

# OOP: Syllabus

- **Introduction to OOP (12 Hrs)**
- **Inheritance, Interfaces and Packages (10 Hrs)**
- **Exception Handling and Multithreading (10 Hrs)**
- **Generics and GUI with JavaFx (12 Hrs)**
- **CERT JAVA Coding Standard (4 Hrs)**

# Chapter 1

## Java Programming Fundamentals

# Programming Languages

- High-level languages are easy for humans to read and write (e.g., C, C++ Java).
- Low-level languages are closer to the hardware (e.g., machine or object code).
- *Compilers* are programs that translate high-level languages into object code.

# The Origins of Java

- Java is a high-level language conceived by James Gosling, Patrick Naughton, Chris Warth, Ed Frank, and Mike Sheridan at Sun Microsystems in 1991.
  - Initially called “Oak”, but was renamed “Java” in 1995.
- Java syntax is similar to the older languages C & C++.
  - It was not designed to replace C++.
- **Original motivation** – need for a platform-independent language for creating softwares to be embedded in various consumer electronic devices / **Internet**
- ([Java SE](#)) lets you develop and deploy Java applications on [desktops](#) and servers
  - The newest version is Java SE 14.

# Progress Check

- Java is useful for the Internet because it can produce \_\_\_\_\_ programs
- Java is the direct descendant of what languages?

# Java Buzzwords

- **Simple**
  - easy to learn, inherits c/c++ syntax, left out difficult constructs like pointers, memory management.
- **Pure Object oriented Language-**
  - classes and objects
- **Java is case sensitive language-**
  - variable val is different from Val, VAL, VaL
- **Robust-**
  - Ability to create robust, reliable programs
    - Exception handling
    - Strictly typed language  
( checks for errors at both compile and run time)
    - Easy memory management  
( automatic deallocation-garbage collection)

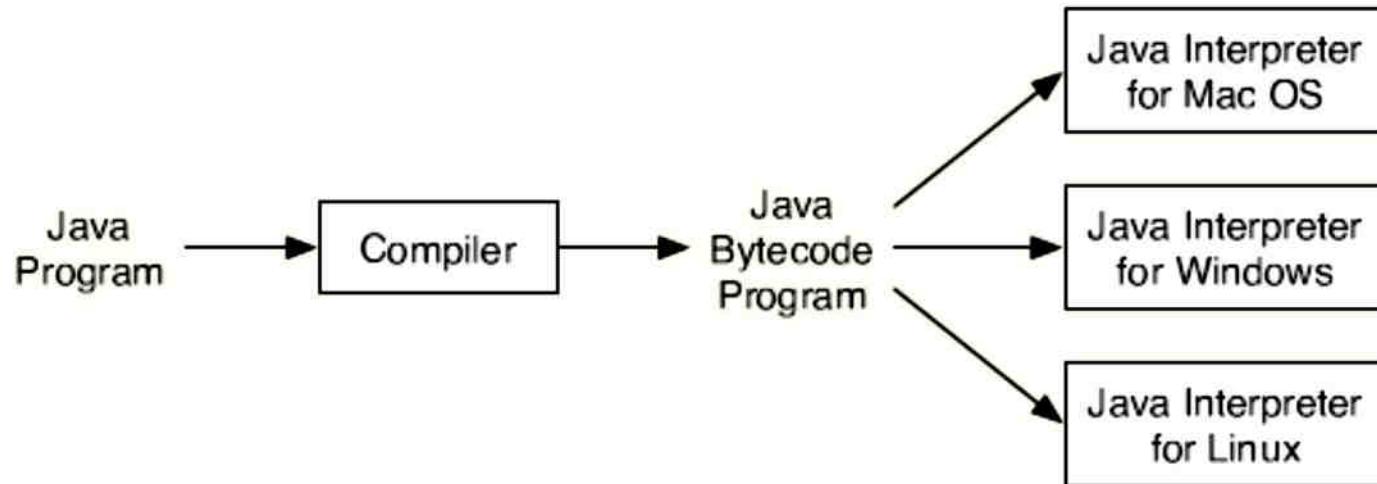
# Java Buzzwords

- **Multi threaded**
  - write programs that do many things simultaneously
- **Architecture neutral**
  - compiled java programs will run on variety of processors using various OSs (The goal- “ Write once, run any where, any time, forever”)
- **Both compiled and interpreted**
  - enables creation of cross-platform programs
- **Dynamic**
  - Resolves type information at run time

# Portability

- Portability is the major aspect of Internet
  - because many different types of computers and OSs are connected to it.
- Because Java is architecture neutral,
  - Java programs are portable.
  - They can be run on any platform without being recompiled.

