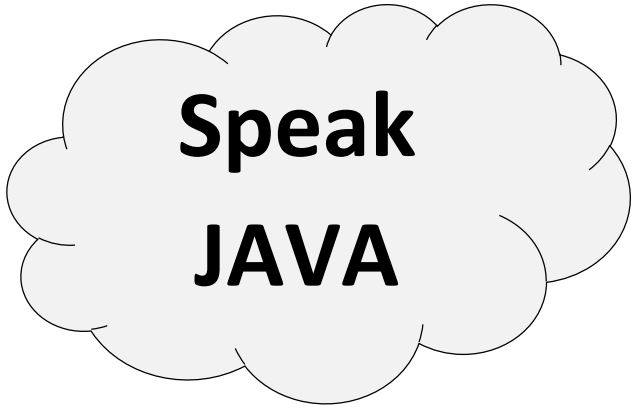
Do You



?

Java Language Fundamentals

First Edition

February 2016

Aleks Rudenko

Trademarks

The following are trademarks of the Oracle Corporation and its affiliates in the United States, other countries, or both:

- Oracle
- Java
- Java Beans
- JDK
- JRE
- JVM
- JavaFX

Other trademarks and registered trademarks are the properties of their respective owners.

Table of Contents

Introduction

There is a broad range of books on Java – from beginner's guides to complete references. The first category tends to concentrate on basic concepts, which does not take you very far from the "Hello world!" program. The books of the second category are simply too big – usually over 1,000 pages. They are very good as reference sources, but a bit overloaded with details and lengthy examples. They are also trying to cover as many topics as possible, which makes the learning curve steeper.

The size of this book is less than two hundred pages so it is definitely not a reference guide. Nor is it a "Java basics" book.

Not concentrating too much on details, and illustrating everything with small comprehensive examples, this book focuses on Java's fundamental features, essential for building real-life applications. The goal is to create an overall understanding of how Java works. The details can be picked up later either from Java original specifications and tutorials, or from those comprehensive guides and complete references.

Starting from the ground up, the book builds a solid foundation of your Java knowledge.

If you want to become a Java expert – this book is a good start. Good luck!

Installing the Software

If you want to experiment with the sample programs provided in this book, you need to have the JDK 8 software installed on your machine. You can download it for free from the Oracle website:

http://www.oracle.com/technetwork/java/javase/downloads/

There are different versions of the software available, suitable for different types of operating systems – Linux, Mac OSX, Solaris, or Windows.

The instructions of how to install and configure the Java 8 platform for different operating systems can be found on this page:

https://docs.oracle.com/javase/8/docs/technotes/guides/install/

Note: all URLs referenced in this book might change in the future.

Working with Examples

Throughout this book numerous examples are used to illustrate the usage of particular features of Java language. The basic structure of an examples is this:

```
// Comments . . .
class className {
   statements . . .
   public static void main (String args[]) {
      statements . . .
      System.out.println (results);
   }
}
```

To fully understand the above syntax, several topics of this book must be read first. At this point, the following explanations would be sufficient for working with examples.

Java is case-sensitive:

thisVariable and **ThisVariable** are two different variables.

Comments:

```
// This is a non-executable comment line

/*
   This is a block of commented lines
   . . .
*/
```

A single statement ends with semicolon:

```
thisVariable = 5;
```

A block of statements is included within curly brackets:

```
{ a=5; b="text"; }
```

The DOT (.) notation:

```
objectA.variable1;  // returns the value of variable1 of objectA
objectA.method1();  // executes the method1 of objectA
```

The code to the left of the dot (.) must be an object, and the code to the right of it must be the object's property – a variable or a method. An expression with multiple dot operators is executed from left to right.

The print statement:

```
System.out.println ("Total = " + 5); // output: Total = 5

System.out.print   (" x=" + 1);      // output: x=1
System.out.print   (" y=" + 2);      // output: x=1 y=2
```

These are commonly used methods of printing the results. Both statements use the "**+**" operator to concatenate all items (arguments) listed within the brackets into a single text line and output it to the console. The **print** method outputs the arguments onto the current line, appending them to the existing contents. The **println** method outputs the results and switches to a new line.

The main method:

Every Java application starts execution with the **main** method (program):

```
public static void main (String args[]) { code }
```

The class:

The **class *className* {..}** is a definition of a class. Each class definition is stored in a file with corresponding name - ***className*.java**.

Object-Oriented Programming

The purpose of any computer program is to perform some manipulations on data. This is also true for the object-oriented programs. The difference, however, between a "traditional", data-oriented program, and an object-oriented program is in the way they view the data.

Traditional programs "see" and can perform some operations on individual or grouped together elementary data elements of primitive data types – numbers, characters, text strings, etc. Sometimes more complex data structures like tables, stacks, or lists, along with specialized manipulation commands, could be built into the language, but that's about it. The highest level of abstraction stops at the above mentioned data structures. If, for example, we want to create a representation of some logical entity, like "Customer", we would need to construct a new group of data elements and build the code to access it.

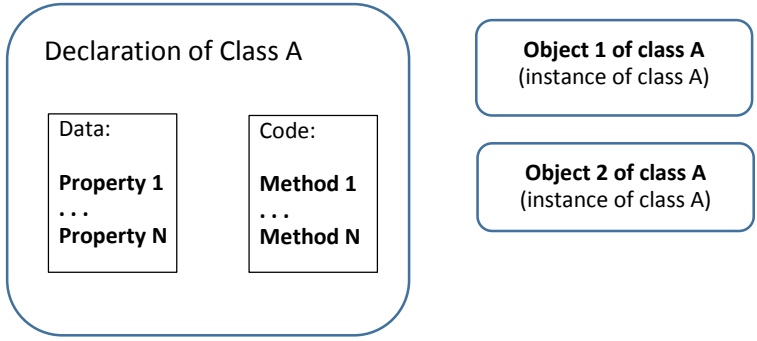
Object-oriented programs bring data abstraction to another level. They can still manipulate the primitive data types – numbers, characters, etc., but the primary focus is on so-called objects. An object is an entity that contains data along with code to access that data. Object-oriented languages provide additional tool sets (commands, operators, other language constructs) allowing for creation and manipulation of objects. In the Java language, for example, we have a construct to define ***classes***, which are templates for building objects, and a special operator ***new*** for creating instances of objects:

```
// Define the class "Apple"
class Apple {
… data and code …
}

// Create a new object of class Apple
obj = new Apple();
```

Object-Oriented Design

As we already mentioned, the main difference between traditional languages and object-oriented programming languages is the level of abstraction in viewing the data. Traditional languages provide means of manipulating primarily primitive data elements, while object-oriented languages are focused on manipulating logical objects. The way manipulation of objects is implemented in the language is based on Object-Oriented Design (OOD) principles. The **OOD** consists of three main concepts – ***Encapsulation***, ***Inheritance***, and ***Polymorphism***. In general, they set up rules on how objects and their contents should be built, and the valid relationships between objects. In order to understand these concepts, we have to get familiar with terminology around objects.

Let's look at the picture below:



Class A is a template (a definition) for building objects. It consists of data definitions called ***properties*** and programs (executable code) called ***methods***.

Object 1 and **Object 2** are the ***instances*** of class A. They hold concrete values of all properties declared in Class A.

There can be multiple classes (templates) and multiple objects (instances) of those classes in a program. Now, let's describe the three OOD concepts.

Encapsulation

Encapsulation principle requires that the internals (data and code) of an object should be protected from being arbitrarily accessed by code outside of the object. The only way to "communicate" with an object and access its properties is through the public methods and public properties. Below is an example of how the principle of encapsulation is implemented in Java.

```
// Declare the class "Apple"
class Apple {
   private String color;
   public void setColor (String c) {
      color = c;
   }
}

// Create the "apple1" object of class Apple
Apple apple1 = new Apple();

// Invoke the public method "setColor" of object "apple1"
// and set the color property to "red"

apple1.setColor("red");

apple1.color = "red";
```

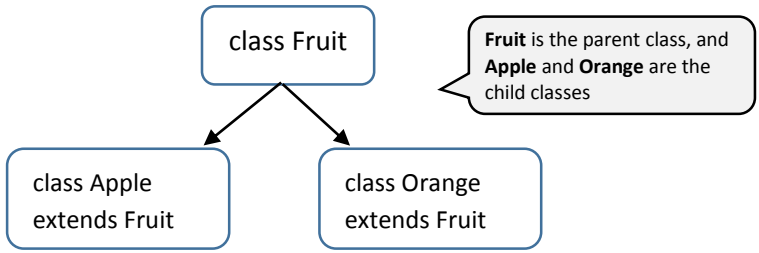
The property **color** is not accessible from the outside of the object

The method **setColor()** is accessible from the outside of the object

Doesn't work because the property color cannot be accessed from outside the class

Inheritance

Inheritance principle applies to the relationship between classes. It states that two classes can have a parent-child connection, in which the child class (a subclass) inherits all the features of the parent class (a superclass).



Below is an example of the inheritance mechanism.

```
// Declare the class "Fruit"
class Fruit {
   private String color;
   public void setColor (String c) {
      color = c;
   }
}

// Declare the class "Apple" as a child of class Fruit
class Apple extends Fruit {
}

// Create the "apple1" object of class Apple
Apple apple1 = new Apple();

// Execute the method "setColor" of object "apple1"
// and set the value of color property to "red"

apple1.setColor("red");
```

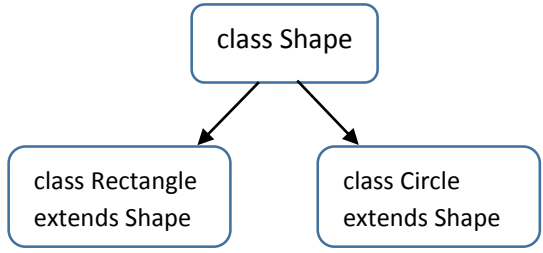
Class **Apple** inherits all properties and methods of the **Fruit** class

Even though the **color** property and the **setColor** methods are not defined in the Apple class, they are accessible in the Apple objects due to inheritance.

Polymorphism

Polymorphism principle dictates that it is allowed to have multiple methods with same the name but different implementations within one or more classes, and the language must be able to determine which variation of the method to execute at run time.

Below are examples of how the Java language implements polymorphism. Let's say, we need to calculate the square footage of different shapes – rectangle, triangle, circle, etc. Each shape is represented by its own class, and each class is a sub-class of the superclass Shape:




```
// Declare the class Shape
class Shape {
   public double footage;
   public void showSquareFootage() {
      System.out.println(footage);
   }
}

// Declare the class Rectangle
class Rectangle extends Shape {
   double width;
   double height;
   Rectangle (double w, double h) {
      width = w;
      height = h;
   }

   public void showSquareFootage() {
      footage = width * height;
      System.out.println(footage);
   }
}
```

This method is executed when an object of class **Rectangle** is created.

This method calculates and displays the square footage of the **Rectangle** object.

```
// Declare the class Circle
class Circle extends Shape {
   double radius;
   Circle (double r) {
      radius = r;
   }

   public void showSquareFootage() {
    footage = radius * radius * 3.1415;
      System.out.println(footage);
   }
}

// Main program

// Create an object of class Shape
Shape shape1 = new Shape();
shape1.showSquareFootage();        // prints "0"
. . .
// Create a rectangle object with width=2 and height=3
shape1 = new Rectangle(2.0, 3.0);  // shape1 is now a rectangle
shape1.showSquareFootage();        // prints "6"
. . .
// Create a circle object with radius=2
shape1 = new Circle(2.0);          // shape1 is now a circle
shape1.showSquareFootage();        // prints "6.2830"
```

This method is executed when an object of class **Circle** is created.

This method calculates and displays the square footage of the **Circle** object.

In the above example the **shape1** variable, depending on the program flow, could represent objects of different subclasses. During execution the language will validate the type of object referenced by **shape1** and select the appropriate **showSquareFootage** method.

Consider now that we want to change the dimensions of a rectangle and at the same time calculate its new square footage. We could add another variation of the **showSquareFootage** method to the declaration of the **Rectangle** class:

```
class Shape {
   public double footage;
   public void showSquareFootage() {
      System.out.println(footage);
   }
}

class Rectangle extends Shape {
   double width;
   double height;
   Rectangle (double w, double h) {
      width = w;
      height = h;
   }
```

```
   public void showSquareFootage() {
      footage = width * height;
      System.out.println(footage);
   }

   public void showSquareFootage(double w, double h) {
      width = w;
      height = h;
      footage = width * height;
      System.out.println(footage);
   }
}
```

This method calculates the square footage of rectangle.

This method changes the dimensions of rectangle and then calculates its square footage.

Now, in order to select and execute the proper method, the language will need to analyze not only the object type but the **signature** of the method as well. The **signature** of a method is the combination of its return type and the types of input parameters. Signature of the first method of this example is "**void+void**" (no output and no input parameters), and signature of the second method is "**void+double+double**" (no output parameters and two input parameters of type **double**).

Thise is how the two variations of the **showSquareFootage** method can be used:

```
// Main program logic

// Create a rectangle object with width=2 and height=3
Rectangle shape1 = new Rectangle(2.0, 3.0);

shape1.showSquareFootage();                  // prints "6"

// Change the rectangle dimensions to 3x4
shape1.showSquareFootage(3.0, 4.0);          // prints "12"
```

Data Types, Variables, and Literals

The Object-Oriented Design discussion gave us an idea of how the Java language is different from other, "traditional" languages. Now let's look at the features making Java similar to non-object-oriented languages.

Most languages, including Java, have the following in common:

- ability to manipulate primitive data types
- variables and literals
- a fixed set of operators
- language constructs and statements for controlling the program flow

In this chapter we will review the primitive data types, variables, and literals. The rest will be covered in later chapters.

Primitive Data Types

A **primitive** data type (could be also referred to as **simple** or **atomic** type) is a data type that is not formed by combining other data types. For example, a *string* is not a primitive data type because it is defined as a set of elements of the *character* data type.

Java defines eight primitive data types:

Date Type	Description	Length, in bits	Range (approximate)
byte	Signed Integer Number	8	-128 to +127
short	Signed Integer Number	16	-32,768 to +32,767
int	Signed Integer Number	32	$-2.1*10^9$ to $+2.1*10^9$
long	Signed Integer Number	64	$-9.2*10^{18}$ to $+ 9.2*10^{18}$
float	Floating Point Number	32	± (1.4e-45 to 3.4e+38)
double	Floating Point Number	64	± (4.9e-324 to 1.8e+308)
char	Unicode Character	16	0 to 65,536
boolean		N/A	true or false

Note: Java does not reveal the length of boolean data type because its implementation is platform-specific.

Variables

In Java, data elements of primitive types can be declared either as **literals** or variables.

A **variable** is a named reference to a location in memory where the data element of specified type resides. The format for declaring a variable is this:

```
type identifier;          // declares the variable identifier
                          // of type type

type identifier = value; // declares and assigns value
                          // to variable identifier of type type
```

Here are some examples of declaring data elements of primitive data types.

```
int k;         // "k" is a reference to a 32-bit area in memory
               // that will hold an integer value

k = 100;       // The above memory is populated with a 32-bit
               // signed binary number (100)

char c = 'A'; // "c" is a reference to the two-byte Unicode
               // character 'A'
```

Note, that variables can reference not only the primitive data types, but objects as well.

Accessing the Variables

At high level, you can view every Java program as blocks of code. Each block is surrounded by curly brackets: **{** statements **}**. Blocks can be separate or included in each other:

```
{ block 1 }

{ block 2 { block 3 } }
```

This structure determines the accessibility, also referred to as **scope,** of variables. A variable declared in a block is accessible within that block and within all included blocks. Once program flow reaches the end of a block all variables declared within that block are destroyed.

Literals

A **literal** specifies an actual value of a data element. For example, 123 will be treated by Java as a 32-bit signed integer number (i.e. type *int*), and 123.4 – as 64-bit floating point number (type *double*). In fact, any whole number value (i.e. without the decimal point) is treated as an integer, and any fractional value (i.e. with the decimal point) is treated as a double precision floating point number.

There are four types of literals in Java: numeric, character, boolean, and string. Each of them is discussed below.

Numeric Literals

Numeric literals can be specified in **decimal**, **hexadecimal**, **octal**, or **binary** notations. Here is the number 10 in different notations:

10 - decimal notation (decimal digits 0-9)

0x0A - hexadecimal notation (**0x** is in front of hexadecimal digits 0-F)

012 - octal notation (**0** is in front of octal digits 0-7)

0b1010 - binary notation (**0b** is in front of binary digits 0-1)

Numeric literals can contain embedded underscores, which are ignored by the compiler but make the values easier to read. Here are examples of valid numeric literals:

1234567890

1_234_567_890

123__456__7890

123_456_789.0

The data type of a literal is defined either implicitly (default data type assignment) or explicitly. Any whole number literal is defaulted to type **int**, and any fractional number literal is defaulted to type **double**. There can be no literals of type **byte** or **short**. The **long** and **float** literals can be defined explicitly by appending the number with '**L**', '**l**', '**F**', or '**f**'. The type **double** can

also be specified explicitly by appending the number with '**D**' or '**d**' but this would be redundant. The table below shows some examples of how the data type of literals is determined.

Literal	**Data Type**
Not available	byte
Not available	short
123	int
123L or 123l	long
123.0	double
123.0D or 123.0d	double
1.23E+2	double
123.0F or 123.0f	float

Character Literals

A character literal is represented with a value within a pair of single quotes. The digits, letters, and special characters that are present on the keyboard, can be specified directly, as follows: 'a', '1', '%', etc. Other symbols should be entered using **escape sequences**. The **escape sequence** is the backslash '\' followed by one or several symbols. There are several predefined escape sequences that represent new line, tab, backspace, etc. Besides that, any Unicode character can be entered directly using escape sequences in octal or hexadecimal notations. Octal character notation has format '**\nnn**' – backslash followed by three octal digits. Hexadecimal character notation has format '\unnnn' – backslash, followed by 'u', followed by four hexadecimal digits. Here is the lower-case letter 'b' literal in different formats:

'b', '\142', '\u0062'

The table below shows available escape sequences.

Escape Sequence	Description
\nnn	Octal character (nnn – octal digits 0-7)
\unnnn	Unicode character (nnnn – hexadecimal digits 0-F)
\b	Backspace
\n	New line
\r	Carriage return
\t	Tab
\f	Form feed

\’	Single quote
\”	Double quote
\\	Backslash

Boolean Literals

Boolean literals can have only two values – **true** and **false**. These values are not converted to any numeric values.

```
boolean b = true;  // "b" is declared and set to true
b = false;         // "b" is set to false
```

String Literals

String literals are formed by enclosing a sequence of characters by double quotes. Here are few examples of valid string literals.

```
"String Literal"
"\"String in double quotes\""
"String on \n two lines"
```

Casting in arithmetic expressions

In general, the operands (variables and literals) of an arithmetic expression could be of different data types. The purpose of the ***casting*** mechanism is to deal with such situations by *promoting* the precision of operands from lower to higher level. Precision is increased from **byte** to **double** as follows:

byte →short → int → long → float → double

There are two types of casting: implicit and explicit. **Implicit casting** is done by the language automatically, based on the actual data types of the operands. **Explicit casting** is requested by the programmer by placing the desired data type in front of a variable or literal.

Below are the rules of how the type of operands is determined or set in arithmetic expressions:

- all integer literals by default are of type **int**
- all non-integer literals by default are of type **double**
- before calculating any arithmetic expression, all **byte** and **short** variables are promoted to type **int**

- for each arithmetic operation (+, -, /, *, etc.), with mixed operand data types, a lower precision operand is promoted to the type of the higher precision operand
- a literal or variable is promoted or demoted to the date type explicitly specified by the type modifier placed in front of it in parenthesis

Example 1

```
double result = 1.5 + 3 / 2; // result = 1.5 + 1 = 2.5

The data type determination and calculation are performed as follows:
(double) result =
(double) 1.5 + (int) 3 / (int) 2 =
(double) 1.5 + (int) 1 =
(double) 1.5 + (double) 1.0 = (double) 2.5
```

Example 2

```
double result = 1.5 + (double) 3 / 2; // result = 1.5 + 1.5 = 3.0

The data type determination and calculation are performed as follows:
(double) result =
(double) 1.5 + (double) 3 / (int) 2 =
(double) 1.5 + (double) 3 / (double) 2.0 =
(double) 1.5 + (double) 1.5 = (double) 3.0
```

Example 3

```
int result = (int) (3.0 / 2.0); // result = 1

The data type determination and calculation are performed as follows:
(int) result =
(int) ( (double) 3.0 / (double) 2.0 ) =
(int) ( (double) 1.5 ) =
(int) 1
```

Operators

In this chapter we will review operators that can be applied to primitive data types. These operators can be divided into the four main categories: **arithmetic**, **bitwise**, **relational**, and **boolean**. In addition, the **assignment** (**=**) and the question mark (**?**) operators will also be discussed here.

The Assignment Operator

The assignment operator "**=**" assigns value to a variable. We've already used it many times in previous examples. The format of the assignment operator is this:

variable = *expression*;

It also can be used to assign a value to several variables in a single statement, like this:

var1 = *var1* = … = *varN* = *expression*;

Examples:

```
int x, y;     // Declare the variables x and y of type integer
x = 1;        // Assign the value 1 to variable x
x = y = 2;    // Assign the value 2 to variables x and y
```

Arithmetic Operators

Arithmetic operators are used in calculations. Java supports the following five basic algebraic operators: **addition** (+), **subtraction** (-), **multiplication** (*), **division** (/), and **modulus** (%). Each of those can be combined with the assignment operator (**=**) to form a so-called **shorthand operator** in this form:

*arithmetic_operator***=**

Examples:

```
double x, y;
x = 2.5;
y = 10;
x += y;      // x = 12.5 The result is same as in x = x + y;
x %= y;      // x = 2.5 The result is same as in x = x % y;
             // note: the modulus operation (%) returns the
             // remainder of the division 12.5 / 10
```

Java also provides two additional arithmetic operators: **increment** (**++**) and **decrement** (**--**) that respectively add 1 to a value or subtract 1 from a value. These operators can be placed either before or after a variable:

```
int x = 1;    // Declare x and set its value to 1
x++;          // Add 1 to x (x=2)
++x;          // Add 1 to x (x=3)
x--;          // Subtract 1 from x (x=2)
--x;          // Subtract 1 from x (x=1)
```

In the above examples the placement of the increment and decrement operators is irrelevant. However, in assignment expressions, their location is important. When the operator is placed **after the variable**, the original value of the variable is used in the calculation of the result; after that the increment/decrement operator is applied to the variable. When placed **in front of a variable**, the increment/decrement operator is first applied to that variable and the new value is used in the calculation.

Examples:

```
int x, y;
x = 1;
y = x++;        // x=2, y=1 (original value of x before the increment)
y = ++x;        // x=3, y=3 (new value of x after the increment)
```

Bitwise Operators

The Java language supports a set of so-called **bitwise** operators that can manipulate integer types (**char**, **byte**, **short**, **int**, **long**) on the bit level. They can be divided into two groups: **bitwise logical** operators and bitwise **shift operators**. The following table lists all the bitwise operators:

Operator	Performed Function	Type of operator
~	Bitwise unary NOT (complement)	Bitwise Logical
&	Bitwise AND	
|	Bitwise OR	
^	Bitwise exclusive OR (XOR)	
<<	Shift left, filling with zeroes	Bitwise Shift
>>	Shift right, propagating the sign bit	
>>>	Shift right, filling with zeroes	

As with the arithmetic operators, any bitwise operator except the NOT (~) can be appended with the assignment operator "=" to form a shorthand:

```
x <<= 1;      // This is equivalent to x = x << 1;
x &= y;       // This is equivalent to x = x & y;
```

The bitwise operators &, |, ^, and ~ apply the logical operators AND, OR, XOR, and NOT to corresponding bits of each operand. The table below shows the results of logical operators for all possible two bit combinations:

Logical operators:		**AND**	**OR**	**XOR**	**NOT**
Operand A	**Operand B**	**A & B**	**A | B**	**A ^ B**	**~A**
0	**0**	0	0	0	1
0	**1**	0	1	1	1
1	**0**	0	1	1	0
1	**1**	1	1	0	0

The bitwise shift operators shift the contents of a signed numeric value to the left or to the right by number of bits specified in the expression. The ">>" and "<<" operators preserve the sign (leftmost bit) of the number, and the ">>>" operator fills the leftmost positions with zeroes.

Examples:

```
byte b;
```

```
b = (byte) 0b11111110;       // -2 in decimal
b = (byte) ~b;               // b = 00000001 (+1 in decimal)

b = (byte) (b << 1);         // b = 00000010 (+2 in decimal)
b <<= 1;                     // b = 00000100 (+4 in decimal)

b = (byte) (b | 0b00000001); // b =  00000101 (+5 in decimal)
b |= (byte) 0b00000010;      // b =  00000111 (+7 in decimal)
```

Relational Operators

The relational operators are used for comparing values of two operands and produce a **Boolean** result of either true or false.

The following table lists all the available relational operators:

Operator	Performed Comparison
==	Operands are equal
!=	Operands are not equal
>	First operand is greater than the second operand
>=	First operand is greater than or equal to the second operand
<	First operand is less than the second operand
<=	First operand is less than or equal to the second operand

Examples:

```
int x = 1;
int y = 2;

if ( x == y) { System.out.println("x is equal to y"); }
else         { System.out.println("x is not equal to y"); }

if ( x != y) { System.out.println("x is not equal to y"); }
else         { System.out.println("x is equal to y"); }

if ( x > y)  { System.out.println("x is greater than to y"); }
else         { System.out.println("x is less or equal to y"); }


if ( x >= y) { System.out.println("x is greater or equal to y"); }
else         { System.out.println("x is less than y"); }

if ( x < y)  { System.out.println("x is less than y"); }
else         { System.out.println("x is greater or equal to y"); }


if ( x <= y) { System.out.println("x is less or equal to y"); }
else         { System.out.println("x is greater than y"); }
```

Boolean Logical Operators

The Boolean logical operators are similar to the bitwise logical operators. Both apply the logical AND, OR, XOR, and XOR to two operands and produce Boolean results, but the bitwise logical operators act on bits, while the Boolean logical operators act on Boolean values.

In addition to "regular" four logical operators (AND, OR, XOR, NOT), Java provides two "improved" Boolean operators – **short-circuit OR** and **short-circuit AND**. They will be explained later in this section.

The following table lists the Boolean logical operators:

Operator	Performed Function	Type of operator
!	Logical unary NOT	Regular logical operator
&	Logical AND	
|	Logical OR	
^	Logical exclusive OR (XOR)	
||	Short circuit OR	Short-Circuit logical operator
&&	Short circuit AND	

Same as with the bitwise operators, the Boolean logical operators (with the exception of the unary NOT operator and the short-circuit operators) can be combined with the assignment operator to form the shorthand as follows:

```
a &= b;     // This is equivalent to a = a & b;
a |= b;     // This is equivalent to a = a | b;
a ^= b;     // This is equivalent to a = a ^ b;
```

The Boolean logical operators (**&**), (**|**), (**^**), and (**!**) apply the logical operators AND, OR, XOR, and NOT to two Boolean operands of an expression. The table below shows the results of Boolean logical operators for all possible combinations of two Boolean operands:

Logical operators:		**AND**	**OR**	**XOR**	**NOT**
Operand A	**Operand B**	**A & B**	**A | B**	**A ^ B**	**!A**
True	**True**	True	True	False	False
True	**False**	False	True	True	False
False	**True**	False	True	True	True
False	**False**	False	False	False	True

The short-circuit operators (**&&**) and (**||**) do not evaluate the right-hand operand of a logical expression when the value of the left-hand operand determines the result regardless of the value of another operand. For example, if A=true, the result of A | B is always true, no matter of what B value is.

