T> thenComparing(Function<? super T, ? extends U> getKey, Comparator<? super U> keyComp)* |  |
| *thenComparingDouble(ToDoubleFunction<? super T> getKey)* |  |
| *thenComparingInt(ToDoubleFunction<? super T> getKey)* |  |
| *thenComparingLong(ToDoubleFunction<? super T> getKey)* |  |
| *static <T, U extends Comparable<? super U>> Comparator<T> comparing(Function<? super T, ? extends U> getKey)* | returns a comparator that obtains its comparison key from *getKey*. |
| *static <T, U> comparator<T> comparing(Function<? super T, ? extends U> getKey, Comparator<? super U> keyComp)* | *keyComp* will be used to compare keys. |
| *comparingDouble(ToDoubleFunction<? super T> getKey)* |  |
| *comparingInt(ToIntFunction<? super T> getKey)* |  |
| *comparingLong(ToLongFunction<? super T> getKey)* |  |

Prior to JDK 8, *Comparator* only defined *compare()* and *equals()*.

Example 1-1: creating and using a custom comparator that operates *compare()* in reverse.

**import** java.util.\*;

**class** Comp **implements** Comparator<String>{

//implementors of Comparator<T> must implement compare(T, T)

**public** **int** compare(String str1, String str2) {

**return** str2.compareTo(str1);

//reverse the operation of compare()

}

}

**class** Nonempty {

**public** **static** **void** main(String args[]) {

TreeSet<String> ts = **new** TreeSet<String>(**new** Comp());

ts.add("C");

ts.add("A");

ts.add("D");

ts.add("E");

ts.add("B");

**for**(String content : ts)

System.***out***.print(content + " "); //displays E D C B A

}

}

Example 1-2: the above can also be accomplished by calling *reversed()* on a natural order comparator.

**import** java.util.\*;

**class** Comp **implements** Comparator<String>{

**public** **int** compare(String str1, String str2) {

//create a natural order comparator

**return** str1.compareTo(str2);

}

}

**class** Nonempty {

**public** **static** **void** main(String args[]) {

Comp c = **new** Comp();

TreeSet<String> ts = **new** TreeSet<String>(c.reversed());

ts.add("C");

ts.add("A");

ts.add("D");

ts.add("E");

ts.add("B");

**for**(String content : ts)

System.***out***.print(content + " "); //displays E D C B A

}

}

*Comparator<T>* is a functional interface.

Example 1-3 and 1-4: a lambda expression can create the above comparator without the class *Comp*.

**import** java.util.\*;

**class** Nonempty {

**public** **static** **void** main(String args[]) {

Comparator<String> c = (str1, str2) -> str1.compareTo(str2);

TreeSet<String> ts = **new** TreeSet<String>(c.reversed());

ts.add("C");

ts.add("A");

ts.add("D");

ts.add("E");

ts.add("B");

**for**(String content : ts)

System.***out***.print(content + " "); //displays E D C B A

}

}

Example 1-4:

**import** java.util.\*;

**class** Nonempty {

**public** **static** **void** main(String args[]) {

TreeSet<String> ts = **new** TreeSet<String>((str1, str2) -> str2.compareTo(str1));

ts.add("C");

ts.add("A");

ts.add("D");

ts.add("E");

ts.add("B");

**for**(String content : ts)

System.***out***.print(content + " "); //displays E D C B A

}

}

Example 2-1: create and use a comparator that orders names by last name. In the case where 2 keys have the same last name, the first name is compared.

**import** java.util.\*;

**class** Comp **implements** Comparator<String> {

**public** **int** compare(String str1, String str2) {

**int** i, j, k;

i = str1.lastIndexOf(" ");

//find index of first letter of last name

j = str2.lastIndexOf(" ");

k = str1.substring(i).compareToIgnoreCase(str2.substring(j));

//create substrings containing the last names and compare them

**if**(k==0) //check entire name if last names match

**return** str1.compareToIgnoreCase(str2);

**else**

**return** k;

}

}

**class** Nonempty {

**public** **static** **void** main(String args[]) {

TreeMap<String, Double> tm = **new** TreeMap<String, Double>(**new** Comp());

tm.put("Florida Man", -3600.78);

tm.put("John Anderson", 123.4);

tm.put("Jimmy Carpenter", 1000.);

tm.put("Mick Gordon", 6666.66);

tm.put("Mary Beckett", 9999.9);

tm.put("Gary Carpenter", 9999.8);

tm.put("Florida Man", tm.get("Florida Man") + 1000000); //deposit 1000000 into Florida Man's account

Set<Map.Entry<String, Double>> set = tm.entrySet();

**for**(Map.Entry<String, Double> content : set) {

System.***out***.print(content.getKey() + ": ");

System.***out***.println(content.getValue());

}

System.***out***.println();

}

}

Example 2-2: using *thenComparing()* to accomplish the same task. *thenComparing()* specifies a second comparator to use if the invoking comparator returns equal.

**import** java.util.\*;

**class** lastName **implements** Comparator<String> {

**public** **int** compare(String str1, String str2) {

**int** i, j;

i = str1.lastIndexOf(" ");

//find index of first letter of last name

j = str2.lastIndexOf(" ");

**return** str1.substring(i).compareToIgnoreCase(str2.substring(j));

}

}

**class** fullName **implements** Comparator<String> {

**public** **int** compare(String str1, String str2) {

**return** str1.compareToIgnoreCase(str2);

}

}

**class** Nonempty {

**public** **static** **void** main(String args[]) {

lastName ln = **new** lastName();

Comparator<String> nameComp = ln.thenComparing(**new** fullName());

TreeMap<String, Double> tm = **new** TreeMap<String, Double>(nameComp); //rest are the same

tm.put("Florida Man", -3600.78);

tm.put("John Anderson", 123.4);

tm.put("Mick Gordon", 6666.66);

tm.put("Mary Beckett", 9999.9);

tm.put("Gary Carpenter", 9999.8);

tm.put("Jimmy Carpenter", 1000.);

tm.put("Florida Man", tm.get("Florida Man") + 1000000); Set<Map.Entry<String, Double>> set = tm.entrySet();

**for**(Map.Entry<String, Double> content : set) {

System.***out***.print(content.getKey() + ": ");

System.***out***.println(content.getValue());

}

System.***out***.println();

}

}

**The Collection Algorithms**

The cllection framework defines several algorithms – static methods within the *Collections* class. They have the following functions:

1. sorting
2. shuffling
3. routine data manipulation – reverse, fill, copy, swap, addAll
4. searching
5. composition – frequency, disjoint
6. finding extreme values
7. checking – return a runtime type-safe view (“dynamically typesafe view”) of a collection (disallows inserting incompatible elements)
8. empty – return an empty collection
9. unmodifiable – return an unmodifable copy of a collection
10. synchronized – return a synchronized copy of a collection

|  |  |
| --- | --- |
| **Algorithm** | **Description** |
| *static <T> boolean addAll(Collection <? super T> c, T … elements)* | add *elements* to *c*. Returns true if added successfully. |
| *static <T> int binarySearch(List<? extends T> list, T target, Comparator<? super T> c)* | search for *target* in *list* ordered according to *c*. returns the position of *target*. |
| *static <T> int binarySearch(List<? extends Comparable<? super T>> list, T target)* | search for *target* in *list*. *list* must be sorted. |
| *static <T> Queue<T> asLifoQueue<Deque<T> c)* | returns a last in first out view of *c*. |
| *static void copy(List<? super T> list1, List<? extends T> list2)* | copies the elements of *list2* to *list1.* |
| *static Enumeration enumeration(Collection<?> c)* |  |
| *static void fill(List list, Object obj)* | assigns *obj* to every element of *list.* |
| *static boolean replaceAll(List list, Object old, Object new)* |  |
| *static void reverse(List list)* |  |
| *static Comparator reverseOrder()* | returns a comparator in reverse order. |
| *static Object max(Collection c, Comparator comp)* | returns the maximum element in *c* determined by *comp* (*comp* optional). |
| *static Object min(Collection c, Comparator comp)* |  |
| *static boolean disjoint* |  |

Example

**import** java.util.\*;

**class** Nonempty {

**public** **static** **void** main(String args[]) {

LinkedList<Integer> ll = **new** LinkedList<>();

ll.add(1);

ll.add(2);

ll.add(3);

Comparator<Integer> revComp = Collections.*reverseOrder*();

//create a reverse order comparator

Collections.*sort*(ll, revComp); //sort ll with the reverse comparator

System.***out***.println("displaying reverse list: ");

**for**(Integer i: ll)

System.***out***.print(i + " ");

System.***out***.println();

Collections.*shuffle*(ll); //shuffle ll

System.***out***.println("displaying shuffled list: ");

**for**(Integer i: ll)

System.***out***.print(i + " ");

System.***out***.println();

Collections.*sort*(ll); //return ll to natrual ordering

System.***out***.println("maximum is: " + Collections.*max*(ll));

System.***out***.println("minimum is: " + Collections.*min*(ll));

}

}

**The *Arrays* Class**

The *Arrays* class provides various methods that bridge the gap between collections and arrays.

|  |  |
| --- | --- |
| **Method** | **Description** |
| *static <T> List asList(T… array)* | returns a list backed by *array*. |
| *static int binarySearch(Object array[], Object target)* |  |
| *static <T> T[] copyOf(T[] source, int length)* | copies *source* to an array with length *length*. Extra spaces filled with 0s/nulls/false’s. |
| *static <T> T[] copyOfRange(T[] source, int start, int end)* | returns a copy of a range within *source* from *start* to *end-1.* |
| *static boolean equals(Object array1[], Object array2[])* |  |
| *static boolean deepEquals(Object[] a, Object b)* | also checks any nested arrays within *a* and *b*. |
| *static void fill(Object array[], Object valueToFill)* |  |
| *static <T> void sort(T array[], Comparator<? super T> c)* | comparator is optional. |
| *static <T extends Comparable<? super T>> void parallelSort(T array[])* | a faster *sort()*. Can provide a comparator as well. |
| *static <T> Spliterator spliterator(T array[])* |  |
| *static <T> Stream stream(T array[])* |  |
| *static void setAll(double array[], IntToDoubleFunction<? extends T> genVal)* | assigns the result of *genVal* to all elements of *array*. |
| *static int compare(Object[] array)* |  |
| *static int compareUnsigned(Object[] array)* |  |
| *static int mismatch(Object[] array)* | returns the location of the first mismatch between 2 arrays. |

Example

**import** java.util.\*;

**class** Nonempty {

**public** **static** **void** main(String args[]) {

**int** array[] = **new** **int**[10];

**for**(**int** i = 0; i<array.length; i++)

array[i] = -1 \* i;

Arrays.*sort*(array);

**for**(**int** i = 0; i<array.length; i++)

System.***out***.print(array[i] + " ");

System.***out***.println();

System.***out***.println(Arrays.*binarySearch*(array, -2));

Arrays.*fill*(array, 1);

**for**(**int** i = 0; i<array.length; i++)

System.***out***.print(array[i] + " ");

}

}

**Legacy Classes and Interfaces**

Prior to the collection framework, several classes and an interface provide an ad hoc method of storing objects.