# Java's solution:

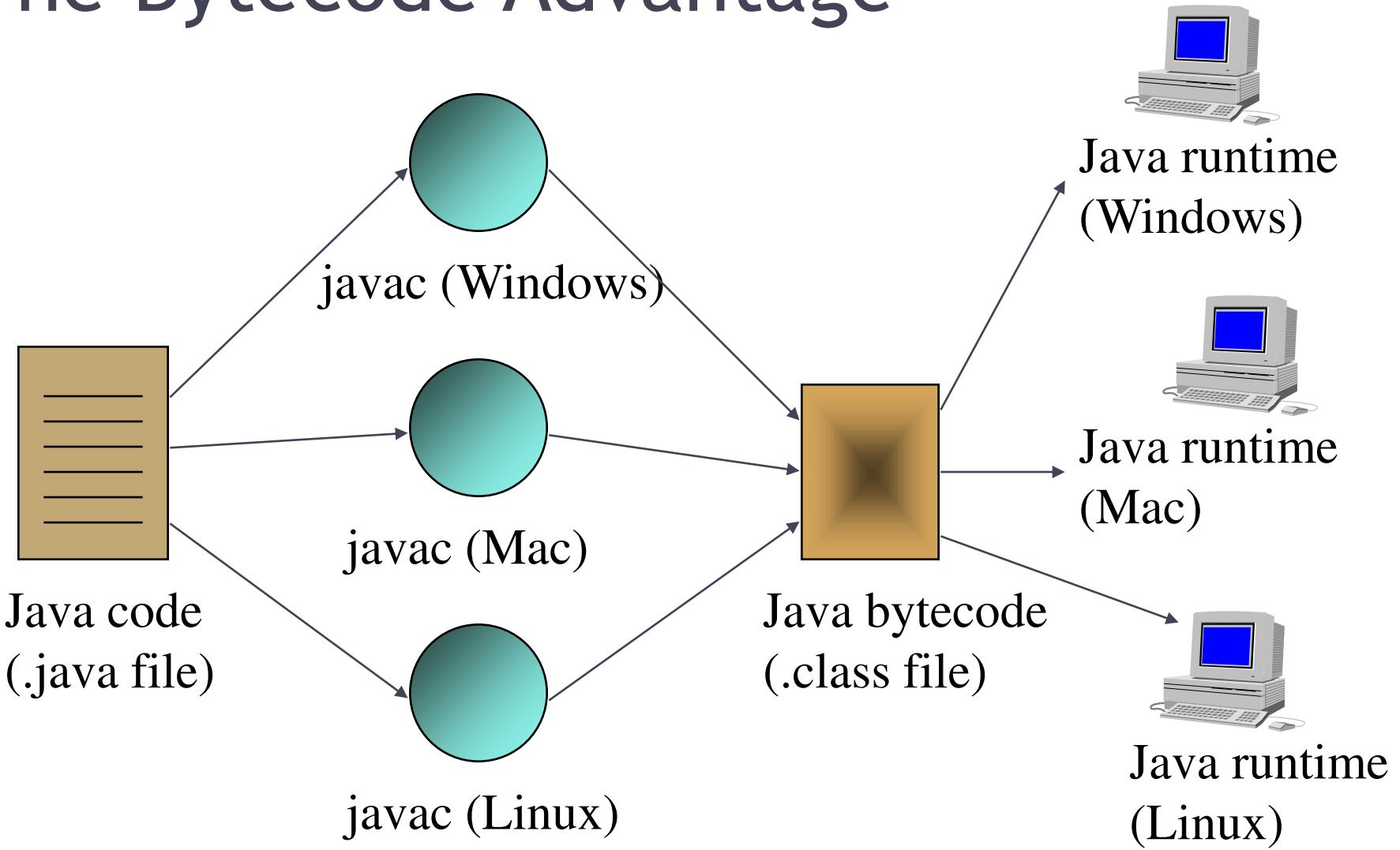


- Execution of **bytecode** by **JVM**
  - is the easiest way to create portable programs by implementing java interpreter for each platform.

# Java bytecode

- The output of a Java compiler(javac) is
  - not executable code ; rather, it is **bytecode**
- **Bytecode** is a highly optimized set of instructions
  - designed to be executed by the Java run-time system, ***Java Virtual Machine (JVM***

# The Bytecode Advantage

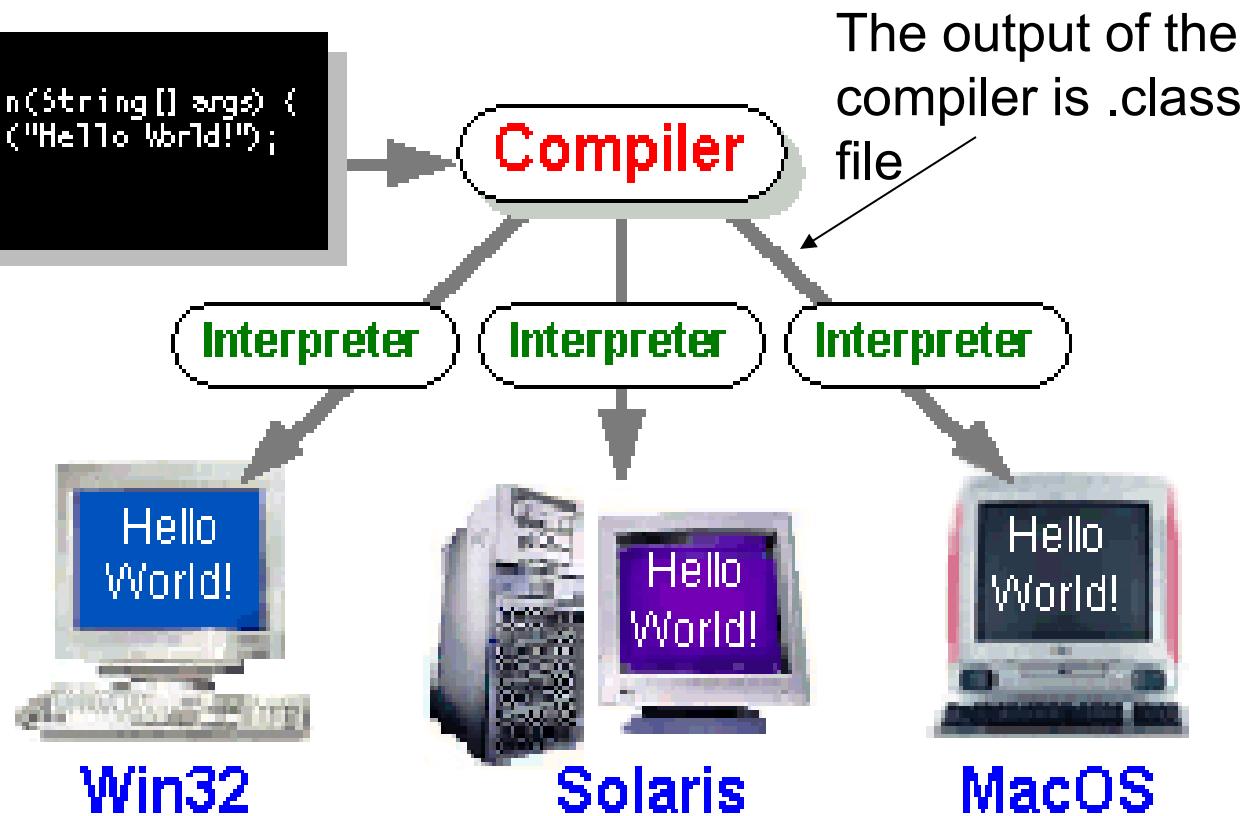


# Java's solution:

## Java Program

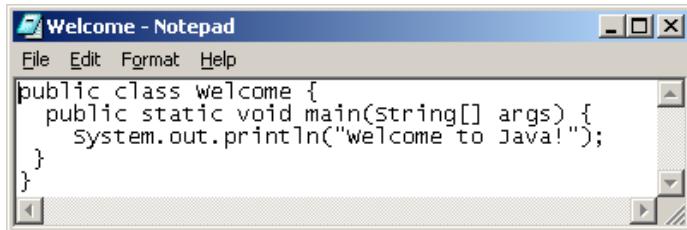
```
class HelloWorldApp {  
    public static void main(String[] args) {  
        System.out.println("Hello World!");  
    }  
}
```

HelloWorldApp.java



The Interpreter's are sometimes referred to as **Java Virtual Machines**

# Creating, Compiling, and Running Programs



Source code (developed by the programmer)

```
public class Welcome {  
    public static void main(String[] args) {  
        System.out.println("Welcome to Java!");  
    }  
}
```

Byte code (generated by the compiler for JVM to read and interpret, not for you to understand)

```
...  
Method Welcome()  
0 aload_0  
...  
  
Method void main(java.lang.String[])  
0 getstatic #2 ...  
3 ldc #3 <String "Welcome to  
Java!">  
5 invokevirtual #4 ...  
6 ...
```

Create/Modify Source Code

Saved on the disk

Source Code

Compile Source Code  
i.e., javac Welcome.java

stored on the disk

Bytecode

Run Bytecode  
i.e., java Welcome

Result

If runtime errors or incorrect result

If compilation errors

# Java Compilers and the JVM

- Java compilers do not generate machine code for a CPU.
- Java compilers generate machine code for the JVM (Java Virtual Machine).
- The JVM machine code (called *bytecode*) is executed by a JVM interpreter program on each computer.

# Java Runtime Environment (JRE)

- a set of programming tools for developing Java applications:
  - Java Virtual Machine (JVM),
  - core classes ( libraries),
  - supporting files.

# Progress Check

- What is a Java bytecode?
- Which two internet programming problems can be solved using bytocode?

# Object-Oriented Programming

- In POP. Programs are organized around code (code acting on data)
- In OOP, code is organized around the data. (data controlling access to code)
- In OOP, we define the **data and the routines** that are permitted to act on the data.
- In Java, **a class** encapsulates the data and the routines acting on the data.