Examples:

```
boolean a, b, c;
a = true;
b = true;

a = !b;      // a=false
a &= b;      // a=false (same as: a = a & b;)
a |= b;      // a=true  (same as: a = a | b;)
a ^= b;      // a=false (same as: a = a ^ b;)
a = a ^ b;   // a=false
```

```
// short-circuit operators

    a = false;
    b = true;
    c = a || b;    // c=true  (a and b both will be evaluated)
    c = b || a;    // c=true  (a will not be evaluated)
    c = a && b;    // c=false (b will not be evaluated)
```

Conditional Operator

Java provides a special ternary (three-way) conditional operator "**?**...**:**", that allows for compact implementation of some IF-THEN-ELSE statements. The format of the "**?**...**:**" operator is this:

variable = *logical_expression* **?** *expression1* **:** *expression2*

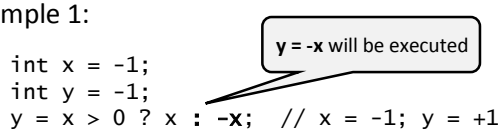
The *logical_expression* is any expression that evaluates to a **Boolean** value. If it evaluates to **true**, then *expression1* is evaluated and assigned to *variable*; otherwise, *expression2* is evaluated and assigned to *variable*. This can be shown in pseudo-code as follows:

IF *logical_expression* = true

THEN *variable* = *value of expression1*

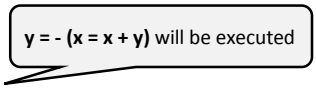
ELSE *variable* = *value of expression2*

Example 1:




```
int x = -1;
int y = -1;
y = x > 0 ? x : -x;  // x = -1; y = +1
```

Example 2:




```
int x = -1;
int y = -1;
y = x > 0 ? x : -(x = x + y);  // x = -2; y = +2
```

In the second example, the **x = x + y** expression is executed first resulting in **x=-2**. Then, expression **-(-2)** evaluates to **+2** and this value is assigned to **y**.

Program Flow Control

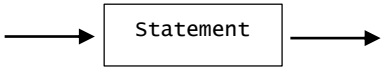
Most computer languages, including Java, support the following fundamental program flow constructs:

- Sequential Statement
- Block of Statements
- Two-Choice (IF structure)
- Multiple-Choice (CASE structure)
- Iteration
- Jump (GOTO statement)

Java's implementation of these constructs is presented in this paragraph.

The Sequential Statement

The **Sequential Statement** does not change the program flow.



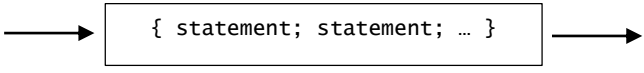
The assignment statement like **x = b;** is an example of a sequential statement. Java also has an **empty statement**, which is just a semicolon (**;**) without any expression in front of it.

The Statement Block

The **Statement Block** is a group of statements and optional declarations enclosed within the curly brackets:

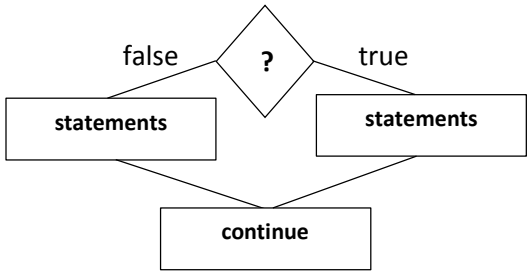
{declarations and statements**}**

The program flow outside of the Statement Block is not affected:



The “if” Statement

The “**if**” statement implements the Two-Choice construct:



The format of the **if** statement is as follows:

```
if ( boolean expression )
   { statements, executed when condition is true }
else
   { statements, executed when condition is false }
```

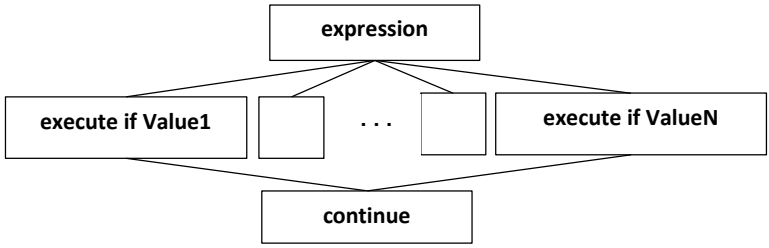
condition

Examples:

```
int x;
x = 0;

if (x < 0)
   x++;
else
    ;

System.out.println("x=" + x);  // prints “x=0”

if (x == 0)
   { x++; }
else
    { x--; }
System.out.println("x=" + x);  // prints “x=1”
```

The “switch” Statement

The **switch** statement implements the Multiple Choice (CASE) construct:



Here is the format of the **switch** statement:

```
switch ( integer or String type expression )
    {
    case value1: statement; . . . statement;
                        break;
    . . .
    case valueN: statement; . . . statement;
                        break;
    default: statement; . . . statement;
    }
```

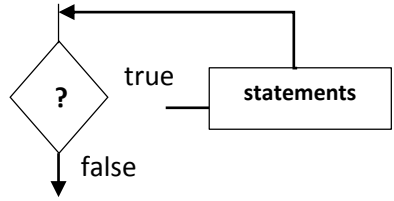
The *expression* must be evaluated to one of the following numeric types – **char**, **byte**, **short**, **int**, or to the type **String**. The result of the *expression* is compared with the values specified in the **case** blocks – *value1* through *valueN*. When a match is found, **all the statements** following that case statement are executed. If we want to execute statements for only one case value, we need to use the **break** statement as shown above. The break statement “jumps” out of the switch construct to the next following statement. The **default** statement group, which is optional, is executed when no value matches found.

Example:

```
int x, y, z;
x = y = z = 1 ;
switch (x += y) {              // evaluates to 1 + 1 = 2
   case 1:   z = 0; break;
   case 2:   z = 1; break;
   default:  z = 999;
}
System.out.println("z=" + z);  // prints: “z=1”
```

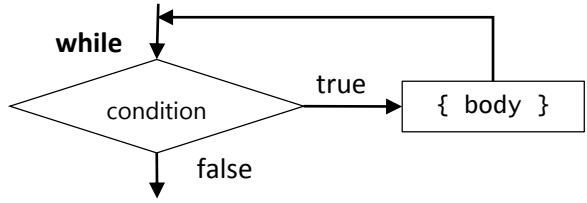
Iteration statements

Java supports three iteration statements – **while**, **do…while**, and **for**. All of them implement the **iteration** construct that executes a block of code until some conditions are met.



The "while" Statement

The **while** statement executes a block of code (body of the loop) while the specified condition is true:



An important point here is that the *condition* is evaluated first, and then the *body* is executed. If the *condition* is false initially, the body of the loop will not be executed at all. The format of the **while** statement is this:

while (*condition*) { *body* }

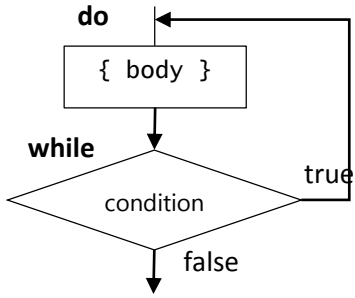
Example:

```
int n;
n = 2;
while (n > 0)
{
   System.out.print(" n=" + n); // prints: "n=2 n=1"
   n--;
}
```

If the initial value of **n** was 0, the above code would not produce any outputs, i.e. the body of the loop would not execute.

The “do…while” Statement

The **do…while** statement is similar to the above **while** statement with one exception – it first executes the body of the loop, then evaluates the specified condition:



The format of the **do…while** statement is this:

do { *body* } while (*condition*) ;

Example:

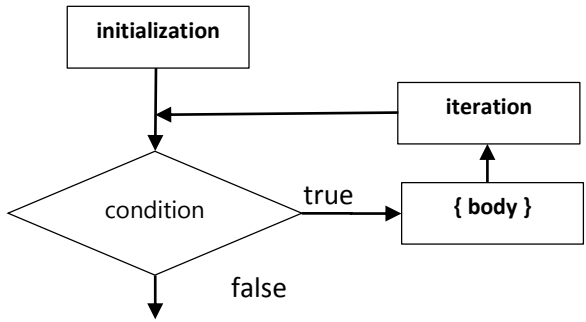
```
int n;
n = 2;
do
{
   System.out.print(" n=" + n); // print: “n=2 n=1”
   n--;
}
while (n > 0);
```

The result, as we see, is exactly as in the previous example with the **while** statement. However, if the initial value of **n** was 0, the body of the loop would execute once and the above code would produce this output:

n=0

The “for” Statement

The **for** statement is used for implementing the so-called “controlled” loops. Controlled loops are basically **while** loops, but with explicitly specified code that initializes and iterates the value of the variable(s) used in the loop controlling condition:



Let’s look at the previous example of the **while** statement:

```
int n;
n = 2;
while ( n > 0 )
{
   System.out.println("n=" + n);
   n--;
}
```

Initialization

Condition

Iteration

Same logic can be coded with the **for** statement as follows:

```
int n;

for ( n = 2;   n > 0;   n-- )
{
   System.out.println("n=" + n);
}
```

Initialization

Condition

Iteration

This is a **“classical”** form of the **for** statement, and its format is:

for (*initialization*; *condition*; *iteration*) { *body* }

Another form of the **for** statement is dealing with collections (arrays, for example) and is called the “**for-each**” loop. We will present it in a few pages.

The **n** in the above example is called a *loop control variable*. There can be more than one loop control variables, as in this example:

```
int x, y, z;
// Find when the sum of x, y, and z drops to zero
for (x=3, y=5, z=7;   x+y+z > 0;   x--, y--, z-- )
{;}  // Empty block; could be specified without curly brackets
System.out.println("x/y/z= " + x + "/" + y + "/" + z);
```

The output of this code will be:

```
x/y/z= -2/0/2
```

Note that the initialization (x=3, y=5, z=7) and the iteration (x--, y--, z--) statements are comma separated.

Internal Loop Control Variables

Another important note about the **for** statement must be made. When the loop control variable is defined within the **for** statement, it is "not visible" outside of the **for** statement:

```
for ( int n = 2;   n > 0;   n-- )
{
   System.out.println("n=" + n);    // the value of n is known
}

// The n variable is not "visible" beyond the for statement
System.out.println("final n=" + n); // Error: the n is unknown
```

The "for" Statement without Loop Control Variables

It's worth mentioning that the **for** statement could be coded without the *initialization* or the *iteration* expressions, or both:

```
int n, max;
n = 0;
max = 100;
boolean reachedMax = true;
for ( ;  !reachedMax;  )   // while NOT reachedMax …
{
   // double the value of n until it exceeds the max
   n += n;
   if (n > max) {reachedMax = true;}
}
System.out.println("n=" + n); // prints: n=128
```

The “for-each” Statement

Consider an example when we need to calculate the average value for a set of numbers, stored in an array. Here is how we can accomplish that with the “classical” **for** statement:

```
int nArray[] = {1, 2, 3, 4, 5};  // array of five numeric values
double avg = 0;

// loop through nArray elements (first element is numbered 0)
// and calculate the sum of all elements

for ( int i = 0;  i < nArray.length; i++ )
{  avg += nArray [i];  }

// divide the sum by the number of elements
avg /= nArray.length;
System.out.println("avg=" + avg); // prints: avg=3.0
```

The same result can be accomplished with the “**for-each**” version of the **for** statement, which has the following format:

for (*type iterativeVariable*: *collection*) { *body* }

The above example would look like this:

```
int nArray[] = {1, 2, 3, 4, 5};  // array of five numeric values
double avg = 0;

// The val will be assigned the value of each array element,
// one by one, in sequential order.

for ( int val: nArray)
{ avg += val; }

avg /= nArray.length;
System.out.println("avg=" + avg); // prints: avg=3.0
```

Please note, that the iteration variable **val** must be of the same or compatible data type as the elements of the collection (array). The compatible data types are those that the original data type can be implicitly promoted to. In this example, the original data type is **int**, which can be promoted to **long**, **float**, or **double**.

The “break” Statement

Java provide two statements – **break** and **continue**, that can break the “natural” execution flow of a loop or any block of statements. The **break** statement causes the termination of a loop or any named block of code before it reaches its end. The **continue** statement immediately starts the next iteration of a loop.

Let’s start with the **break** statement. We have already seen it in the **switch** construct where it was used in the **case** expressions to jump out of the **switch** statement.

Breaking Out from Labeled Blocks

The **break** statement has the following format:

break [*label*];

The ***label*** parameter is optional and is used to jump out of any named (labeled) block of code. That block of code does not have to be a loop or a switch statement. Consider this example:

```
int var;
var = 2;

block1:
   {
      if (var != 1) break block1;
      System.out.println("Block 1 executed");
   }

block2:
   {
      if (var != 2) break block2;
      System.out.println("Block 2 executed”);
   }
```

The output of this code will be “**Block 2 executed**”. Since the variable **var** is not equal to 1, **break block1** will jump out of **block1** to the next statement.

Note that this technique will also work when the named blocks are nested within each other. The break can jump from an inner block out of any outer block:

```
int var = 2;

block1:
{
   block2:
   {
      block3:
      {
            if (var == 1) break block1;
            if (var == 2) break block2;
            if (var == 3) break block3;
            System.out.println("Last statement of block3");
      }
      System.out.println("Last statement of block2");
   }
   System.out.println("Last statement of block1");
}
```

The output of this code will be "**Last statement of block1**". The (var==2) condition will be true, and **break block2** will jump out of blocks **block3** and **block2** to the next statement of **block1**.

Breaking Out from Loops

When the **break** statement is used without its ***label*** parameter, it interrupts execution of the loop containing that **break** statement. In case of nested loops, labels can be used to identify each loop, and the ***label*** parameter will tell which loop to terminate.

Examples:

```
int i;
// The for loop
for (i=9; i>0; i--)
{
  System.out.println("i=" + i);
  if (i==7) break;
}
// The while loop
i = 9;
while (i>0)
{
 System.out.println("i=" + i);
 if (i==7) break;
 i--;
}
// The do...while loop
i = 9;
do
{
 System.out.println("i=" + i);
 if (i==7) break;
 i--;
} while (i>0);
```

All three loops iterate the variable **i** from 9 down to 0, but all will stop when **i** reaches 7, producing same output:

```
i=9
i=8
i=7
```

Here is an example of breaking out from nested loops:

```
int x, y = 0;
loopA: for (x=9; x>0; x--) {
   loopB: for (y=9; y>0; y--) {
     if (x==7 & y==7) break loopA;
}
System.out.println("x/y=" + x + "/" + y); // prints: x/y=7/7
```

The “continue” Statement

The **continue** statement can be used only within the body of a loop to stop the current iteration and start the next iteration. In case of labeled nested loops, the **continue** statement can specify which loop to resume.

The **format** of the continue statement is:

continue [*label*]

The ***label*** parameter specifies the name (label) assigned to a loop and is optional.

The **continue** statement works similar to the **break** statement, but instead of terminating the loop, it forces the next iteration of the specified loop.

Example:

```
// simple for loop
int i;
for (i=4; i>1; i--) {
  if (i==3) continue;             // go to next iteration when i = 3
  System.out.println("i=" + i);  // will print i=4 and i=2
}

// nested loops
int x,y = 0;
loopA: for (x=0; x<=1; x++) {
   loopB: for (y=0; y<=1; y++) {
     if (x == y ) continue loopA;
     System.out.println(x + "/" + y); // will print only 1/0
  }
}
```

Arrays

An **array** is a multi-dimensional collection of data elements of the same type and referenced by a common name and a numeric index, e.g. - myArray[3].

Java provides a special syntax for handling arrays.

Declaring Arrays

There are two equivalent formats of the array declaration:

type arrayName [ ];

type [ ] *arrayName*;

For example, **int myArray [ ];** declares the myArray variable as the name of a one-dimensional array of integer data elements. While each data element of the array is of type **int**, the myArray variable is said to be of type **int[]**.

Note: No physical array is actually created at the time of declaration.

Creating Arrays

After the array is declared, it can be created in one of two ways. The first way is to use an **array initializer** when the array is declared:

```
int myArray [] = { 1, 2, 3, 4, 5 };  // the array of five integers
```

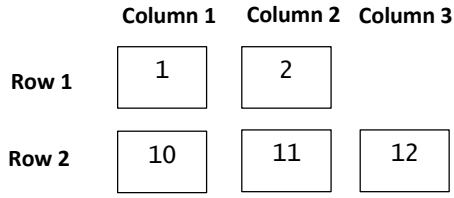
Another way of creating arrays is by using the **new** statement:

```
int myArray [];       // declares myArray as an array variable
myArray = new int[5]; // creates the array {0, 0, 0, 0, 0}
myArray [0] = 1;      // assigns 1 to the first element of the array
myArray [2] = 3;      // assigns 3 to the third element of the array
```

Note, that the **new** statement creates an array and assigns the default values to all of its elements. The default value depends on the array elements' type; for numbers - **0**, for Boolean - **false**, and for classes - **null**.

Multi-Dimensional Arrays

The easiest way to explain how the multi-dimensional arrays are created is by reviewing an example. Let's say we want to create the following two-dimensional array:



Using an array initializer, the above array can be created with this statement:

```
int [][] N = { {1, 2}, {10, 11, 12} };
```

The array will consist of five elements and the values to the array elements will be assigned as follows:

```
N [0] [0] = 1;
N [0] [1] = 2;
N [0] [2]      // ← element does not exist
N [1] [0] = 10;
N [1] [1] = 11;
N [1] [2] = 10;
```

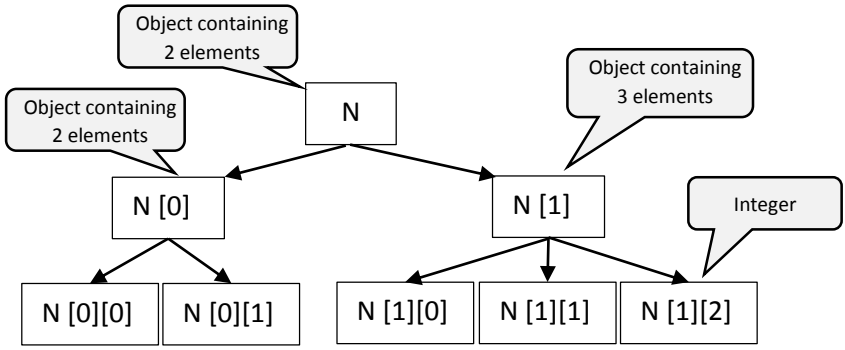
We can also create this multi-dimensional array using the **new** statement:

```
int [][] N;
N = new int [2] [3]; // the 2x3 array with 6 elements created
```

However, the result of the above code is a 2 by 3 table with 6 elements. To get rid of the N[0][2] element, we need to rewrite the code:

```
int [][] N;

N = new int [2] [];  // two columns with undefined number of rows
N [0] = new int [2]; // first row will contain 2 columns
N [1] = new int [3]; // second row will contain 3 columns
```

To avoid confusion with how arrays (or tables) are implemented in other languages, you should envision Java arrays as hierarchical structures of objects. The **N** array we just reviewed can be viewed as follows:



As objects, **N**, **N[0]**, and **N[1]**, have the **length** property, which is very useful when we want to iterate through the array:

```
// This code prints all elements of the two-dimensional array N

for (int x=0; x<N.length; x++)
{
  for (int y = 0; y < N[x].length; y++)
  {
      System.out.println(N[x][y]);
  }
}
```

Methods

A **method** is a block of code that can be referenced and called for execution by its name. It is constructed as follows:

accessAttributes returnType methodName (*parameters*) { *body* }

Example:

```
public int myMethod (int x, int y) {
   return x + y ;
}
```

This method accepts two integer numbers and returns its sum to the calling program (i.e. to another method).

The **access attributes** are optional. They will be discussed in detail in a separate chapter.

The **return type** can be of any primitive type, an array, a class, or **void**, which indicates that no data is returned by the method.

The **method name** can be any name of your choice.

The **parameters** are optional.

The **body** is a set of variables declarations and executable statements.

The "main" Method

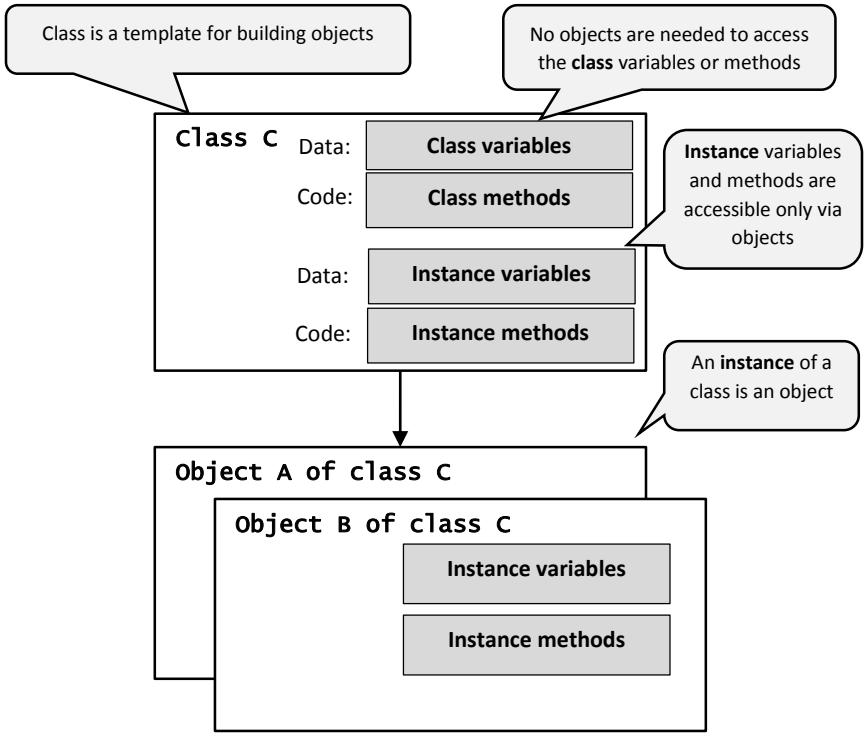
The **main** method is a special method that starts execution of all Java applications. Java looks for this method in all classes presented for execution, and expects it to be defined exactly like this:

```
public static void main (String args[]) { body }
```

Classes and Objects

Java is an Object-Oriented language, meaning that its primary purpose is to manipulate objects. An **Object** is a logical entity consisting of **data** and **code**. All objects are built from templates called **classes**, which makes them central to the Java language. Any functionality you wish to implement in Java, must be constructed as a class.

The following diagram illustrates the relationship between classes and objects:



Declaring Classes

Classes are declared using the **class** statement:

```
class className
   }
      type variable1;
      . . .
      type variable;

      type method1(parameters);
      . . .
      type methodN(parameters);
   }
```

Example:

```
// Class Temperature provides functionality to convert
// temperatures from Celsius to Fahrenheit and vice versa

class Temperature
{
   // Class variables and methods:
   static double t;           // temperature value
   static double FtoC ()      // method for converting °F to °C
      {return (temp - 32 ) / 1.8;}
   static double CtoF ()      // method for converting °C to °F
      {return temp * 1.8 + 32;}

   // Instance variables and methods:
   double temp;               // temperature value
   double toCelsius ()        // method for converting °F to °C
      {return (temp - 32 ) / 1.8;}
   double toFahrenheit ()     // method for converting °C to °F
      {return temp * 1.8 + 32;}
}
```

Class Variables and Methods

The code in previous example declares a new class Temperature with two sets of variables and methods. The main difference between those sets is in the usage of the **static** scope modifier. All variables and methods declared as **static** are called **class variables** and **class methods**, and can be used without creating any objects. Other variables and methods are called **instance variables** and **instance methods**, and can be used only through object instances.

The code below illustrates the usage of the class variable **t** and class methods FtoC() and CtoF() of the class Temperature declared above:

```
   double temp;

   // Class variables and methods can be referenced via class name
   Temperature.t = 77.0;
   temp = Temperature.FtoC();
   System.out.println(Temperature.t + "F=" + temp +"C");

   // Class variables and methods can be referenced
   // by their names directly (if the names are unique in the program)
   t = 25.0;
   temp = CtoF();
   System.out.println(t + "C=" + temp +"F");

This code will print:
77.0F=25.0C
25.0C=77.0F
```

Initializing Class Variables

During program execution, when a class is first used, i.e. when it is loaded into the Java Virtual Machine (JVM), its class variables are assigned default values. For numeric variables, the default value is zero. If there is a need to initialize a class variable to a non-default value, it can be done in the declaration statement itself, or in a **statement block** declared as **static**.

The example below declares the Temperature class with two class variables - **tF** and **tC.** The tF gets its initial value during declaration, and the tC is initialized in the static statement block. Note that the declaration of the tC must be made outside of the static statement block, otherwise its scope will be restricted to that block only.

Example:

```
// When the Temperature class is loaded into JVM
// the class variable tF will be assigned the value 77.0
// and the tC=FtoC(); code will run to calculate the tC value.

class Temperature {
   static double tF = 77.0;   // class variable tF = 77.0
   static double tC;          // class variable tC = 0.0
   static { tC = FtoC(); }    // static statement block will convert
                              // tF to °C and put it into tC
   // method FtoC converts °F to °C
   static double FtoC ()   {return (tF - 32 ) / 1.8;}
}
```

Everything that was said so far about the class variables and methods can be summed up in three rules:

Declaration Rule:

Class **variables**, **methods**, and **statement blocks** must be declared using the **static** modifier

Accessibility Rule:

Class **variables** and **methods** can be referenced by their names directly if the name is unique, or via the class name in all cases

Execution Rule:

All **static** declarations and statement blocks {…} of a class will be processed (i.e. executed) one time only, when the class is first used in the program

Instance Variables and Methods

The variables, methods, and statement blocks in a class declaration, that do not have the **static** modifier, are called **instance variables**, **instance methods**, and **instance statement blocks**. Their usage is subject to the following rules:

Declaration Rule:

Instance **variables**, **methods**, and **statement blocks** must be declared without the **static** modifier

Accessibility Rule:

Instance **variables** and **methods** can be used only after an object (instance) of the class is created, and must be referenced via the object name

Execution Rule:

All **instance** declarations and statement blocks {…} of a class are processed (i.e. executed) each time an object (i.e. instance) of the class is created.

The following example shows how objects could track their creation sequential number.

Note: The **new** statement (discussed in the next section) creates an object of specified class.

```
class myClass
{
   static int totalCount;               // class variable
          int myNumber;                 // instance variable
          { myNumber = ++ totalCount; } // instance statement block

   // Main program
   public static void main (String args[])
   {
      myClass obj1 = new myClass();      // creates the object obj1
      myClass obj2 = new myClass();      // creates the object obj2
      System.out.println(obj1.myNumber); // prints 1
      System.out.println(obj2.myNumber); // prints 2
   }
}
```

The class variable **totalCount** will be initialized to zero when the class is first referenced in the **main** program. The instance statement block will run when object **obj1** and **obj2** are created, incrementing the **totalCount** by 1 and assigning its new value to the instance variable **myNumber**. Note that objects obj1 and obj2 will have their own copies of the myNumber variable.

The previous example contains one thing that was not covered yet - the **new** statement, which creates objects. Let's take a closer look at it and the whole process of creating new objects.

Working with Objects

Objects are the core of Java language, making it is very important to have a good understanding of how to create objects and how to use their contents.

Creating Objects

The process of creating a new object is called **instantiation**, and is performed using the **new** statement. The format of the **new** statement is this

className objectReferenceVariable = **new** *className*(*parameters*);

Example:

```
myClass object1 = new myClass();
```

This is what happens during execution of the above statement:

1) "**myClass object1"** declares the variable **object1** of type **myClass**. This variable can be used to reference any object of class **myClass**, that's why such variables are also called **object references**.

2) The **new** operator creates a new object; all instance declarations of variables, statement blocks, and methods are processed.

3) The **myClass()** method is invoked to optionally initialize the instance variables of the newly created object.

4) The reference between the variable **object1** and the new object is established.

Constructors

The **myClass()** method mentioned above is called a **constructor**, which is a method with the same name as the class name and without the return type specified. It is invoked by the JVM when an object of the class is being created. The purpose of a constructor is to initialize the instant variables of the class. If you do not declare any constructors, the JVM will create an empty one: **className() { }**, which will do nothing. However, once at least one constructor is declared by the programmer, the default constructor is gone and cannot be used. Please remember that a constructor is invoked right after the instance declarations and statement blocks are processed. Below is an example of a class that keeps track of its objects:

```
class testClass
{
   static int count;       // Total objects counter; defaults to zero
   int myNumber;           // Object's sequential number

   {myNumber = ++count;}  // instance block increments the count
                          // and assigns it to object's myNumber

   testClass() {}         // Constructor without parameters

   testClass(int n) {     // Second constructor; assigns specified
      myNumber = n;       // sequence number to new object
   }

  // ------Execution--------

  public static void main (String args[])
  {
   testClass obj1 = new testClass();
   testClass obj2 = new testClass();
   testClass obj3 = new testClass(10);
   System.out.println(obj1.myNumber); // prints 1
   System.out.println(obj2.myNumber); // prints 2
   System.out.println(obj3.myNumber); // prints 10
  }
}
```

Instance block executes every time an object is created

Create three object of class **testClass**

Note, that we defined two different constructors with the same name. This is called "**method overloading**".