The legacy interface is:

* *Enumeration –* superceded by *Iterator*

The legacy classes are:

* *Vector* – superceded by *ArrayList*; updated for collection framework
* *Stack* – implements a last-in, first-out stack
* *Dictionary* – obsolete, fully superceded by *Map*
* Hashtable – concrete implementation of *Dictionary*; superceded by *HashMap*; updated for collection framework
* Properties ­– subclass of *Hashtable*, maintains lists of values in which the keys and values are both Strings. The value is called the property in a *Properties* object.
  + information contained in a *Properties* object can be easily stored or loaded with *store()* and *load()*. This makes it convenient for implementing simple databases.

example: a phonebook program

**import** java.util.\*;

**import** java.io.\*;

**class** Nonempty {

**public** **static** **void** main(String args[]) **throws** IOException {

Properties phonebook = **new** Properties();

BufferedReader br = **new** BufferedReader(**new** InputStreamReader(System.***in***));

String name, number;

FileInputStream fin = **null**;

**boolean** changed = **false**;

**try** {

fin = **new** FileInputStream("phonebook.dat");

//open phonebook.dat

} **catch**(FileNotFoundException e) {

}

**try** {

**if**(fin != **null**) {

phonebook.load(fin); //load contents of phonebook.dat

fin.close();

}

} **catch**(IOException e) {

System.***out***.println("error reading files");

}

**do** {

System.***out***.println("enter name ('quit' to stop): ");

name = br.readLine();

**if**(name.equals("quit")) **continue**;

System.***out***.println("enter number: ");

number = br.readLine();

phonebook.setProperty(name, number);

changed = **true**;

} **while**(!name.contentEquals("quit"));

**if**(changed) {

FileOutputStream fout = **new** FileOutputStream("phonebook.dat"); //save the file if it was changed

phonebook.store(fout, "phone book");

fout.close();

}

**do** {

System.***out***.println("enter the name to find ('quit' to quit): "); //look up a name

name = br.readLine();

**if**(name.equals("quit")) **continue**;

number = (String) phonebook.get(name);

System.***out***.println(number);

} **while**(!name.equals("quit"));

}

}

bkmk

CH20

***java.util* Part 2**

This chapter examines *java.util* classes and interfaces that are not part of the collection framework.

*StringTokenizer*

**Parsing** – division of text into a set of discrete parts (tokens). The *StringTokenizer* class provides the first step in the parsing process, known as the **lexer (lexical analyzer)** or **scanner**.

*StringTokenizer* implements *Enumeration*, thus one can enumerate individual tokens within a string with it.

*StringTokenizer* is mainly used for legacy code, the modern alternative is regular expression (ch30).

To use *StringTokenizer*, specify an input string and a string containing delimiters (characters that separate tokens) e.g. “,;:”. It has the constructors

1. *StringTokenizer(String input*
2. *StringTokenizer(String input, String delimiters)*
3. *StringTokenizer(String input, String delimiters, boolean delimAsToken)*

*StringTokenizer* methods

* *int countTokens()* – returns the number of tokens left to be parsed
* *boolean hasMoreTokens()*
* *boolean hasMoreElements()*
* *Object nextElement()*
* *Object nextToken()*

example: parse “key=value” pairs, separated by semicolons.

**import** java.util.StringTokenizer;

**class** Nonempty {

**static** String *str* = "key1=value:1;" + "key2=value:2;" + "key3=value:3";

**public** **static** **void** main(String args[]) {

StringTokenizer st = **new** StringTokenizer(*str*, "=;");

**while**(st.hasMoreTokens()) {

String key = st.nextToken();

String val = st.nextToken();

System.***out***.println(key + "\t" + val);

/\* displays

key1 value:1

key2 value:2

key3 value:3

\*/

}

}

}

*BitSet*

The *BitSet* class creates a special type of array that holds bit values in boolean forms, with variable size. It has the constructors

* *BitSet()*
* *BitSet(int size)*

all bits are initialized to false. To set a bit is to make it true.

Methods:

* *void and(BitSet bs)* – ANDs the contents of the invoking *BitSet* with *bs* and place the result into the invoking *BitSet*.
  + related: *or(), xor()*
* *void andNot(BitSet bs)* – for each set bit in *bs*, clear the corresponding bit in the invoking *BitSet*.
* *int cardinality()* – returns the number of set bits in the invoking *SetBit*.
* *void clear()*
* *Object clone()*
* *boolean equals(Object bs)*
* *void flip(int index)* – reverses the bit at *index.*
* *boolean get(int index)*
* *boolean intersects(BitSet bs)* – returns true if at least one pair of corresponding bits in the invoking *BitSet* and *bs* are set.
* *boolean isEmpty()*
* *int length()* – returns the number of bits required to hold the contents of the invoking *BitSet*.
* *int size()*
* *int nextClearBit(int startIndex)*
  + related: *nextSetBit(), previousClearBit(), previousSetBit()*
* *void set(int index, boolean b)* – sets the bit at *index* to *b*.
* *byte[] toByteArray()*
* *long[] toLongArray()*
* *static BitSet valueOf(byte[] v)* – returns a *BitSet* containing the bits in *v*.
  + additional parameter types: *long[], ByteBuffer, LongBuffer*

example

**import** java.util.BitSet;

**class** Nonempty {

**public** **static** **void** main(String args[]) {

BitSet even = **new** BitSet(6);

BitSet odd = **new** BitSet(6);

**for**(**int** i=0; i<6; i++) {

**if**((i%2)==0) even.set(i);

**if**((i%2)!=0) odd.set(i);

}

System.***out***.println("displaying b1: " + even);

System.***out***.println("displaying b2: " + odd);

odd.and(even);

System.***out***.println("displaying odd AND even: " + odd);

//{1,3,5}&{0,2,4} == {}

odd.or(even);

System.***out***.println("displaying odd OR even: " + odd);

//{}|{0,2,4} == {0,2,4}

odd.xor(even);

System.***out***.println("displaying odd XOR even: " + odd);

//{0,2,4}^{0,2,4} == {}

}

}

*Optional, OptionalDouble, OptionalInt, OptionalLong*

Added in JDK 8, these classes offer a way to handle situations in which a value may or may not be present. Previously one would use *null*, but this may lead to exceptions.

*Optional* is declared with

*class Optional<T>*

An *Optional* instance may contain a type *T* value or be empty. It defines no constructors. Create an instance by calling one of its methods, e.g. *of()*, *empty()*.

Methods:

* *boolean isPresent()* – returns true if the invoking object contains a value. (added JDK11)
* *boolean isEmpty()*
* *T get()* – returns the value; throws an exception if value dne
* *T orElse(T defVal)* – returns the invoking object’s value or *defVal* if it doesn’t have one
  + related: *orElseGet()*­, *orElseThrow()*
  + these methods substitute for calling *isPresent()* then *get()*
* *static <T> Optional<T> empty()* ­– returns an *Optional* object with no value
* *boolean equals(Object comparand)*
* *void ifPresent(Consumer<? super T> func)* – calls *func* if a value is present and passes the value to *func*.
* *void ifPresentOrElse(Consumer<? super T> func, Runnable onEmpty)*
* *static <T> Optional<T> of(T val)* – creates an *Optional* instance containing *val*.
  + *ofNullable()* allows null parameter

Example:

**import** java.util.\*;

**class** Nonempty {

**public** **static** **void** main(String args[]) {

Optional<String> zero = Optional.*empty*();

Optional<String> abc = Optional.*of*("abc");

**if**(zero.isEmpty()) System.***out***.println("zero is empty");

**if**(zero.isPresent()) System.***out***.println("abc has value");

String str = zero.orElse("alternate string");

//assign "alternate string" to str

}

}

*OptionalDouble*, *OptionalInt*, *OptionalLong* work much like *Optional*, except they only work with their respective types. Their methods are called “*getAsDouble()”* “*getAsInt()*” etc.

*Date*

The *Date* class implements *Comparable* and encapsulates the current date and time. Many of its methods are deprecated by Java 1.1 and moved to *Calendar* and *DateFormat*. It defines 2 constructors:

* *Date()* – initializes with current date and time.
* *Date(long millisec)* – initiates with *millisec* ms elapsed since Jan 1 1970.

Methods:

* *boolean after(Date date)* – returns true if the invoking *Date* contains a date later than *date*
* *boolean before(Date date)*
* *int compareTo(Date date)*
* *Object clone()*
* *boolean equals(Object date)*
* *static Date from(instant t)* – returns a *Date* object corresponding to *t*
  + note: *instant* describes a instantaneous point in time
* *Instant toInstant()*
* *long getTime()* – returns the time passed in Jan 1 1970.
* *int hashCode()*
* *void setTime(long time)* – sets the time to the given *time*
* *String toString()*

The date and time can be obtained in terms of milliseconds, or as an *Instant* object. For more detailed information about date and time, use *Calendar* class instead.

Example:

**import** java.util.\*;

**class** Nonempty {

**public** **static** **void** main(String args[]) {

Date date = **new** Date();

System.***out***.println(date); //returns "Sat May 09 13:25:18 EDT 2020"

**long** ms = date.getTime();

System.***out***.println("Time in milliseconds passed since Jan 1 1970 GMT is: " + ms);

}

}

*Calendar*

The abstract class *Calendar* allows its subclasses to convert time from milliseconds to more useful formats. It defines no public constructors. It has the subclass *GregorianCalendar*.

*Calendar* defines several *protected* instance variables:

* *boolean areFieldsSet* – indicates if the time components have been set
* *int[] fields* – holds the components of the date and time. The components are *Calendar.HOUR*, *Calendar.DATE*, *Calendar.MONTH*, etc.
* *boolean[] isSet* – indicates if each time component has been set
* *long time* – holds the current time
* *boolean isTimeSet* – indicates if the current time has been set

Methods:

* *static Calendar getInstance()* – returns a ­*Calendar* object for the default locale and time zone.
  + the returned object will be concrete.
* *static Calendar getInstance(TimeZone tz, Locale locale)* – *tz* and *locale* are optional
* *int get(int field)* – returns the value of the time component specified by *field*.
* *final Date getTime()*
* *TimeZone getTimeZone()*
* *abstract void add(int which, int val)* – adds (as in +) *val* to the time/date component specified by *which*.
* *boolean after(Object cal)* – returns true if the invoking *Calendar* object has a date that is later than the one specified by *cal*.
* *boolean before(Object cal)*
* *boolean equals(Object cal)*
* *final void clear(int which)* – clears the time component specified by *which*
* *final void clear()*
* *Object clone()*
* *void set(int which, int val)*
* *void set(int year, int month, int day, int hour, int min, in sec)* – the *hour*, *min*, *sec* are optional
* *final boolean isSet(int which)*
* *final void setTime(Date d)*
* *void setTimeZone(TimeZone tz)*
* *final Instant toInstant()*

Example:

**import** java.util.\*;

**class** Nonempty {

**public** **static** **void** main(String args[]) {

String[] months = {"Jan", "Feb", "Mar", "Apr",

"May", "Jun", "Jul", "Aug",

"Sep", "Oct", "Nov", "Dec" };

Calendar cal = Calendar.*getInstance*();

//this line fulfills the same purpose:

// GregorianCalendar cal = new GregorianCalendar();

System.***out***.println("The current date is: " + months[cal.get(Calendar.***MONTH***)] + " "+ cal.get(Calendar.***DATE***) + " " + cal.get(Calendar.***YEAR***));

//change the date to April 1 2030 1pm

cal.set(Calendar.***DATE***, 1);

cal.add(Calendar.***MONTH***, -1);

cal.add(Calendar.***YEAR***, 10);

cal.set(Calendar.***HOUR***, 13);

}

}

*GregorianCalendar*

*GregorianCalendar* is a concrete implementation of *Calendar*. Calling *Calendar.getInstance()* typically returns a *GregorianCalendar* instance.