# The **Key traits** of Object Oriented Programming

- *Encapsulation*
- *Polymorphism*
- *Inheritance*

# Encapsulation

- ***Encapsulation*** - is the mechanism
  - that binds together code and the data it manipulates,
  - and keeps both safe from outside interference and misuse
- ***Encapsulation***
  - Self-contained **black box**
  - **Object** supports encapsulation
- Encapsulation: Java
  - Basic Unit of Encapsulation: **Class**
  - Object: An instance of Class
  - **Clasa Members:** Instance variables **and** methods

# Polymorphism

- ***Polymorphism*** - is a feature that
  - allows **one interface to be used for a general class of actions.**
  - The specific action is determined by the exact nature of the situation.
- ***Example***"
  - **get** method – to retrieve next item from the list
    - List: FIFO, FILO, Based on some priority
- In general
  - One interface, multiple methods
    - A generic interface to a group of related activities
- Helps to reduce complexity
  - Compiler should select specific action as it applies to each situation

# Inheritance

- ***Inheritance*** - the process by which one object can acquire the properties of another object
- Supports Hierarchical(top-down) classification
  - Example: Food <- Fruit <- Apple <- Red Delicious Apple
- **Using Inheritance**
  - An object need only define its unique qualities
    - It can inherit its general attributes from its parent

# Progress Check

- Name the principles of OOP
- What is the basic unit of encapsulation in Java?

# The Java Development Kit (JDK)

- To compile and run java programs JDK must be installed in the m/c, which can be downloaded using the below link
- [www.oracle.com/technetwork/java/javase/downloads/index.html](http://www.oracle.com/technetwork/java/javase/downloads/index.html)
- Provide two primary programs
  - ✓ Java compiler: **javac**
  - ✓ Application launcher or java interpreter : **java**
- It runs in the command prompt environment and use command line tools

```
public class First
{
    public static void main(String[] args)
    {
        System.out.println("First Java application");
    }
}
```