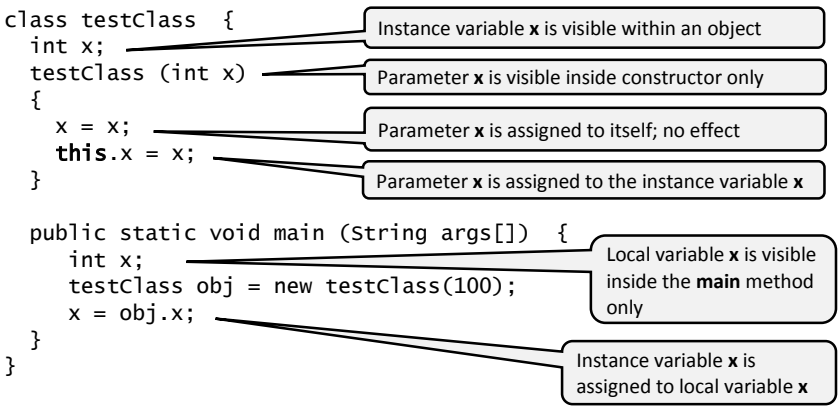
Method Overloading

Method overloading is a mechanism used by Java to distinguish between methods having the same names but different parameter lists. Each method in Java is distinguished by its **signature** – the name of the method along with the list of parameter types. The signature of the testClass() method could be expressed as "**testClass**", and the signature of the testClass(int n) would be "**testClass,int**".

The "this" Keyword

In previous examples, instance variables were referenced in constructors directly by their names. But what will happen when a variable with the same name as the instance variable is declared in the **main** method? This situation is perfectly legal because all variables declared within the **main** method are not "visible" outside of that method.

Another confusing situation would be when the name of an instance variable is also used in the constructor's parameter list. This problem can be solved with the use of the keyword **this**, which represents the current object. See this example:

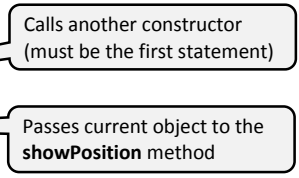


The **this** keyword can be used not only to refer instance variables, but also to invoke one constructor from another, or to pass the current object as an argument between methods.

Important note: The **this** keyword can be used only in constructors.

Another example of using the **this** keyword:

```
class Position {
   int x, y, z;
   Position (int x, int y, int z) {
      this.x = x;
      this.y = y;
      this.z = z;
   }
   Position (int x) {
      this(x, 0, 0);
      showPosition(this);
   }
   void showPosition (Position p) {
      System.out.println("My position: "+p.x+","+p.y+","+p.z);
   }

   public static void main (String args[])  {
      Position obj = new Position(5);
   }
}
```



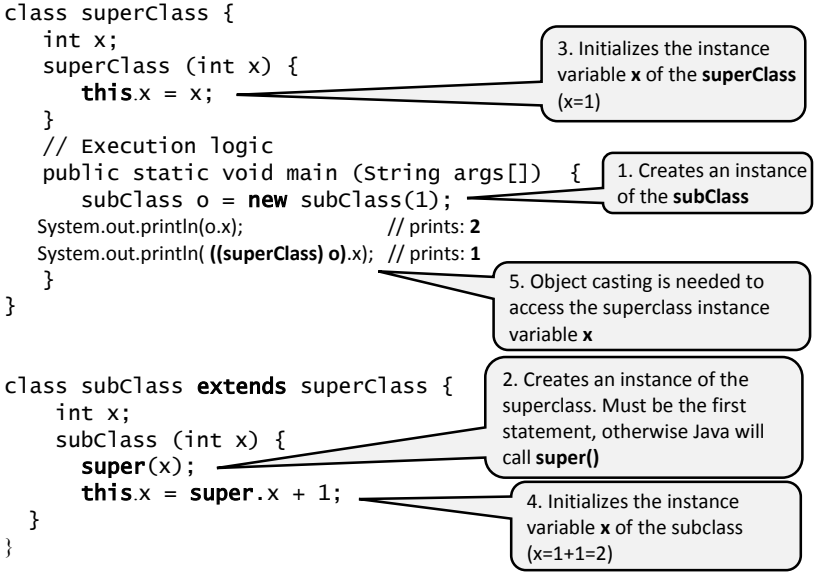
The above code prints: **My position: 5,0,0**

The “super” Keyword

One of the Object-Oriented Design principles is **inheritance**. **Inheritance** represents the parent-child relationship between classes, when a child class (**subclass**) inherits all the features of the parent class (**superclass**). Java uses the “**extends**” keyword in class declaration to establish inheritance:

```
class superClass { body }
class subClass extends superclass { body }
```

Example of creating an object of a subclass:



When an object of a subclass is created, a superclass object is created as well, implicitly or explicitly. It means that any subclass object consists of the subclass instance and its parent class (superclass) instance.

In the above example, the user-supplied subclass constructor **subClass()** creates the superclass instance by calling the superclass constructor **superClass()** via the **super(x)** statement. The keyword **super**, when used in the subclass constructor, represents the current object of the parent class. In the example, it is also used to access the instance variable **x** of the superclass.

Object Casting

Note, that the **super** keyword can be used only in constructors. So how do we access the instance variables or instance methods of a superclass from a subclass object? In the above example the variable **x** is defined in both the subclass and the superclass. The subclass' **x** can be referenced via the subclass object name as follows: *subclassObject*.**x**, but the superclass' **x** is hidden. The solution to this problem is **object casting**. We can cast a subclass object to any of its superclass objects. Consider an example with two classes **Parent** and **Child**, both declaring an instant variable **x**. Class **Parent** is superclass of class **Child**. An instance (object) of class **Child** would contain two instances of variable **x**. The code below shows how to access all of the different instance variables by utilizing the object casting mechanism:

```
class Parent { int x; }

class Child extends Parent { int x; }

class Test {

  public static void main (String args[]) {
    int y;

    // A Child object referenced by a variable of type Child:
    Child  c = new Child();
    y = c.x;                // reference to the x of class Child
    y = ((Parent) c).x;     // reference to the x of class Parent

    // A Child object referenced by a variable of type Parent:
    Parent p = new Child();
    y = p.x;                // reference to the x of class Parent
    y = ((Child) p).x;      // reference to the x of class Child
  }
}
```

This table summarizes the accessibility of instance variables of super and sub-classes:

Object reference	**class Parent**		**class Child extends Parent**	
	Instance **variables**		Instance **variables**	
	unique	same as in Child	unique	same as in Parent
Parent **p1** = new Parent()				
p1.	can access	can access	-	-
Parent **p2** = new Child()				
p2.	can access	can access	can access	-
((**Child**) **p2**).	can access	-	can access	can access
Child **c** = new Child()				
c.	can access	-	can access	can access
((**Parent**) **c**).	can access	can access	-	-
super.	can access	can access	-	-

Method Overriding

Method overriding is a special mechanism allowing for declaring instance methods with the same signatures (name and parameter types) in a superclass and any of its subclasses.

This table presents different ways of accessing instance methods of super and sub-classes. Please note that **object casting** does not give you availability to access overridden methods of a superclass from a subclass. Once a subclass object is created, only the subclass' instance per each overridden method is accessible.

Object reference	**class Parent**		**class Child extends Parent**	
	Instance **methods**		Instance **methods**	
	unique	overridden	unique	overridden
Parent **p1** = new Parent()				
p1.	can access	can access	**-**	**-**
Parent **p2** = new Child()				
p2.	can access	**-**	**-**	can access
((Child) p2).	can access	**-**	can access	can access
Child **c** = new Child()				
c.	can access	**-**	can access	can access
super.	can access	can access	**-**	**-**
((Parent) c).	can access	**-**	**-**	can access

Determining the Type of an Object

When working with objects, sometimes it might be not clear what class an object belongs to. For example, if class **A** is extended by classes **B** and **C**, an object reference of type **A** can reference objects of classes A, B, and C:

```
A obj;
obj = new A();   // obj is a reference to an object of class A
obj = new B();   // obj is a reference to an object of class B
obj = new C();   // obj is now references an object of class C
```

An object reference of type Object can refer any object, of any class:

```
Object obj;
obj = String("text");
obj = Integer(123);
obj = MyClass();
```

So, how can we determine the real type of an object referred by a variable? The answer is – with the help of the **instanceof** operator.

The format of the **instanceof** operator is:

objectReference **instanceof** *classType*

Example:

```
Object obj;
obj = new String("text");
. . .
obj = new Date();
. . .
if  (obj instanceof String)
     System.out.println("Object of type String");

if  (obj instanceof Date)
     System.out.println("Object of type Date");
```

Summary

Here is the summary of what we have learned thus far about the objects:

Objects are instantiated (i.e. created) using the **new** statement, which has the following format:

className **new** *constructor*;

A **constructor** is a special method with the same name as the class name that is invoked automatically by the JVM when an object of the class is being created.

The format of the constructor is: ***className*(*parameters*) {*body*}**
The *parameters* and *body* are optional, the return type is omitted.

The constructor is invoked after all instance declarations are processed and instance blocks (if present) are executed.

A class can have several constructors with different **signatures** (i.e. with different sequences of argument types).

An empty constructor (**className() {}**) is automatically created by Java when no constructors are declared explicitly.

The default constructor (**className() {}**) will not be created by the JVM if at least one explicitly declared constructor is present.

A constructor can invoke another constructor of the same class by executing the **this(*parameters*);** as the first statement of the constructor's body.

Instance variables can be referenced in a constructor via "**this**" keyword as follows: **this.*variableName***

The keyword "**super**" represents a superclass (parent) of a subclass (child) and can be used in the constructor of a subclass to invoke the superclass constructors - **super(*parameters*);**, to access the superclass instance variables - **super.*variable*;**, and to call the superclass methods – **super.*method*();**.

A superclass and its subclass can declare instant variables with the same names. The **object casting** mechanism should be used if we want to access non-unique instance variables of a superclass from a subclass:

```
( (superClass) subclassObject ).x
// points to variable x of superClass
```

A superclass and its subclasses can declare instant methods with same signatures (names and parameter list). The **method overriding** mechanism makes sure that proper method is instantiated when an object is created.

The **instanceof** operator checks if an object belongs to specified class.

Type Wrappers

The usage of primitive data types has its limitations. Many of the data structures operate on objects. For example, you cannot construct a set of integers (a set is a collection of objects with distinct values which will be discussed later in the book). That's why Java provides **type wrappers** – classes that encapsulate a primitive type within an object. There is one wrapper for each primitive type:

`Double`, `Float`, `Long`, `Integer`, `Short`, `Byte`, `Character`, and `Boolean`.

Each of the above classes offers a wide range of methods (most of them static) allowing various manipulations over the underlying primitive type including the conversions between different primitive types.

Example:

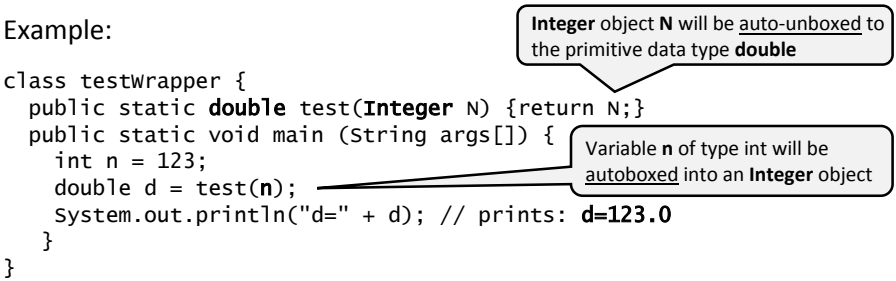
```
Integer N = Integer.valueOf(10); // N is an Integer object
int n = N.intValue();
double d = N.doubleValue();
String s = Integer.toBinaryString(n);
System.out.println("n="+ n + " d=" + d + " s=" + s);
```

The output of this program is "**n=10 d=10.0 s=1010**".

Autoboxing

In JDK5, Java introduced the **autoboxing/auto-unboxing** feature. **Autoboxing** automatically creates a type wrapper object and encapsulates a primitive type into it when an object of that type is needed. **Auto-unboxing** is the reverse process; it retrieves a primitive type from a type wrapper object when it is needed.

Example:




```
class testWrapper {
  public static double test(Integer N) {return N;}
  public static void main (String args[]) {
    int n = 123;
    double d = test(n);
    System.out.println("d=" + d); // prints: d=123.0
   }
}
```

Strings

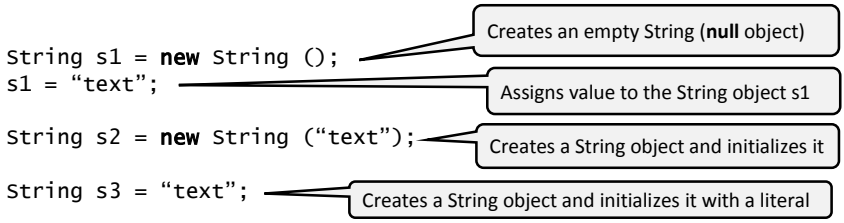
In Java, a sequence of characters is represented by the class **String**. The **String** class is a regular class, and can be handled as such. However, due to its extensive usage, Java added some convenient features that make handling of strings more intuitive. In this chapter, we will review all the special features of the String class.

String Literals, Creating Strings

The **string literal** is a sequence of characters enclosed within double quotes:

```
"First line of a string literal. \n Second line."
```

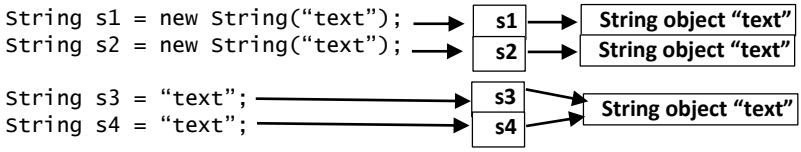
String literals can be used to create **String** objects directly:



As you can see in this example, a **String** object can be created without using the **new** operator – just by direct assignment of a string literal to a String variable. This might look strange at first, but there is a simple explanation:

A stand-alone string literal in Java is an object of class String.

For each unique string literal Java creates one object that can be referenced by different object variables. The **new** operator, on the other hand, always creates a new object regardless of the literal value passed to the constructor:



Since a stand-alone string literal represent a String object, you can work with it as an object, without creating a reference. For example, you can invoke the **length()** instance method of the String class to get the length of a literal, as follows:

```
int n = "text".length();   // n = 4
```

Another important fact about the String object:

> Once created and initialized, a String object cannot me modified. Its value cannot be appended, truncated, or replaced. If you need a modifiable string – use the StringBuffer class (not covered in this book).

Comparing Strings

It is important to understand that there is a difference between comparing two actual string values and comparing two references to String objects:

```
String s1 = "text";
String s2 = "text";
boolean b = (s1 == s2); // b=true
```

s1 and **s2** are pointing to same String object having value "text"

```
String s3 = new String ("text");
String s4 = new String ("text");
boolean b = (s3 == s4); // b=false
```

s3 and **s4** are pointing to different String objects having same value "text"

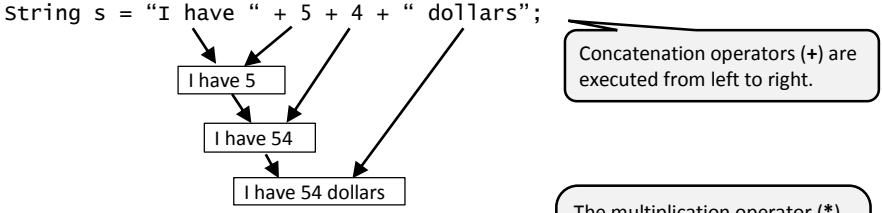
To compare the actual values of two strings (which in fact are encapsulated within String objects), we need to use the instance method **equals()** of class String:

```
String s1 = new String("text");
String s2 = new String("text");
boolean b = ( s1.equals(s2) );   // b = true
```

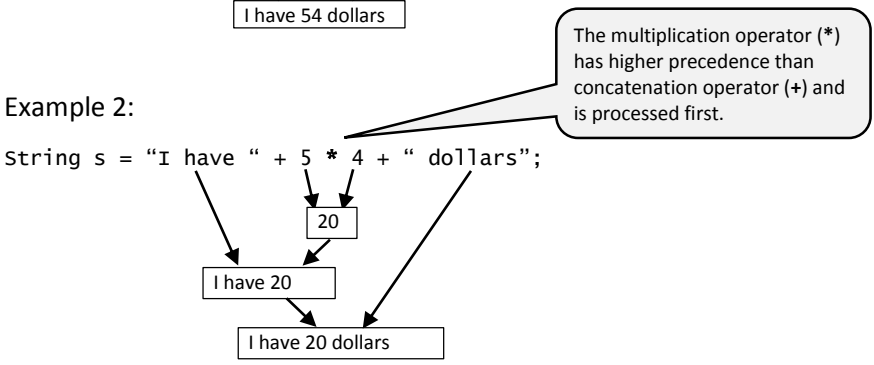
String Concatenation Operator

The **string concatenation operator** (**+**) combines two operands into one string. The primitive data type operands (boolean, char, int, double, etc.) are converted into their string representation, and object operands are converted into a string by calling the instance method **toString()**. This type conversion is performed on an operational basis, from left to right, as shown in the following examples.

Example 1:



Example 2:



String Methods

In this paragraph we will present some commonly used methods of class **String**.

Determining String Length

```
String s = "text";
int lng = s.length();        // lng = 4
lng = "my string".length();  // lng = 9
```

Comparing Strings

The **compareTo()** method compares two strings lexicographically. The comparison is based on the Unicode value of each character in the strings and is performed by comparing one character at a time, from left to right, until unequal characters are found or until the end of one of the strings is reached. The value returned by the compareTo() method is calculated as difference between the numeric values of unequal characters ('a' - 'b' = -1). If one of the strings reaches its end, the method returns the difference of the length of the strings.

```
String s = "ab";
int n;
n = s.compareTo("ab");    // n = 0   ("ab" = "ab")
n = s.compareTo("aa");    // n = +1  ("ab" > "aa")
n = s.compareTo("bb");    // n = -1  ("ab" < "bb")
n = s.compareTo("abaa");  // n = -2  ("ab" < "abaa")
```

Accessing String Characters

```
String s = "abcdef";
char c = s.charAt(0);        // c = 'a'
char c = s.charAt(5);        // c = 'f'
```

Searching for a Character

```
String s = "abcbcd";
int pos;
pos = s.indexOf('g');     // pos=-1 (character 'g' not found)
pos = s.indexOf('b',2);   // pos=+3 (first 'b' after position 2
                          //         is in position 3)
pos = s.lastIndexOf('c'); // pos=+4 (last symbol 'b' is in position 4)
```

Searching for a Substring

```
String s = “abcbcd”;
int pos;
pos = s.indexOf(‘abd’); // pos = -1 (sub-string ‘abd’ not found)
pos = s.indexOf(‘bc’);  // pos = 1 (sub-string ‘bc’ is in position 1)
```

Extracting a Substring

```
String s = “abcdef”;
String s1 = s.substring(1,3); // s1 = “bc”
String s2 = s.substring(1);   // s2 = “bcdef”
```

Creating a new String from existing String

```
String s = “ abc ”;
String s1 = s.trim();             // s1 = “abc”
String s2 = s.replace(‘ ’,’+’);  // s2 = “+abc+”
```

Creating a Character Array from a String

```
String s = “abcd”;

char[] c = s.toCharArray();

s.getChars(0, 2, c, 2);
```

c[0]=’a’
c[1]=’b’
c[2]=’c’
c[3]=’d’

Copy characters in positions 0, 1 of string **s** into array **c** starting from position 2.

c[0]=’a’
c[1]=’b’
c[2]=’**a**’
c[3]=’**b**’

Creating an Array of Bytes from a String

This method converts each character of a string from Unicode (16-bit) into the local 8-bit encoding (usually ASCII) and creates an array of bytes.

```
String s = “abcd”;

byte[] b = s.tgetBytes();
```

b[0]=97 (‘a’ in ASCII)
b[1]=98 (‘b’ in ASCII)
b[2]=99 (‘c’ in ASCII)
b[3]=100 (‘d’ in ASCII)

Creating a String from an Array of Characters

The static **copyValueOf** method of class String create a String object from an array of characters. It has the following two formats:

```
static copyValueOf(char[] charArray);

static copyValueOf(char[] charArray, int startPos, int numberOfEle-
ments);
```

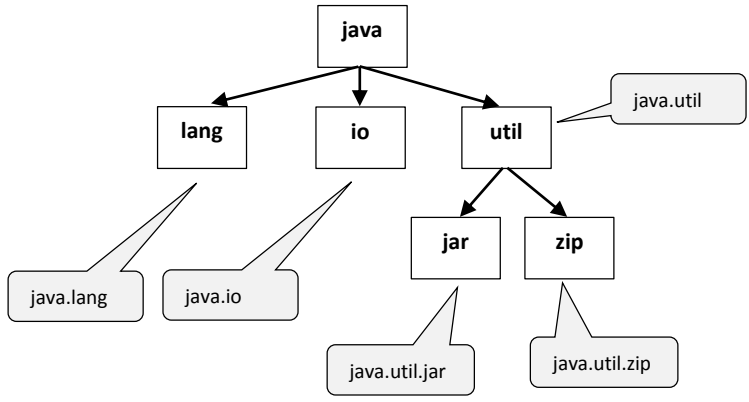
Example:

```
char[] c = {‘a’, ‘b’, ‘c’, ‘d’};

String s1 = String.copyValueOf(c);     // s1 = “abcd”;

String s2 = String.copyValueOf(c,1,2); // s2 = “bc”;
```

Packages

A **package** is a container for classes. All Java classes are spread across more than 200 packages. The fundamental classes reside in the **java.lang** package. Examples of other packages are **java.io**, **java.util**, **java.net**, **java.math**, etc.

Packages are arranged in a tree-like structure:



The java.lang package is always available to the Java compiler. When your program needs to use a class residing in a different package, that class or the whole package containing that class must be **imported** into the program:

```
import java.math.*;
import java.io.FileReader;
   class myClass { . . . }
```

All classes from package java.math will be available

Class **FileReader** from the package **java.io** will be available

User-defined classes are also placed into packages. For example, the myClass from the above example will be placed in a default, "no-name" package. To place it into a particular package, the first statement of the program (not counting comment lines) must be the **package** statement:

```
package mypack.io;
import java.io.FileReader;
class myClass { . . . }
```

The **myClass** class will be placed into the **mypack.io** package.
If mypack.io does not exist it will be created.

Note that classes, after compiling, retain information about the package they belong to. So, if you copy a class from one package into another, the

new copy will not work. However, there is one exception. When a class is compiled without the **package** statement, it can be placed in any named package and be accessible by all classes in that package.

The implementation of the package concept varies depending on the environment in which Java is installed (Linux, Mac, Windows, etc.) and therefore it is not included in the scope of this book.

Modifiers

The following entities of the Java language – classes, interfaces, variables, and methods – can be declared with so-called **modifiers** controlling their behavior and/or accessibility. Here, myMethod is declared with two modifiers: **public** and **static**.

```
public static void myMethod() { … };
```

Class Modifiers

There is a limited set of modifiers that can be applied to classes:

- public
- abstract
- final

The **public** modifier makes the class accessible from any package. If this modifier is not specified, the class will be accessible from the package it was declared.

The **abstract** modifier defines a class in which one or more methods are declared as skeletons - with the return type and the list of input parameters, but without the body. Abstract classes will further be explained in the "Abstract Classes" section.

The **final** modifier prohibits the class from being extended by other classes.

Access Level Modifiers

Access level modifiers control the visibility of variables and methods of a class from other classes in the same or different packages. There are four access level modifiers (in order from less to more restrictive):

public → protected → *[no modifier]* **→ private**

Example:

```
public void myMethod() { … };
protected int n;
double x;
private String name;
```

The following table shows how access modifiers affect the visibility of variables and methods of one class from its sub-classes and also from other classes residing in the same or different packages.

	Package One		Package Two	
class A	**subclass of A**	**class B**	**subclass of A**	**class C**
public	visible	visible	visible	visible
protected	visible	visible	visible	-
no modifier	visible	visible	-	-
private	visible	-	-	-

The “static” Modifier

The **static** modifier indicates that the variable or method can be accessed before any object of the class is created. A good example is the **main** method that starts execution of an application. It must be declared **static** because it is called before any object exists.

The “final” Modifier

The **final** modifier can be applied to variables, methods, and classes to “finalize” their declaration.

A **final variable** cannot be modified after it is assigned an initial value. That effectively makes a final variable a constant:

```
final double pi = 3.1415926;
```

A **final method** cannot be overridden. If a class declares some method as final, the sub-classes of that class cannot create a method with the same signature as the final method.

A **final class** cannot have sub-classes, i.e. it cannot be inherited.

Generics

Generics is a feature of Java allowing the use of generic names when specifying the class type of object parameters in the declarations of classes, interfaces, methods, or constructors. For example, if two methods have same functionality but one receives an integer parameter, and another receives a double parameter, we can declare just one method and supply that method with a "**generic**" type of the parameter, which will be substituted with the proper type during compilation.

The best way to understand all the advantages and restrictions of **generics** is by going through examples, which are illustrated below.

Generic Methods

Let's say that we want to create a method that acts on an object without knowing what its type is. One way of accomplishing that is by using the **Object** class – the superclass of all objects. We could accept an Object instance and then cast it to the desired type.

For example, method **sumArea** summarizes the total area occupied by objects of type **A** and **B** by processing one object at a time. All we know about objects A and B is that they are unrelated (i.e. belong to separate class hierarchies) and both have the **getArea** method returning their area value. The **sumArea** method can be implemented as follows:

```
class A { public double getArea() {return 10.0;}
class B { public double getArea() {return 20.0;}

class myClass
   double totalArea;
   public void sumArea(Object obj) {
      if (obj instanceof A) { totalArea += ((A) obj).getArea(); }
      if (obj instanceof B) { totalArea += ((B) obj).getArea(); }
   }
}
```

Cast the **obj** to correct type

instanceof operator determines the type of **obj**

Now, can we use **generics** to get rid of **instanceof** and object casting? The short answer is – Yes and No. At this point we come to a very important fact about the **generics**:

Generics is a compiler feature, not a run-time feature. All class type references are resolved at compile time.

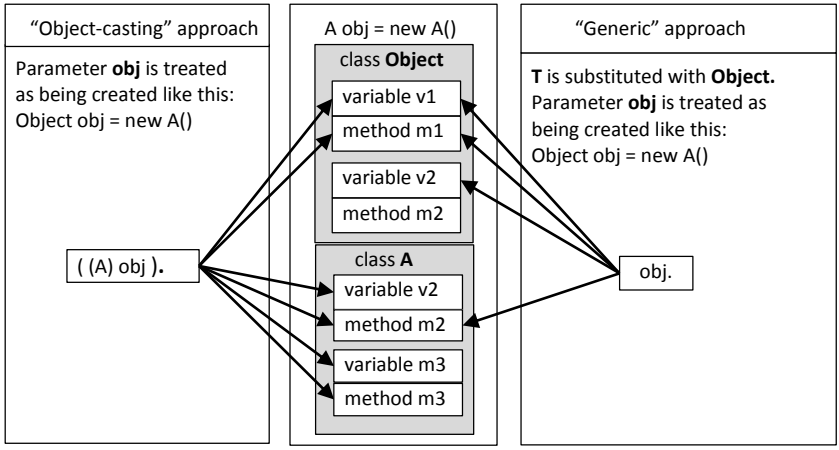
For example, the **sumArea** method can be re-written in generic notation as follows:

The **T** parameter is not a real class name, but a generic name.
It must be placed before the return type.

```
public <T> void sumArea(T obj) {
   totalArea += obj.getArea();
}
```

In the object-casting example, when the method **sumArea** receives an object of class **A**, the object reference **obj** is treated as if it was created by the statement **Object obj = new A();**. By explicitly casting **obj** to the original class **A** we acquire access to all its variables and methods.