*GregorianCalendar* defines the fields *AD* and *BC*.

It has two constructors:

* GregorianCalendar(**int** year, **int** month, **int** day, **int** hour, **int** min, **int** sec)

The *hour*, *min* and *sec* are optional. (Time is set to midnight if time not provided) And

* GregorianCalendar(TimeZone tz, Locale lc)

One of *tz* or *lc* can be missing.

*GregorianCalendar* defines the method

* *boolean isLeapYear(int year)*

*TimeZone*

This abstract class deals with time zones, and computes daylight saving time. It only has the default constructor.

Key methods:

* *String getID()* – returns the name of the invoking *TimeZone*.
* *static TimeZone getTimeZone(String tzName)*
* *static String[] getAvailableIDs()* – returns all time zones in a String array.
* *void setID(String tzName)*
* *abstract boolean useDaylightTime()*
* *abstract int getOffset(int era, int year, int month, int dayOfMonth, int dayOfWeek, int ms)*

*SimpleTimeZone*

extends *TimeZone*. Has constructor

* SimpleTimeZone(int timeDelta, String tzName)

where *timeDelta* specifies the offset relative to GMT.

*Locale*

Produces objects that describe geographical or cultural regions. This allows the program to run differently depending on the environment (e.g. the date format is different from place to place). It has the constructors

* Locale(String Lang)
* Locale(String Lang, String country)
* Locale(String Lang, String country, String variant)

and methods

* *static void setDefault(Locale lc)* – sets the JVM’s default locale to that of *lc*.
* *static Locale getDefault()*
* *final String getDisplayCountry()*
* *final String getDisplayLanguage()*
* *final String getDisplayName()* – gets the description of the locale
* JDK 7 upgraded *Locale* with new features, methods, and the *Locale.Builder* class. New methods include *getScript()*, *toLanguageTag()*, and *getISOCountries()*.

*Random*

The *Random* class generates pseudorandom numbers, using uniformly distributed sequences. It has the constructors

* Random()
* Random(long seed)

where *seed* specifies the starting point of the sequence. Instantiating multiple *Random* with the same seed would result in the same sequence being used.

*Random*’s core methods are named *nextX()* and returns a random object (number or boolean) of type X, e.g.

* *int nextInt(int range)* – returns a random *int* from 0 to *range*.
* *int nextGaussian()* – returns a random number from a gaussian distribution centered at 0 with std=1.

Other methods:

* *void nextBytes(byte vals[])* – fills *vals* with randomly generated bytes.
* *void setSeed(long newSeed)*

JDK 8 added three methods that support the stream API :

* *DoubleStream doubles()*
* *IntStream ints()*
* *LongStream longs()*

each returns a stream containing a pseudorandom sequence.

*Timer* and *TimerTask*

*Timer* and *TimerTask* allows the scheduling of a task for execution at some future time. They are much simpler to use than the *Thread* equivalent. The 2 classes work together: *Timer* schedules and times the task for execution, whereas *TimerTask* contains the task to be executed.

*TimerTask* implements the *Runnable* interface, and has 1 constructor:

* protected TimerTask()

It defines the methods *run()*, *cancel()* and *scheduledExecutionTime()* (which returns the time of the last scheduled execution).

*Timer* has the constructors

* Timer()
* Timer(String TimerName, boolean daemonThread)

the parameters are optional. *daemonThread* determines if the timer uses a daemon thread (a daemon thread will only execute when the rest of the program does).

It defines the methods *schedule()* and *scheduleAtFixedRate()* which prepares a task for execution at a specified time and then repeat that task at a specified rate, *cancel()* which cancels the timer and *purge()* which deletes cancelled tasks. For non daemon threads, *cancel()* would need to be called to end the thread.

Example:

**import** java.util.\*;

**class** ttask **extends** TimerTask {

**public** **void** run() {

System.***out***.println("timer task executed.");

}

}

**class** Nonempty {

**public** **static** **void** main(String args[]) {

ttask myTask = **new** ttask();

Timer myTimer = **new** Timer();

myTimer.schedule(myTask, 1000, 500);

//schedule the task to an initial delay of 1s, then execute every 0.5s

**try** {

Thread.*sleep*(5000); //run for 5s

} **catch**(InterruptedException e) {}

myTimer.cancel();

}

}

*Currency*

*Currency* encapsulates information about currency.

*Formatter*

*Formatter* allows for diaplaying numbers, strings, time and date in various formats. It works by converting the content and storing it in a buffer, which can then be obtained by the program. The buffer can be supplied automatically, or manually specified (it can also be a file).

*Formatter* has the constructors:

* *Formatter()*
* *Formatter(Appendable buf)* – specifies a buffer. if null, a *StringBuilder* will be allocated.
* *Formatter(Appendable buf, Locale loc)*
* *Formatter(String filename) throws FileNotFoundException*
* *Formatter(String filename, string charset) throws FileNotFoundException, UnsupportedEncodingException*
* *Formatter(File outF) throws FileNotFoundException* – *outF* specifies the open file to receive the output
* *Formatter(OutputStream outStrm)*

It has the methods:

* *void close()* – closes the invoking *Formatter*, releasing resources. A closed *Formatter* cannot be reused.
* *void flush()* – flushes the format buffer, causing its content to be written to its destination (apply mostly to *Formatter* tied to a file).
* *Formatter format(String fmtString, Object …args)* – formats *args* according to *fmtString*. Returns the invoking object.
* *Formatter format(Locale loc, String format, Object …args)*
* *IOException ioException()* – returns the IOException (or null) thrown by the output destination
* *Locale locale()*
* *Appendable out()*
* *String toString()* – returns the formatted String.

Basics

Formatted strings are created with *format()*. *fmtString* has 2 types of contents: one is simply copied to the output buffer; the other contains format specifiers that define how arguments are to be displayed. A format specifier begins with a percent sign followed by a single character. In general, the number of arguments and format specifiers are equal, and they are matched from left to right.

Example:

Formatter fmt = new Formatter();

fmt.format(“formatting %s with Java: %d %f”, “numbers”, 10, 4.1);

which creates a *Formatter* containing “formatting numbers with Java: 10 4.1”. The specifiers *%s* (for String), *%d* (for integer), *%f* (for floating points) are replaced with the respective arguments on the right.

The formatted string is obtained by calling *toString()*, or passing the *Formatter* to *println()*. I.e. both of the following print the preceding example’s string:

String str = fmt.toString();

System.out.println(str);

or

System.out.println(fmt);

After using a *Formatter*, call *close()* on it to close it.

The complete list of format specifiers:

|  |  |
| --- | --- |
| Format Specifier | Converts |
| *%a*  *%A* | hexadecimal floating point |
| *%b*  *%B* | boolean |
| *%c*  *%C* | character (singular) |
| *%d* | decimal integer |
| *%e*  *%E* | scientific notation |
| *%f* | decimal floating point |
| *%g*  *%G* | uses *%e* or *%f* depending on the data |
| *%h*  *%H* | hash code |
| *%o* | octal integer |
| *%n* | inserts new line |
| *%s*  *%S* | String |
| *%t*  *%T* | time and date |
| *%x*  *%X* | hexadecimal integer |
| *%%* | inserts “%” |

The upper- and lower-case specifiers return letters in upper or lower-case respectively.

*%a* returns a string in the form of 0x*sig*p*exp* where *sig* is the fractional portion of the significand and and *exp* is the exponent. E.g. 512 turns into *0x1.0p9* in this format.

*%t* requires a suffix to specify the type of time and date, e.g. *&tb* indicates abbreviated month name, *&tD* indicates month/day/year.

Example:

**import** java.util.\*;

**class** exercise {

**public** **static** **void** main(String[] args) {

Calendar cal = Calendar.*getInstance*();

Formatter fmt = **new** Formatter();

fmt.format("%tr", cal); //display 12 hour time

System.***out***.println(fmt); //displays 5:17:34 PM

fmt.close();

fmt = **new** Formatter();

fmt.format("%tc", cal); //display complete time

System.***out***.println(fmt); //displays Thu May 28 17:19:34 2020

fmt.close();

fmt = **new** Formatter();

fmt.format("%tl:%tM", cal, cal); //display hour and minute

System.***out***.println(fmt); //displays 5:17

fmt.close();

}

}

*%n* and *%%* are unique in that they do not match arguments. Instead they display a new line or percent sign. e.g.

fmt.format("line 1 %n line 2 %d%%", 100);

displays

line 1

line 2 100%

Specifying a minimum field width

Placing an integer between % and the specifier code specifies the minimum field width. The output would be padded if it’s not long enough.

Examples:

1. *%5s* pads a string shorter than 5 characters with spaces
2. *%08d* pads a number shorter than 8 digits with 0s

Minimum field width modifiers are often used to produce tables so that the columns are aligned.

Example:

**for**(**int** i=1; i<=10; i++) {

Formatter fmt = **new** Formatter();

fmt.format("%4d %4d %4d", i, i\*i, i\*i\*i);

System.***out***.println(fmt);

fmt.close();

Specifying precision

Placing a period between % and the specifier code creates a precision specifier.

For floating points, this determines the number of decimal places displayed. E.g. *&10.4f* displays 4 decimal places and has a minimum width of 10.

For scientific notations, this determines the number of significant digits.

The default precision is 6.

For strings, the precision specifier determines the maximum length. E.g. *%5.7s* displays a string between 5 and 7 characters long.

Format Flags

Format flags are single characters following the %.

|  |  |
| --- | --- |
| Flag | Effect |
| *-* | left justification (add paddings on the right) |
| *#* | alternate conversion format |
| *0* | pad with 0s |
| *+* | precede positive output with + |
| *space* | precede positive output with space (can be used to line up positive and negative values) |
| *(* | enclose negative values with () (no longer displays “-“) |
| *,* | numeric values include grouping separators (e.g. 123,456) |

Argument Index

Normally, format specifiers match with arguments from left to right. Argument index allows control over which argument to match.

Argument index follows % and has the form *n$* where n is the index (beginning with 1).

Example:

fmt.format("%3$d %2$d %1$d", 1, 2, 3);

displays

3 2 1

One utility is to reuse an argument without having to type it again. Example:

fmt.format("%d in hexadecimal is %1$x", 255);

displays

255 in hexadecimal is ff

There is also relative index which reuses the last argument. Relative index follows % and has the form *<.* For example, the last example can be done with

fmt.format("%d in hexadecimal is %<x", 255);

Closing a Formatter

A *Formatter* can be closed with *close()*. Additionally, *Formatter* implements *AutoCloseable*, meaning it can be automatically closed with *try-with-resources* statements.