Figure 1-4 The First class

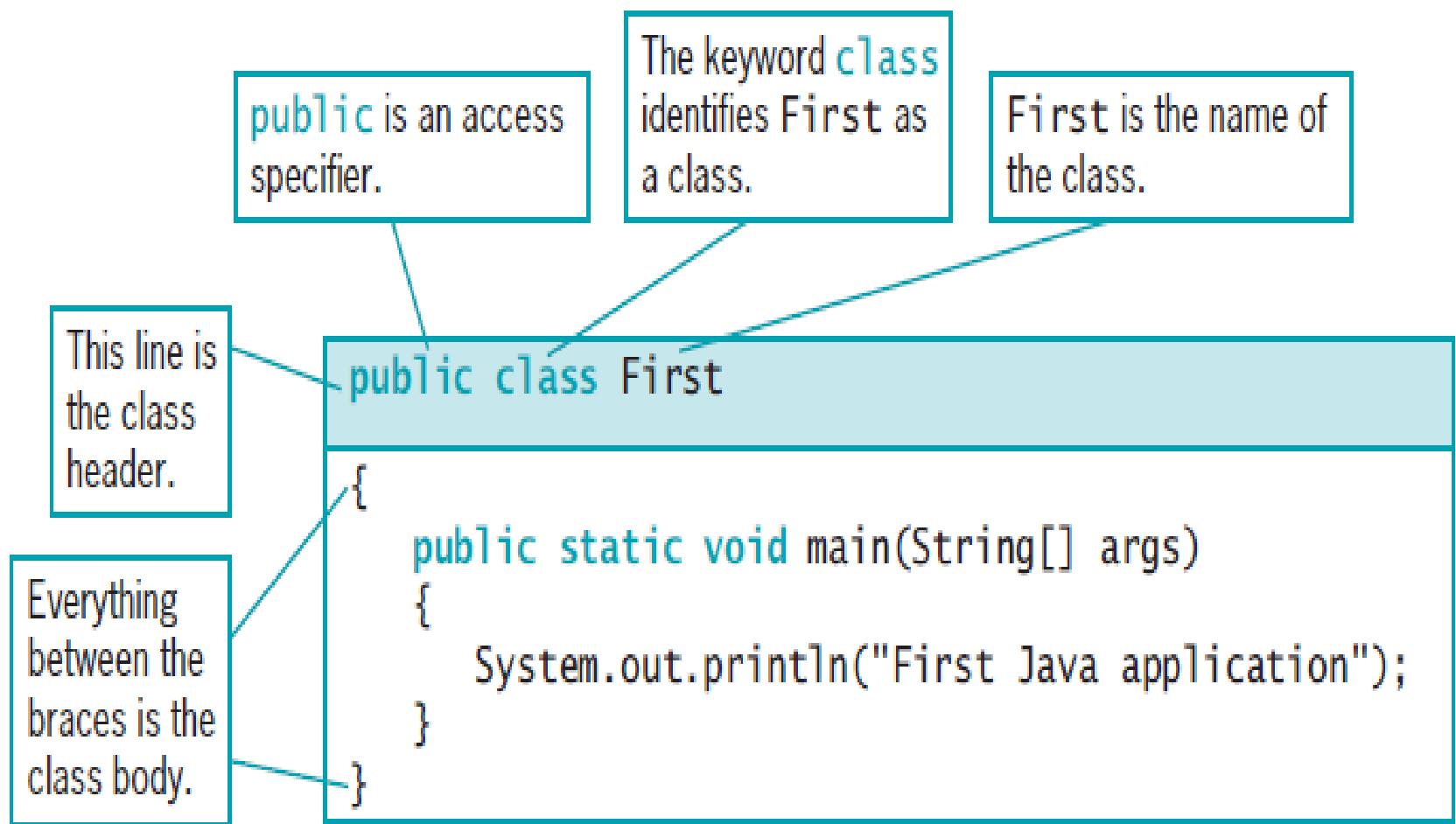


Figure 1-6 The parts of a typical class

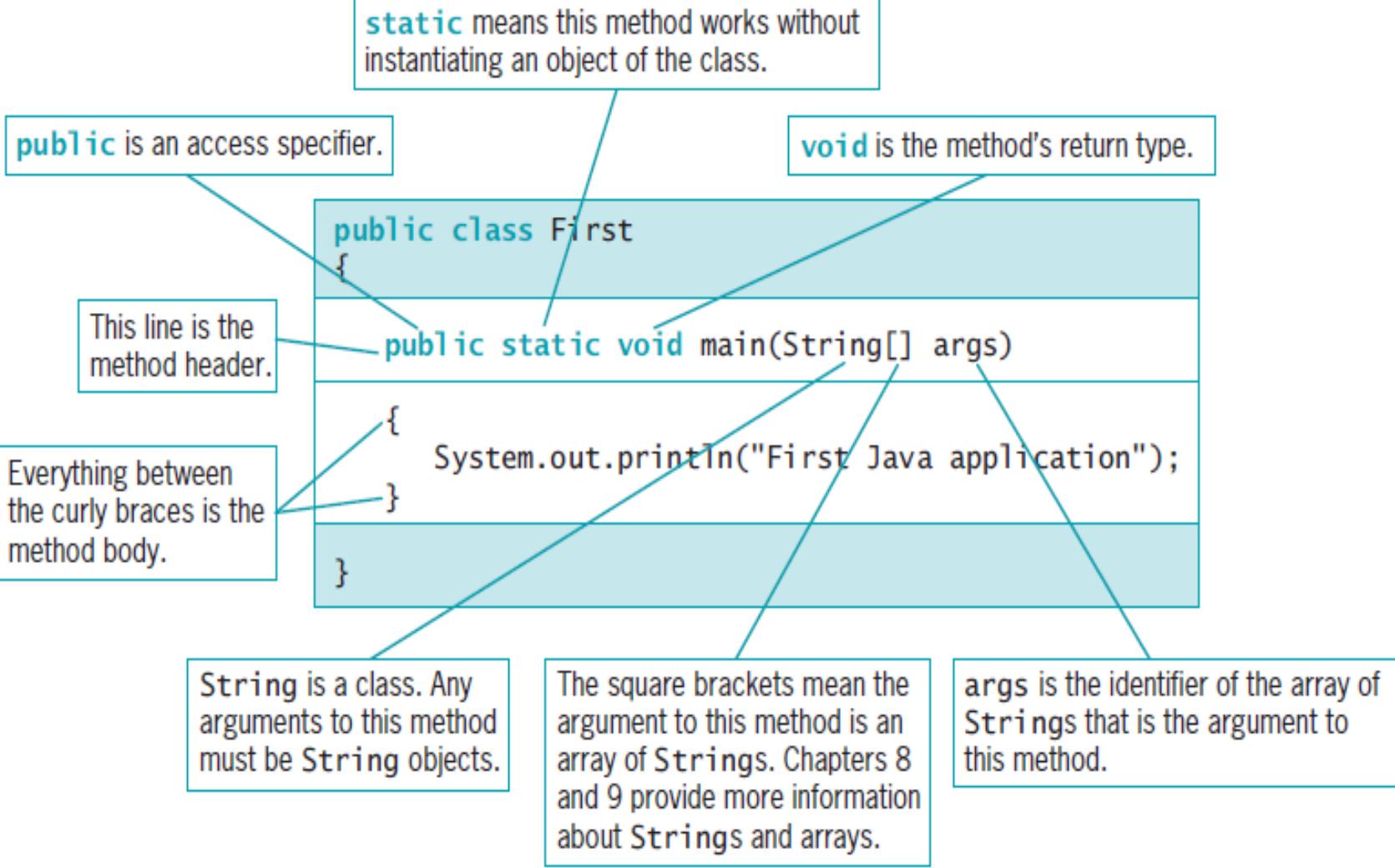


Figure 1-7 The parts of a typical main() method

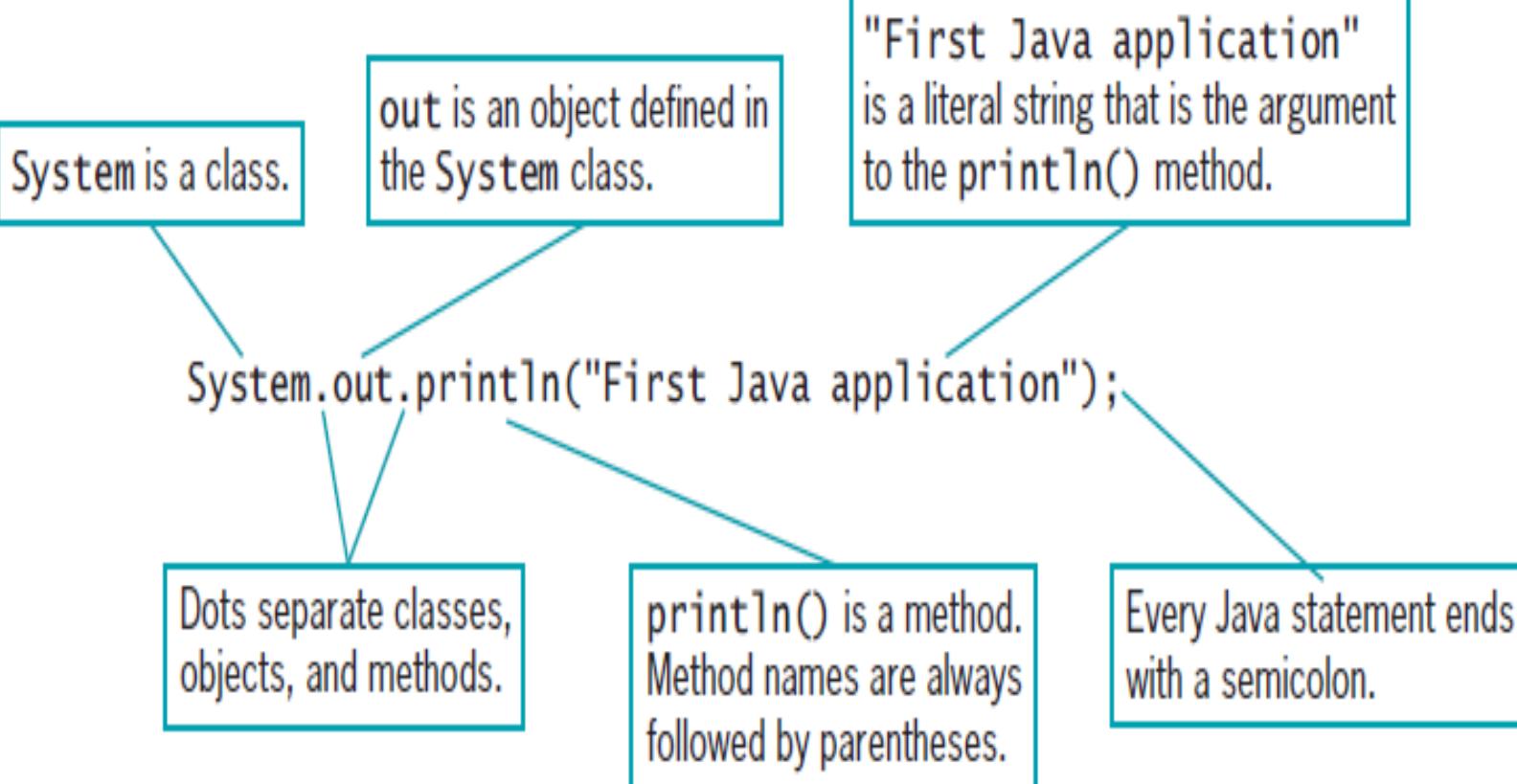


Figure 1-5 Anatomy of a Java statement

# Comments: Single vs Multi-line

```
/*
This is a simple Java program.
Call this file Example.java.
*/
Public class Example {
    // A Java program begins with a call to main().
    public static void main(String[] args) {
        System.out.println("Java drives the Web.");
    }
}
```

# First Simple Java Program

```
/*
This is a simple Java program.
Call this file Example.java.
*/
Public class Example {
    // A Java program begins with a call to main().
    public static void main(String[] args) {
        System.out.println("Java drives the Web.");
    }
}
```

- When a class member is preceded by **public**, then that member may be accessed by code outside the class in which it is declared
- The keyword **static** allows **main( )** to be called without having to instantiate a particular instance of the class
- The keyword **void** simply tells the compiler that **main( )** does not return a value

# Progress Check

- Where does a Java program begin execution?
- What does System.out.println() do
- What is the name of the java compiler? What do you use to run java program?