The **generics** mechanism works somewhat differently. At compile time, the **type-variable T** must be resolved to a real class name. Since we did not provide any hints to what it might be, the **T** will be substituted with **Object** – the superclass for all classes – and that will be the only change made to the source. Without casting, object reference **obj** declared as "**Object obj = new A()"** will only have access to the instance variables and methods of class Object and also to the methods of Object overridden by class A. This diagram shows which properties of object A would be accessible in both cases:



As we see, without explicit casting, the "**generics**" code does not have access to the class **A** instance variables and methods of object **obj**. This problem can

be solved by making a few changes to the class structure and also providing more information to the generic method.

First, we can tell the generic method what class (or classes) can substitute the type-variable T:

```
public <T extends AB> void sumArea(T obj) { … }
```

The **<T extends *class*>** is called a **bounded type**, which sets the **upper bound** for classes that can be referenced by the type-variable **T**.

Second, the new class (**AB**) should be made a superclass of classes **A** and **B** and should define all instance methods we want to use within the generic method **sumArea**:

Note: class AB is not required to be abstract

```
abstract class  AB { public abstract double getArea(); }

class A extends AB { public double getArea() {return 10.0;}
class B extends AB { public double getArea() {return 20.0;}
```

Now our **obj** will be viewed as being declared with "**AB obj = new AB()**", and the **getArea** method of classes **A** and **B** will be reachable via the method overriding mechanism. Simply put, the declaration of the generic method was transformed from this:

```
public <T> void sumArea(T obj) {
   totalArea += obj.getArea();
}
```

to this:

```
public void sumArea(AB obj) {
   totalArea += obj.getArea();
}
```

Note that if we need access to the variables of classes A and B we can do it only through methods declared in the superclass AB and overridden in A and B.

Generic Classes

First of all, it is very important to understand that a **generic class** is not much different from any regular Java class. What makes it "generic" is a small addition to the class declaration syntax. Here is an example of a generic class declaration:

```
class G <T>  { … }
```

The **<…>** notation identifies class G as generic

The **<T>** notation is a **class type parameter** in which **T** is a **type-variable**, i.e. parameter representing a class name. It indicates that an arbitrary (any) class name can be used in the declarations of variables and methods of the class:

```
class myClass <T> {
   public T myValue;
   public void setValue(T obj) {
      myValue = obj;
   }

   public static void main (String args[]) {

      myClass<String> c = new myClass<String>();
      c.setValue("test");
      System.out.println(c.myValue);   // print: "test"

      myClass<Integer> n = new myClass<Integer>();
      n.setValue(123);
      System.out.println(n.myValue);   // print: "123"

   }
}
```

Note that we have to specify the class type parameter explicitly when creating objects of generic classes. This information is used by Java for **type safety** checking, making sure, for example, that an object reference of type **myClass<String>** does not point to an object of type **myClass<Integer>**.

The class declaration shown in the previous example (`class myClass <T>`) allows substitution of the **type-variable T** with any class name. In most cases, it is not desirable. Usually, we want to restrict the range of classes for which generic class objects can be built. We can achieve this by using another form of generic class declaration:

```
class G <T extends classname> { ... }
```

The *classname* parameter sets the upper boundary for class names that are allowed to be specified by the type-variable **T** and enforces that only the *classname* class or any class extending the *classname* can be specified by the **T** parameter. This is a part of the **type safety** mechanism provided by **generics**.

The example below shows the **type safety** mechanism in action:

```
abstract class  AB { public abstract String myName(); }

class A extends AB { public String myName() {return "A";}
class B extends AB { public String myName() {return "B";}

class G <T extends AB> {
   String name;

   G (T obj) { name = obj.myName();  } // class G constructor


 public static void main (String args[])
 {

    A a = new A();
    G g;

    g = new G ( a );
    g = new G<AB> ( a );


    G<B> b;
    b = new G<B> ( new B() );


    g = b;

    b = g;
  }
}
```

The type-variable **T** can specify only class **AB** or its subclasses

Error! You have to specify explicitly the object type you are passing to the constructor of class G

Compiler will check if the reference **a** is of type **AB** or its subclasses

Ensures that **b** can reference only **G** objects created with the input parameter of type **B**.

Compiler will check that the input parameter **new B()** is of type **B** (and it is), and that **b** can reference a **G** object created with the input parameter of this type.

OK. The **g** and **b** are both of type **G**

"Unsafe conversion" error: **b** can only reference **G** objects created with input parameter of type **B**, but **g** could point to the **G** object created with input parameter of type **AB**, **A**, or **B**

Passing Generic Classes as Parameters

Objects of generic classes can be passed as parameters to methods, in the same way as other Java objects:

```
class GenericClass <T> {…}
. . .
public void myMethod(GenericClass g) {…}
. . .
GenericClass<A> a = new GenericClass<A>( );
GenericClass<B> b = new GenericClass<B>( );

myMethod(a);
myMethod(b);
```

The **a** variable references an object of type GenericClass<A>

The **b** variable references an object of type GenericClass<B>

Valid calls to myMethod()

As you see, **myMethod()** can accept any object of type **GenericClass** regardless of the type-variable **T** used in creating those objects. We can add some **type safety** checking to the process by specifying restrictions for the input parameter of myMethod(). This is accomplished with the help of the **wildcard parameter <?>**. The example below demonstrates the usage of the **<?>** parameter in method declaration:

```
abstract class  AB { ... }
class A extends AB { ... }
class B extends AB { ... }
. . .

class G <T extends AB> {
. . .


public void myMethod(G<?> g) {…}


public void myMethod(G<? extends AB> g) {…}


public void myMethod(G<? super B> g) {…}
```

Objects of generic class G can be of types G, G<AB>, G<A>, or G<B>

Object **g** can be of any type allowed by the class G declaration

Object **g** can be of type **G<AB>** or its sub-classes (i.e. G<AB>, G<A>, G<B>)

Object **g** can be of type **G<B>** or its super-classes (i.e. G<B> or G<AB>)

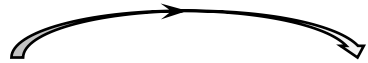
In this example <**? extends AB**> sets the **upper boundary** for passing object references, and <**? super B**> sets the **lower boundary**.

Generic Interfaces

Generic interfaces are declared in a way similar to the declaration of generic classes, for example:

```
interface GenericInterface <T> {
   public void myID (T id);
}
```

The type-variable(s) in the interface declaration are passed to the interface from the class implementing that interface:



```
class myClass <T> implements myInterface <T> {…}

interface myInterface <T> { … }
```

This is why generic interfaces can only be implemented by generic classes.

It is possible to restrict the scope of type-variables in the class declaration, in the interface declaration, or in both:

```
class myClass <T extends AB> implements myInterface <T> {…}

interface myInterface <T extends ABC> { … }
```

The Java compiler performs **type safety checking** by comparing the bounds set for the type arguments in the class to the bounds set in the interface. The class type arguments must be within bounds of the interface class arguments.

Note: Whether the type-variable **T** is used (or how it is used) in the bodies of the class or interface, is irrelevant to the type safety checking process – the type safety checking will be performed anyway.

Below are examples of valid as well as incorrect declarations.

```
class AB {}
class A extends AB {}
class B extends AB {}

interface myInterface <T extends AB> {…}

class myClass <T> implements myInterface <T> {…}


class myClass <T extends A> implements myInterface <T> {…}

class myClass <T extends String> implements myInterface <T> {…}

class myClass implements myInterface <T> {…}

class myClass <X> implements myInterface <T> {…}
```

myInterface can be implemented only by the class **AB** or its sub-classes

Valid declaration

Valid; class **A** is subclass of **AB**

Error! Class **String** is not a subclass of **AB**

Error! Class **myClass** must be generic

Error! The generic type variables in **myClass** and **myInterface** must have same name

Generic Constructors

It is important to remember that constructors are methods whose function is to create instances (objects) of classes. As methods, they can be declared as generic. The class itself can be either generic or non-generic.

We have already seen constructors of generic classes in previous examples. Here is another example:

```
class GenericClass <T> {
   public T key;
   GenericClass(T k) {
      this.key = k;
   }
}
. . .
GenericClass<Integer> g = GenericClass<Integer>(123);
```

This is a constructor of a generic class, but it is not a generic constructor!

Creates object **g** of type GenericClass<String> and sets **key**=123

The above constructor is not generic. To be generic it should be able to accept generic parameters different from those specified in the class declaration, like this:

```
class GenericClass <T> {
   public T key;
   public Object value;
   <X> GenericClass(T k, X v) {
      this.key = k;
      this.value = v;
   }
}
. . .
GenericClass<String> g = GenericClass<String>(123,"test");
```

This is a generic constructor.

Creates object **g** of type GenericClass<String> and sets **key**=123 and **value**="test"

The **<X>** in front of the constructor indicates that this constructor can accept the type-variable **X**, which is not related to any type parameters in the class declaration.

Very important note: all generic type-variables of a constructor (in this case - X) are local to the constructor. Therefore, you should not use the names of type variables declared by the class:

```
class GenericClass <T> {
   public T key;
   <T> GenericClass(T k) {
      this.key = k;
}}
```

This **T** is local to this constructor.

Error! Type mismatch: the **T** declared in class and **T** declared in constructor are different type-variables.

By default, type-variables can specify any class type or interface type. To set boundaries to the allowed values of a type-variable, use the **extends** keyword along with this format:

```
<T extends class> constructorName (…) {…}
```

The ***T*** type-variable is restricted to the type ***class*** or any of its sub-classes.

Example:

```
class GenericClass <T> {…}
<V extends ABC> GenericClass (T obj1, V obj2) {…}
```

The above type-variable **V** can be of type ABC or any of its sub-classes.

Generic constructors can also be used in non-generic classes.

Example:

```
class nonGeneric {

   String name;

   <T extends myInterface> nonGeneric(T o) {
      this.name = o.getName();
   }

   public static void main (String args[]) {

     nonGeneric g = new nonGeneric( new A() );

     System.out.println( g.name );    // prints: "Class A"
   }
}

interface myInterface { String getName(); }

class A implements myInterface {
   public String getName() {return "Class A";}
}
```

Only classes implementing the interface **myInterface** can be passed to this constructor.

Notice the usage of the interface name in the constructor declaration:

```
 <T extends myInterface>
```

This is valid, because when a class implements an interface, objects of that class can be referenced by variables of the interface type.

Lambda Expressions

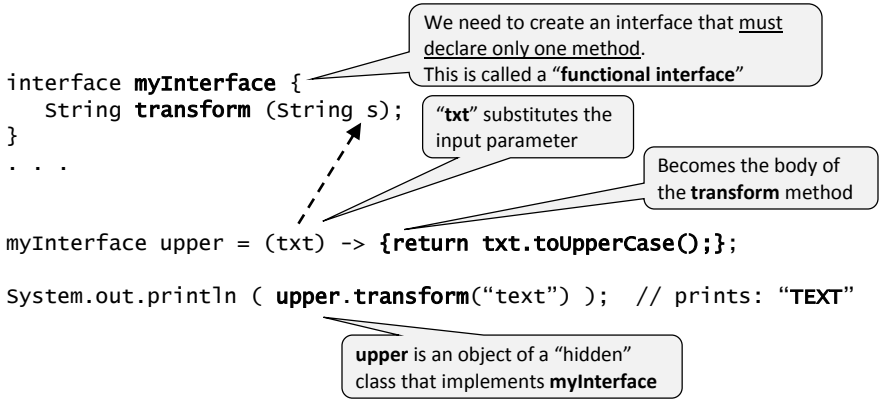
A **lambda expression** is a special construct of the Java language used for defining custom methods outside of any class. In reality, the **lambda expression** automatically declares a "hidden" class, creates an instance of that class, and attaches the specified method to it.

Let's illustrate this process with an example. We will implement the same functionality using the "traditional" approach and also using the lambda expressions. The task is to create a method converting any text to uppercase.

Here is the "traditional" way:

```
class myClass {
   String transform(String s) { return s.toUpperCase();}
}
. . .
myClass c = new myClass();
System.out.println ( c.transform("text") );  // prints: "TEXT"
```

The same **transform** method declared using lambda expression:



It seems that the "lambda" approach does not save us much effort – instead of declaring a class, we declare an interface, and instead of creating an object of the class we use the lambda expression that defines the body of the method and also creates an object of a "hidden" class. Even worse – the lambda expression approach looks like an "un-object-oriented" way to achieve the same thing that could be done the "object-oriented" way.

All of the above is true, but sometimes it is not practical to create a new class for just one or for a rarely used method. Also, the same interface can be used for creating of many different methods. For example, here is another method that uses the same interface **myInterface** presented earlier:

```
myInterface lower = txt -> txt.toLowerCase();
System.out.println ( lower.transform("TEXT") );  // prints: "text"
```

To fully explore this capability, the package **java.util.function** provides several predefined functional interfaces to be used in lambda expressions. One of these functional interfaces – **Function<T,R>**, applies its method **apply()** to object of type **T** and returns the result as an object of type **R**. The [partial] declaration of the **Function** interface looks as follows:

```
public interface Function <T, R> {
  R apply(T t);
  . . .
}
```

Let's use this interface and a lambda expression to create a method that transforms digits into words:

```
import java.util.function.*;
. . .
Function<Integer, String> spell = N ->

{
   String digits [] =
      {"zero", "one", "two", "three", "four",
       "five", "six", "seven", "eight", "nine"};

   return digits[N.intValue()];
};

Integer N = Integer.valueOf(4);

System.out.println(spell.apply(N));  // prints: "four"
```

spell is an object of a "hidden" class that implements the **Function** interface

N will be the input parameter to **apply()**

This block will become the body of the **apply()** method

Returns N'th element of array **digits**

Creates object of type **Integer** with the value of 4

Calls the **apply()** method of the object **spell**.

In previous examples we were explicitly declaring the object references of functional interface types:

```
Function<Integer, String> spell
myInterface lower
```

spell and **lower** are object references of types **Function** and **myInterface**

Then, lambda expressions were assigned to these object references. This method of creating lambda expressions clearly shows what we're trying to achieve, but is not mandatory. In general, it is not necessary to explicitly declare functional interface object references. A lambda expression itself can be viewed (and used) as an object of the corresponding functional interface. For example, we can pass a lambda expression as a parameter to a method if that parameter is declared as a functional interface.

Here is an example; the **IntPredicate** is a factory-supplied functional interface declaring the following **test** method:

```
boolean test(int value)
```

We are going to utilize it for selecting numbers within a specified value range: (n > min) & (n < max). Here is what needs to be done:

- Create an object of type **IntPredicate**
- Substitute its **test** method with the desired conditional expression
- Use the **test** method to check if a number is within the range

```
import java.util.function.*;

class testLambda
{
   static int min = 0;
   static int max = 100;

   public static IntPredicate filter(IntPredicate p) {return p;}

   public static void main (String args[]) {

       int i = 100;

       if filter( n -> (n > min & n < max) ).test(i)
          System.out.println("within range");
       else
          System.out.println("out of range");
   }
}
```

The **filter()** method accepts object of type **IntPredicate** (functional interface) and returns same object

The "**n -> (n > min & n < max)**" is an object of type **IntPredicate**

Explanation: the **filter** method accepts an object of type **IntPredicate** and returns it back. The goal here is to pass a lambda expression to this method and get it back as an object. A lambda expression itself does not tell us what

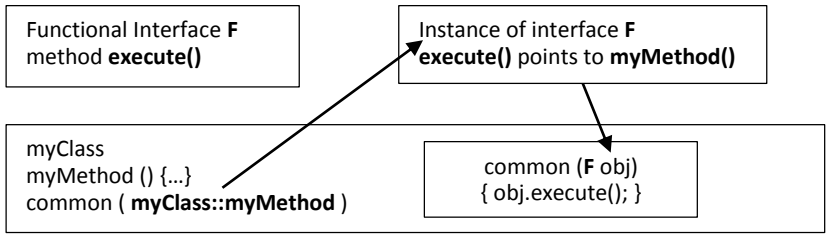
the functional interface it is used for, so we need to provide this information to the Java compiler either by directly assigning the lambda expression to a functional interface variable, or by passing the lambda expression as a parameter of a functional interface type. So, in this example, the lambda expression **n -> (n>min & n<max)** creates an object of the **IntPredicate** type, substitutes its **test** method with the **(n>min & n<max)**, and passed this object to the **filter** method. The **filter** method returns this object back, so the expression **filter(n->(n>min&n<max))** is in fact an object reference of type **IntPredicate**. Finally, we use this object reference to call the instance method **test** (which, at this moment, reflects the lambda expression):

```
filter(n -> (n > min & n < max)).test(i)
```

The outcome of the above program: "**out of range**".

Method Reference

The **method reference** feature allows you to create a reference to a method without executing the method. That reference then can be passed to another method for execution using the double colon "**::**" operator. The following diagram and corresponding program code illustrates the process:




```
interface F {
  void execute ();
}

class myClass {
  static void common(F obj) { obj.execute(); }

  void myMethod() {…}

  public static void main (String args[]) {
     common(myClass::myMethod);
  }
}
```

The **common()** method is defined to accept an object of the functional interface **F** and run its **execute()** method. At run time, the method reference operator (**::**) creates an instance of the functional interface **F** (because the **common()** method expects it) whose **execute()** method points to **method1()**. Then, the instance of **F** is passed to the **common()** method, and **method1()** is executed. Note that method passing as reference must have the same signature (return type and the types of parameters) as the method defined by the functional interface.

Below are some examples converting the string “Test Text” to upper or lower-case.

```
interface Transform {
  String doit (String s);
}

class myClass {


   static void printAs (Transform obj) {
      System.out.println( obj.doit(“Test Text”) );
   }

   static String upper (String s) {return s.toUpperCase();}

   String lower (String s) {return s.toLowerCase();}


   public static void main (String args[]) {

      Transform lowercase, uppercase;

      printAs (myClass :: upper);

      myClass mc = new myClass();
      printAs (mc :: lower);

      lowercase = mc :: lower;
      printAs (lowercase);

  }
}
```

The signature of method **doit()** is <String>*method*<String>
All methods passing as references must have this signature.

The **printAs()** method accepts an object of type **Transform** and executes its **doit()** method

The “**::**” operator creates a new **Transform** object with the **doit** method referencing the static method **upper**

The “**::**” operator creates a new **Transform** object with the **doit** method referencing the instance method **lower**

Constructor reference

Constructor reference is similar to method reference.

Let's say, the **Location** class holds two coordinates, x and y:

```
class Location {
   int x, y;
   Location () {x = y = 0;}
   Location (int x, int y) {
      this.x = x;
      this.y = y;
   }
}
```

Default constructor (without arguments)

Constructor with two arguments

We want to be able to build the Location objects not only "traditional way", with the **new** keyword, but also by calling a **create** method accepting two coordinates.

First, we declare a functional interface with the **create** method accepting two integers and returning an object of class Location:

```
interface myConstructor {
   Location create (int x, int y);
}
```

Then, during execution, we create an instance (i.e. object) of the functional interface with the reference to one of the **Location** class constructors:

```
myConstructor mc = Location :: new;
```

Will try to find a constructor of class **Location** with same argument list as in the **create()** method declared by **myConstructor** interface.
If found, the **create()** method will be referencing the found constructor.

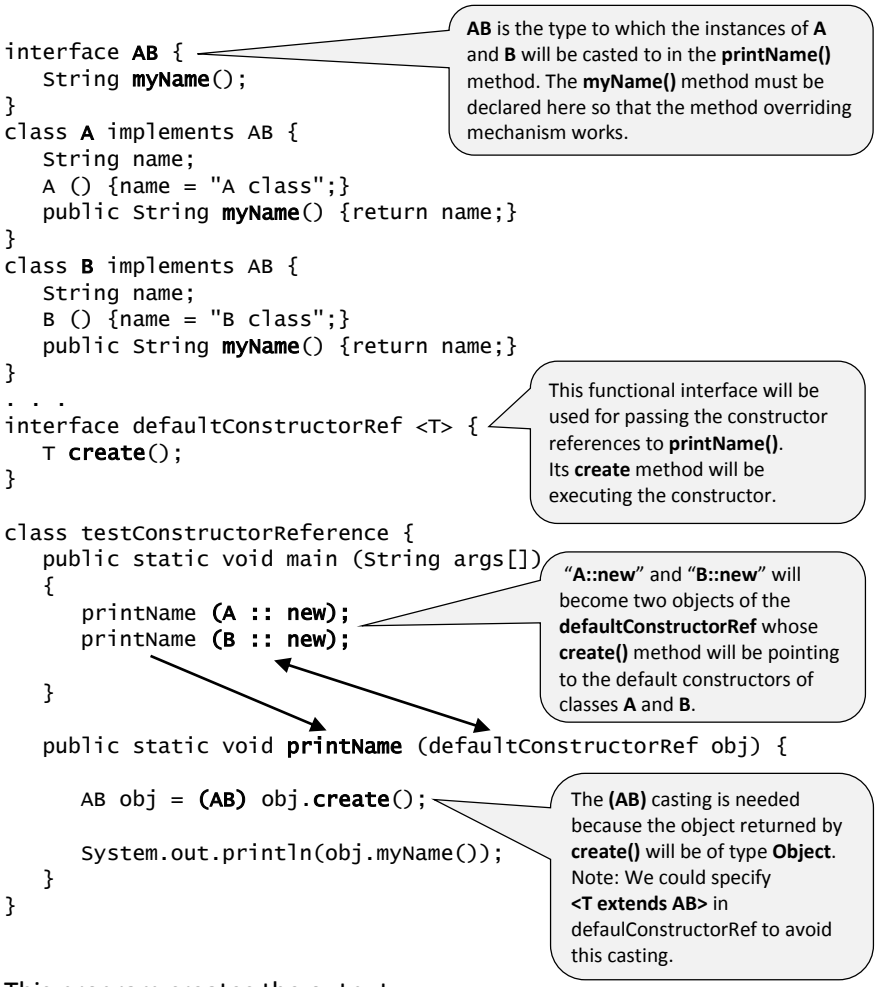
Finally, we use the instance method **create()** to invoke a proper constructor of the class **Location**:

```
Location a = mc.create( 7, 8 );
```

This example demonstrates the basics of creating and using constructor references. However, having two different ways of invoking the same constructor seems to be more confusing than beneficial.

So here is a more practical example - we will be passing default constructor references of two different classes (**A** and **B**) to the **printName()** method. This

method will use the received constructor reference to create an object and invoke its **myName** method.




```
interface AB {
   String myName();
}
class A implements AB {
   String name;
   A () {name = "A class";}
   public String myName() {return name;}
}
class B implements AB {
   String name;
   B () {name = "B class";}
   public String myName() {return name;}
}
. . .
interface defaultConstructorRef <T> {
   T create();
}

class testConstructorReference {
   public static void main (String args[])
   {
      printName (A :: new);
      printName (B :: new);

   }

   public static void printName (defaultConstructorRef obj) {

      AB obj = (AB) obj.create();

      System.out.println(obj.myName());
   }
}
```

This program creates the output:

```
A class
B class
```

Inner Classes

An **inner class** is a class declared within another class or within a method:

```
class classA {           // ← top level class (outer class)
   class classB {        // ← inner class
     classC {            // ← inner class
     }
   }
}

public void myMethod() {
   class classD {        // ← inner class (AKA local class)
   }
}
```

Static Inner (Nested) Classes

A **static inner class** is a class with the **static** modifier and declared within another class:

```
 class outerClass {
    static class innerClass {
    }
 }
```

A static inner class can access only static variables and methods of the outer class and has no association with the instances of the outer class. In this regard it behaves exactly like any other class declared outside of the outer class, so more appropriate name for such classes would be **static nested classes**.

Example of using static inner (nested) classes:

```
public class Outer {
   static String myName = "Outer";
   static int N = 0;

   static class Nested {
      static String myName = "Nested";
      int myN;
      Nested() { myN = ++N; }
   }
```

N will keep track of total created instances of the nested class

Constructor of the nested class; increments object's sequential number

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

      System.out.println(myName);

      System.out.println(Nested.myName);
      System.out.println(Outer.Nested.myName);

      Nested nested1 = new Nested();
      Outer.Nested nested2 = new Outer.Nested();

      System.out.println(nested1.myN);  // Prints: 1
      System.out.println(nested2.myN);  // Prints: 2
   }
}
```

This is how we can access static variables of the outer class

This is how we can access static variables of the nested class

This is how we can create objects of the nested class

Non-Static Inner Classes

A **non-static inner class** is a class declared within another class as an instance property (i.e. without the **static** modifier). An object of a non-static inner class has access to the instance variables and methods of the outer class object in which it was created. Objects of non-static inner classes can be created either within instance methods of the outer class, or with the **new** operator prefixed with the outer class object name:

```
public class Outer {
   int x = 0;

   class Inner { void innerMethod() { x++; }; }

   public static void main (String args[]) {
       Outer outer1 = new Outer();

       Inner inner1 = outer1.new Inner();
       Outer.Inner inner2 = outer1.new Inner();

       outer1.instanceMethod();
   }

   void instanceMethod() {
       Inner inner3 = new Inner();
       Outer.Inner inner4 = this.new Inner();
   }
}
```

Non-static inner class can access the instance variables (and methods) of the outer class

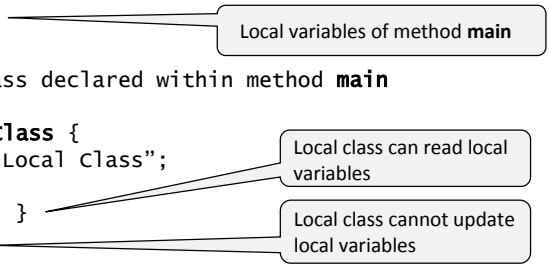
Valid methods of creating the inner class objects using outer object reference

Valid methods of creating inner class objects from within instance methods of outer class objects

Local Inner Classes

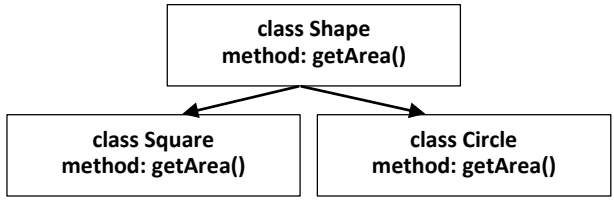
A class declared within a method is called a **local class**.