Example:

**import** java.util.\*;

**class** exercise {

**public** **static** **void** main(String[] args) {

**try** (Formatter fmt = **new** Formatter()) {

fmt.format("using %s with try-with-resources", "formatter");

System.***out***.println(fmt);

}

}

}

*printf()*

*printf()* automatically uses *Formatter* to create a formatted string and displays it on the console. This is more convenient than using *Formatter* directly. (More on Ch21)

*Scanner*

*Scanner* reads formatted input and converts it into binary. It can read from the console, a file, or any source that implements *Readable* or *ReadableByteChannel* (e.g. *InputStream*, *Path*, *String, FileReader*) (note: *System.in* is an object of type *InputStream*).

Basics

*Scanner* reads tokens from the source related to the *Scanner*. A token is delineated by delimiters (whitespace by default). *Scanner* reads a token by matching it with a pattern.

The majority of *Scanner* methods enter into 2 categories:

* *boolean hasNextX()* – where *X* is a data type. Returns true if the next token to be read is of type *X*.
  + *boolean hasNext()* returns true if any token is available
* *X nextX()* – returns the next token of type *X*.
  + *String next()* returns the next token

General steps to using *Scanner*:

1. call *hasNextX()* to determine if input of type *X* is available
2. if available, read it with *nextX()*
3. repeat until input is exhausted
4. close the *Scanner* with *close()*

Example: read the console until a non integer is encountered.

**import** java.util.\*;

**class** exercise {

**public** **static** **void** main(String[] args) {

Scanner sc = **new** Scanner(System.***in***);

**int** i;

**while**(sc.hasNextInt()) {

i = sc.nextInt();

System.***out***.println(i);

}

}

}

Example: compute user input until “done” is encountered.

**import** java.util.\*;

**class** exercise {

**public** **static** **void** main(String[] args) {

Scanner sc = **new** Scanner(System.***in***);

**int** count = 0;

**double** sum = 0.;

System.***out***.println("Enter numbers to average. Input \"done\" when done");

**while**(sc.hasNext()) {

**if**(sc.hasNextDouble()) {

sum += sc.nextDouble();

count++;

}

**else** {

String next = sc.next();

**if**(next.equals("done")) **break**;

**else** {

System.***out***.println("Error, input \"done\" to print result.");

**return**;

}

}

}

sc.close();

System.***out***.println(sum / count);

}

}

note: *nextDouble()* converts any number into a double, hence integers work too. Likewise for other *nextX()* methods.

Example: previous example but reading from a file.

**import** java.util.\*;

**import** java.io.\*;

**class** exercise {

**public** **static** **void** main(String[] args) **throws** IOException {

**int** count = 0;

**double** sum = 0.;

FileWriter fout = **new** FileWriter("avg.txt");

fout.write("1 2 3 4 5 6 done");

fout.close();

FileReader fin = **new** FileReader("avg.txt");

Scanner sc = **new** Scanner(fin);

**while**(sc.hasNext()) {

**if**(sc.hasNextDouble()) {

sum += sc.nextDouble();

count++;

}

**else** {

String next = sc.next();

**if**(next.equals("done")) **break**;

**else** {

System.***out***.println("Error, input \"done\" to print result.");

**return**;

}

}

}

sc.close();

System.***out***.println(sum / count);

}

}

Notice that *fin* is never closed. When a *Scanner* is closed, its associated *Readable* is also closed.

*Scanner* implements *AutoCloseable*, meaning it can be closed with try-with-resources statements.

Example: scanning multiple input types.

**while**(sc.hasNext()) {

**if**(sc.hasNextInt()) {

//does stuff

}

**else if** (sc.hasNextDouble()) {

//does stuff

}

**else if** (sc.hasNextBoolean()) {

//does stuff

}

**else** {

//does stuff

}

}

}

Note: keep in mind ints would be read as doubles by *hasNextDouble()*. The order of the *nextX()* methods is relevant.

Until now the tokens are delimited with whitespace. However it is possible to change delimiters by calling *useDelimiter()*. For example,

sc.useDelimiter(", \*");

tells *sc* to match a comma and zero or more whitespaces as delimiters.

The current delimiter can be obtained with *delimiter()*.

Other Features

* *String findInLine(Pattern patn)* – searches for the specified pattern in the next line. Return the matching token or *null*.

Example: locate the age field and return the age.

**import** java.util.\*;

**class** exercise {

**public** **static** **void** main(String[] args){

String ex = "Name: Tom Age: 28";

Scanner sc = **new** Scanner(ex);

sc.findInLine("Age:");

**if**(sc.hasNext())

System.***out***.println(sc.next());

**else**

System.***out***.println("pattern not found");

sc.close();

}

}

* *String findWithinHorizon(Pattern patn, int conut)* – searches for the specified pattern within the next *count* number of characters. Returns the matching token or *null*. If *count* is zero, then all input is searched.
* *Scanner skip(Pattern patn)* – when *patn* is matched, *skip()* advances beyond it.
* *Scanner reset()*
* *Stream<String> tokens()* – returns all tokens

*ResourceBundle, List ResourceBundle,* and *PropertyResourceBundle*

These 3 classes of *java.util* aid internationalization.

Ch21

**Input/Output: *java.io***

**The *File* Class**

*File* does not operate on streams; rather it deals with files directly.

A directory in Java is treated as a *File*, with a list of filenames that can be examined with *list()*.

*File* has the constructors

File(String directoryPath)

File(String directoryPath, String filename)

File(File dirObj, String filename)

File(URI uriObj)

where *dirObj* is a *File* object that specifies a directory; *uriObj* is a *URI* object that describes a file.

Examples:

File f1 = new File(“/”); //only specifies a directory

File f2 = new File(“/”, “autoexec.bat”);

File f3 = new File(f1, “autoexec.bat”); //f2 and f3 refer to the same file

note: if using backslash \ in directory as per Windows convention, specify its escape sequence *\\*.

*File* defines methods that obtain standard properties of a *File* object: *getPath(); getName(); getParent(); canWrite(); canRead(); exists();* etc.

Some require explanation:

* *boolean isFile()* – returns true if called on a file and false if called on a directory
* *boolean isAbsolute()* – returns true if the file has an absolute path (complete path)
* *long length()* – returns the size of the file in bytes
* *boolean renameTo(File newName)*
* *boolean delete()* – deletes a file, or a directory if it is empty

*File* also supports *compareTo()* since it implements *Comparable*.

*Directory*

Creating a *File* without a file name makes it a *Directory*. *isDirectory()* checks if a *File* is a directory. *list()* can be called on directories to return a list of files and directories inside.

Example: displaying the contents of JDK13

**import** java.io.\*;

**public** **class** FileEx {

**public** **static** **void** main(String[] args) {

String dir = "C:\\Program Files\\Java\\jdk-13.0.2";

File f1 = **new** File(dir);

**if**(f1.isAbsolute())

System.***out***.println("f1 specifies an absolute path");

**else**

System.***out***.println("f1 specifies a relative path");

**if**(f1.isDirectory()) {

System.***out***.println("contents of f1: ");

String s[] = f1.list();

**for**(**int** i=0; i<s.length; i++) {

File f = **new** File(dir + "\\" + s[i]);

**if**(f.isDirectory())

System.***out***.println(s[i] + " is a directory");

**else**

System.***out***.println(s[i] + " is a file");

}

} **else**

System.***out***.println("f1 is not a directory");

}

}

*FilenameFilter* Interface

*FilenameFilter* can be an argument in *list()* to restrict the files returned.

*FilenameFilter* defines only *accept()*, with the form

boolean accept(File directory, String filename)

which returns true for files that should be included in the *list*.

Example: use an implementation of *FilenameFilter* to only list .txt files.

**import** java.io.\*;

**class** ExtensionFilter **implements** FilenameFilter {

String ext;

ExtensionFilter(String extName) {

**this**.ext = "." + extName;

}

**public** **boolean** accept(File dir, String name) {

**return** name.endsWith(ext);

}

}

**public** **class** FileEx {

**public** **static** **void** main(String[] args) {

String dir = "C:\\Program Files\\Java\\jdk-13.0.2";

File f1 = **new** File(dir);

FilenameFilter filter = **new** ExtensionFilter("txt");

String s[] = f1.list(filter);

**for**(**int** i=0; i<s.length; i++)

System.***out***.println(s[i]);

}

}

*listFiles()*

*listFiles()* is an alternative to *list()*, and returns *File* arrays instead of Strings. *listFiles()* can also specify a *FilenameFilter* as its argument.

Creating Directories

*File* provides two methods for creating directories: *mkdir()* and *mkdirs()*. *mkdir()* can only create a directory on an already existing path; whereas *mkdirs()* can create a directory and all parents to the directory.

*AutoCloseable*, *Closeable*, *Flushable* Interfaces

*AutoCloseable*, defined in *java.lang*, provides support for try-with-resources statements. It defines only *close()*, which is called automatically at the end of a try-with-resources statement.

Closeable and *Flushable* are defined in *java.io*. *Closeable* extends *AutoCloseable*, thus any class that implements *Closeable* also implements *AutoCloseable*. It is implemented by all I/O classes that opens a stream.

*Flushable* allows a class to force buffered output to the written to the attached stream. It is implemented by all I/O classes that writes to a stream.

I/O Exceptions

Two exceptions are notable: *IOException* and *SecurityException*.

Two Ways to Close a Stream

Streams need to be closed when they are no longer needed, else memory leaks can occur.

To close a stream, either call *close()*, or use a try-with-resources statement.

The Stream Classes

Java I/O is based on 4 abstract classes: *InputStream*, *OutputStream* (designed for byte streams), *Reader*, *Writer* (designed for character streams). Byte streams should be used when working with binary objects (though it can be used with any type of object); character streams should be used when working with characters/strings.

The Byte Streams

*InputStream*

*OutputStream*

*FileInputStream*

*FileInputStream* creates an *InputStream* that can read a file. It has the constructors

FileInputStream(String filePath)

FileInputStream(File fileObj)

Example: two ways to link to the same file.

FileInputStream f0 = new FileInputStream(“/autoexec.bat”);

File f = new File(“/autoexec.bat”);

FileInputStream f1 = new FileInputStream(f);

Example: reading the .java file itself, and showing how to read, skip and inspect available data.