# A Second Example

```
class Example2 {  
    public static void main(String[] args) {  
        int var1; // this declares a variable  
        int var2; // this declares another variable  
  
        var1 = 1024; // this assigns 1024 to var1  
        System.out.println("var1 contains " + var1);  
  
        var2 = var1 / 2;  
        System.out.print("var2 contains var1 / 2: ");  
        System.out.println(var2);  
    }  
}
```

# A Third Example

```
class Example3 {  
    public static void main(String[] args) {  
        int w; // declare an int variable  
        double x; // declare a floating-point variable  
  
        w = 10; // assign w the value 10  
        x = 10.0; // assign x the value 10.0  
        System.out.println("Original value of w: " + w);  
        System.out.println("Original value of x: " + x);  
        System.out.println(); // print a blank line  
  
        // now, divide both by 4  
        w = w / 4;  
        x = x / 4;  
        System.out.println("w after division: " + w);  
        System.out.println("x after division: " + x);  
    }  
}
```

# Converting Gallons to Liters

```
class GalToLit {  
    public static void main(String[] args) {  
        double gallons; // holds the number of gallons  
        double liters; // holds conversion to liters  
  
        gallons = 10; // start with 10 gallons  
  
        liters = gallons * 3.7854; // convert to liters  
        System.out.println(gallons + " gallons is " +  
                            liters + " liters.");  
    }  
}
```

# The if Statement

- Simplest form:

*if ( condition ) statement;*

- Example:

```
if (3 < 4) System.out.println("yes");
```

- Relational operators:

<, >, <=, >=, ==, !=

# Example of if Statements

```
class IfDemo {  
    public static void main(String[] args) {  
        int a, b;  
        a = 2;  
        b = 3;  
  
        if(a < b)  
            System.out.println(a + " is less than " + b);  
  
        // this won't display anything  
        if(a == b)  
            System.out.println("you won't see this");  
    }  
}
```

# The for Loop

- General form:  
$$\text{for}(\textit{initialization} ; \textit{condition} ; \textit{iteration}) \\ \quad \textit{statement};$$
- *Initialization* is normally for setting a loop control variable.
- *Condition* is for stopping the loop.
- *Iteration* is normally for updating the loop control variable.

# Example of a for Loop

```
class ForDemo {  
    public static void main(String[] args) {  
        int count;  
  
        for(count = 0; count < 5; count = count+1)  
            System.out.println("This is count: " + count);  
  
        System.out.println("Done!");  
    }  
}
```

# Code Block

- A list of statements inside braces
- Does not need to end in a semicolon
- Example:

```
if (w < h) {  
    v = w*h;  
    w = 0;  
}
```

# Comments in Java

- Java supports **single line** and **multi-line** comments very similar to C++.

## Example 1

```
//This is an example of single line comment
```

## Example 2

```
/*
    This is my first java program.
    This will print 'Hello World'
    as the output
*/
```

# Documentation comment

- This type of comment is used to produce an HTML file that documents your program
- Example 3  
The documentation comment begins with a `/**` and ends with a `*/`
- They can be extracted into an HTML file using JDK's javadoc command.

# Indentation Practices

- The Java compiler doesn't care about indentation.
- Use indentation to make your code more readable.
  - ***Indent one level*** for each opening brace and
  - ***move back out*** after each closing brace.

# Progress Check

- How is a block of code created? What does it do?
- In Java statements are terminated by a \_\_\_\_\_
- All Java statements must start and end on one line. T/F

# Java Keywords

<b>abstract</b>	<b>assert</b>	<b>boolean</b>	<b>break</b>	<b>byte</b>	<b>case</b>	<b>catch</b>	<b>char</b>	<b>class</b>	<b>const</b>
<b>continue</b>	<b>default</b>	<b>do</b>	<b>double</b>	<b>else</b>	<b>enum</b>	<b>extends</b>	<b>final</b>	<b>finally</b>	<b>float</b>
<b>for</b>	<b>goto</b>	<b>if</b>	<b>implements</b>	<b>import</b>	<b>instanceof</b>	<b>int</b>	<b>interface</b>	<b>long</b>	<b>native</b>
<b>new</b>	<b>package</b>	<b>private</b>	<b>protected</b>	<b>public</b>	<b>return</b>	<b>short</b>	<b>static</b>	<b>strictfp</b>	<b>super</b>
<b>switch</b>	<b>synchronized</b>	<b>this</b>	<b>throw</b>	<b>throws</b>	<b>transient</b>	<b>try</b>	<b>void</b>	<b>volatile</b>	<b>while</b>

**const** and **goto** are reserved but not used.

Reserved words : **true**, **false** and **null**

# Java Identifiers

- An identifier is a name given to
  - a method, variable, class or other user-defined item.
- Identifiers are one or more characters long.
- The dollar sign(**\$**), the underscore( **\_** ), any letter of the alphabet(**a-z, A-Z**), and any digit(**0-9**) can be used in identifiers.
- Should not be a keyword or reserved word

# Java Identifiers

- The **first character** in an identifier cannot be a digit.
  - e.g. **12a** is an **invalid** identifier
- **Java is case sensitive:** Upper case and lower case are different.
  - **myvar** and **MyVar** are different identifiers.
- Legal identifiers
  - **Test , x, y2, maxLoad, sample34, \$up, \_top**

# Java class Libraries

- A package which gets imported automatically to all the java programs
  - ‘**java.lang**’
- **java.lang**’ package has
  - many built-in classes and methods ( **System** and **String** classes) and
  - **println** and **print** methods.
- Different packages for
  - **I/O , GUI, Event handling, networking, multithreading, exception handling etc.**

# Exercises

- Which of the following variable names is invalid?
  - A. count
  - B. \$count
  - C. count27
  - D. 67count
- What is wrong with each of the following commands
  - javac Example.class
  - Java Example.class

# Exercises

- Assume x is a variable that is declared as type int. What is wrong with each of the following statements?
  - A. `x=3.5;`
  - B. `if(x=3) x=4;`
  - C. `x= “34”;`
- Write a program that prints out the first 20 squares(1,4,9,16.....400), one per line. Use a for loop
- Modify previous exercise so that it prints the sum of the first 20 squares( $1+4+9+\dots+400$ ).

# Chapter 2

Introducing Data Types and  
Operators

# Java Types

- Java is a strongly typed language; that is, the compiler type-checks all statements.
- Two categories of types:
  1. **Primitive type ( 8 types)**
  2. **Object/Reference type**

# Java Primitive Types

- **boolean** (true/false)
- **char** (characters)
- **float** (single-precision floating point numbers)
- **double** (double-precision floating point)
- **byte** (8-bit integers)( useful when we work with raw binary data)
- **short** (16-bit integers)
- **int** (32-bit integers) most widely used(index, loop var)
- **long** (64-bit integers) used when int is not large enough

# Ranges of Values for Integer Types

- Whole valued signed numbers
- byte: -128 to 127
- short: -32,768 to 32,767
- int: -2,147,483,648 to 2,147,483,647
- long: -9,223,372,036,854,775,808 to  
9,223,372,036,854,775,807

Java doesn't support unsigned integers

# Example Using long Integers

```
class Inches {  
    public static void main(String[] args) {  
        long cubicInches, inchesPerMile;  
  
        // compute the number of inches in a mile  
        inchesPerMile = 5280 * 12;  
        // compute the number of cubic inches  
        cubicInches = inchesPerMile * inchesPerMile *  
                      inchesPerMile;  
        System.out.println("There are " + cubicInches +  
                           " cubic inches in a cubic mile.");  
    }  
}
```

Output: There are 254358061056000 cubic inches in a cubic mile.