```
public static void main (String args[]) {
   int x = 1;
   int y = 2;

   // A local class declared within method main

   class myLocalClass {
     String s = "Local Class";
     int z;
     { z = x + y; }
     { x = 2; }
   }

   myLocalClass loc = new myLocalClass();

   System.out.println("z=" + loc.z); // prints: z=3
}
```



Abstract Classes

Abstract class is a class in which one or more methods are declared, but not defined (i.e. a without a body). To understand why we need abstract classes, consider this class hierarchy:



Class Shape declares the getArea method for calculating the area of a particular shape. This method is overridden with actual implementation in each of the subclasses – Square and Circle. If we create an object of class Shape, its getArea() method would have no meaning. Of course, we could make it to return a fake value, like zero, to indicate that this is not a true shape. A better approach, however, would be to altogether disallow the creation of class Shape objects. That's when **abstract classes** and **abstract methods** come in hand.

Example:

```
abstract class Shape {
   String myName;
   public abstract double getArea(double arg);
   public String getName() {return myName;}

   public static void main (String args[]) {

      Shape a = new Circle();

      Double area = a.getArea(1);

      System.out.println("area=" + area); // prints: "area=3.1415"
   }
}

class Circle extends Shape {
   public double getArea(double arg) {
        return (arg * arg * 3.1415);
   }
}
```

Abstract class cannot be instantiated

Abstract method cannot have a body

Abstract class can declare non-abstract methods

A subclass of an abstract class **must override all abstract methods** with "real" ones, otherwise the subclass will become abstract too.

Anonymous Classes

An **anonymous class** is a class declared within an expression. The definition of the anonymous class begins with the keyword **new** and is enclosed within round brackets:

(new *className*() {*body of the class*})

This is equivalent to creating an instance of some unnamed class extending the ***className***. Being an instance of a class, the anonymous class definition can be used in various expressions:

```
someVariable = (anonymous class definition).method();
someMethod (anonymous class definition);
```

Consider a situation when we have class Circle calculating the area of a circle with this formula: **area = 3.14** * **radius * radius**. We want to keep the existing class Circle, but increase the precision of this calculation by using a new formula: **area = 3.1415926** * **radius * radius**.

Here is how we can accomplish this with an anonymous class:

```
class Circle {
   public double getArea (double r) {return 3.14 * r * r;};
}

class testClass {
  public static void main (String args[])

     Circle a = new Circle();
     System.out.println(a.getArea(2)); //prints: 12.56


     a = ( new Circle() {
                 public double getArea (double r)
                 {return 3.1415926 * r * r;}
               }
         );

     System.out.println(a.getArea(2)); // prints: 12.5663704
  }
}
```

Original formula from class **Circle** is used

The anonymous class extends class **Circle** and overrides the **getArea()**

New formula is used because **a** is now referencing the anonymous class

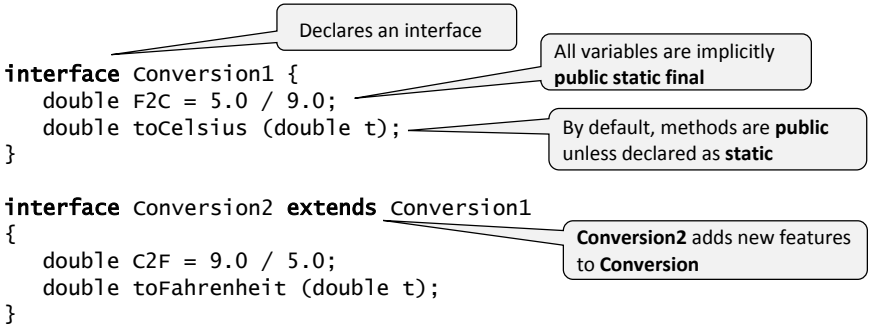
Interfaces

An **interface** is a collection of constants, abstract methods, and default methods. The main purpose of interfaces is to supply useful constants and methods to classes, and also to expand the polymorphism mechanism beyond the class hierarchies.

An interface can extend one or more other interfaces. Interfaces cannot be instantiated (you cannot create an object of an interface), but they can be implemented by a class. The class implementing an interface must override all abstract methods of the interface, otherwise the class will be abstract.

The example below demonstrates the common technique of working with interfaces.

First, we declare two interfaces: Conversion1 and Conversion2. The Conversion2 interface extends the Conversion1 and adds one constant and one method.




```
interface Conversion1 {
   double F2C = 5.0 / 9.0;
   double toCelsius (double t);
}

interface Conversion2 extends Conversion1
{
   double C2F = 9.0 / 5.0;
   double toFahrenheit (double t);
}
```

Next, we declare two classes implementing the Conversion2 interface. Both classes will have access to the constants declared in Conversion1 and Conversion2 interfaces, and they must also implement all methods declared in those interfaces.

```
// Approximate temperature conversions
class myClass1 implement Conversion2 {
   public double toFahrenheit(double t) {return t * 2 + 32;}
   public double toCelsius(double t)    {return (t - 32 ) / 2;}
}
```

```
// Precise temperature conversions
class myClass2 implements Conversion2
{
   public double toFahrenheit(double t) {return t * C2F + 32;}
   public double toCelsius(double t)    {return (t - 32 ) * F2C;}
```

Finally, we create two objects of classes myClass1 and myClass2 and execute their instance methods **toFahrenheit()** and **toCelcius()**.

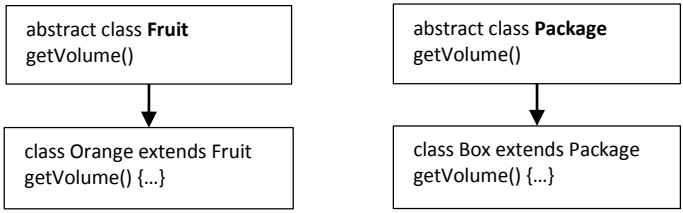
```
   public static void main (String args[])
   {
      Conversion2 obj;

      obj = new myClass1();
      System.out.println(obj.toFahrenheit(25)); // prints: 82.0
      System.out.println(obj.toCelsius(77));    // prints: 22.5

      obj = new myClass2();
      System.out.println(obj.toFahrenheit(25)); // prints: 77.0
      System.out.println(obj.toCelsius(77));    // prints: 25.0
   }
}
```

Note that same variable "**obj"** of type **Conversion2** can be used to reference objects of classes **myClass1** and **myClass2**. This technique employs the polymorphism mechanism and instructs Java to find correct instance methods dynamically, at run time, based on the actual type of the object referenced by "**obj**".

Interfaces vs. Abstract Classes

You might have noticed already that an interface and abstract classes are similar. Both can declare constants and provide method templates for further implementation by other classes. So the question is – what can we do with interfaces that cannot be done with classes?

Let's take a look at an example:

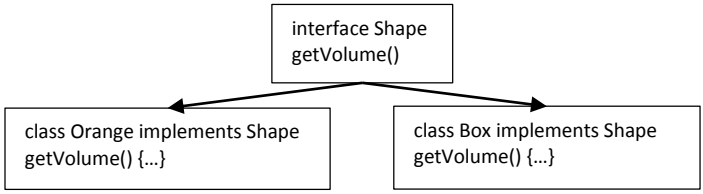


The getVolume() method of the Orange and Box objects can be accessed as follows:

```
Orange  orange = new Orange(); orange.getVolume();
Fruit   fruit  = new Orange(); fruit.getVolume();

Box     box    = new Box();    box.getVolume();
Package pack   = new Box();    pack.getVolume();
```

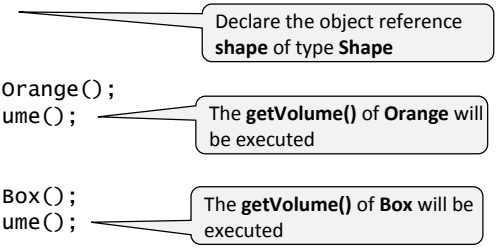
Since the Orange and Box classes belong to different class hierarchies, it's not possible to create an object reference that can be used for referencing both objects.

Interfaces allow us to do that because the class hierarchy and the interface hierarchy are unrelated. Interfaces break the class hierarchy boundaries. Otherwise unrelated classes can implement the same interface, and a variable of that interface type can reference the objects of both classes.



Example: using a single interface variable to reference class objects in different class hierarchies.

```
interface Shape {
   double getVolume();
}

class Orange  implements Shape {
   double getVolume() { implementation code };
}
class Box     implements Shape {
   double getVolume() { implementation code };
}

. . .

Shape shape;

shape = new Orange();
shape.getVolume();

shape = new Box();
shape.getVolume();
```



Note: the polymorphism mechanism is employed to find and execute the **getVolume()** method of the actual object referenced by the **shape** variable.

Default and Static Methods in Interfaces

Sometimes it might be beneficial to have interfaces with fully implemented methods (i.e. methods with a body). This can be accomplished in two ways: by declaring **default methods** or **static methods**.

A **default method** is an instance method that has a body and is prefixed with the **default** modifier. Default methods can be, but are not required to be, overridden in the classes implementing the interface.

```
interface Shape {
   default double getVolume() {return 1.0;}
}
```

A **static method** is a class-level method that has a body and is prefixed with the **static** modifier. Static methods can be, but are not required to be overridden in classes implementing the interface.

```
interface Shape {
   static String getShapeName(Object obj)
   { return obj.getClass().getName(); }
}
```

Example:

```
class Orange implements Shape {

   public static void main (String args[]) {
     double volume;
     String name;

     Shape obj = new Orange();


name = Shape.getShapeName(obj);

     volume =  obj.getVolume();



     System.out.println("Shape="+ name);    // prints: "Shape=Orange"
     System.out.println("Volume="+ volume); // prints: "Volume=0"
   }
}
```

Create an object of the Orange class

Use the **static** method of interface **Shape** to get the name of the object referenced by **obj**

Use the **default** instance method getVolume() of interface **Shape**

Exceptions

An **exception** is an interruption of normal program flow due to an error. Some errors are fatal and cannot be recovered from, while others are not critical and can be handled by the application (e.g. file not found situation). Java recognizes two types of exceptions: **checked exceptions** and **unchecked exceptions**.

Unchecked exceptions are not anticipated**.** For example, a JVM failure or division by zero are not expected situations. Usually these errors are due to issues with the Java runtime environment or bugs in the application code. The way to deal with unchecked exceptions is to fix the environment issues (if any) or get rid of bugs in your application (debugging).

> **Unchecked exceptions** are not anticipated and not required to be processed by the application.

Checked exceptions are explicitly declared in some Java classes or in user-defined classes and could be **thrown** (i.e. created) under some circumstances. For example, the **FileReader** class will throw the **FileNotFoundException** if the file you're trying to open cannot be found. A checked exception must be caught and processed in the method where the exception occurs, or it can be passed through (propagated) to the calling method (if any), all the way to the JVM.

> **Checked exceptions** are declared in Java or user-defined classes and could be thrown under some circumstances.
> **Checked exceptions** are anticipated and must be processed or passed through by the application.

Java provides a special mechanism for handling exceptions. When an error situation occurs during execution of an application, Java interrupts the application and triggers the exception handling mechanism. This process is referred to as **throwing an exception**. After that, the exception can be handled either by the application, or by the JVM. The details of the whole process are described in the next section.

Handling Exceptions

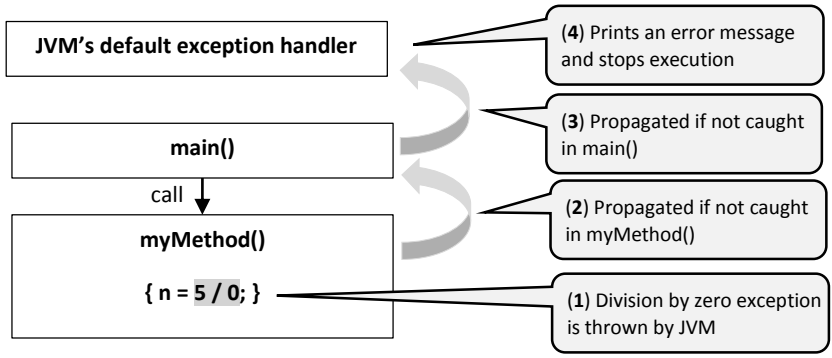
Java's exception handling mechanism includes a built-in **default exception handler** and special language constructs for catching or throwing the exceptions. The default exception handler is executed when your program does not process the exception; it prints out some diagnostic information and terminates the program. A program can build its own custom code for handling specific exceptions by utilizing these five commands: **try**, **catch**, **finally**, **throw**, and **throws**.

As we mentioned before, exceptions (i.e. error situations) could be anticipated (**checked exceptions**) or unanticipated (**unchecked exceptions**). Checked exceptions are explicitly declared in some Java classes (or in user-defined classes or methods) and must be handled by methods that use those classes or methods. Unchecked exceptions are usually unexpected (e.g. division by zero) and could be left unhandled. Java supports a predefined set of unchecked exceptions. The similarities and differences in handling of checked and unchecked exceptions are explained below.

Handling Unchecked Exceptions

When an unexpected error occurs, Java creates an object of appropriate **unchecked exception** type that contains information about the exception, and passes that object to the application. If the current method does not have logic for catching the exception, Java propagates (passes) it to the calling method if one exists, or to the JVM.

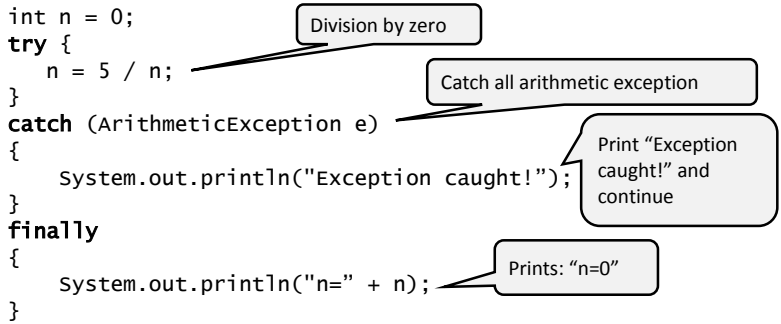
This diagram illustrates how the JVM handles unchecked exceptions:



To handle unchecked exceptions in your application, you can use the following constructs:

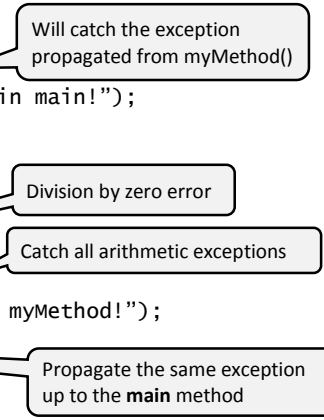
```
try {
   code you want to monitor for possible exceptions
}
catch (exceptionType1 e) {
   code processing the exception of type exceptionType1
}
. . .
catch (exceptionTypeN e) {
   code processing the exception of type exceptionTypeN
}
finally {
   code to be executed regardless of the results of try or catch
blocks
}
```

Example of handling **unchecked** (not anticipated) exceptions:



Note that once an exception is caught, the propagation process stops and normal program execution resumes. It is possible, however, to pass an exception to the calling method even after the exception was caught and processed. This is done by using the **throw** statement:

```
// Explicit propagation of unchecked exceptions

public static void main (String args[]) {
   try {
      myMethod();
   }
   catch (ArithmeticException e) {
        System.out.println("Exception in main!”);
   }
}

public static void myMethod () {
   int n = 0;
   try { n = 5 / n; }
   catch (ArithmeticException e) {
      System.out.println("Exception in myMethod!”);
      throw e;
   }
}
```



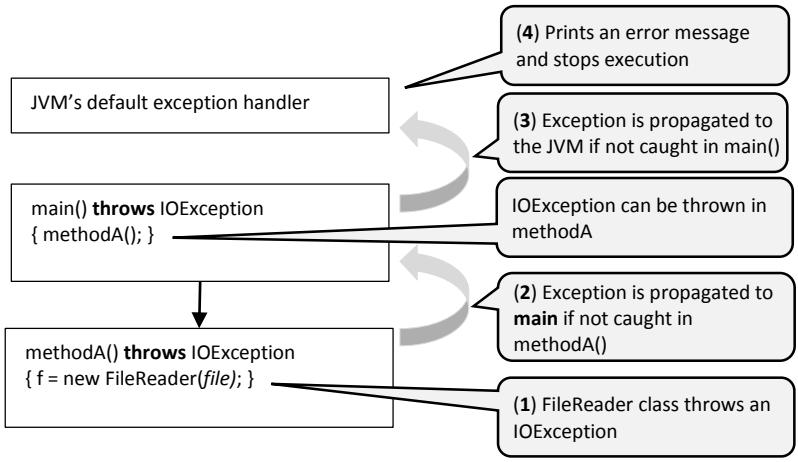
The above code will produce this output:

```
Exception in myMethod!
Exception in main!
```

Handling Checked Exceptions

Checked exceptions are exceptions explicitly declared and thrown by some Java classes or by user programs. If your program uses a Java class or calls a method that could throw a checked exception, you must catch that exception or explicitly propagate it up to the calling method.

This diagram illustrates how the checked exceptions should be processed:



Note that this method must be declared with the **throws** ***someException*** clause if it can create and throw its own exception or if it catches a checked exception and then re-throws it with the **throw** ***someException*** statement.

Example:

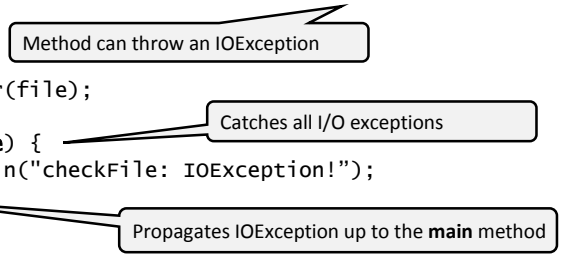
```
// Explicit propagation of checked exceptions
import java.io.*;
class myClass {
public static void main (String args[]) {
   String file = "not_existing_file";
   try {
      checkFile(file);
   }

   catch (IOException e) {
        System.out.println("main: checkFile failed!");
   }
}
```

FileReader class resides in java.io

Can throw an IOException

Will catch an IOException propagated from checkFile

```
 public static void checkFile (String file) throws IOException {
   FileReader f;
   try {
     f = new FileReader(file);
   }
   catch (IOException e) {
      System.out.println("checkFile: IOException!");
      throw e;
   }
 }
}
```



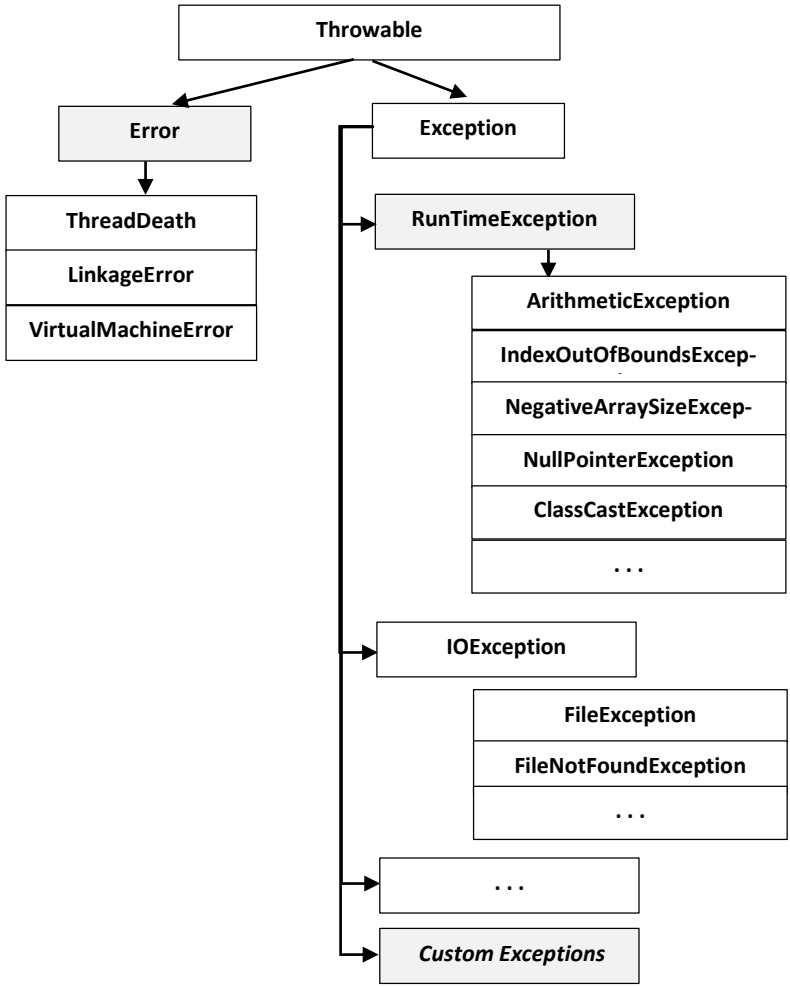
The above code produces the output:

```
checkFile: IOException!
main: checkFile failed!
```

In addition to the pre-defined set of checked and unchecked exceptions provided by Java, you can build your own exceptions, similar to **checked exceptions**. To understand how it's done, we will first review the exceptions classes.

Exceptions Class Hierarchy

For every exception, Java creates an object containing some useful information about the exception. Each type of exception is represented by its own class – FileNotFoundException, IndexOutOfBoundsException, etc. All these classes are sub-classes of the top class **Throwable**. The diagram below shows the hierarchy of the exceptions classes:



All **Error** and **RunTimeExceptions** are **unchecked** exceptions and are not required to be caught and handled. The rest are **checked** exceptions that must be handled by application logic if they can be thrown during execution.

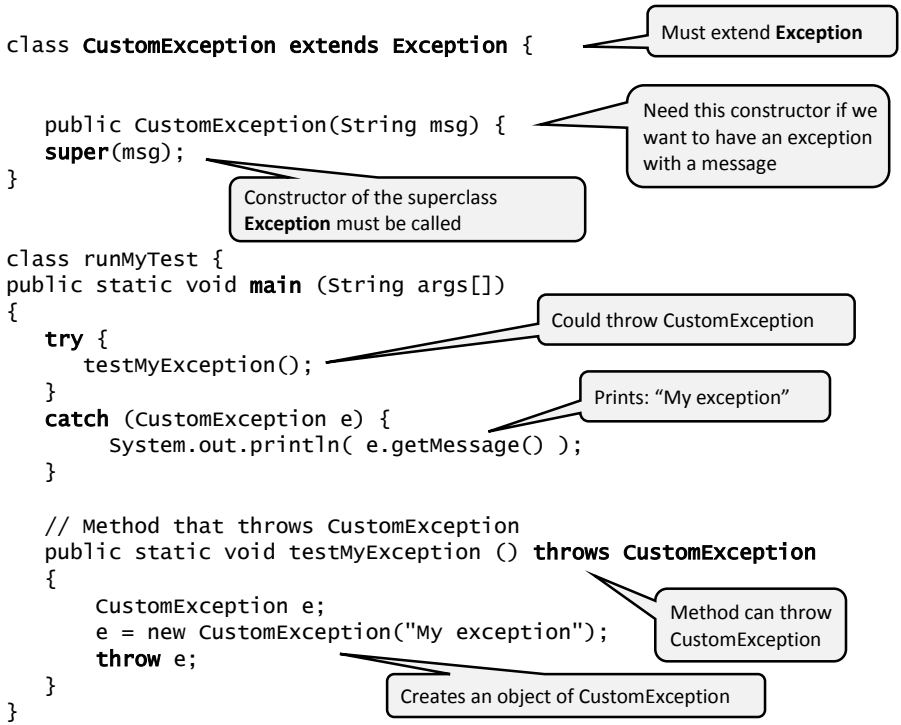
Class **Throwable** provides several useful methods that can be used in the exception handling logic of your program. Here are some of them:

String getMessage()	Returns the detailed message string for this exception
void printStackTrace()	Sends the above message and the trace information to the standard error stream
String toString()	Returns a short description of this exception

Now, that we know how the exceptions classes are organized, let's take a look at how to create a custom one.

Creating Custom Exceptions

A custom, user-defined exception can be built as a sub-class of the **Exception** class as follows:

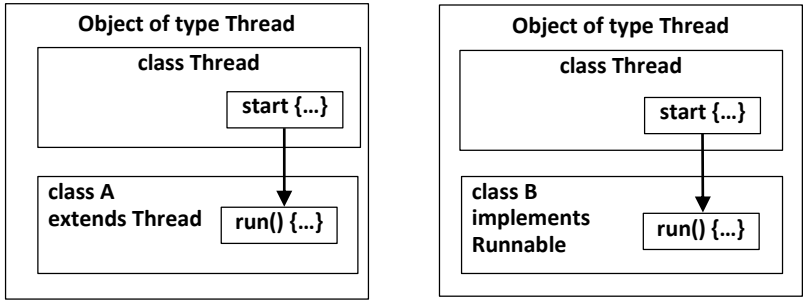


Threads

When two or more parts of a program are running concurrently, such program is called a **multithreaded** program. Each path of execution is called a **thread**. Every Java program has at least one thread that starts with the **main** method.

Starting Threads

An application program can create as many threads as needed. To be able to create a thread, a class needs either to extend the **Thread** class, or implement the **Runnable** interface. In both cases, the class creating a thread must implement the **run()** method. The below diagram illustrates the two ways of creating threads. Note that in both cases an object of type **Thread** must be present and have access to the **run()** method of your class:



As the diagram shows, a thread is started by the **start()** method of class **Thread** that calls the **run()** method defined in the class starting the thread.

In this example a new thread is started by an object of class **myClass** which extends the **Thread** class:

```
class myClass extends Thread {
   public void run() {…}
   public static void main (String args[]) {
      myClass th1 = new myClass();
      th1.start();
   }
}
```

When your class extends the **Thread** class, the only method that you have to implement is the **run()** method. This method should define the processing

that will be done by the new thread. Note, that the **run()** method defined by the class **Thread** is empty and does nothing.

You can also implement (i.e. override) other methods of class **Thread**, but with one exception – the **start()** method. If you implement the **start()** method, and even if it calls the **run()** method, no threads will be started, **run()** will be executed as a regular method of your class, within the current thread.

This method of creating threads is completely acceptable, but if you are not intending to override other methods of class **Thread** besides the **run()** method, a better approach would be to implement the **Runnable** interface.

In this example a new thread is created by an object of class **myClass** implementing the **Runnable** interface:

```
class myClass implements Runnable {
   public void run() {…}
   public static void main (String args[]) {
      Thread th2 = new Thread( new myClass() );
      th2.start();
   }
}
```

The Runnable interface declares only one method – **run()**, so your class needs to implement just that method. To start a new thread, the program has to create an object of class **Thread** using the following constructor:

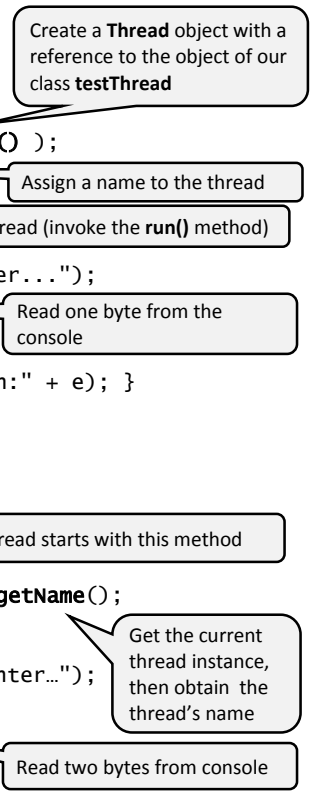
```
Thread (Runnable target)
```

Here, *target* is an instance of your class.

It's important to reiterate that a thread can be started only by the **start()** method of a **Thread** object. In the first example, when a sub-class of Thread was instantiated, an object of class Thread was created automatically by calling the **super()** constructor. In the second example, we created a **Thread** object explicitly, and at the same time passed an object of the class **myClass** to it.

Here is a sample program that starts one thread.

```
import java.io.IOException;

class testThread implements Runnable
{
   public static void main (String args[])
   {
     Thread th = new Thread( new testThread() );
     th.setName("Thread 1");
     th.start();

     try {
          System.out.println("main: hit enter...");
          System.in.read();
        }
     catch (Exception e)
        { System.out.println("main exception:" + e); }

     System.out.println("main ended");
   }


   public void run()
   {
      String name = Thread.currentThread().getName();

      try {
          System.out.println("thread: hit enter…");
          System.in.read();
          System.in.read();
      }
      catch (Exception e) {
        System.out.println("thread exception:" + e);
      }

      System.out.println(name + " thread ended");
   }
}
```



The above program will prompt you to hit Enter twice. The first request will come from the thread, the second – from the main method. You should hit Enter twice to first stop the execution of the main method, then the thread execution.

Daemon Threads

When you execute the previous program, you will notice that the **main** thread ends once you hit Enter, but the thread will continue running until you hit Enter again. Threads that continue execution after the parent thread dies are called **user threads**. Another type of threads is called **daemon threads**. A daemon thread dies once its parent thread dies. By default, a newly created thread is a user thread; to make it a daemon, you need to mark it as such by calling **setDaemon(true)**. Make sure it is done before you start the thread:

```
Thread th = new Thread( new myClass() );
th.setDaemon(true);
th.start();
```

Interrupting Thread Execution

A thread can be interrupted by another thread via the instance method **interrupt()**:

```
Thread th = new Thread( new myClass() );
th.start();
. . .
th.interrupt();
```

The interrupt method sets a flag in the thread object, which has to be checked in the **run()** method to have any effect.

```
public void run() {
   . . .
   if (Thread.interrupted() )
      { System.out.println("I was interrupted"); }
}
```

Note that **interrupt()** is an instance method, and **interrupted()** is a static method.

Neither of these methods stop tread execution. They merely allow to "send a signal" to the thread, but how to process this "signal" is up to the thread being interrupted.

Waiting on a Thread to Die

If the main thread depends on the execution of its child threads, it can use the **join()** method of class **Thread** to "connect" to the child thread and wait until it finishes.