**import** java.io.\*;

**public** **class** FileEx {

**public** **static** **void** main(String[] args) {

**int** size;

**try** (FileInputStream f = **new** FileInputStream("C:\\Users\\Myron\\eclipse-workspace\\File\\src\\FileEx.java")) {

System.***out***.println("total available bytes is: " + (size = f.available()));

**int** n = size/40;

System.***out***.println("Reading the first " + n + " bytes of the file");

**for**(**int** i=0; i<n; i++) { //first way to read

System.***out***.print((**char**) f.read());

} //read() returns the next byte of data as int

System.***out***.println();

System.***out***.println("Still available bytes: " + f.available());

System.***out***.println("Reading the next " + n + " bytes simultaneously");

**byte** b[] = **new** **byte**[n];

**if**(f.read(b) != n) {

//read(byte[] b) returns the number of bytes read

System.***err***.println("Couldn't read");

}

System.***out***.println(**new** String(b,0,b.length));//second way to read

System.***out***.println("\n Still available bytes: " + f.available());

f.skip(size/2); //skips over half of all bytes

System.***out***.println("Available bytes after skipping: " + f.available());

} **catch**(IOException e) {

System.***out***.println("I/O error: " + e);

}

}

}

*FileOutputStream*

*FileOutputStream* implements *AutoCloseable*, *Closeable*, *Flushable* and creates an output stream used to write bytes to a file. It has the constructors

FileInputStream(String filePath)

FileInputStream(File fileObj)

FileInputStream(String filePath, boolean append)

FileInputStream(File fileObj, boolean append)

*append* determines if the file is opened in append mode.

*FileOutputStream* does not require the file to already exist. If the file is not, it will be created when the *FileOutputStream* is created. However it cannot open read only files.

Example: converting a String to its byte array equivalent, then writing it to 3 files.

**import** java.io.\*;

**public** **class** OutputStreamEx {

**public** **static** **void** main(String[] args) {

String source = "This string will be \n"

+ "divided into 3 parts\n"

+ " and put into 3 files";

**byte** buf[] = source.getBytes();

FileOutputStream f0 = **null**;

FileOutputStream f1 = **null**;

FileOutputStream f2 = **null**;

**try** { //closing files manually

f0 = **new** FileOutputStream("file1.txt");

f1 = **new** FileOutputStream("file2.txt");

f2 = **new** FileOutputStream("file3.txt");

f0.write(buf); //writes everything

**for**(**int** i=0; i<buf.length; i+=2) //operates every other loop

f1.write(buf[i]);

f2.write(buf, buf.length-buf.length/4, buf.length/4);

//only writes the last quarter

} **catch**(IOException e) {

System.***out***.println("IO error.");

} **finally** {

**try** {

**if**(f0 != **null**) f0.close();

} **catch**(IOException e) {

System.***out***.println("error closing file1");

}

**try** {

**if**(f1 != **null**) f1.close();

} **catch**(IOException e) {

System.***out***.println("error closing file2");

}

**try** {

**if**(f2 != **null**) f2.close();

} **catch**(IOException e) {

System.***out***.println("error closing file");

}

}

}

}

*ByteArrayInputStream*

*ByteArrayInputStream* is an implementation of an input stream that uses a byte array as the source. It has the constructors

ByteArrayInputStream(byte[] array)

ByteArrayInputStream(byte[] array, int start, int numBytes)

the second specifies a subset of the array to create the *ByteArrayInputStream* from.

A byte array can be obtained by calling *getBytes()* on a String.

*close()* has no effect on a *ByteArrayInputStream*. It is not necessary to close a *ByteArrayInputStream*.

*ByteArrayInputStream* implements *mark()* and *reset()*. If *mark()* has not been called, *reset()* sets the stream pointer to the start of the array.

Example: initializing 2 *ByteArrayInputStream*’s, and using *reset()* to read one twice.

**import** java.io.\*;

**public** **class** ByteArrayInputStreamEx {

**public** **static** **void** main(String[] args) {

String source = "abcdefg";

String source2 = "abc";

**byte** b1[] = source.getBytes();

**byte** b2[] = source2.getBytes();

ByteArrayInputStream input1 = **new** ByteArrayInputStream(b1,0,3);

ByteArrayInputStream input2 = **new** ByteArrayInputStream(b2);

//both ByteArrayInputStream’s refer to "abc" as their source

**for**(**int** i=0; i<2; i++) {

**int** c;

**while**((c = input1.read()) != -1) {

**if**(i==0)

System.***out***.print((**char**) c);

**else**

System.***out***.print(Character.*toUpperCase*((**char**) c));

} //print first loop in lowercase and second loop in uppercase

System.***out***.println();

input1.reset();

}

}

}

*ByteArrayOutputStream*

*ByteArrayOutputStream* is an implementation of an output stream that uses a byte array as the destination. *ByteArrayOutputStream* has the constructors

ByteArrayOutputStream()

ByteArrayOutputStream(int bufSize)

The first form creates a buffer of 32 bytes; the second form specifies the buffer size. The buffer size is increased automatically if needed. The buffer is held in the protected *buf* field of *ByteArrayOutputStream*.

A byte array can be obtained by calling *getBytes()* on a String.

*close()* has no effect on *ByteArrayOutputStream*. It is not necessary to close a *ByteArrayOutputStream*.

Example: writing to the same *ByteArrayOutputStream* (resetting it in between) and then writing its content to an *OutputStream*.

**import** java.io.\*;

**public** **class** ByteArrayOutputStreamEx {

**public** **static** **void** main(String[] args) {

ByteArrayOutputStream f = **new** ByteArrayOutputStream();

String s = "example for bytearrayoutputsteram";

**byte** buf[] = s.getBytes();

**try** {

f.write(buf);

} **catch**(IOException e) {

System.***out***.println("error writing to buffer");

}

**byte** b[] = f.toByteArray(); //print the content of f

**for**(**int** i=0; i<b.length; i++)

System.***out***.print((**char**) b[i]);

System.***out***.println();

f.reset();

**for**(**int** i=0; i<3; i++)

f.write('x');

System.***out***.println(f.toString()); //now prints "xxx"

**try**(OutputStream op = **new** FileOutputStream("xxx.txt")) {

f.writeTo(op); //write “xxx” to “xxx.txt”

} **catch**(IOException e) {

System.***out***.println("IO error: " + e);

}

}

}

Filtered Byte Streams

Filtered streams are wrappers around input/output streams with some extended functionalities, such as buffering and translation. Their methods are identical to those in *InputStream* and *OutputStream*.

They have the constructors:

FilterOutputStream(OutputStream os)

FilterInputStream(InputStream is)

Buffered Byte Streams

For byte-oriented streams, a buffered stream extends a filtered stream class, and attaches a memory buffer to the I/O stream, allowing Java to perform I/O operations more than one byte at a time, improving performance.

The buffer also allows for skipping, marking, and resetting the stream.

The buffered byte streams are *BufferedInputStream*, *BufferedOutputStream*, *PushbackInputStream*. They are described in detail next.

*BufferedInputStream*

*BufferedInputStream* wraps around any Input Stream. It has the constructors

BufferedInputStream(InputStream is)

BufferedInputStream(InputStream is, int bufSize)

A good size for the buffer depends on the system configuration. A good guess is 8192 bytes.

Example: using *mark()* to mark a point in the stream and using *reset()* to go back to it.

**import** java.io.\*;

**public** **class** BufferEx {

**public** **static** **void** main(String[] args) {

String s = "this is a &copy; copyright symbol \n" +

"this &copy is not";

//HTML entity references begin with & and end with ;

**byte**[] buf = s.getBytes();

ByteArrayInputStream in = **new** ByteArrayInputStream(buf);

**int** c;

**boolean** mark = **false**;

**try**(BufferedInputStream f = **new** BufferedInputStream(in)) {

**while**((c = f.read()) != -1) {

**switch**(c) {

**case** '&':

**if**(!mark) {

f.mark(32);

mark = **true**;

} **else**

mark = **false**;

**break**;

**case** ';':

**if**(mark) {

mark = **false**;

System.***out***.print("(c)");

} **else**

System.***out***.print((**char**) c);

**break**;

**case** ' ':

**if**(mark) {

mark = **false**;

f.reset();

System.***out***.print("&");

} **else**

System.***out***.print((**char**) c);

**break**;

**default**:

**if**(!mark)

System.***out***.print((**char**) c);

**break**;

}

}

} **catch**(IOException e) {

System.***out***.println("IO error: " + e);

}

}

}

*BufferedOutputStream*

Unlike buffered input, buffered output has no additional functionality and only improves performances. Use *flush()* to write the contents of the buffer. *BufferedOutputStream* has the constructors

BufferedOutputStream(OutputStream os)

BufferedOutputStream(OutputStream os, int bufSize)

*PushbackInputStream*

*Pushback* reads a byte from an input stream and then returns it. This “peeks” at the coming byte without disrupting it. *PushbackInputStream* has the constructors

PushbackInputStream(InputStream is)

PushbackInputStream(InputStream is, int numBytes)

*numBytes* specifies the length of the pushback buffer.

*PushbackInputStream* has the method *unread()*:

void unread(int b)

void unread(byte[] buffer, int offset, int numBytes)

the first form pushes back a byte. The next *read()* will return *(byte) b*. The second form pushes back a portion of a byte array.

Example: use *unread()* to make the stream read “world”.

**import** java.io.\*;

**public** **class** PushbackIS {

**public** **static** **void** main(String[] args) {

**byte**[] buff = **new** **byte**[10];

**byte**[] hello = **new** **byte**[] {'H', 'e', 'l', 'l', 'o'};

InputStream is = **new** ByteArrayInputStream(hello);

**try**(PushbackInputStream pis = **new** PushbackInputStream(is, 10)) {

**for**(**int** i=0; i<hello.length; i++) {

buff[i] = (**byte**) pis.read();

System.***out***.print((**char**) buff[i]);

}

System.***out***.println();

**byte**[] b = **new** **byte**[] {'w', 'o', 'r', 'l', 'd'};

pis.unread(b);

**for**(**int** i=0; i<b.length; i++) {

buff[i] = (**byte**) pis.read();

System.***out***.print((**char**) buff[i]);

} //displays "world”

} **catch**(IOException e) {

System.***out***.println("IO error: " + e);

}

}

}

*SequenceInputStream*

*SequenceInputStream* allows concatenating multiple *InputStream*’s. Its constructor differs from other *InputStream*’s in that it uses either a pair of *InputStream*’s or an *Enumeration* of *InputStream*’s as its argument:

SequenceInputStream(InputStream is1, InputStream is2)

SequenceInputStream(Enumeration <? extends InputStream> streamEnum)

the class fulfills read requests from the first *InputStream* until it runs out, then switches over to the next one(s). Each file is closed when its stream reaches the end. Closing *SequenceInputStream* causes all unclosed streams to close.

Example:

**import** java.util.\*;

**import** java.io.\*;

**class** Enumerator **implements** Enumeration<FileInputStream> {

**private** Enumeration<String> files;

**public** Enumerator(Vector<String> f) {

**this**.files = f.elements();

}

**public** **boolean** hasMoreElements() {

**return** files.hasMoreElements();

}

**public** FileInputStream nextElement() {

**try** {

**return** **new** FileInputStream(files.nextElement().toString());

} **catch**(IOException e) {

**return** **null**; //must return something, else error occurs

//null causes NullPointerException, which is caught in main()

}

}

}

**public** **class** SequenceIS {

**public** **static** **void** main(String[] args) {

**int** c;

Vector<String> files = **new** Vector<String>();

files.addElement("file1.txt");

files.addElement("file2.txt");

files.addElement("file3.txt");

Enumerator ise = **new** Enumerator(files);

InputStream is = **new** SequenceInputStream(ise);

**try** {

**while**((c=is.read()) != -1)

System.***out***.print((**char**) c);

} **catch**(NullPointerException | IOException e) {

System.***out***.println("Exception occurred: " + e);

} **finally** {

**try** {

is.close();

} **catch**(IOException e) {

System.***out***.println("Error closing SequenceInputStream.");

}

}

}

}

**Skipping ahead. Skippoint = P1075**

*Console*

The *Console* class reads from and writes to the console. Most of its functionalities are also available through *System.in* and *System.out*. It can simiplify some console interactions, such as reading strings.