# Floating Point Types

## Numbers with fractional precision

- float
  - 32 bits wide
  - approximately 7 decimal places of accuracy
  - range of values is approximately  
 $-3.4 \times 10^{38}$  to  $+3.4 \times 10^{38}$ .
- double:
  - 64 bits wide
  - approximately 15 decimal places of accuracy
  - range of values is approximately  
 $-1.8 \times 10^{308}$  to  $+1.8 \times 10^{308}$

double is most widely used (all math function in java library use double)

Math.sqrt(double)

# Examples Using Floating Point Type

```
double x;  
x = 3.1416;  
x = 0.;  
  
x = 6.02E23; // 6.02 x 1023  
x = -3.6E-4; // -3.6 x 10-4
```

# Characters

- Java uses *Unicode*.
- A **char** uses 16 bits with a range of values of 0 to 65,535. ( defines a fully international characters set( Latin, Greek, Arabic etc.)
- Each value represents a different character.
- The ASCII character set is a subset. (0-127)
- Use single quotes to denote characters.
- Example:

```
char ch;  
ch = 'A';  
ch++; // now ch = 'B'  
ch = 90; // now ch = 'Z'
```

# Booleans

- Represents true/false values
- The words **true** and **false** are reserved words.
- Examples:

```
boolean b;  
b = true;  
if (b) System.out.println("yes");  
b = (3 > 4); // assigns false to b  
System.out.println(3>4); // displays false
```

```
if ( 1 )  
// error, expects boolean const or expression
```

# Literals

- Literals refer to fixed values or constants.
- Examples: 'A', 23, 23.45, true, “true”, 10L, 4.5F
- The only **boolean literals** are **true** and **false**.
- **Floating point literals** include a decimal point and/or an exponent.
- float f=2.7f;
- double d=2.7;
- double d=1.234E2; //  $1.234 \times 10^2$
- d= 1.234e-2; //  $1.234 \times 10^{-2}$

# Character Literals

- Surround a character with single quotes.
- Special characters need escape sequences:

single quote: \'

tab: \t

double quote: \"

backspace: \b

backslash: \\

newline: \n

carriage return: \r

form feed: \f

- **Example:**

```
char ch;
```

```
ch = '\t'; ch='\"'; // assigns single quote ch
```

# String Literals

- Surround a sequence of characters with double quotes.
- **Examples:**

```
String s;  
s = "abc";  
s = "abc\ndef";  
s = ""; // empty string
```

# Declaring Variables

- To declare a variable use the form:  
*type variablename;*
- You can declare several variables of the same type at once:  

```
int x, y, z;
```
- You can **initialize** variables when you declare them:  

```
int x = 3;      int y = 4, z = 5, w;
```

  - **Dynamic initialization at run time**
  - ```
float radius=4.5f;
```
  - ```
float volume=3.1416*radius*radius; // dynamic initialize
```

# Scope of Variables in Methods

- A variable can be declared within any code block in a method.
- The *scope* of a variable is the part of the program in which the variable can be used.
- In general, a variable's scope consists of *the code from where it is declared to the end of the code block in which it was declared*.
- Lifetime of a var is confined to its scope

```
class ScopeDemo
{
    public static void main(String a[] )
    { int x=2;// known to all code in main
        if( x==2)
        { //starts new scope
            int y=3; // y is created and known only
                      //to this scope
            System.out.println( x+” “ +y); // 2 3
        }
        y=7; // error because it got destroyed
        x=5; //ok
    }
}
```

# Arithmetic Operators

(Applied to built-in numeric type and char( subset of integer))

- addition: +
- subtraction: -
- multiplication: \*
- division: /
- modulus: % ( remainder of division)
  - (works with both integers and floating point numbers  
sign of the result is same as sign of the I operand )
- increment: ++ (postfix , prefix)
- decrement: -- (postfix , prefix)

# Arithmetic Operator Examples

- $9 + 4$  yields 13
- $9 - 4$  yields 5
- $9 * 4$  yields 36
- $9 / 4$  yields 2 (when two integers are divided, the remainder is thrown away)
- $9.0/4$  yields 2.25
- $9 \% 4$  yields 1 (the remainder when 9 is divided by 4)
- $-9 \% 7.5$  yields -1.5
- $x++$  and  $++x$  increment the value of  $x$  by 1
- $x--$  and  $--x$  decrement the value of  $x$  by 1

# Relational Operators

The outcome is boolean value

Operator	Meaning
<code>==</code>	equal to
<code>!=</code>	not equal to
<code>&gt;</code>	greater than
<code>&lt;</code>	less than
<code>&gt;=</code>	greater than or equal to
<code>&lt;=</code>	less than or equal to

# Relational operators

- Can be applied to all numeric types and char
- int i1,i2; char ch1,ch2; String s1,s2;float f1,f2;
- if ( i1==i2), if (ch1 !=ch2), if (s1==s2), if ( f1>f2)
- == and != ( all objects can be compared for equality or no equality)
- boolean b1=true,b2=false;// only for equality or non  
if ( b1==b2) // ok  
if (b1 !=b2)// ok  
if(b1>b2) or if (b1<b2)// not ok as they are not ordered

# Logical Operators

(Operands must be of boolean type, Result is also boolean type)

Operator	Meaning
&	AND
	OR
^	XOR (exclusive OR)
	short-circuit OR
&&	short-circuit AND
!	NOT

# Truth Table

p	q	p & q	p   q	p ^ q	! p
False	False	False	False	False	True
True	False	False	True	True	False
False	True	False	True	True	True
True	True	True	True	False	False

# Short circuit Logical operators( &&, ||)

- The only difference between normal and short circuit version is that
  - the normal operands will always evaluate each operand,
  - but short circuit versions will evaluate the second only when necessary.

E.g. if( dr !=0 & (nr/dr >10))

//whether I opr is true/ false it evaluates second opr,  
may cause divide by zero error

if( dr !=0 && (nr/dr >10))

- No risk of causing run time error
- Saves time

# Assignment Operator

- General form:

*var = expression;*

- Variations:

x = y = z = 3;

x += 10; // adds 10 to x

- Shorthand Assignment Operators( compound)  
(compact ,more efficient bytecode can be generated)

=	-=	*=	/=
%=	&=	!=	^=

# Automatic Type Conversions ( Widening conversion)

- A variable of one type A can be assigned a value of another type B if:
  - the two types are compatible
  - the type A is larger than the type B
- In that case, the value is automatically converted from type B to type A.
- **Example:**

```
double x = 3;      // the integer 3 is converted to
                   // the double 3.0

doubleVar=intVar;
intVar=doubleVar; // not ok
doubleVar=longVar; // ok
longVar=doubleVar; // not ok
```

# USING A CAST

## (Narrowing Conversion)

- If automatic type conversion doesn't work, use a type cast.
- **Example:**

```
int x = (int) 3.14; // assigns 3 to x
```

```
byte b = (byte) 256; // assigns 0 to b
```

```
byte b1 = (byte) 128; // assigns -128 to b1
```

# Operator Precedence

	++ (postfix)	-- (postfix)					
High	++ (prefix)	-- (prefix)	~	!	+ (unary)	- (unary)	(type-cast)
	*	/	%				
	+	-					
	>>	>>>	<<				
	>	>=	<	<=	instanceof		
	==	!=					
	&						
	^						
	&&						
	? :						
Low	=	op=					