```
Thread th = new Thread( new myClass() );
th.start();

try  {
      th.join(1000);
      th.join();
}
catch (InterruptedException e) {
      System.out.println("Exception: " + e);
}
```

Wait 1sec for thread **th** to die

Wait forever for thread **th** to die

This exception can be thrown by the join() method

Synchronization

In multithreaded programming, it is often critical to restrict the access to shared resources to only one thread at a time. Java offers a resource locking mechanism that can be employed in two ways – through **synchronized methods** or **synchronized statement blocks**. Both approaches will be discussed in few moments.

First, to understand Java's synchronization and locking mechanism we need to know two key terms: the **monitor** and the **lock**. When a thread acquires a lock, it is said to have entered the monitor. This does not add much to the understanding of the topic itself, but is important because these terms are used in all Java manuals.

Synchronized Methods

You can declare any method of your class as synchronized:

```
synchronized static public void method1() {…}
synchronized public void method2() {…}
```

By doing so, you activate the locking mechanism either on the class level, or on the object level, or both.

The **object-level synchronization** works as follows:

All synchronized instance methods of an object are controlled by the same lock. Only one synchronized instance method of that object can be executed at any point in time.

The **class-level synchronization** works as follows:

All synchronized static methods of a class are controlled by the same lock. Only one synchronized static method of the class can be executed at any point in time.

Note that object-level synchronization and class-level synchronization work independently of each other.

Synchronized Statement Blocks

There might be situations when you cannot use the synchronized methods such as when you're not allowed to modify the class. In such circumstances you can use a **synchronized statement block**. Here is how a synchronized statement block is defined:

```
synchronized ( object ) { statement block }
```

This statement can be used anywhere in the thread execution path (i.e. in the **run()** methods or any other methods called by it). The *statement block* will be locked on the *object* and will be accessible only by one thread at a time.

An important note must be made regarding the *object* used in the above construct. The *object* could be an object of any class, even a dummy object created just for the purpose of being a lock. As long as all threads have access to the same object – synchronization on the statement block will work.

The example below demonstrates the usage of statement block synchronization.

Example: only one thread at a time can execute the synchronized block

```
import java.io.IOException;

class testSyncBlock implements Runnable {

   Object lock;

   public static void main (String args[])
   {
     testSyncBlock mainThread = new testSyncBlock();

     mainThread.lock = new Object();

     Thread th1 = new Thread(mainThread, "thread1");
     Thread th2 = new Thread(mainThread, "thread2");
     th1.start();
     th2.start();

     try {
          th1.join();
          th2.join();
     }
     catch (InterruptedException e) { System.out.println(e); }
   }


   public void run()
   {
      String name = Thread.currentThread().getName();
      System.out.println(name + " started");


      synchronized (lock)
      {
         try
         {
           System.out.println("hit enter...");
           System.in.read();
         }
         catch (Exception e) { System.out.println(e); }
      }
   }
}
```

Declare the instance reference **lock** of type Object

The **lock** object will be used only for synchronization

Create and start two threads

Wait for both threads to finish

Synchronizing on the object **lock**

Beginning of the synchronized block

End of the synchronized block

Advanced Inter-Thread Communication

In most cases synchronized methods and synchronized statement blocks are sufficient for establishing safe access to shared resources. However, there might be a need for more granular control over the synchronization process. The following three methods defined by the Object class allow two or more threads to communicate with each other:

`wait()`, `notify()`, `notifyAll()`

The **wait()** method releases the lock held by the current thread and puts the thread in a waiting state.

The **notify()** method wakes up one of the waiting threads (if any). The choice of which thread to wake up is arbitrary (i.e. a random thread will be chosen for you by the JVM).

The **notifyAll()** method wakes up all threads waiting on the lock held by the current thread.

The above methods must be used within either synchronized methods or synchronized statement blocks.

The example below illustrates how communication between threads could be established. The program creates two threads reading the same array of strings, one element at a time. Once the next element of the array is processed by one thread, the thread gives control to another thread and goes to sleep.

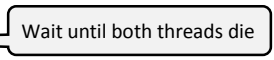
```
import java.io.IOException;

class testNotify implements Runnable {
   Object lock = new Object();
   String[] s = {"one", "two", "three", "four"};
   int count = 0;

   public static void main (String args[])
   {
     testNotify me = new testNotify();

     Thread th1 = new Thread(me, "thread1");
     Thread th2 = new Thread(me, "thread2");
     th1.start();
     th2.start();
```

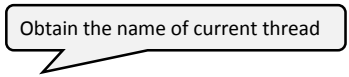
The **lock** object will be used for synchronization

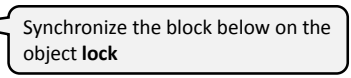
Create two threads with the reference to object **me**

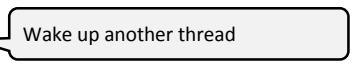
Start both threads; the **run()** method of object **me** will be called

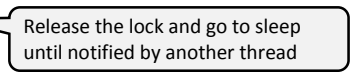
```
      try {
           th1.join();
           th2.join();
      }
      catch (InterruptedException e) { System.out.println(e); }
   }

   public void run()
   {
      String name = Thread.currentThread().getName();

      synchronized (lock)
      {
         try
         {
            while (count < s.length)
            {
               System.out.println(name + ": " + s[count++]);

               lock.notify();

               lock.wait();
            }
         }
         catch (Exception e) { System.out.println(e); }

         lock.notifyAll();
      }
   }
}
```

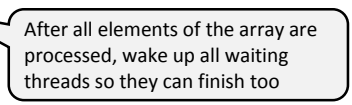












Note that the **wait()**, **notify()**, and **notifyAll()** methods must be invoked by the object on which the current thread is locked – object **lock** in this example. Otherwise, the **IllegalMonitorStateException** would be thrown.

The output of the above program is this:

```
thread1: one
thread2: two
thread1: three
thread2: four
```

Collections Framework

The **Collections Framework** is a set of interfaces and classes that provide a standardized approach for managing groups of objects. The cornerstone of the whole concept is the **collection**, a generic term for a set of objects grouped together by some means. Based on the way collections are built and accessed, all collections can be divided into three major categories:

LIST - a sequence of objects; examples: arrays, sorted lists, queues, stacks.

SET - a group of unique objects; no duplicates allowed.

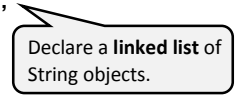
MAP - a group of key/value pairs of objects.

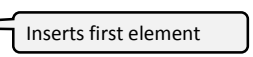
Each group can be divided further into sub-groups. For example, a **list** could be sorted, linked, organized as a stack, etc. However, regardless of the differences between collections, most of the operations upon them are performed in a similar manner, through standardized interfaces. This chapter provides an overview of such operations.

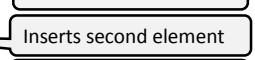
Creating Collections

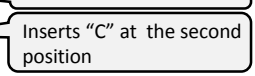
Most of the collections use the **add()** method of the corresponding class for putting new elements into collections:

```
// Create a LinkedList collection
LinkedList<String> list = new LinkedList<String>();
list.add("A");
list.add("B");
list.add(1, "C");
System.out.println(list);   // prints: [A, C, B]
```









```
// Create a TreeSet collection

TreeSet<Integer> intset = new TreeSet<>();
intset.add(3);
intset.add(5);
intset.add(1);
System.out.println(intset); // prints: [1, 3, 5]
```

Declare a **sorted set** of Integer objects.

Some collection classes (Vector, Stack, PriorityQueue, etc.), as well as all the map classes do not use the **add()** method for adding elements to the collection. For example, the **Vector** class, uses the **addElement()** method, the **PriorityQueue** class uses the **push()** method, and the map classes use the **put()** method:

```
TreeMap<Integer,String> map = new TreeMap<>();

map.put (2,"two");
map.put (1,"ten");
map.put (3,"three");
map.put (1,"one");

System.out.println(map); // prints: {1=one, 2=two, 3=three}
```

Declares a sorted Key/Value map of the Integer/String type.

The **1/one** pair will override the previous **1/ten** entry because the keys must be unique.

Retrieving Collections' Elements

The most common way of retrieving an element from a collection is via the **get()** method or its variations (e.g., **getFirst()**, **getLast()**):

```
LinkedList<String> list = new LinkedList<String>();
list.add("A");
list.add("B");
list.add(0, "C");

System.out.println(list.get(1);)        // prints: "A"
System.out.println(list.getFirst();)    // prints: "C"
System.out.println(list.getLast();)     // prints: "B"

TreeMap<String,String> map = new TreeMap<>();
map.put ("key1","A");
map.put ("key2","B");

System.out.println(map.get("key1");)    // prints: "A"
System.out.println(map.get("key3");)    // prints: "null"
```

Retrieves the second element of the list

Retrieves the value associated with the "**key1**"

Updating Collections

Every collection defines a set of methods allowing you to make changes to already created collections by removing or updating their elements. These methods can differ among the collection classes. The only method that is common to all collections is **remove()**. Below are a few examples of performing update operations on different collections.

```
TreeMap<String,String> map = new TreeMap<>();
map.put ("key1","A");
map.put ("key2","B");
map.put ("key3","C");

map.remove ("key1","A");


map.replace ("key2","D");
System.out.println(map); // prints: {key2=D, key3=C}
```

Removes the "**key1/A**" pair from the map

Replaces the "**key2/B**" with the "**key2/D**"

```
Vector<String> v = new Vector<>();
v.addElement ("A");
v.addElement ("B");
v.addElement ("C");


v.removeElementAt (1);
System.out.println(v);      // prints: [A, C]

v.removeAllElements ();
```

Removes the element at position 1 (i.e. second element)

Removes all elements from this vector and sets its size to zero

Iterating through Collections

There are situations when you need to access a collection sequentially, reading its elements one by one, in one or both directions. There are two ways to accomplish this: by using the **for-each loop**, and by using **iterators** - the **Iterator**, **ListIterator**, and **Spliterator** classes. However, a collection must implement the **Iterable** interface in order to use either of these methods. The map classes, for example, do not implement the Iterable interface, so you have to obtain a collection-view of a map using the **entrySet()** method (i.e. convert the map into a **SET-type** collection) in order to work with the map as a "real" collection.

The "for-each" Loop

The first way of cycling through a collection is by use of the **for-each** loop. This is the easiest way to access the elements of a collection sequentially, one by one, in one direction.

Example:

```
// Iterating through list

LinkedList<String> list = new LinkedList<String>();
list.add("A");
list.add("B");
list.add("C");

for (String s : list)
{
     System.out.println(s); // Prints each element of the list
}

// Iterating through a map
TreeMap<String,String> map = new TreeMap<>();

map.put ("key1","A");

Set<Map.Entry<String, String>> mapset = map.entrySet();

for ( Map.Entry<String, String> m : mapset)
{
     System.out.println(m);            // prints: key1=A
     System.out.println(m.getKey());   // prints: key1
     System.out.println(m.getValue()); // prints: A
}
```

Iterates through the **list**. Each element is assigned to **s**, one by one.

Puts one key/value pair into the map

Creates a set-type collection from the map elements

Cycles through the **mapset**. Each map entry element is assigned to **m**.

Let's take a closer look at the **mapset** declaration. The instance method **entrySet()** transforms the TreeMap **map** into the Set-type collection **mapset** of type **Set<Map.Entry<String,String>>**. **Set** indicates that this is a Set collection; **Map.Entry** is the type of objects stored in the collection; and **<String,String>** specifies the type of key/value pairs stored in the **Map.Entry** objects.

By explicitly declaring the type of objects stored in the collection as **Map.Entry**, we also gain access to all methods declared in the **Map.Entry** interface. In this example we used the methods **getKey()** and **getValue()**.

The **mapset** can also be defined as follows:

```
Set mapset = map.entrySet();
```

Though, in this case, we would know only that the **mapset** collection contains objects of type **Object**. We could use them as is or cast them to the proper type to perform any meaningful operations on the underlying map elements.

Note that the **for** statement must explicitly specify the type of objects fetched from a collection:

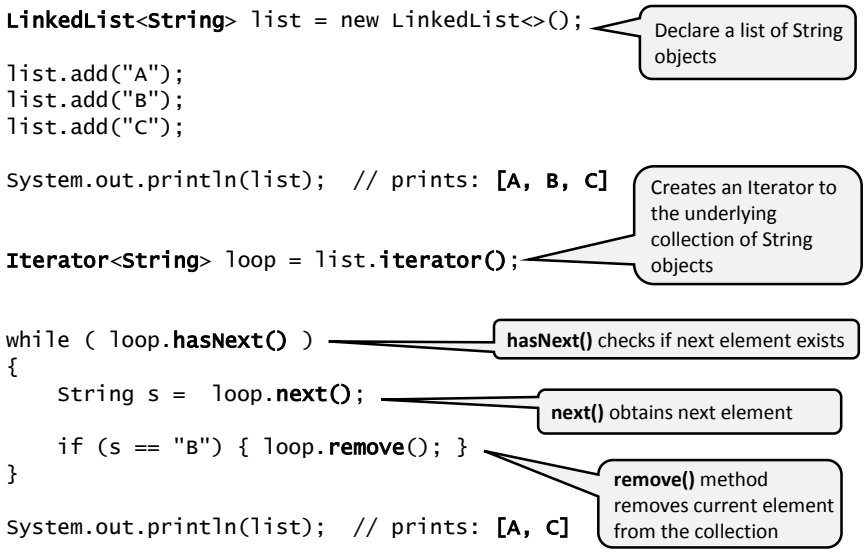
```
for ( Map.Entry<String, String> m : mapset)
```

Iterator

The **Iterator** interface is used to provide sequential access to the elements of collections, and for removing elements from collections.

In order to use an iterator, a collection class must implement the **Iterable** interface and define the **iterator()** method returning an **Iterator** object for this collection.

Example:



The above code removes the string “B” from the LinkedList **list**.

List Iterator

The **ListIterator** interface extends the functionality of **Iterator** by providing methods allowing to traverse **LIST-type** collections in both directions and modify the list during iteration.

Example:

```
// Iterate through a LinkedList collection backwards

LinkedList<String> list = new LinkedList<>();

list.add("A");
list.add("B");
list.add("C");
System.out.println(list);  // prints: [A, B, C]

ListIterator<String> loop = list.listIterator( list.size() );

// Iterate through the list backwards

while ( loop.hasPrevious() )

{
   String s =  loop.previous();

   loop.set(s + "1");
}

System.out.println(list);  // prints: [A1, B1, C1]
```

List of String objects

Create a **ListIterator** to the underlying list of String objects

The iterator will be positioned at the last element of the list

The **hasPrevious()** method checks if the previous element exists

Retrieves the previous element from the list

Updates the last retrieved element: append it with '**1**'

Spliterator

The newest type of iterator added in JDK 8 is the **Spliterator** interface. It was designed for traversing collections or other sources of data elements and optionally partitioning them for parallel processing. We said "collections or other sources elements of data elements" because the **Spliterator** can be created not only for collections, but also for arrays, I/O channels, or generator functions.

Note that **Spliterator** consumes the source. A new **Spliterator** must be created once all the elements of the source are processed.

In the below example we create a spliterator for an array of integers, split it in two, and calculate the sum of all values stored in the array.

```
import java.util.*;

class testSpliterator {
  static int total = 0;

  public static void sum (Integer i) { total += i; );  }

  public static void main (String args[]) {

    Integer[] integers = {1, 2, 3, 4};

    Spliterator<Integer> sp1 = Arrays.spliterator(integers);

    Spliterator<Integer> sp2 = sp1.trySplit();

    while (sp1.tryAdvance (i -> total += i));

    if (sp2 != null) {
       while (sp2.tryAdvance (i -> sum(i) ));
    }

    System.out.println("Total=" + total); // prints: Total=10
  }
}
```

Creates a Spliterator for the array of integers

Creates another Spliterator by dividing the source in half

Cycle through the first Spliterator updating the **total**

Cycle through the second Spliterator updating the **total**

Pay attention to the **tryAdvance()** method. It checks if the next element of the source is present and, if not – returns the boolean **false**. Otherwise it executes the **accept()** method of the functional interface **Consumer** and returns the **true**. We substitute the **accept()** method via two different lambda expressions: "**i -> total += i**" and "**i -> sum(i)**".

Note that the **while** statements have no bodies (however, they could) because all the processing is done though the lambda expressions.

Comparator

Any ordered collection (e.g. **TreeSet**) needs to have some kind of compare algorithm to determine which element of the collection is "greater" than the other. In fact, there are default algorithms for each type of objects that can be stored in ordered collections. These algorithms are implemented via the **Comparator** functional interface that defines the **compare()** method. In case we want to alter the default sequence of objects in a collection, we can build a collection that points to a custom comparator.

Note that the Comparator can be used to provide an ordering for collections of objects that don't have a natural ordering.

The example below shows how to create a custom comparator to sort an array of integers into descending order.

This **reverse()** method will be used to substitute the **compare()** method of the functional interface **Comparator**.

```
import java.util.*;
class testComparator {
  public static int reverse (Integer n1, Integer n2) {
    return n2 - n1;
  }

  public static void main (String args[]) {
    Integer[] nn = { 1, 2, 3 };
    Arrays.sort(nn, (n1, n2) -> reverse (n1, n2) );
    System.out.println(nn[0] + "," + nn[1] + "," + nn[2]);
  }
}
```

The **reverse(n1,n2)** substitutes the **compare()** method of the functional interface **Comparator**.

The output of this program will be "**3,2,1**".

Note: The **Arrays** class defines several sort methods; we've used one:

```
static void sort( T[] a, Comparator<? super T> c)
```

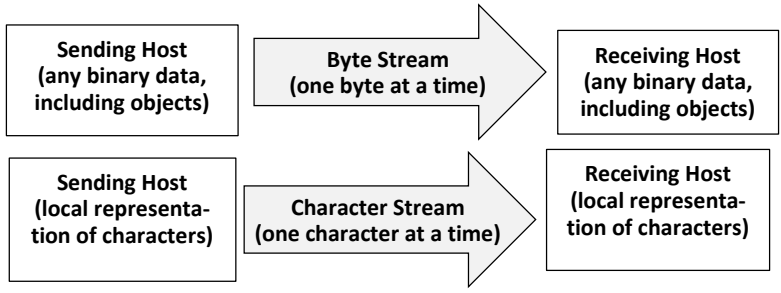
Here is how it is executed:

- **T** becomes an **Integer**
- The **integers** array substitutes the first operant **T[]**
- The lambda expression **(n1, n2) -> reverse (n1, n2)** is transformed into an object of the **Comparator<Integer>** functional interface, whose **compare()** method body is implemented as { **reverse (n1, n2);** }
- During the sorting, **reverse()** is used in determining which one of two integers is greater, as follows:

  n1>n2 : result>0; n1<n2 : result<0; n1=n2 : result=0

I/O Streams

A **stream** is an abstract representation of an input, where source data is retrieved from, or an output, where destination data is sent to. Java recognizes two types of streams – **byte streams** and **character streams**. The **byte streams** should be used for sending or receiving binary objects, and the **character streams** should be used when working with characters and the "character" representation of data is needed.

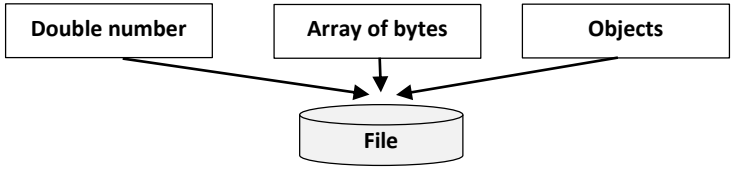


Note that the transmission of data via byte or character streams is done one byte or one character at a time. There are buffered streams that group the data into buffers, but those streams are still based on the underlying byte or character streams.

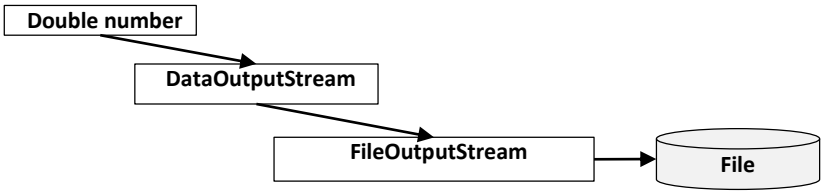
This chapter provides an overview of the concept behind streams and illustrates the usage of some input and output streams.

Byte Output Streams

Imagine that an application generates the following types of data and wants to write them out into a file:



One way to accomplish this would be by creating different classes for each type of input data, but Java's approach to this task is by building a "chain" of streams, each performing necessary data transformation and passing the result to the next stream:



The **FileOutputStream** class provides several methods for writing data out into files. For example, it defines the method **write(byte[] b)** that writes the specified array of bytes to a file. If we had an array ready, we could use this method as follows:

```
byte[] b = {97, 98, 99};
FileOutputStream fileout;
fileout = new FileOutputStream("myfile.txt");
fileout.write(b);
```

However, class **FileOutputStream** does not deal with primitive data types (double, float, int, etc.). So in order to write out a double number, we need to engage the **DataOutputStream** class, which implements the **writeDouble** method. **DataOutputStream** converts the received argument to an array of bytes and passes each byte, one by one, to the underlying stream by calling the **write** method of the underlying stream:

```
DataOutputStream dataout;
dataout = DataOutputStream(fileout);
```

dataout will be calling the **fileout.write()** method for each output byte.

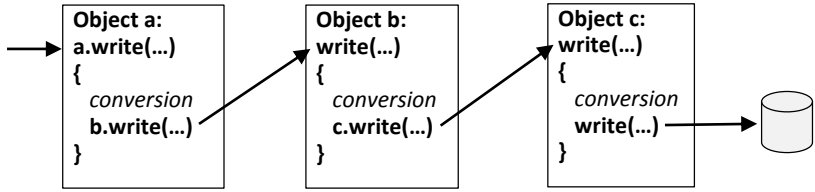
Now, we can call the **writeDouble** method of **DataOutputStream** that will convert the double number to bytes and pass them to **FileOutputStream** via the **write** method:

```
dataout.writeDouble(123.4);
```

In general, the chain of streams is constructed from the bottom up:

```
C c = new C();
B b = new B(c);
A a = new A(b);
// or just:
A a = A(new B(new C)));
```

The data is passed between output streams as follows:



In order of this mechanism to work, the following requirements must be met:

Each output stream must be a subclass of the class **OutputStream** and has the constructor ***streamName*** **(OutputStream s)**

Each output stream has to implement the following methods defined in the OutputStream class:

- write(int i) - mandatory
- write(byte[] b) - optional
- write(byte[] b, int offset, int length) - for buffered outputs

Classes in the output stream chain should not extend each other; otherwise it would be not possible to call the **write** method of the top class due to polymorphism.

There are quite a few **byte output streams**, but reviewing all of them would not add much to the general understanding of the concept of input/output streams, so we give you a single example of creating a custom byte output stream.

```
// Example of sending a double number to custom myOutputStream
// though DataOutputStream and BufferedOutputStream

import java.io.*;

class myOutputStream extends OutputStream {

   public void write(int i) {  }

   public void write(byte[] b, int offset, int len)
   {
     System.out.println("Received " + len + "bytes");
   }
}

class testMyStream {
   public static void main (String args[])
   {
      myOutputStream()      myout;
      BufferedOutputStream  buffout;
      DataOutputStream      dataout;

      try {
         myout = new myOutputStream();
         bufout = new BufferedOutputStream(myout);
         dataout = new DataOutputStream(bufout);

         dataout.writeDouble(123.4);

         dataout.flush();
      }

      catch (IOException e)
      {
         System.out.println("I/O error");
      }
   }
}
```

Where the output stream classes are

Not used by the buffered output, but must be implemented

This method will be called by **bufout**

The 123.4 will be converted to 8 bytes by **dataout** and put into a buffer by **bufout**

Push the data out of the buffer

An IOException can be thrown by the output stream classes; must be caught

The above program produces this output:

"Received bytes: 8"

Character Output Streams

All character data in Java is represented in the Unicode format. The character output stream classes convert Unicode characters into their local representation (e.g. ASCII format) before sending them to output devices or other streams. Another function of the character output streams is to generate a character representation of any primitive type, as well as objects. This is implemented via the **print** and **println** methods of the class **PrintWriter**. These methods are overloaded (i.e. there are multiple versions of them) and can accept all primitive data types, Strings, character arrays, or objects.

Similar to the byte output stream classes that extend the **OutputStream** class, the **character stream classes** extend the **Writer** abstract class and implement its **write** methods.

The following program uses two different character output streams to write an array of characters into a file:

```
import java.io.*;

class testCharStreams {
   public static void main (String args[]) {
      char[] c = {'a', 'b', 'c'};

      FileWriter fileout;
      PrintWriter printout;

      try {
         fileout = new FileWriter("write.txt");
         fileout.write(c, 0, 3);
         fileout.flush();

         printout = new PrintWriter("print.txt");
         printout.println(123.4);
         printout.flush();
      }

      catch (IOException e) {
         System.out.println("I/O error");
      }
   }
}
```

Put the first three characters of array **c** into an internal buffer

Push data out of the buffer

Convert the double number 123.4 into a character representation and put it into a buffer

Push data out of the buffer

Byte Input Streams

Byte input streams are used to receive (read) a sequence of bytes. All classes of this group have to extend the **InputStream** abstract class and implement its **read** methods:

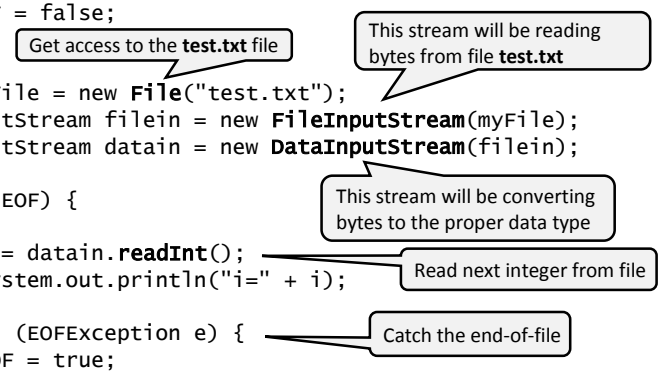
```
abstract int read ();
int read (byte[] b);                        // optional
int read (byte[] b, int offset, int len); // optional
```

Note: the **read** methods return the integer value **-1** when the end of the stream is reached.

In addition to the **read** methods, the **InputStream** class provides several other methods for supporting the so-called **markable streams**. With **markable streams**, you can mark a current position within the stream and later return to that position. The **markSupported** method can be used to check if a stream is markable.

Below is an example of reading integers from a file:

```
// Assumption: a set of integer values were written into the
// "test.txt" file via DataOutputStream.
import java.io.*;
class testByteInputStream {
   public static void main (String args[]) {
      boolean EOF = false;
      int i;
         try {
         File myFile = new File("test.txt");
         FileInputStream filein = new FileInputStream(myFile);
         DataInputStream datain = new DataInputStream(filein);

         while (!EOF) {
            try {
               i = datain.readInt();
               System.out.println("i=" + i);
            }
            catch (EOFException e) {
               EOF = true;
            }
         }
      }
      catch (IOException e) { System.out.println("I/O Error"); }
   }
}
```



Character Input Streams

Character input streams are represented by a group of classes extending the abstract class **Reader**. Their purpose is to read characters from different sources in their native (local) encoding format and convert them into the Unicode format. Each input stream class has to implement the **read** methods:

```
int read ();          // read a single character
int read (char[] c);  // read characters into an array
// reads into a portion of an array:
abstract int read (char[] c, int off, int len);
```

Note: the **read** methods return the integer value **-1** when the end of the stream is reached.

In addition to the **read** methods, the **Reader** class provides several other methods for supporting the so-called **markable streams**. With **markable streams**, you can mark a current position within the stream and later return to that position. The **markSupported** method can be used to check if a stream is markable.

Example:

```
// Print every other character of the alphabet
import java.io.*;
class testStringReader {
   public static void main (String args[]) {
      String s = "abcdefghijklmnopqrstuvwxyz";
      int i = 0;
      try
      {
         StringReader sReader = new StringReader(s);

         do
         {
               sReader.skip(1);
               i = sReader.read();
               if (i > 0)
                  System.out.print( (char) i);
         } while (i != -1);
      }
      catch (IOException e) {
         System.out.println("Error reading file");
      }
   }
}
```


This program creates the output: "**dfhjlnprtvxz"**

Try-With-Resources

There is one nice enhancement added to the stream handling process in JDK7. It allows to automatically release the resources held by a stream (e.g. close the file) once we are done using the stream. This feature can be used with the classes implementing the **AutoClosable** interface, which declares only one method – **close()**. The **close** method will be called when the **try** block ends.

The new format of the **try** block:

```
try ( resource declaration and initialization )
{  usage of the resource  }
catch (. . .) {. . .}
```

Example:

```
class myStream extends OutputStream{
   public void write(int i) {}
   public void write(byte[] b, int offset, int lng) {
     System.out.println("Received bytes: " + lng);
   }
}

class testStreams {
   public static void main (String args[]) {

   try (
         DataOutputStream dataout =
            new DataOutputStream (
            new BufferedOutputStream (
            new myStream ()))
       )
   {
      dataout.writeInt(123);
      // dataout.flush();
   }
   catch (IOException e) {
         System.out.println("I/O error");
   }
}
```

Start of the resource (stream) declaration

The **flush()** or **close()** would be required in a "traditional" **try** block.

The **dataout.close()** will be automatically called at the end of the **try** block

Serialization

Byte output streams have the capability of writing Java objects from computer memory into output devices or data streams, and **byte input streams** can read them back into the computer memory. The process of writing an object into a stream is called **serialization**, and the process of reading an object from a stream is called **deserialization**.

Every Java object is a complex structure that may include methods, variables, and even other objects. That structure is referred to as an "**object graph**". Java encodes all data into a special internal format before writing it out into a stream.

The implementation of serialization/deserialization is done by the **ObjectOutputStream** and **ObjectInputStream** classes. Their main methods are **writeObject()** and **readObject()**, respectively.

The **writeObject** method can serialize any object, as long as its class implements the **Serializable** interface. The **Serializable** interface does not declare any constants or methods; it is an indication that objects of this class are allowed to be serialized.

Example:

```
// Write one object of class myClass into file object.txt
import java.io.*;
class myClass implements Serializable {
   private String name;

   myClass (String s) { this.name = s; }

   public String myName() {return name;}

   public static void main (String args[]) {

      // Write one object of class myClass to file object.txt
      try (
               ObjectOutputStream objout =
               new ObjectOutputStream(
               new FileOutputStream(
               new File("object.txt"))) ;
       )
      {
          objout.writeObject(new myClass("First object"));
      }
```

myClass constructor

Instance method returning the **name** variable

objout.writeObject() method will be writing objects into the file object.txt via the FileOutputStream

Create a new myClass object and write it to a file

```
      catch (IOException e) {
         System.out.println("I/O error");
      }

// Read one object from the object.txt file
      try (
               ObjectInputStream objin =
               new ObjectInputStream(
               new FileInputStream(
               new File("object.txt"))) ;
           )
      {

          Object obj = objin.readObject();


          myClass myObj = (myClass) obj;

          System.out.println(myObj.myName());

       }

      catch (IOException e) {
         System.out.println("I/O error");
      }

      catch (ClassNotFoundException e) {
         System.out.println("Class not found error");
      }
   }
}
```

The **objin.readObject()** method will be reading objects from file object.txt via FileInputStream

Read an object from file as type **Object**

Cast the type of the just-read object to **myClass**

Execute the instance method **myName**

When you serialize an object, you can choose not to serialize some of its data members - primitive types or other objects. For this, you declare a variable as **transient**. In the above example, to prevent the **name** variable from being written out to the file, declare it as follows:

```
private transient String name;
```

During deserialization the transient variables are set to their default values – **zero** for numbers, **true/false** for Boolean, and **null** for objects.

In the above example, if the **name** variable was declared as transient, the output of the program would be "**null**" because strings are objects in Java.

A special situation will occur if your **serializable** class extends a **non-serializable** class. During the deserialization process, an instance of the non-serializable class (your superclass) will be created automatically using the no-argument constructor. You have to make sure that such a **no-argument constructor** of the superclass exists.

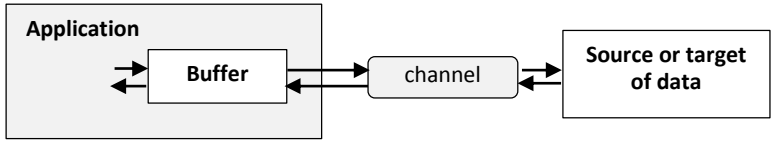
New Input/Output System - NIO

Since the inception of the language, Java's approach to the input/output operations was **stream-oriented**. The transmission of data between the application and an external entity (e.g. a disk file) was seen as a stream of single bytes or characters.

Based on the direction of the stream and the type of data transmitted, streams are divided into four categories: **byte input stream**, **byte output stream**, **character input stream**, and **character output stream**. Each of these streams is represented by a group of Java classes, each extending, respectively, **InputStream** class (byte input stream), **OutputStream** class (byte output stream), **Reader** class (character input stream), or **Writer** class (character output stream).

The common characteristic of all the above streams is that they are **blocking** data streams. It means that the current execution thread of your application is put on hold until the I/O request is completed.

In version 1.4 Java first introduced an alternative input/output system named **New Input Output**, or just **NIO**, in which the I/O concept changed from the **stream-oriented** to **buffer/channel-oriented**.



Here are the highlights of NIO features:

- All data is transmitted via a channel into or out of a buffer. The application manipulates data directly in the buffer.
- Unlike standard streams, a channel can transmit data in both directions.
- Channels are capable of operating in blocking mode, same as streams, and in non-blocking mode. In the non-blocking mode, the thread that issues an I/O request is **not** put on hold until the I/O operation completes, but continues its execution.
- NIO also added the capability of handling multiple channels within a single thread.

In addition to the above features, over time NIO was enhanced to work together with standard I/O streams and to perform file system operations (i.e. manipulation of files and directories).

Here are the main NIO classes and interfaces:

Buffer

The **Buffer** class is used to create buffers and manipulate the buffer's contents. There are sub-classes of the Buffer class to hold different types of primitive data – **ByteBuffer**, **CharBuffer**, **IntBuffer**, etc.

Channel

The **Channel** is actually an **interface** implemented by various channel classes: **SocketChannel**, **DatagramChannel**, **SeekableByteChannel, FileChannel**, etc. It represents a connection to an entity capable of performing I/O operations (file, network socket, etc.)

Selector

The **Selector** class allows a single application thread to handle multiple channels.

(The usage of the **Selector** class is beyond the scope of this book.)

Files

The **Files** class provides methods to operate on files and directories. It also can open standard byte streams (i.e. **InputStream** and **OutputStream**) for reading from or writing to a file.

The first example illustrates the usage of **Buffer**, **Channel**, and **Files** for passing unformatted data (bytes) between the application program and a disk file via a blocking channel.

```
// Example of using NIO for writing to and reading from files

import java.io.*;
import java.nio.*;
import java.nio.file.*;
import java.nio.channels.*;

class testNIO {
   public static void main (String args[]) {

      byte[] bytes = {97, 98, 99};

      ByteBuffer buffer = ByteBuffer.allocate(16);

      try {
          Path path = Paths.get("testNIO.txt");

          SeekableByteChannel channel =
            Files.newByteChannel (path,
               StandardOpenOption.CREATE,
               StandardOpenOption.WRITE,
               StandardOpenOption.READ );

          buffer.put(bytes);

          buffer.rewind();
          channel.write(buffer);

          channel.position(1);
          buffer.rewind();
          channel.write(buffer);

          channel.truncate(3);
          channel.position(0);

          buffer.rewind();
          int i = channel.read(buffer);
          System.out.println("file length = " + i); // prints: 3

          channel.close();
      }
      catch (InvalidPathException e) {
            System.out.println("Invalid path");
      }
      catch (IOException e) {
         System.out.println("I/O error");
      }
   }
}
```

Byte array; represents "abc" in character format

Allocates a 16-byte buffer

Creates a path to file **testNIO.txt**

Creates a read/write channel to file **testNIO.txt**; Allocates the file if it does not exist

Puts the **bytes** array into a buffer

Repositions at the first byte of the buffer

Puts the buffer into a channel and into the file

Writes out the same buffer again starting with the second byte of the file

Truncates the file to 3 bytes

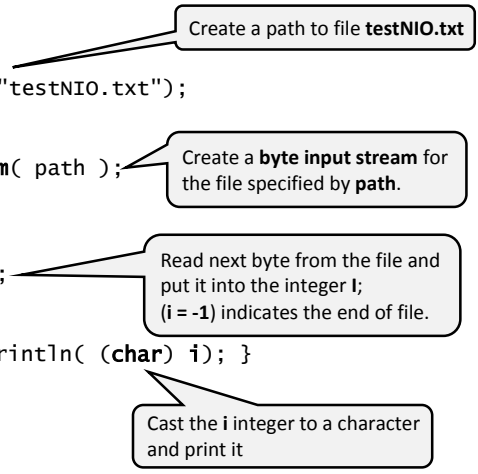
Repositions at the beginning of the file

Reads the file into a buffer

After the execution of the code, the file testNIO.txt will contain "**aab**".

The next example demonstrates how NIO and standard stream I/O can work together to read the contents of a file.

```
// Print the testIO.txt file created in previous example

import java.io.*;
import java.nio.file.*;

class testNIOStream {
   public static void main (String args[]) {
      int i;

      try
      {
         Path path = Paths.get("testNIO.txt");

         InputStream stream =
            Files.newInputStream( path );

         do
         {
              i = stream.read();

              if (i != -1)
                 { System.out.println( (char) i); }
         }   while (i != -1);

         stream.close();
      }

      catch (InvalidPathException e) {
         System.out.println("Invalid path");
      }
      catch (IOException e) {
         System.out.println("I/O error");
      }
   }
}
```



The last example in this chapter demonstrates some of NIO's capabilities when working with the file system.

```
// Print all the directories on the C: drive

import java.io.*;
import java.nio.file.*;
import java.nio.file.attribute.*;

class testNIOFileTree {
   public static void main (String args[]) {

      try {
         path = Paths.get("c:/");
         Files.walkFileTree( path, new CustomFileVisitor() );
      }
      catch (IOException e) {
         System.out.println(e);
      }
   }
}

class CustomFileVisitor extends SimpleFileVisitor <Path> {

   public FileVisitResult       visitFile (Path path,
               BasicFileAttributes attr)
               throws IOException {
      return FileVisitResult.CONTINUE;
   }

   public FileVisitResult preVisitDirectory (Path path,
               BasicFileAttributes attr)
               throws IOException {
      System.out.println(path.getFileName());
      return FileVisitResult.CONTINUE;
   }

   public FileVisitResult visitFileFailed (Path path,  IOException e)
      throws IOException {
      return FileVisitResult.CONTINUE;
   }
}
```

Starting point of traversing the file system

For each file, one of four pre-defined methods will be called. These methods are defined in the class **SimpleFileVisitor**. We need to override them if we want non-default behavior.

This method is called when a file is visited; we do nothing and continue the scan.

This method is called before a directory is visited; we print the directory name and continue.

This method is called when an error occurs while reading a file; we do nothing and continue.

Note: The FileVisitResult enumeration defines other constants besides CONTINUE, controlling the execution of the method walkFileTree. The SKIP_SUBTREE constant, for example, will force walkFileTree to skip visiting the entries of the current directory.

The Stream API

Stream API is a new **a**pplication **p**rogramming **i**nterface added in JDK 8. It is not an extension to or a replacement of the previously discussed I/O streams (byte and character streams). The **stream API** introduced a new stream concept. It sees any stream as a sequence of objects on which various manipulations are performed. Using the stream API classes, we can apply different transformations to the stream (sorting, filtering, etc.) and produce either another stream (i.e. sequence of objects) or a final result. There are no ultimate "destination points" like files or network sockets. For example, we might want to find the array element with the highest value. In this case the result is not a destination, but rather some value derived from the source data, and the whole process resembles a database query. Note that the data source remains unchanged in stream processing.

Let's start with a very small and simple example.

```
// Find the lowest value stored in an array of integers

import java.util.*;
import java.util.stream.*;

class testStreamAPI {
   public static void main (String args[]) {

     int[] integers = {8, 5, 7, 2, 3, 4};

     IntStream intstream = Arrays.stream(integers);

     OptionalInt val = intstream.min();

     System.out.println(val.getAsInt());
   }
}
```

The static method **stream()** of class **Arrays** converts the array **integers** into a sequential stream of integers

The **min()** method of **IntStream** finds the minimum element of the stream and returns it as an **OptionalInt** object

The **getAsInt()** method of the **OptionalInt** object retrieves the integer value.

What should be noted about this example is that the **min()** method applied to the stream produces a final result, i.e. the result is not a stream anymore. Methods that produce final results are known as **terminal operations**, which consume the stream. Other methods that return the result as another stream are called **intermediate operations**. The next example presents two-stage stream processing involving intermediate and terminal methods.

Let's say that we have an array of double numbers and we want to calculate the sum of all elements, but first round them up to the next larger integer. This can be accomplished with a single line of code, but we will present and explain each step of the process.

The first step is to convert the array of doubles into a stream. We can use the static method **stream** of the class **Arrays** that returns a sequential stream of type **DoubleStream**:

```
double[] doubles = {1.5, 2.6, 3.3};
DoubleStream dblstream = Arrays.stream(doubles);
```

The next step is to transform the stream of doubles into a stream of integers while rounding each double to the next larger integer. We can utilize the **mapToInt** method of the interface **DoubleStream** for this. The **mapToInt** method is declared as follows:

```
IntStream mapToInt(DoubleToIntFunction mapper)
```

It accepts an object of the functional interface **DoubleToIntFunction**, applies its method **applyAsInt** to each element of the stream, and returns the new **IntStream**. So, we need to create an object of type **DoubleToIntFunction**, implementing the method **applyAsInt** according to our needs. The easiest way to do this is by using a lambda expression:

```
DoubleToIntFunction mapper = n -> (int) (n + 0.5);
```

This statement will create the object **mapper** of type DoubleToIntFunction with the applyAsInt method defined as follows:

```
int applyAsInt (double n) { return (int) (n+0.5); }
```

Now we're ready to transform a DoubleStream into an IntStream:

```
IntStream intstream = dblstream.mapToInt(mapper);
```

The final step is to calculate the sum of all elements of the integer stream by using the **sum()** method of **IntStream**:

```
int sum = intstream.sum();
```

When put together, the code will look as follows:

```
double[] doubles = {1.5, 2.6, 3.3};

DoubleStream dblstream = Arrays.stream(doubles);
DoubleToIntFunction mapper = n -> (int) (n + 0.5);
IntStream intstream = dblstream.mapToInt(mapper);
int sum = intstream.sum();
```

The result calculated by this code is **8**.

Now, as mentioned before, the whole process can be expressed in one line:

```
int sum = Arrays.stream(doubles).mapToInt(n -> (int) (n + 0.5)).sum();
```

Note, if we need to put more complex logic into the mapper (i.e. into the **applyAsInt** method), we can define a custom method and specify it in the lambda expression:

```
public static int round(double n) {
   int i = (int) (n + 0.5);            // round the double number
   System.out.println("in=" + n + " out=" + i);
   Return i;
}
. . .
int sum = Arrays.stream(doubles).mapToInt(n -> round(n)).sum();
```

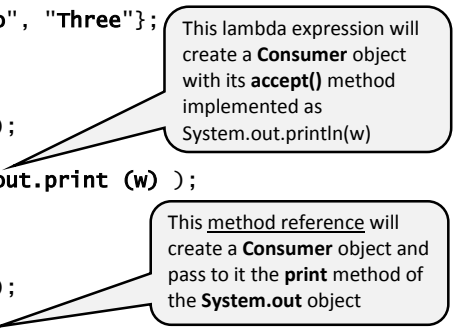
The **mapToInt()** method is an **intermediate operation** - it accepts a stream and returns another stream. The **sum()** method is a **terminal operation** – it consumes the stream and returns a final result.

The next example demonstrates the process of iterating through all elements (objects) of a stream. We utilize the **forEach()** instance method of class **Stream** to perform the loop. The **forEach()** method is declared as follows:

`forEach` (`Consumer`<? super T> **action**)

For each object of the stream, the **forEach()** method invokes the **accept()** method of **Consumer**, providing it with the object retrieved from the stream. The **Consumer** is a functional interface, so an object of type Consumer can be created either via a lambda expression or by using a method reference. Both approaches are presented in the following example.

```
// Using the forEach() method of class Stream
import java.util.*;
import java.util.stream.*;

class testStream {

   public static void main (String args[]) {

     String[] words = {"One", "Two", "Three"};

     Stream<String> stream;

     stream = Arrays.stream(words);

     stream.forEach( w -> System.out.print (w) );


     stream = Arrays.stream(words);

     stream.forEach( System.out::print );

   }
}
```



The above code prints two lines:

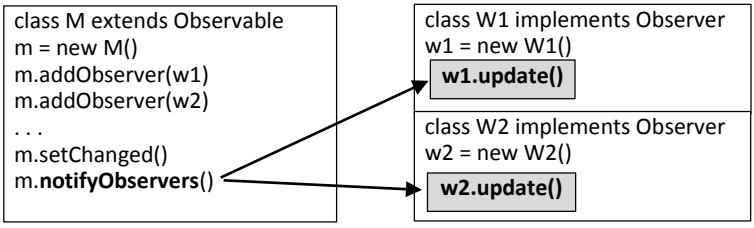
"OneTwoThree"
"OneTwoThree"

Before we conclude this overview of the Stream API, we want to mention that one of the benefits offered by the Stream API is parallel processing. The operations on a stream can occur in parallel, assuming the environment supports parallelism. To switch to parallel processing, use the **parallel()** method:

```
int sum = Arrays.stream(doubles)
          .parallel()
          .mapToInt(n -> round(n)).sum();
```

Observable and Observers

The **Observable** class, together with the **Observer** interface offers a simple mechanism of establishing communication between objects. One object (observable object) can be watched (observed) by one or more other objects (observers). When needed, the object being observed can notify its observers, and they can perform some actions.



Example:

```
import java.util.*;
public class myClass extends Observable implements Observer {
   String myName;
   public myClass (String s) {myName = s;}

   public void update(Observable m, Object arg)
      { System.out.println(myName + " received a " + arg +
        " from " + m.myName);  }

   public static void main (String args[])
   {
      myClass m   = new myClass ("manager");

      m.addObserver(new myClass ("Bob");
      m.addObserver(new myClass ("Joe");

      m.setChanged();

      m.notifyObservers("message");
   }
}
```

Objects of **myClass** can act as observer and observable.

Create two new objects and register them as observers of object **m**.

The **setChanged()** must be called before each notification

The **update()** method of each observer will be called;
Note: **m** will be passed to **update()** automatically.

The output of this program:

"**Bob** received a **message** from **manager**"
"**Joe** received a **message** from **manager**"

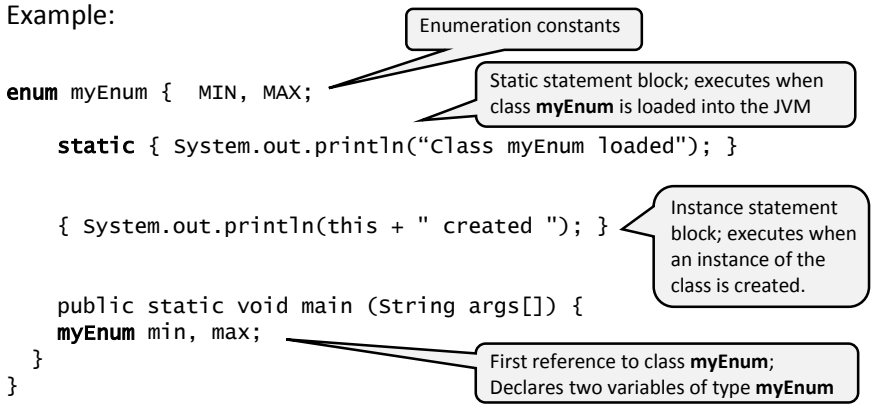
Enumerations

The idea behind **enumerations** is to provide an easy way of defining sets of constants not bound to any primitive or object types, which usually restricts the scope of their usage. Java's solution to this problem was a new type of classes, whose instances are treated as constants. Here is a sample declaration of an enumeration class:

```
enum myEnum { MIN, MAX }
```

Notice the usage of **enum** instead of **class**. The MIN and MAX are the instances (objects) of the class **myEnum** and are called **enumeration constants**. They are implicitly declared as **public**, **static**, **final**. The enumeration constants are created (instantiated) automatically, when the enumeration class is first referenced in the program. Therefore, you do not need to use the **new** operator (in fact, you cannot) to create an object of the enumeration class. Besides these differences, the enumeration classes are still Java classes, and as such, they can have static and instance variables, methods, statement blocks, and even constructors.

Example:




```
enum myEnum {  MIN, MAX;

    static { System.out.println("Class myEnum loaded"); }

    { System.out.println(this + " created "); }

    public static void main (String args[]) {
    myEnum min, max;
  }
}
```

This program produces the output:

```
MIN created
MAX created
Class myEnum loaded
```

Now, let's see what we can do with the enumerations. The best way to illustrate the benefits of enumerations is to go through examples. Imagine

that we have a method that monitors weather conditions at major national airports. By employing enumeration, we can ensure, at compile time, that our method receives requests only for a predefined set of airports, which reduces the risk of the run-time errors:

```
class testEnumAirport {

   enum Airport { JFK, PHL, MIA; }

   public static void main (String args[]) {
      Airport a;
      a = Airport.MIA;
      System.out.println(getTemp(a)); // prints: 80.0

      // System.out.println(getTemp("JFK"));
   }

   public static double getTemp (Airport a) {
     switch (a) {
       case JFK: return 60.0;
       case PHL: return 70.0;
       case MIA: return 80.0;
       default:  return 0.0;
     }
   }
}
```

An enumeration can be declared within another class

It's impossible to provide the **getTemp()** method with an object of any other type except **Airport**

It looks beneficial, but what if we want to provide some other information about the airports, like state and city names? This can also be easily achieved with the help of enumerations. As an object, each enumeration constant can have instance variables, and those variables could hold the state code and city name. All we need to do is to declare the instance variables, define a constructor that initializes them, and provide their initial values for each enumeration object:

```
class testEnumAirport {
   enum Airport {
     JFK("NY","New York"),
     PHL("PA","Philadelphia"),
     MIA("FL","Miami");

     String state, city;

     Airport (String s, String c)  {this.state = s; this.city=c;}
   }

   public static double getTemp(Airport a) {return 80.0;}
   public static String getCity(Airport a) {return a.city;}
   public static String getState(Airport a) {return a.state;}
```

Specify the initial values of instance variables for each enumeration

Instance variables

Constructor initializing the instance variables

```
   public static void main (String args[]) {
       Airport a;
       a = Airport.MIA;
       System.out.println( getCity(a) + “, ”
                                      + getState(a) + ” temp=”
                                      + getState(a) );
   }
}
```

This program produces the output: **Miami, FL temp=80.00**

We have mentioned already that enumerations are classes of a special type. When you create an enumeration class, it implicitly extends the regular **Enum** class. All the methods defined by the **Enum** class are available to any enumeration you create. We will present some commonly-used methods.

The static method **values()** returns an array of enumeration constants in the order they were declared:

```
enum Airport { JFK, PHL, MIA; }
. . .

   for ( Airport a : Airport.values() )
       { System.out.println(a); }
```

for-each loop can be used to iterate over the enumeration objects

The static method **valueOf()** returns the enumeration constant (object) with the specified name:

```
Airport a = Airport.valueOf(“PHL”);
```

The instance method **ordinal()** returns the position of the enumeration constant within the declaration, staring with zero:

```
enum Airport { JFK, PHL, MIA; }
. . .

System.out.println(Airport.JFK.ordinal()); // prints: 0
System.out.println(Airport.PHL.ordinal()); // prints: 1
System.out.println(Airport.MIA.ordinal()); // prints: 2
```

Regular Expressions

The validation and manipulation of the contents of text strings can be found in many applications. For example, we might need to check if an email address entered has valid format. The **Regular Expressions API** is a powerful tool that a Java programmer can utilize to perform text validation, parsing, tokenization, or other type of manipulations.

The **Regular Expression API** consists of two utility classes: **Pattern** and **Matcher**. The **Pattern** class is used to build regular expressions, which are referred to as **patterns**. The **Matcher** class is used to apply the pattern to a text string. The usage of these two classes is quite simple but the challenge is in mastering regular expression skills, which, unfortunately, is beyond the scope of this book. Nevertheless, we will go through several examples to make you familiar with the operations that can be performed on text strings, and also to cover the basics of regular expressions.

Regular Expressions Basics

A regular expression is a sequence of characters defining a pattern that can be matched against a text string. For example, the regular expression “abc” can be used to check if a text contains the “abc” sequence.

There are several categories of regular expressions. We will present three of them.

The first category of regular expressions is called “**characters**” and includes most of the characters and also the escape sequences, like **\t** (tab character), or **\n** (newline character). “abc” is an example of the **character** regular expression.

The second category of regular expressions is called “**character classes**” and represents a range of valid characters. For example, the **[a-zA-Z]** pattern will match any lower-case or upper-case letter.

For some of the **character classes** constructs Java provides a shorthand form:

```
.       - matches any character
\s      - matches the white space character
\d      - matches any digit
```

```
\w      - matches any word character
. . .
```

The third category of regular expression constructs is called "**quantifiers**". Quantifier can be appended to any **character** or **character class** construct to specify the allowed number of occurrences for the construct:

```
?       - occurs once or not at all
*       - occurs zero or more times
+       - occurs one or more times
{n}     - occurs exactly n times
{n,}    - occurs at least n times
{n,m}   - occurs from n to m times
```

There are other regular expression groups, but their review is beyond the scope of this book.

Regular Expressions Examples

Before we start with examples, we need to mention that the **Pattern** and **Matcher** classes have no constructors. **Pattern** objects are created by the static method **compile()** of class **Pattern**, and **Matcher** objects are created by the instance method **matcher()** of class **Pattern**.

Example: Validate if the whole text matches the pattern.

```
String regex;
Pattern p;
Matcher m;

p = Pattern.compile("[a-z]+");
m = p.matcher("test") ;
System.out.println(m.matches()); // prints: true
```

[a-z]+ matches one or more lower case letters

Create Matcher object **m**

Example: Validate the email format.

```
regex = "[a-zA-Z0-9]+"
        + "@"
        + "\\w+"
        + "\\."
        + "com";
p = Pattern.compile(regex);
m = p.matcher("John123@site.com");
System.out.println(m.matches()); // prints: true
```

Matches any number of letters or digits

Matches the character '**@**'

Matches any number of word characters

Matches the character '.'

Matches the string "**com**"

Example: Extract the user name and website address from email.

```
p = Pattern.compile("@");
String[] ss = p.split("John@site.com");
```

```
System.out.println(ss[0]);  prints: John
System.out.println(ss[1]);  prints: site.com
```

The next example is more complicated. It uses the **splitAsStream()** method of the **Pattern** class to obtain the distinct (unique) words from the input text. It also uses a lambda expression to print the results from within the **forEach()** method of the Stream class.

Example: Extract the distinct (unique) words from string.

```
import java.io.*;
import java.util.stream.*;
import java.util.regex.*;

class testRegex
{
   public static void main (String args[])
   {
     String text = "a b  c  a b   c  def b b b g";

     p = Pattern.compile(" ");
     Stream<String> stream = p.splitAsStream(text);
     Stream<String> unique = stream.distinct();
     unique.forEach(w -> System.out.print(w + " "));

     // same result can be achieved in one statement:

     Pattern.compile(" ")
                    .splitAsStream(text)
                    .distinct()
                    .forEach(w->System.out.print(w + " "));
   }
}
```

The output of the above program is: "**a b c def g ".**

Example: Replace all blank sequences with semicolons.