*Console* has no constructors. To obtain an instance, call *System.console()*, which returns a reference if possible.

Notably *Console* has the *readPassword()* method, which reads a string from the console without displaying it. In practice, the array holding the password should be erased after testing the password.

Example:

**import** java.io.\*;

**public** **class** ConsoleEx {

**public** **static** **void** main(String[] args) {

String str;

Console con;

con = System.*console*();

**if**(con == **null**) {

System.***out***.println("no console available");

**return**;

}

str = con.readLine("Enter a string: ");

con.printf("Here is your string %s\n", str);

}

}

CH22

**Exploring NIO**

NIO (new I/O) was added in java 1.4. JDK 7 enhanced NIO (sometimes called NIO.2). NIO does not replace java.io.

Fundamentals

NIO is built on **buffers** and **channels**. A buffer holds data. A channel is an open connection to an I/O device.

**Buffers**

Buffers are subclasses of the *Buffer* class, located in the *java.nio* package. All buffers have the following properties: current position, limit, capacity. (limit is the index value 1 past the last valid location; capacity is the number of elements the buffer can hold. They are often equal.) All buffers support marking and resetting.

Buffer classes are specific to the type of data they hold: *ByteBuffer, IntBuffer, FloatBuffer, CharBuffer, MappedByteBuffer* (maps a file to a buffer), etc.

Buffers have various *get()* and *put()* methods, which gets data from or gets data into a buffer. Other operations include *allocate()* (allocates buffer manually), *wrap()* (wraps an array inside a buffer), *slice()* (creates a subsequence of a buffer), etc.

notable *get()*’s and *put()*’s:

* *get()* – returns the byte at the current position
* *get(int index)* – returns the byte at the specified index
* *get(byte[] vals, int start, int num)* – copies elements from the buffer into *vals*.
* *put(byte b), put(byte[] vals, int start, int num)* – copies *b*/*vals* into the invoking buffer, beginning at current position.

**Channels**

Channels are defined in *java.nio.channels*, implements *Channel* and extends *Closeable* and *AutoCloseable*.

One way to obtain a channel is by calling *getChannel()*, supported on certain I/O classes such as *FileInputStream* and *FileOutputStream*. The type of channel returned depends on the type of the invoking object.

Another way to obtain a channel is by calling one of the static methods defined by *Files*, such as *newByteChannel()*.

Channel has *read()* and *write()*, which takes a *ByteBuffer bb* as argument. *read()* reads bytes from the invoking channel into *bb* until the buffer is full or the channel is exhausted; *write()* writes the contents of *bb* to the invoking channel.

*Charsets* and *Selectors*

*NIO.2* Enhancements

The *Path* Interface

*Path* is packaged in *java.nio.file* which encapsulates a path to a file.

notable methods:

* *Path getFileName()*
* *Path getName(int index)* – obtains an element in a path (index 0 refers to the path nearest the root)
* *int getNameCount()* – returns the number of elements in the invoking path beyond the root.
* *Path toAbsolutePath()* – returns the invoking *Path* as an absolute path.

The *Files* Class

*Files* provides many static methods that perform various actions on a file. The file is specified by its *Path*.

Several methods take type *OpenOption* arguments. *OpenOption* is an enumeration with the values:

* APPEND
* CREATE
* CREATE\_NEW (only creates the file if it does not already exist)
* DELETE\_ON\_CLOSE
* DSYNC
* READ
* SPARSE
* SYNC
* TRUNCATE\_EXISTING
* WRITE

The *Paths* Class

As of JDK 11, the new recommendation is to use the *Path.of()* method. It has the form

static Path of(String pathname, String …parts)

The parameters provide two different ways to pass the path: either as a single String, or as a series of Strings.

Prior to JDK 11, to get an instance of *Path*, *Paths* defines the *get()* method.

Obtaining a *Path* to a file does not open or create it.

The File Attribute Interfaces

Files has various attributes: size, time of creation, time of last modification, whether it is a directory, etc. Attributes are represented by various interfaces defined in *java.nio.file.attribute*.

The *FileSystem*, *FileSystems*, *FileStore* Classes

Using the NIO System

Because all file channel operations are byte-based, the type of buffer used will be *ByteBuffer*.

Before opening a file, a *Path* describing it is needed. The preferred way to do so is to use *Path.of()*.

Using NIO for Channel-based IO

* reading a file via a channel

The most common way to read data from a file using a channel: manually allocate a buffer and then explicitly read the file and write the data onto the buffer.

To open a file, first create its *Path*, then call *Files.newBytesChannel()*:

static SeekableByteChannel newByteChannel(Path path, OpenOption …how) throws IOException

*SeekableByteChannel* is an interface that describes a channel for file operations. The channel must be closed after using it.

The buffer will be obtained by calling *ByteBuffer.allocate()*.

Afterwards, call *read()* on the channel to pass data to the buffer. Each call fills the buffer with data from the file. *read()* returns the number of bytes read, or -1 when reaching the end of the file.

Example:

**import** java.io.\*;

**import** java.nio.\*;

**import** java.nio.channels.\*;

**import** java.nio.file.\*;

**public** **class** NIOChannel {

**public** **static** **void** main(String args[]) {

**int** count;

Path filepath = **null**;

**try** { //obtains a path

filepath = Path.*of*("NIOChannel.txt");

} **catch**(InvalidPathException e) {

System.***out***.println("Path error: " + e);

**return**;

}

**try**(SeekableByteChannel fCh = Files.*newByteChannel*(filepath)) { //obtains a channel

//no open option specified - opens the file for reading

ByteBuffer buf = ByteBuffer.*allocate*(128);

**do** {

count = fCh.read(buf);

**if**(count != -1) {

buf.rewind();

//resets the current position to the start of the buffer

//allows it to read again

**for**(**int** i=0; i<count; i++) {

System.***out***.print((**char**) buf.get()); //reads from the buffer

}

}

} **while**(count != -1);

System.***out***.println();

} **catch**(IOException e) {

System.***out***.println("IO error: " + e);

}

}

}

The above can be streamlined so that obtaining the *Path* and obtaining the channel is done in the same try block:

**try**(SeekableByteChannel fCh = Files.*newByteChannel*(Path.*of*("NIOChannel.txt"))) {//…

Another way to read a file is to map it to a buffer. This approach does not require any read operation, as the buffer automatically contains the contents of the file.

First, obtain a *Path* that describes the file.

Then obtain a channel to that file by calling *Files.newByteChannel()*. This returns a *SeekableByteChannel* which can be cast to *FileChannel*.

Then map this channel to a buffer by calling *map()* on the channel, which is defined by *FileChannel*. It has the form

MappedByteBuffer map(FileChannel.MapMode mode, long pos, long size) throws IOException

where *FileChannel.MapMode* determines what type of operations are allowed. It has the values *READ\_ONLY, READ\_WRITE, PRIVATE*. *MappedByteBuffer* is a subclass of *ByteBuffer*.

Example:

**import** java.io.\*;

**import** java.nio.\*;

**import** java.nio.channels.\*;

**import** java.nio.file.\*;

**public** **class** NIOMappedChannel {

**public** **static** **void** main(String[] args) {

**try**(FileChannel fCh =

(FileChannel) Files.*newByteChannel*(Path.*of*("NIOChannel.txt")) {

**long** size = fCh.size();

MappedByteBuffer buf = fCh.map(FileChannel.MapMode.***READ\_ONLY***, 0, size);

//maps the file to a buffer

**for**(**int** i=0;i<size;i++) {

System.***out***.print((**char**) buf.get());

}

} **catch**(IOException | InvalidPathException e) {

System.***out***.println("Error: " + e);

}

}

}

* Writing to a file via a *Channel*

The most common way to write to a file using a channel is by manually allocaating a buffer, writing to that buffer, and then perform a write operation to write to a file.

To open the file, first obtain a *Path* to it and obtain a *SeekableByteChannel* by calling *newByteChannel()*. It has the form

static SeekableByteChannel newByteChannel(Path path, OpenOption …how) throws IOException

The open options include *StandardOpenOption.WRITE*, *StandardOpenOption.CREATE*, etc.

Example: write the alphabet to a file.

**import** java.io.\*;

**import** java.nio.\*;

**import** java.nio.channels.\*;

**import** java.nio.file.\*;

**public** **class** ChannelOut {

**public** **static** **void** main(String[] args) {

**try**(FileChannel fCh =

(FileChannel) Files.*newByteChannel*(Path.*of*("ChannelOut.txt"), StandardOpenOption.***WRITE***, StandardOpenOption.***CREATE***)) {

ByteBuffer buf = ByteBuffer.*allocate*(26);

**for**(**int** i=0; i<26; i++)

buf.put((**byte**) ('A' + i)); //writes some bytes to the buffer

//writes the alphabet

buf.rewind();

//resets the buffer; otherwise write() will think there is no data in the buffer

//could also use flip() in this case

fCh.write(buf);

//this will overwrite the previous contents of ChannelOut.txt if it has any

} **catch**(InvalidPathException | IOException e) {

System.***out***.println("error: " + e);

}

}

}

In general, the buffer must be reset between read and write operations.

*flip()* is similar to *rewind()*; it sets the buffer’s current position as the new limit, and then resets the buffer’s position to zero. In cases where the buffer is already at the end, *flip()* is equivalent to *rewind()*.

Another way to write to a file is to map it to a buffer. This approach does not require any write operation, as the data written to the buffer is automatically written to the file.

Example: the previous example using the map approach

**import** java.io.\*;

**import** java.nio.\*;

**import** java.nio.channels.\*;

**import** java.nio.file.\*;

**public** **class** MappedChannelOut {

**public** **static** **void** main(String[] args) {

**try**(FileChannel fCh =

(FileChannel) Files.*newByteChannel*(Path.*of*("ChannelOut.txt"), StandardOpenOption.***WRITE***, StandardOpenOption.***READ***, StandardOpenOption.***CREATE***)) {

/\*because a mapped buffer can either be read-only or read/write, the READ option needs to be added \*/

ByteBuffer buf = fCh.map(FileChannel.MapMode.***READ\_WRITE***, 0, 26);

//map the file into a buffer

**for**(**int** i=0; i<26; i++)

buf.put((**byte**) ('A' + i));

//writes the alphabet

} **catch**(InvalidPathException | IOException e) {

System.***out***.println("error: " + e);

}

}

}

* Copying a file using NIO

One of the ways to copy files is by calling the static method ­*Files.copy()*:

static Path copy(Path src, Path dest, CopyOption …how) throws IOException

*CopyOption* can be

* *StandardCopyOption.COPY\_ATTRIBUTES*
* *StandardCopyOption.NOFOLLOW\_LINKS*
* *StandardCopyOption.REPLACE\_EXISTING*

Example: (notice that it is substantially shorter than the one in Ch.13)

**import** java.io.\*;

**import** java.nio.\*;

**import** java.nio.channels.\*;

**import** java.nio.file.\*;

**public** **class** NIOCopy {

**public** **static** **void** main(String[] args) {

**if**(args.length != 2) {

System.***out***.println("Usage: copy from to");

**return**;

}

**try** {

Path source = Path.*of*(args[0]);

Path tgt = Path.*of*(args[1]);

Files.*copy*(source, tgt, StandardCopyOption.***REPLACE\_EXISTING***);

} **catch**(InvalidPathException | IOException e) {

System.***out***.println("Error: " + e);

}

}

}

Using NIO for Stream-Based I/O

As of NIO.2, NIO can open an I/O stream. To do so, obtain a *Path* and call *newInputStream()* or *newOutputStream()*, static methods defined by *Files*. Once opened, any method defined by *InputStream* or *OutputStream* can be used.