# Type conversion in Expressions

```
class Expn_typePromotion
{
    public static void main(String[] args)
    {
        byte b=42;
        char ch='a';
        Short s=1024;
        int i=50000;
        long l=3999999999l;
        float f=5.645f;
        double d=.1234;

        double res=(f*b)+(i/ch)+(d*s);
        System.out.println(res);
        double res1=l*d;
    }
}
```

```
byte b;  
int i;  
i=b*b; // ok  
b=b*b; // not ok
```

```
char ch1='a', ch2='b';  
char ch3=ch1+ch2; // not ok  
char ch3=(char)(ch1+ch2); // ok
```

# Exercise

- What is wrong with this fragment?

```
for(i=0; i<10; i++) {  
    int sum;  
    sum=sum+I;  
}  
System.out.println("Sum is: " + sum);
```

# Exercise

- Write a program that finds all of the prime numbers between 2 and 10
- Which of the following statements are legal?
  - `int x = false;`
  - `int x = 3 > 4;`
  - `int x = int y = 3;`
  - `int x = 3.14;`
  - `int x = '350'`
  - `boolean b = (boolean) 5;`
  - `char c = "3";`

# Chapter 3

## Program Control Statements

# Control Statements

- Selection statements  
(if -else if -else, switch)
- Iteration statements  
( while, do while, for, for -each)
- Jump statements  
( break, continue, return)

# The if Statement

- Complete form:

```
if ( condition )      statement;  
else                  statement;
```

- The two target statements can be code blocks.
- You can nest **if** statements to form a ladder:

```
if (x > y) System.out.println("x");  
else if (y > z) System.out.println("y");  
else if (x > z) System.out.println("z");  
else if (x == z) System.out.println("a");  
else System.out.println("z");
```

# Switch Statement General Form

```
switch( expression ) {  
    case constant1:  
        statement sequence  
        break;  
    case constant2:  
        statement sequence  
        break;  
    case constant3:  
        statement sequence  
        break;  
    ...  
    default:  
        statement sequence  
}
```

# The for Statement

- General form:

```
for ( initialization ; condition ; iteration )  
    statement;
```

- Examples:

```
for(int x = 10; x > 0; x--)  
{ System.out.println(x);  
}  
x=20; // x not known here
```

```
for(int x = 0, int y = 0; x+y < 10; x++, y++)  
    total += x + y;
```

```
for( ; ; )  
    System.out.println("Infinite loop");
```

# The while Statement

- General form:

while ( *condition* ) *statement*;

- Examples:

```
while(x < 10) {  
    System.out.println(x);  
    x++;  
}
```

```
while(true)  
    System.out.println("Infinite loop");
```

# The do-while Statement

- General form:

```
do {  
    statements;  
} while ( condition );
```

- Example:

```
char ch;  
do  
{  
    ch = (char) System.in.read();  
    System.out.println(ch);  
} while(ch != 'x');
```

# Use **break** to exit a loop

- The **break** statement can be used to exit loops.

- Example:

```
// Using break with nested loops.

class BreakLoop3
{
    public static void main(String args[])
    {
        for(int i=0; i<3; i++)
        {
            System.out.print("Pass " + i + ": ");
            for(int j=0; j<100; j++)
            {
                if(j == 10) break; // terminate loop if j is 10
                System.out.print(j + " ");
            }
            System.out.println();
        }
        System.out.println("Loops complete.");
    }
}
```

# Use break as a form of goto

```
// Using break as a civilized form of goto.  
class LabeledBreak  
{ public static void main(String args[])  
{  
    boolean t = true;  
    first: {  
        second: {  
            third: {  
                System.out.println("Before the break.");  
                if(t) break second; // break out of second block  
                System.out.println("This won't execute");  
            } // end of third  
        } // end of second  
        System.out.println("This is after second block.");  
    } // end of first  
} // end of main()  
} // end of class
```

```
// Using break to exit from nested loops
```

```
class BreakLoop4
{
    public static void main(String args[])
    {
        outer: for(int i=0; i<3; i++)
        {
            System.out.print("Pass " + i + ": ");
            for(int j=0; j<100; j++) {
                if(j == 10) break outer; // exit both loops
                System.out.print(j + " ");
            }
            System.out.println("This will not print");
        }
        System.out.println("Loops complete.");
    }
}
```

```
// This program contains an error.  
class BreakErr  
{  
    public static void main(String args[])  
    {  
        one: for(int i=0; i<3; i++)  
        {  
            System.out.print("Pass " + i + ": ");  
        }  
        for(int j=0; j<100; j++)  
        {  
            if(j == 10) break one; // WRONG  
            System.out.print(j + " ");  
        }  
    }  
}
```

# The **continue** Statement

- The **continue** statement inside a loop causes the rest of the body of the loop to be skipped and a jump to occur directly to the conditional statement to begin the next iteration of the loop.
- Example:

```
int x = 0;  
while(x < 10)  
{  
    x++;  
    if(x % 2 == 0) continue;  
    System.out.println(x);  
}
```

// Using continue with a label.

```
class ContinueLabel
{
    public static void main(String args[])
    {
        outer: for (int i=0; i<10; i++)
        {
            for(int j=0; j<10; j++)
            {
                if(j > i)
                {
                    System.out.println();
                    continue outer;
                }
                System.out.print(" " + (i * j));
            }
            System.out.println();
        }
    }
}
```

# Exercise

- Is the following fragment valid?

```
for(int i = 0; i<num; i++)
```

```
    sum += i;
```

```
    count = i;
```

# Exercise

- In the following fragment, after the break statement executes, what is displayed?

```
for(i=0; i<10; i++) {  
    while(running) {  
        if (x<y) break;  
        //...  
    }  
    System.out.println("After while");  
}  
System.out.println("After for");
```

# Exercise

- What does the following fragment print?

```
for(i=0; i<10; i++) {  
    System.out.println(i + " ");  
    if ((i%2) == 0) continue;  
    System.out.println();  
}
```

# Exercise

- Write a program that uses a loop to print a list of 100 numbers consisting of alternating 1's and -1's starting with 1.
- Write a program that prints out the pattern of stars given below:

\*

\*\*

\*\*\*

\*\*\*\*

\*\*\*\*\*

# Chapter 4

Introducing Classes, Objects, and  
Methods

# Class Fundamentals

- Classes are templates or blueprints that specify how to build objects.
- Objects are instances of a class.
- A class can be used to create any number of objects, all of the same form, but possibly containing different data.
- Well-designed classes group logically connected data with methods for acting on that data.

# Class General Form

```
AccessSpecifier class classname
{
    // declare instance variables
    datatype varname;

    // declare 1 or more constructors
    classname( parameters ) {
        // body of constructor
    }

    // declare methods/member functions
    Return_type methodname( parameters ) {
        // body of method
    }
}
```

# Class General Form explanation

- Collectively, the **methods** and **variables** defined within a class are called *members* of the class
- **Variables** defined within a class are called **instance variables** because each instance of the class contains its own copy of these variables

# Example of a Class

- **Vehicle** class declaration:

```
class Vehicle {  
    int passengers, fuelCap,  
    int mpg; //miles per gallon  
  
// gallon is a unit of measurement for liquids  
}
```

- **VehicleDemo** class declaration:

```
class VehicleDemo{  
    public static void main(String[] args) {  
        Vehicle van; // object reference is declared  
        van= new Vehicle();  
        Vehicle car = new Vehicle();  
        car.mpg = 25;  
        car.fuelCap = 12;  
    }  
}
```

# Object Creation

- First, you must declare a variable of the class type
- This **variable does not define an object** ; Instead, it is simply a variable that can *refer* to an object
- Second, you must acquire an actual, physical copy of the object and assign it to that variable. You can do this using the **new** operator
- The **new operator dynamically allocates memory** for an object and returns a reference to it:

```
Vehicle van ; // declare reference to object
```

```
van = new Vehicle(); // allocate a Vehicle object
```

```
Vehicle van = new Vehicle();
```

OR

# Reference Variables

- The local variables **van** and **car** refer to different objects.
- Both objects have the same form (with 3 instance variables **passengers**, **fuelCap**, and **mpg**), but they have their own copies of the 3 instance variables.
- To **access instance variables and methods, use the dot (.) operator:**

```
car.mpg = 25;  
car.fuelCap = 12;
```

# Assignment of References

- If you assign

```
van = car;
```

then **both variables refer to the same object.**
- In that case, if you change an instance variable's value in **van**, it will change the value in **car** as well, since they refer to the same object.
- The object that **van** previously referred to is garbage collected if no other references to the object exist.