```
     String text = "One    Two Three         Four";

     System.out.println
     (
        Pattern.compile("\\s+").matcher(text).replaceAll(";")
     );

     // prints: One;Two;Three;Four
```

Matches any number of blanks

Reflection API

Every Java program performs some manipulations on objects of various types - classes, interfaces, enumerations, arrays, primitive types, etc. At run time, for each type of objects used by the program, the Java Virtual Machine (JVM) creates an instance of class **Class**, which provides methods to examine the properties of the object's type. For example, when you declare a class **A**, the JVM builds a corresponding **Class** object, providing all information about class A – its methods, interfaces, annotations, fields, etc.

The **Reflection API** is a very important component of the Java language. It is used extensively by Java Beans (not covered by this book), by various test tools, when working with annotations, etc. You can also use it for debugging purposes.

Obtaining the Class object

There are three primary methods of obtaining the Class object for a particular class.

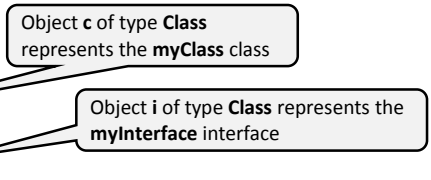
When you have an instance of the class you want to explore, you can call the instance method **getClass()** defined by class **Object** and available to all Java objects:

```
class myClass { ... }
. . .
myClass obj = new myClass();

Class c = obj.getClass();
```

Object **c** of type **Class** represents the **myClass** class

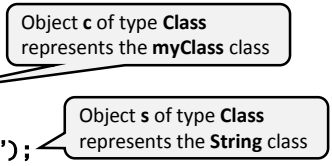
In the case when no instances of a class are available you can employ the so-called **class literal** to obtain the **Class** object. A **class literal** is formed by the class name appended with **.class**:

```
interface myInterface {
   Void myMethod();
}


class myClass { ... }
. . .

Class c = myClass.class;

Class i = myInterface.class;
```



Finally, you can use the static method **forName()** of the class **Class**. This method accepts a fully-qualified name of a class and returns its **Class** object. Note, that the forName() method can throw the **ClassNotFoundException**.

```
class myClass { ... }
. . .

Class c = Class.forName("myClass");

Class s = Class.forName("java.lang.String");
```



Discovering Class Members

Once you get the Class object for the class being examined, you can use various methods of **Class** to obtain more information about the class. We will give you one example of obtaining the names of all methods and fields declared by a class. Please check the **java.lang.reflect** package for other reflection classes and methods.

```
import java.lang.reflect.*;

class myClass {
   private String name;
   void setName (String s) {this.name = s;}
   String getName () {return name;}

   public static void main (String args[]) {

      Class c = myClass.class;

      try {

         Method[] methods = c.getDeclaredMethods();
         for (Method m: methods) {
            System.out.print(" Method:" + m.getName());
         }

         Field[] fields = c.getDeclaredFields();
         for (Field f: fields) {
            System.out.print(" Field:" + f.getName());
         }
      }
      catch (Exception e) {System.out.println(e);}
   }
}
```

Object **c** of type **Class** represents the **myClass** class

Returns an array of **Method** objects

Returns an array of **Field** objects

The above code produces the output:

```
Method:main Method:getName Method:setName Field:name
```

Annotations

Annotations provide a standard way of documenting Java programming code. The following characteristics make the annotations distinct from other documenting means:

- Annotations are closely associated with the program constructs they annotate; this reduces the possibility of misrepresentation.
- Annotations can be used as instructions to the Java compiler to perform certain actions, e.g. to suppress some warning messages.
- Annotations are Java objects that can be accessed at run time by the application program containing annotations or by third-party tools.

Important note: annotations have no effect at run time.

How do we create an annotation?

In a nutshell, annotations are regular Java classes, only declared and handled differently. All annotation classes (referred to as **annotation types**) are created by implementing the **Annotation** interface. However, instead of using the **implements** keyword in the class declaration, we specify the annotation class as **@interface**:

```
@interface myAnnotation {
   Type method();
   Type method() default value;
   . . .
}
```

Only methods without bodies are allowed in the annotation declarations

Example:

```
@interface Author {
   String name();
   String version() default "1.0";
}
```

A default value returned by the method can be declared.

Any declaration in the program source, even the annotation itself, can be annotated. Annotation should precede the declaration being annotated:

```
@Author (name="John Doe", version="1.2")
class myClass { … }
. . .


@Author (name="Bob")
public void myMethod (int n) { … }
```

Class **myClass** is annotated with the **Author** information

Method **myMethod** is annotated. it's OK to skip the **version** because it was defined with default return value

What can we do with annotations?

Before answering this question, we need to review the lifecycle of the Java program, which can be broken into four stages:

- Source code
- Compilation
- Byte code
- Execution (run time)

The initial question needs to be broken down for each stage: “What can we do with annotations in the source code, during compilation, while in byte code, and at the run time?”

As we have already seen, the source code contains the annotation declarations. From the programmer’s standpoint, not much can be discussed here except the specifics of the declaration syntax, which will be covered later.

The next stage is compilation. The compiler interprets the annotation declarations and creates proper Java objects, but does the compiler do anything else with the annotations? The answer is yes - there is a set of **pre-defined** (i.e. built-in) annotations, which serve as instructions to the compiler. They all are defined in the **java.lang.annotation** package:

```
@Retention
@Target
@Inherited
@Override
@SuppressWarnings
@Deprecated
@SafeVarargs
@Repeatable
@FunctionalInterface
```

As all the annotations, the pre-defined annotations should precede the annotated declaration (or another annotation). For example, the following code will instruct the compiler to check if the **someMethod()** is indeed an override of the same method of the superclass. If the superclass of the current class does not declare the someMethod(), a compile-time error will occur.

```
@Override
public void myMethod() {…}
```

Other, not pre-defined annotations (user-defined, or declared in other Java packages), are controlled by the **@Retention** pre-defined annotation. The **@Retention** annotation can specify three values defined by the **RetentionPolicy** enumeration:

```
@Retention (RetentionPolicy.SOURCE)
@Retention (RetentionPolicy.CLASS)   ← default value
@Retention (RetentionPolicy.RUNTIME)
```

Note: @Retention can annotate only another annotation declaration.

Here is the effect of the above constants:

- **SOURCE**– annotation stays only in the source (.java file) and is discarded after compilation.
- **CLASS** – annotation will be stored in the bytecode (.class file) but will be discarded before the execution; this is the default value.
- **RUNTIME** – annotation will be available at run time.

The **SOURCE**-level annotations are not processed by the Java compiler (with the exception of the pre-defined annotations). However, as of Java SE 6, you can add ***annotation processors*** to the Java compiler, which are stand-alone, custom tools for processing annotations. They are out of the scope of this book.

The **CLASS**-level annotations are too specialized and therefore are also beyond the scope of our discussion, so let's talk about run-time processing.

Processing Runtime Annotations

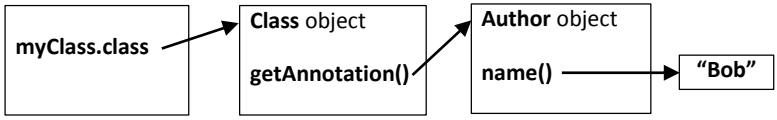
When the **@Retention(RetentionPolicy.RUNTIME)** annotation is specified for another annotation (custom defined, or declared in one of the Java packages), it makes that annotation available at the run time. Since Java does not provide any special run-time mechanisms of handling the annotations, it's up to us what to do with them.

As we already mentioned, annotations are objects of Java classes implementing the **Annotation** interface. What it means is that if we can obtain a reference to the object of some annotation, we can invoke its method(s) to retrieve the declared values:

```
@interface Author {
   String name();
}

@Author (name="Bob")
class myClass {
   public static void main (String args[]) {
      Author a = (myClass.class).getAnnotation(Author.class);
      System.out.println(a.name()); // would print: Bob
   }
}
```

To understand how to access an instance of an annotation, we need to recall the purpose of annotations, which is to provide some supplemental information about other declarations. The **@Author (name="Bob")** above creates an object of type **Author**, associated with the class **myClass**, and whose method **name()** returns the value "Bob".

As we explained previously in the **Reflection** chapter, at run time, each class is represented by a **Class** object. The association between the annotation and the annotated class can be presented with this diagram:



myClass.class is the instance of the **Class** object representing **myClass**. (We could use the **getClass()** method on myClass instance, but such an instance might not be created). Then, we can use the instance method **getAnnotation()** to obtain the instance of the **Author** annotation. The final step is to execute the **name()** method of the annotation to retrieve the value "Bob".

Here is the statement obtaining the Author object annotating myClass:

```
Author a = (myClass.class).getAnnotation(Author.class);
```

Note that we provided **Author.class** to the **getAnnotation()** method. This was necessary because myClass could have more than one annotation, and the getAnnotation method would need to know which one to retrieve.

Here is a sample code accessing an annotation at run time:

```
import java.lang.reflect.*;

@interface Author {
   String name();
}

@Author (name=”Bob”)
class myClass
{
   public static void main (String args[])
   {
      Author a = (myClass.class).getAnnotation(Author.class);
      System.out.println(a.name()); // prints: “Bob”
   }
}
```

Declares the annotation **Author**

Annotates the **myClass**

Obtains the instance of **Author** associated with **myClass**

The same technique can be used for annotated methods, fields, etc. For example, if the above annotation was used to annotate the **myMethod(String s)** method, the access to the annotation object would be as follows:

```
Author a = (myClass.class)
           .getMethod(“myMethod”, String.class)
           .getAnnotation(Author.class);
```

Annotation Types

Annotation coding semantics allows dropping some parts of the annotation constructs when the corresponding information can be derived from the context. For example, if the annotation does not have any parameters, the following two declarations are equal:

```
@interface MyAnnotation { }
@interface MyAnnotation
```

This is driven by the number of elements an annotation has. Based on this, Java recognizes the following types of annotations:

- Normal Annotation
- Single Element Annotation
- Marker Annotation

A **Normal Annotation** is an annotation explicitly declaring all components of the annotation constructs, for example:

```
@interface Book {
   String name();
   int edition() default 1;
}

@ Book ( { name="My Book",version=2; } )
class myClass {…}
```

Declares the **Book** annotation with two elements – **name** and **edition**

Annotates **myClass** with **Book** annotation; both elements are specified.

A **Single-Element Annotation** is a shorthand designed for use with annotations declaring a single element. If the element name is **value**, it can be omitted from the annotation:

```
@interface Name {
 String value()
}

@Name ( value="Bob" )
@Name ( "Bob" )
```

It is also valid to use single-element annotations for annotations with multiple elements, so long as one element is named **value** and all other elements have default values. All below constructs are valid:

```
@interface BookName {
   String value();
   int edition() default 1;
}

@ BookName ( { name="My Book",edition=2; } )
@ BookName (name="My Book")
@ BookName ("My Book")
```

Single-value annotation with other elements declared with default values

A **Marker Annotation** is shorthand for annotations without elements. All below constructs are valid:

```
@interface Book { }
@interface Book

@Book ()
@Book
```

Marker annotation

Restricting the Usage of Annotations

The pre-defined annotation **@Target** can be used to specify the types of items to which another annotation can be applied. Valid values are defined by the enumeration **ElementType**:

```
@Target (ElementType.ANNOTATION_TYPE)
@Target (ElementType.CONSTRUCTOR)
@Target (ElementType.FIELD)
@Target (ElementType.LOCAL_VARIABLE)
@Target (ElementType.METHOD)
@Target (ElementType.PACKAGE)
@Target (ElementType.PARAMETER)
@Target (ElementType.TYPE)
@Target (ElementType.TYPE_PARAMETER)
@Target (ElementType.TYPE_USE)

Example:

// @Author can only be used to annotate constructors and methods:

@Target ( { ElementType.CONSTRUCTOR, ElementType.METHOD } )

@interface Author {
 String name()
}

@ AUthor ( name = "Joe" )
class myClass {…}

@ AUthor ( name = "Bob" )
myClass () {…}

@ Author ( name = "Mike" )
public void myMethod () {…}
```

Invalid annotation:
cannot annotate classes

Valid annotation:
can annotate class constructors

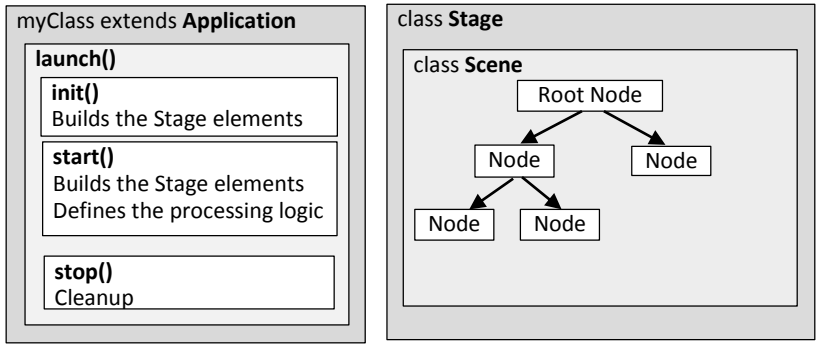
Valid annotation:
can annotate methods

JavaFX API

JavaFX is a set of graphics and media packages that enables developers to create rich client applications with high-performance modern graphical user interfaces featuring audio, video, graphics, and animation.

Each JavaFX application consists of two major parts: **application** and **stage**. The **application** (represented by the Application class) is responsible for building the **stage** and its elements, and for defining the processing logic. The **stage** (represented by the Stage class and other classes) is a hierarchical collection of objects defining the visual appearance of the **application**.

This diagram presents the structure of JavaFX applications:



The **Stage** defines the environment for the JavaFX application. A default Stage instance is provided by the Java run-time environment. However, you can create multiple Stage instances (objects).

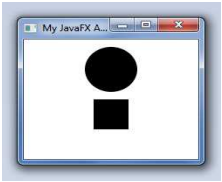
The **Scene** is a container (a window) that holds the contents (i.e. elements) of the **Stage**. The contents of a **Scene** create what is referred to as a **scene graph**.

The **Root Node** determines the layout of the elements of the **Scene**.

The **Node** represents an element of the **Scene**. It could be a button, a text box, a geometrical shape, an animation, etc.

The **Application** class launches the JavaFX applications. The **launch()** method executes the **init()** method, then **start()**, and after the **Stage** is finished (e.g. the window is closed) – the **stop()** method.

The following example uses JavaFX to create a window displaying a circle and a rectangle:




```
import javafx.application.*;
import javafx.stage.*;
import javafx.scene.*;
import javafx.scene.shape.*;
public class testJavaFX extends Application {
  Circle    circle;
  Rectangle rectan;
  public static void main(String[] args) {
     System.out.println("JavaFX: launch");

     launch(new String[1]);
  }
  // -------------
  public void init() {
    System.out.println("JavaFX: init");
    System.out.println(Thread.currentThread().getName());

    circle = new Circle (100, 40, 30);
    rectan = new Rectangle (80, 80, 40, 40);
  }
  // --------------
  public void start(Stage stage) {
     System.out.println("JavaFX: start");
     System.out.println(Thread.currentThread().getName());

     Group rootNode = new Group();

     rootNode.getChildren().add(circle);
     rootNode.getChildren().add(rectan);

     Scene scene = new Scene(rootNode, 200, 200);

     stage.setTitle("My JavaFX Application");
     stage.setScene(scene);
     stage.show();
  }
  // -------------
  public void stop() {
    System.out.println("JavaFX: stop");
    System.out.println(Thread.currentThread().getName());
  }
}
```

Creates an instance of class testJavaFX and executes the **init()**, **start()**, and **stop()** methods. The **String[]** parameter must be provided but can be empty.

The **init()** method is optional; all declarations can be made in **start()**

We can create the nodes in the **init()** method.

Build the stage and start the execution of the JavaFX application

Declare a group of nodes

Add two nodes to the group

Place the nodes into a 200x200 pixel window

Place the scene on the stage

Open the window. Start executing JavaFX application.

The **stop()** method is optional;

In addition to the new window displayed, this program prints the following text to the console:

```
JavaFX: launch
JavaFX: init
```

```
JavaFX: - Launcher
JavaFX: start
JavaFX: Application Thread
JavaFX: stop
JavaFX: Application Thread
```

Note 1: The **main()**, **init()**, and **stop()** methods are optional. The **start()** method is sufficient for starting a JavaFX application.

Note 2: The **init()** method is running on the so-called "**launching**" thread, and the **start()** and **stop()** methods are running on a separate "**application**" thread. The output produced by the example illustrates the latter by displaying the thread names during execution of each method.

As mentioned, same application can be written without the main(), init(), and stop() methods:

```
import javafx.application.*;
import javafx.stage.*;
import javafx.scene.*;
import javafx.scene.shape.*;

public class testJavaFX extends Application {

  public void start(Stage stage) {

     Group rootNode = new Group();

     Circle    circle = new Circle (100, 40, 30);
     Rectangle rectan = new Rectangle (80, 80, 40, 40);

     rootNode.getChildren().add(circle);
     rootNode.getChildren().add(rectan);

     Scene scene = new Scene(rootNode, 200, 200);

     stage.setTitle("My JavaFX Application");
     stage.setScene(scene);
     stage.show();
  }
}
```

Layouts

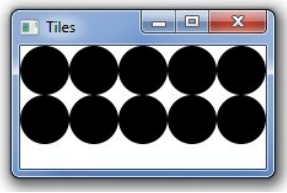
In the previous example, the location of the nodes (the circle and the rectangle) within the scene (i.e. window) was specified explicitly. The root node **Group** provides a container for holding other nodes, without any attempts to arrange them in any particular way.

Layout containers or panes can be used to allow for flexible and dynamic arrangements of the UI controls within a scene graph of a JavaFX application.

The JavaFX Layout API provides several container classes that can hold components and arrange them according to pre-defined layout models. The layout classes reside in the **javafx.scene.layout.Pane** package. We will show you an example of using one of these layouts – **TilePane**. It places the nodes in uniformly sized cells:

```
import javafx.application.*;
import javafx.stage.*;
import javafx.scene.*;
import javafx.scene.layout.*;
import javafx.scene.shape.*;

public class testJavaFX extends Application {

  public void start(Stage stage) {

     TilePane tile = new TilePane();

     for (int i = 0; i < 10; i++) {
        tile.getChildren().add(new Circle(20));
     }

     Scene scene = new Scene(tile, 200, 100);
     stage.setTitle("Tiles");
     stage.setScene(scene);
     stage.show();
  }
}
```

This program displays the following window:



Event Handling

When you interact with a GUI application, each action triggers an event. Mouse move, mouse click, keyboard input, scroll, screen touch, swipe, etc. – all are examples of events that can be captured by the JavaFX application and processed.

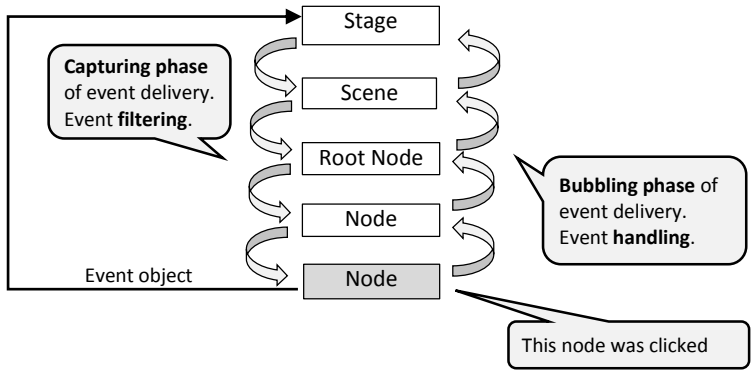
The event handling mechanism employed by JavaFX is easier explained through an example. Let's say that we clicked on a button on the screen. This action triggers a "**mouse click**" event. In JavaFX, an event is an instance of the **javafx.event.Event** class or any subclass of **Event**, so JavaFX creates a corresponding **Event** object for the event. The event object then "travels" through the object hierarchy of the JavaFX application, from the Stage object to the target node that was clicked (a Button), and then backwards. By "travel" we mean "presented for processing". The first path is called "**capturing phase**":

Stage → Scene →Root Node → *branch nodes* (if any) → Target Node

The second path is called "**bubbling phase**":

Target Node → branch nodes → Root Node → Scene → Stage

The following diagram illustrates the event handling mechanism:



Any time during the capturing or bubbling phase the event can be flagged as "consumed" by executing the **consume()** instance method of class Event. This stops further event propagation.

Presenting (i.e. delivering) an event to an object for processing does not imply that the event will be or can be processed by that object. To be able to accept

and process an event, every object must register itself as an event handler and implement some kind of event handling logic. Since events are delivered to each object twice – during the capturing (or filtering) phase, and during the bubbling (or handling) phase, the object can set up either an **event filter**, or **event handler**, or both.

All nodes (i.e. subclasses of the Node class), as well as the Stage and Scene classes define methods for registering and implementing **event filters** and **event handlers**:

```
<T extends Event> void addEventFilter (EventType<T> eventType,
                       EventHandler<? super T> eventFilter)

<T extends Event> void addEventHandler (EventType<T> eventType,
                       EventHandler<? super T> eventHandler)
```

There are also so-called **convenience** methods for registering the **event handlers** for different types of events. They are basically shorthand for the addEventHandler methods. For example, this method registers an event handler for the “mouse clicked” event:

```
void setOnMouseClicked(EventHandler<? super MouseEvent> value)
```

Let’s practice and create a couple of event handlers using the **addEventHandler()** and **setOnMouseClicked()** methods.

The setOnMouseClicked method declares one argument – an object of generic functional interface EventHandler, whose generic class parameter can be the MouseEvent or any superclass of MouseEvent (e.g. EventHandler<MouseEvent>, EventHandler<Event>, etc.).

The EventHandler functional interface declares the following method:

```
void handle(T event)
```

Next, we need to create an object of type **EventHandler<MouseEvent>**, implement its **handle()** method, and pass this object as an argument to the **setOnMouseClicked()** method. This can be accomplished in several different ways, three of which are presented below.

The first way of registering an event handler with the **setOnMouseClicked()** method is by using an **anonymous class**:

```
node.setOnMouseClicked
(
  new EventHandler<MouseEvent>()
      {
         public void handle(MouseEvent e)
         {
           … event handling logic …
         }
      }
);
```

Create an object of type EventHandler<MouseEvent>

Implement the **handle()** method of the **EventHandler** functional interface

The second way of using the setOnMouseClick() method is by using a **lambda expression**:

```
node.setOnMouseClicked
(
   e -> {… event handling logic …}
);
```

This lambda expression creates an object of type EventHandler<MouseEvent> with its **handle()** method implemented as specified in {…}

The third way of using the setOnMouseClick() method is by using a **method reference**:

```
class myClass {
   public static myClickHandler (MouseEvent e) {
      … event handling logic …
   }
   public static void main(String[] args) {
      . . .
      node.setOnMouseClicked (

         myClass::myClickHandler
      );
   }
}
```

This method reference expression creates an object of type EventHandler<MouseEvent> with its **handle()** method now pointing to the **myClickHandler()** method

Note that the **myClickHandler** method must have same signature as the **handle** method of the **EventHandler** interface, which is declared as follows:

```
void handle (T event)
```

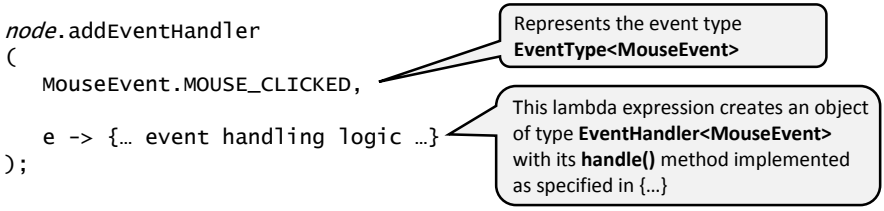
The **T** is the event type we want to handle; in our case it is the **MouseEvent**.

Now let's create an event handler using the **addEventHandler()** method. The format of the addEventHandler is this:

```
<T extends Event> void addEventHandler (EventType<T> eventType,
                      EventHandler<? super T> eventHandler)
```

The first difference of this method from the previously reviewed setOnMouseClicked method is that the EventHandler object can be of any event type, not just MouseEvent. In fact, the EventHandler can be created for the Event class and any subclass of the Event class. The second difference – we need to tell the method which event type we will be processing by passing it an object of type **EventType<T>**, where T can specify the Event class and any of its subclasses.

Using a lambda expression, we can build an event handler for the "mouse click" event as follows:

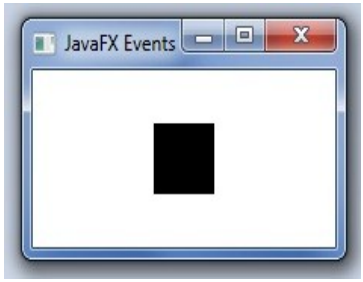


Note: The `MouseEvent.MOUSE_CLICKED` in this example is a static variable of type EventType<MouseEvent> defined in the MouseEvent class, representing the "mouse clicked" event type. Its definition is:

```
public static final EventType<MouseEvent> MOUSE_CLICKED
```

The registration of **event filters** is similar to the registration of **event handlers** with the exception that there are no **convenience** methods provided.

With the next example we will conclude this brief overview of the JavaFX API features. The example illustrates everything we have learned so far about the JavaFX event handling mechanism. When executed on a Windows system, it will display the following window:



After you click anywhere within the black rectangle, the following messages will be printed on the console:

```
Capturing phase: Stage clicked
Capturing phase: Scene clicked
Capturing phase: Group clicked
Capturing phase: Rectangle clicked

Bubbling phase: Rectangle clicked
Bubbling phase: Group clicked
Bubbling phase: Scene clicked
Bubbling phase: Stage clicked
```

To stop the JavaFX application, close the displayed window.

The JavaFX event handling example:

```
import javafx.application.*;
import javafx.stage.*;
import javafx.scene.*;
import javafx.scene.shape.*;
import javafx.event.*;
import javafx.scene.input.MouseEvent;

public class testJavaFX_events extends Application {
  public void start(Stage stage) {

     stage.addEventFilter(MouseEvent.MOUSE_CLICKED,
       e -> System.out.println("Capturing phase: Stage clicked"));

     stage.addEventHandler(MouseEvent.MOUSE_CLICKED,
       e -> System.out.println("Bubbling phase: Stage clicked"));

     Rectangle rect = new Rectangle(80, 30, 40, 40);

     rect.addEventFilter(MouseEvent.MOUSE_CLICKED,
       e -> System.out.println("Capturing phase: Rectangle clicked"));

     rect.setOnMouseClicked (
       e -> System.out.println("Bubbling phase: Rectangle clicked"));

     Group rootNode = new Group();

     rootNode.addEventFilter(MouseEvent.MOUSE_CLICKED,
       e -> System.out.println("Capturing phase: Group clicked"));

     rootNode.setOnMouseClicked (
       e -> System.out.println("Bubbling phase: Group clicked"));

     rootNode.getChildren().add(rect);

     Scene scene = new Scene(rootNode, 200, 100);

     scene.addEventFilter(MouseEvent.MOUSE_CLICKED,
       e -> System.out.println("Capturing phase: Scene clicked"));

     scene.setOnMouseClicked ( new EventHandler<MouseEvent>() {
       public void handle(MouseEvent e)
          { System.out.println("Bubbling phase: Scene clicked"); }
       } );

     stage.setTitle("JavaFX Events");
     stage.setScene(scene);
     stage.show();
  }
}
```

Reference Material

https://docs.oracle.com/javase/8/docs/technotes/guides/install/

Instructions on how to install and configure the Java platform for different operating systems.

http://www.oracle.com/technetwork/java/javase/downloads/

Java SE downloads. The latest Java Development Kit (JDK) and Java Runtime Environment (JRE) can be downloaded from this page.

http://docs.oracle.com/en/java/

The starting point for browsing the Java SE (Standard Edition), Java EE (Enterprise Edition), and Java Micro Edition Embedded documentation.

http://docs.oracle.com/javase/tutorial/

Java SE tutorials. Hundreds of complete, working examples, and dozens of lessons.

http://docs.oracle.com/javase/8/docs/api/index.html

Java SE 8 API Specifications. Complete specifications on all Java classes and interfaces.

https://docs.oracle.com/javase/specs/

Java Language and Virtual Machine Specifications.

http://docs.oracle.com/javase/8/javafx/api/toc.htm

JavaFX API Specifications. All “javafx” packages are presented here.

Index