Example: display a file’s content.

**import** java.io.\*;

**import** java.nio.file.\*;

**import** java.util.\*;

**public** **class** NIOStream {

**public** **static** **void** main(String[] args) {

Scanner sc = **new** Scanner(System.***in***);

System.***out***.println("Input path to file to open.");

String in = sc.next();

sc.close();

**int** i;

**try**(InputStream fin = Files.*newInputStream*(Path.*of*(in))) {

**do** {

i = fin.read();

**if**(i != -1) System.***out***.print((**char**) i);

} **while**(i != -1);

} **catch**(InvalidPathException | IOException e) {

System.***out***.println("Error caught: " + e);

}

}

}

Using NIO for Path and File System Operations

NIO.2 offers a better way to perform the functionalities of the *File* class. The following examines two example operations:

* obtaining information about a Path and a File

Some attributes can be accessed with *Path* methods, e.g. name, parent, path, absolute path; others with *Files* methods, e.g. *isExecutable(), isHidden(), isReadable(), exists()*. A list of attributes can be obtained by calling *Files.readAttributes()* which contains more attributes.

Example:

**import** java.io.\*;

**import** java.nio.file.\*;

**import** java.nio.file.attribute.\*;

**public** **class** NIOAttribute {

**public** **static** **void** main(String[] args) {

Path filepath = Path.*of*("file1.txt");

**if**(Files.*exists*(filepath))

System.***out***.println("File exists");

**else**

System.***out***.println("File does not exist");

System.***out***.println("File name: " + filepath.getName(0));

//filename is closest to the root since relative path

System.***out***.println("Path: " + filepath);

System.***out***.println("Absolute path: " + filepath.toAbsolutePath());

System.***out***.println("Parent: " + filepath.getParent());

**if**(Files.*isWritable*(filepath))

System.***out***.println("File is writable");

**if**(Files.*isReadable*(filepath))

System.***out***.println("File is readable");

**try** {

**if**(Files.*isHidden*(filepath))

System.***out***.println("File is hidden");

**else**

System.***out***.println("File is not hidden");

BasicFileAttributes attribs =

Files.*readAttributes*(filepath, BasicFileAttributes.**class**);

**if**(attribs.isDirectory())

System.***out***.println("The file is a directory");

**if**(attribs.isRegularFile())

System.***out***.println("The file is a normal file");

**if**(attribs.isSymbolicLink())

System.***out***.println("The file is a symbolic link");

System.***out***.println("File was last modified at " + attribs.lastModifiedTime());

System.***out***.println("File size is " + attribs.size() + "bytes");

} **catch**(IOException e) {

System.***out***.println("I/O error: " + e);

}

}

}

* List the contents of a directory

If a path describes a directory, then its contents can be read. To do so, first obtain a directory stream by calling *newDirectoryStream()*. *DiectoryStream<Path>* implements *AutoCloseable* and *Iterable<Path>*, so it can be iterated over. Each entry is represented by a *Path* instance.

Note that the iterator for a directory stream can only be obtained once.

Example:

**import** java.io.\*;

**import** java.nio.file.\*;

**import** java.nio.file.attribute.\*;

**public** **class** NIODirectory {

**public** **static** **void** main(String[] args) {

String directory = "C:\\Users\\Myron\\eclipse-workspace\\File";

**try**(DirectoryStream<Path> dirst = Files.*newDirectoryStream*(Path.*of*(directory))) {

System.***out***.println("Contents of directory " + directory + ":");

**for**(Path entry : dirst) {

BasicFileAttributes attribs = Files.*readAttributes*(entry, BasicFileAttributes.**class**);

**if**(attribs.isDirectory())

System.***out***.print("directory: ");

**else**

System.***out***.print("file: ");

System.***out***.println(entry.getName(4));

}

} **catch**(InvalidPathException | IOException e) {

System.***out***.println("Error: " + e);

}

}

}

The contents of a directory can be filtered in two ways:

The easiest way: use

static DirectoryStream<Path> newDirectoryStream(Path dirname, String wildcard)

which obtains files that match the wildcard. The wildcard can be either a complete filename or a glob – a string that defines a general pattern using the following

|  |  |
| --- | --- |
| glob | Effect |
| *\** | matches zero or more of any character |
| *?* | matches any character |
| *\*\** | matches zero of more of any character across directories |
| *[chars]* | matches any one character in *chars* (including \* and ?). Include a range using a hyphen, e.g. [x-z]. |
| *{globlist}* | matches any one of the globs specified in the list. Globs are comma separated. |

Example:

Files.newDirectoryStream(Path.of(dirname), “{Path,Dir}\*.{java,class}”)

which obtains a directory stream containing only files that begin with “Path” or “Dir” and end with “.java” or “.class”.

The second way to filter a directory is to use

static DirectoryStream<Path> newDirectoryStream(Path dirPath, DirectoryStream.Filter<? super Path> filefilter)

where *DirectoryStream.Filter* is an interface that specifies *Accept()* which returns true on certain criteria.

This form of *newDirectoryStream()* has the advantage of being able to filter based on something other than filename.

Example: filter to only include writable files

**import** java.io.\*;

**import** java.nio.file.\*;

**import** java.nio.file.attribute.\*;

**public** **class** NIOFilter {

**public** **static** **void** main(String[] args) {

String dirname = "C:\\Users\\Myron\\eclipse-workspace\\File\\.settings";

DirectoryStream.Filter<Path> how = **new** DirectoryStream.Filter<Path>() {

**public** **boolean** accept(Path filename) **throws** IOException {

**if**(Files.*isWritable*(filename))

**return** **true**;

**return** **false**;

}

}; //creates a filter that returns true for writable files

**try**(DirectoryStream<Path> dirst = Files.*newDirectoryStream*(Path.*of*(dirname), how)) {

System.***out***.println("Directory of " + dirname);

**for**(Path entry: dirst) {

BasicFileAttributes attribs =

Files.*readAttributes*(entry, BasicFileAttributes.**class**);

**if**(attribs.isDirectory())

System.***out***.print("<DIR> ");

System.***out***.print(" ");

System.***out***.println(entry.getName(5));

}

} **catch**(IOException e) {

System.***out***.println("IO error: " + e);

}

}

}

List a directory tree with *walkFileTree()*

*Files.walkFileTree()* allows processing multiple directories in a directory tree. It has the form

static Path walkFileTree(Path root, FileVisitor<? super Path> fv)

where *root* is the path to the starting point of the directory walk. *FileVisitor* is an interface that defines how files are visited when a directory is traversed. Its implementation determines how the directory tree is traversed.

*FileVisitor*’s methods return a *FileVisitResult*, an enumeration with the values *CONTINUE, SKIP\_SIBLINGS, SKIP\_SUBTREE, TERMINATE*.

A simple implementation of *FileVisitor* is provided in *SimpleFileVisitor*. Example: displaying all files in the directory tree with *//File* as its root.

Example:

**import** java.io.\*;

**import** java.nio.file.\*;

**import** java.nio.file.attribute.\*;

**class** MyFileVisitor **extends** SimpleFileVisitor<Path> {

**public** FileVisitResult visitFile(Path path, BasicFileAttributes attribs) **throws** IOException {

System.***out***.println(path);

**return** FileVisitResult.***CONTINUE***;

}

}

**class** VisitFile {

**public** **static** **void** main(String[] args) {

String dir = "C:\\Users\\Myron\\eclipse-workspace\\File";

**try** {

Files.*walkFileTree*(Path.*of*(dir), **new** MyFileVisitor());

} **catch**(IOException e) {

System.***out***.println("IO error: " + e);

}

}

}

In this example, *visitFile()* simply displays the files. More sophisticated actions include copying/filtering files or performing actions on them.

CH23

**Networking**

**Networking Basics**

A *socket* is an endpoint in a network. Numbered sockets on machines are called *ports*.

A server *listens* to a port for connection requests. If it accepts the request, the server establishes a connection between the server’s port and the client’s port.

Socket communication takes place via protocols.

* *Internet Protocol (IP)* is a low level protocol that breaks data into packets and send them.
* *Transmission Control Protocol (TCP)* is a higher level protocol responsible for sorting, checking and then reliably sending packets.
* *User Datagram Protocol (UDP)* offers fast, connectionless but unreliable transmission of packets.
* Once connection has been established, higher level protocols come into effect.

The *address* is a unique number that identifies each computer. IPv4 addresses consist of 4 8-bit values (for 32-bit in total); IPv6 addresses consist of 8 16-bit chunks (for 128-bits in total). The domain name is mapped to an IP address by the *Domain Naming Service (DNS)*.

*java.net* Networking Classes and Interfaces

*InetAddress* Class

*InetAddress* encapsulates the IP address and domain name. It has no constructors; to create an object, use one of the factory methods (static methods that return an instance of the class they are in). 3 commonly used ones are *getLocalHost()*, *getByName(String hostName)* and *getAllByName(String hostName)*.

Example:

**import** java.net.\*;

**public** **class** InetFactory {

**public** **static** **void** main(String[] args) **throws** UnknownHostException {

InetAddress address = InetAddress.*getLocalHost*();

System.***out***.println(address);

address = InetAddress.*getByName*("www.google.com");

System.***out***.println(address);

InetAddress[] addresses = InetAddress.*getAllByName*("www.reddit.com");

**for**(**int** i=0;i<addresses.length;i++) {

System.***out***.println(addresses[i]);

}

}

}

returns

LAPTOP-JTMCRL2L/192.168.1.4

WWW.google.com/172.217.0.228

[www.reddit.com/151.101.125.140](http://www.reddit.com/151.101.125.140)

*InetAddress* has two subclasses for IPv4 and IPv6 addresses: *Inet4Address* and *Inet6Address*. Since they are subclasses, instances of *InetAddress* can refer to either.

TCP/IP Sockets

A socket can connect java’s I/O system to other programs either locally or on other machines on the Internet.

There are two kinds of TCP sockets in java: the *ServerSocket* class for servers is designed for listening; the *Socket* class for clients is designed to connect to server sockets and initiate protocol exchanges.

Creating a *Socket* object implicitly establishes a connection. These are the constructors

Socket(String hostName, int port) throws UnknownHostException, IOException

Socket(InetAddress ipAddress, int port) throws IOException

*Socket* methods return the port, *InetAddress*, input and output streams of the calling *Socket*.

*Sockets* need to be closed with *close()* or try-with-resources.

Example:

**package** networking;

**import** java.net.\*;

**import** java.io.\*;

**public** **class** SocketEx {

**public** **static** **void** main(String[] args) **throws** Exception {

**int** c;

Socket s = **new** Socket("whois.internic.net", 43);

//creates a socket connected to port 43: internic.net

InputStream in = s.getInputStream();

OutputStream out = s.getOutputStream();

String str = (args.length == 0 ? "OraclePressBooks.com" : args[0]) + "\n";

**byte**[] buf = str.getBytes();

out.write(buf);

**while**((c=in.read()) != -1)

System.***out***.print((**char**) c);

s.close();

}

}

the Socket connects to the host “whois.internic.net”, the InterNIC website that handles whois requests. Port 43 is the whois port. Then, the string containing the name of a website is converted to byte array and sent out of the socket with *write()*. Then, the response is read by inputting from the socket.

URL

The uniform resource locator (URL) provides a reliable and reasonably intelligible way to uniquely identify or address information on the internet.

The URL is based on four components:

1. the protocol to use (almost always HTTP)
2. the host name or IP address of the host
3. the optional port number (delimited on the left by a colon and on the right by a slash) (defaults to 80)
4. the file path

Java’s URL class has several constructors:

* URL(String urlSpecifier)
* URL(String protocolName, String hostName, int port, String path)
* URL(String protocolName, String hostName, string path)
* URL(URL urlObj, String urlSpecifier)

each can throw a *MalformedURLException*. The second and third forms allows the URL to be broken up into its components.

Example: retrieving information from a *URL* object.

**package** networking;

**import** java.net.\*;

**public** **class** URLdemo {

**public** **static** **void** main(String[] args) **throws** MalformedURLException {

URL google = **new** URL("http://www.google.com/");

System.***out***.println("Protocol: " + google.getProtocol());

System.***out***.println("Port: " + google.getPort());

System.***out***.println("Host: " + google.getHost());

System.***out***.println("File: " + google.getFile());

System.***out***.println("Ext:" + google.toExternalForm());

}

}

The port returned is -1, meaning a port was not explicitly set.

*URLConnection*

The *URLConnection* class is for accessing the attributes, properties or content information of a *URL*. Obtain a *URLConnection* instance by calling *url.openConnection()*.

Example:

**package** networking;

**import** java.net.\*;

**import** java.util.\*;

**import** java.io.\*;

**public** **class** URLdemo {

**public** **static** **void** main(String[] args) **throws** Exception {

URL google = **new** URL("http://www.google.com/");

URLConnection gc = google.openConnection();

**long** date = gc.getDate();

**if**(date==0)

System.***out***.println("No data information");

**else**

System.***out***.println("Date: " + **new** Date(date));

**long** exp = gc.getExpiration();

**if**(exp==0)

System.***out***.println("No expiration information");

**else**

System.***out***.println("Expiration date: " + **new** Date(exp));

**long** lm = gc.getLastModified();

**if**(lm==0)

System.***out***.println("No information regarding last modification.");

**else**

System.***out***.println("Last modified: " + **new** Date(date));

**long** len = gc.getContentLengthLong();

**if**(len==-1)

System.***out***.println("No content length information");

**else**

System.***out***.println("Content length is: " + len);

**int** c;

**if**(len!=0) {

System.***out***.println("Content: ");

InputStream is = gc.getInputStream();

**while**((c=is.read()) != -1)

System.***out***.print((**char**) c);

is.close();

}

}

}

*HttpURLConnection*

*HttpURLConnection* is a subclass of *URLConnection* that provides support for HTTP connections. Obtain an instance of it by calling *url.openConnection()* and then casting the result to an *HttpURLConnection*.

Example:

**package** networking;

**import** java.net.\*;

**import** java.util.\*;

**public** **class** HttpConnect {

**public** **static** **void** main(String[] args) **throws** Exception {

URL google = **new** URL("http://www.google.com");

HttpURLConnection gCon = (HttpURLConnection) google.openConnection();

System.***out***.println("Request method: " + gCon.getRequestMethod());

System.***out***.println("Response code is " + gCon.getResponseCode());

System.***out***.println("Response message is " + gCon.getResponseMessage());

Map<String, List<String>> hdrMap = gCon.getHeaderFields();

Set<String> hdrField = hdrMap.keySet();

System.***out***.println("\nHere is the header: ");

**for**(String k:hdrField) {

System.***out***.println("Key: " + k + " Value: " + hdrMap.get(k));

}

}

}

*URI*

A Uniform Resource Identifier (URI) is a standard way to identify a resource. URLs are a subset of URIs.

TCP/IP Server Sockets

The *ServerSocket* class is used to create servers that listen for client programs and connect to them on published ports. *ServerSocket* has the constructors

ServerSocket(int port)

ServerSocket(int port, int maxQueue)

ServerSocket(int port, int maxQueue, InetAddress localAddress)

*maxQueue* specifies the maximum amount of connections can be pending before the socket refuses further connection requests (default is 50). *localAddress* specifies the IP address on a multihomed host. These constructors throw an *IOException*.

Datagrams

Datagrams are an alternative to TCP transmission, which sacrifices reliability for speed. Java implements datagrams with two classes: *DatagramPacket* contains the data; *DatagramSocket* sends or receives *DatagramPackets*.

*DatagramSocket*

*DatagramSocket* has the constructors

DatagramSocket()

DatagramSocket(int port)

DatagramSocket(int port, InetAddress ipAddress)

DatagramSocket(SocketAddress address)

the empty constructor binds to any unused port on the local computer; *port* specifies the port to bind to. All can throw a *SocketException*.

The two key methods of *DatagramSocket* are *send()* and *receive()*:

void send(DatagramPacket packet) throws IOException

void receive(DatagramPacket packet) throws IOException

*DatagramSocket* can be closed with *close()* or try-with-resources.

*DatagramPacket*

*DatagramPacket* has the constructors

DatagramPacket(byte[] data, int size)

DatagramPacket(byte[] data, int offset, int size)

DatagramPacket(byte[] data, int size, InetAddress ipAddress, int port)

DatagramPacket(byte[] data, int offset, int size, InetAddress ipaddress, int port)

*data* specifies a buffer that will receive data.

Example: sending a message from the server to the client and displaying it.

**package** networking;

**import** java.net.\*;

**public** **class** Datagrams {

**public** **static** **int** *serverPort* = 998;

**public** **static** **int** *clientPort* = 999;

**public** **static** **int** *buffer\_size* = 1024;

**public** **static** DatagramSocket *ds*;

**public** **static** **byte**[] *buffer* = **new** **byte**[*buffer\_size*];

**public** **static** **void** Server() **throws** Exception {

**int** pos=0;

**while**(**true**) {

**int** c = System.***in***.read();

**switch**(c) {

**case** -1:

*ds*.close();

**return**;

**case** '\r':

**break**;

**case** '\n':

*ds*.send(**new** DatagramPacket(*buffer*,pos,InetAddress.*getLocalHost*(),*clientPort*));

pos=0;

**break**;

**default**:

*buffer*[pos++] = (**byte**) c;

}

}

}

**public** **static** **void** Client() **throws** Exception {

**while**(**true**) {

DatagramPacket p = **new** DatagramPacket(*buffer*,*buffer*.length);

*ds*.receive(p);

System.***out***.println(**new** String(p.getData(),0,p.getLength()));

}

}

**public** **static** **void** main(String[] args) **throws** Exception {

**if**(args.length==1) {

*ds* = **new** DatagramSocket(*serverPort*);

*Server*();

} **else** {

*ds* = **new** DatagramSocket(*clientPort*);

*Client*();

}

}

}

on the server port, the buffer records the command line argument, sending when encountering a new line character. The client side then receives this message.

*java.net.http* Package

*java.net.http*, also referred to as the HTTP client API, is added in JDK 11 and provides support for HTTP clients. It is designed as a superior alternative to *HttpURLConnection*.

The HTTP Client API has three core elements:

* *HttpClient*
* *HttpRequest*
* *HttpResponse*

which work together to support the request/response features of HTTP. The general procedure is: first create an instance of *HttpClient*. Then construct an *HttpRequest* and call *send()* on the *HttpClient*. *send()* returns the response.

*HttpClient*

*HttpClient* encapsulates the HTTP request/response mechanism. It is the foundation of the HTTP client API. It supports synchronous and asynchronous communications.

*HttpClient* is an abstract class, and instances are built with factory methods. The *HttpClient.Builder* interface provides several methods to configure the *HttpClient*. *newBuilder()* obtains an *HttpClient* builder. Next, call *build()* on the builder.

Example: creating a default *HttpClient*:

HttpClient hc = HttpClient.newBuilder().build();

A default *HttpClient* can also be obtained by calling *newHttpClient()*.

*HttpRequest*

*HttpRequest* is an abstract class, and instances are built with builders. Create an *HttpRequest.Builder* instance with *newBuilder()*. *HttpRequest.Builder* provides several methods to configure the *HttpRequest*. Next, call *build()* on the builder.

*HttpResponse*

*HttpResponse<T>* is is a generic interface where *T* specifies the type of body.

When a request is sent, an *HttpResponse* instance is returned containing the response.

T body()

returns a reference to the body.

Example:

**package** networking;

**import** java.net.\*;

**import** java.net.http.\*;

**import** java.util.\*;

**import** java.io.\*;

**public** **class** NetHttp {

**public** **static** **void** main(String[] args) **throws** Exception {

HttpClient hc = HttpClient.*newHttpClient*(); //creates an HttpClient with default settings

HttpRequest hr = HttpRequest.*newBuilder*(**new** URI("http://www.google.com/")).build(); //create a request

HttpResponse<InputStream> resp = hc.send(hr, HttpResponse.BodyHandlers.*ofInputStream*()); //send the response and get the response

//body type is InputStream

System.***out***.println("Response code is " + resp.statusCode());

System.***out***.println("Request method is " + hr.method());

HttpHeaders hdrs = resp.headers();

Map<String, List<String>> hdMap = hdrs.map();

Set<String> hdField = hdMap.keySet();

System.***out***.println("\nDisplaying the headers.");

**for**(String s: hdField) {

System.***out***.println("[KEY]: " + s + " [VALUE]: " + hdMap.get(s));

}

System.***out***.println("\nDisplaying the body.");

InputStream in = resp.body();

**int** c;

**while**((c = in.read()) != -1) {

System.***out***.print((**char**) c);

}

}

}

*BodyHandlers* can handle the body as many types: for instance, *ofString()* stores it in a String, in which case it can be displayed with

System.out.println(resp.body);

or *ofLines()* which stores the body as a stream of lines.

**Skipped Ch24-30**

CH25

Introducing AWT: working with windows, graphics and text

The Abstract Window Toolkit (AWT) is Java’s first GUI framework

CH31

Introducing Swing

*Swing* is used for GUIs more often than the *AWT*. *Swing* provides more powerful and flexible components.