# Assignment of References

*Note:*

*When you assign one object reference variable to another object reference variable, you are not creating a copy of the object, you are only making a copy of the reference*

# Methods

- General form:  
*return-type methodname ( parameters ) {  
    statements;  
}*
- Parameters are local variables that receive their values from the caller of the method.
- The return type can be any valid type or **void** if the method doesn't return a value.

# Example Method

- The following method can be added to the **Vehicle** class:

```
void range() {  
    System.out.println("range: " + fuelCap * mpg);  
}
```

- If the **VehicleDemo** class's **main()** method calls

```
car.range();
```

then "range: 300" will be displayed.

# Returning from a **void** Method

- Two ways to return from a **void** method:
  - when the method's closing brace is encountered
  - when a **return** statement is encountered
- Examples:

```
void sayHello() {  
    System.out.println("Hello");  
}
```

```
void sayHello() {  
    System.out.println("Hello");  
    return;  
}
```

# Returning a Value

- To return a value from a method, you must use a **return** statement of the form:  
`return value;`
- Example:

- In the **Vehicle** class:

```
int range() {  
    return fuelCap * mpg;  
}
```

- In **VehicleDemo** class's **main()** method:

```
int range = car.range();  
System.out.println(range); // prints "300"
```

# Using Parameters

- A *parameter* is a variable whose scope is the method body and whose initial value is specified by the caller.
- Example:

- In the **Vehicle** class:

```
double fuelNeeded(int distance) {  
    return (double) distance / mpg;  
}
```

- In **VehicleDemo**'s **main( )** method:

```
System.out.println(car.fuelNeeded(750));  
// prints "30.0"
```

# Constructors

- A special member function used to initialize instance variables of the class when objects are created
- Syntactically it is similar to a method and is automatically invoked when object is created
- Name is same as that of the class
- Have no return type, not even void
- Example constructor for **Vehicle** class:

```
Vehicle() { // Zero argument constructor
    fuelCap = 12;
    mpg = 25;
    passengers = 5;
}
```

# **Default constructor**

Consider **Vehicle van = new Vehicle();**

- **new Vehicle()** is calling the **Vehicle()** constructor
- When you do not explicitly define a constructor for a class, then Java provides a default constructor for the class
- The **default constructor** automatically initializes all numeric instance variables to zero

# Constructors with Parameters

- Example:

- In the **Vehicle** class:

```
Vehicle(int p, int f, int m) {  
    passengers = p; fuelCap = f; mpg = m;  
}
```

- In the **VehicleDemo**'s **main( )** method:

```
Vehicle car = new Vehicle(5, 12, 25);  
Vehicle van = new Vehicle(7, 24, 21);
```

# The Keyword **this**

- A reference variable that refers to the current object.
- **this** is an implicit argument that refers to the object on which the method is called.
- **this** is useful for making it clear that you are referring to an instance variable.
- Example in **Vehicle** class:

```
double fuelNeeded(int distance) {  
    return (double) distance / this.mpg;  
}
```

# Instance variable hiding

- It is illegal in java to declare two local variables with the same name inside the same or enclosing scopes.
- Interestingly, you can have local variables, including formal parameters to methods, which overlap with the names of the class instance variables. We can see the following example for this explanation.

## *Example:*

```
Vehicle(int p, int f, int m) {  
    passengers = p; fuelCap = f; mpg = m;  
}
```

# Instance variable hiding

- However, when a local variable has the same name as an instance variable, the **local variable hides the instance variable**. This is why passengers, fuelCap, mpg were not used as the names for the parameters to the Vehicle() constructor inside the Vehicle class. If they had been, then **passengers** would have referred to the formal parameter, hiding the **instance variable passengers**.

*Example:*

```
Vehicle(int passengers, int f, int m) {  
    passengers = passengers; // passengers refers to formal parameter  
    fuelCap = f;  
    mpg = m;  
}
```

# Instance variable hiding

- While it is usually easier to simply use different names, there is another way around this situation. Because this lets you refer directly to the object, you can use it to resolve any name space collisions that might occur between instance variables and local variables.

// Use this to resolve **name-space collisions**.

```
Vehicle(double passengers, double fuelCap, double mpg) {  
    this. passengers = passengers;  
    this. fuelCap= fuelCap;  
    this.mpg = mpg;  
}
```

# Progress Check

1. What is the difference between a class and an object?
2. How is class defined?
3. What two things does a class contain?
4. What does new do?
5. Each object has its own copies of the class's \_\_\_\_\_
6. What is this?
7. When is the constructor executed?
8. Does the constructor have a return type?
9. Can the constructor have one or more parameters?
10. What happens when one reference variable is assigned to another?

# Garbage Collection

- In some languages, such as C++, dynamically allocated objects must be manually released by use of a **delete** operator
- Java takes a different approach; it handles deallocation for you **automatically**
- The technique that accomplishes this is called **garbage collection**

# Garbage Collection

- When no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed
- There is no explicit code need to destroy objects as in C++

# The finalize( ) Method

- Sometimes an object will need to perform some action when it is destroyed
- `finalize()` is a method of `Object` class which is the super class for every class you create
- For example, if an object is holding some non-Java resource such as a file handle or window character font, then you might want to make sure these resources are freed before an object is destroyed

# The finalize( ) Method

- By using finalization, you can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector
- To add a finalizer to a class, you simply define the **finalize( )** method
- Inside the **finalize( )** method you will specify those actions that must be performed before an object is destroyed

# The finalize( ) Method

- The finalize( ) method has this general form:

protected void finalize()

```
{  
// finalization code here  
}
```

- The keyword **protected** is a specifier that prevents access to finalize( ) by code defined outside its class

# The finalize( ) Method

- It is important to understand that finalize( ) is only called just prior to garbage collection
- It is not called when an object goes out-of-scope, for example
- This means that you cannot know when—or even if—finalize( ) will be executed
- Therefore, your program should provide other means of releasing system resources, etc., used by the object

# Exercise

Q1. Define a class to represent a complex number called **Complex**. Provide the following member functions:-

- i. To assign initial values to the Complex object (using constructor).
- ii. To display a complex number in  $a+ib$  format.
- iii. To add 2 complex numbers. (the return value should be complex)

Write a main function to test the class.

Q2. Create a Swapper class with two integer instance variables x and y and a constructor with two parameters that initialize the two variables. Also include three methods: a getX( ) method that returns x, a getY( ) method that returns y, and a void swap( ) method that swaps the values of x and y. Then create a SwapperDemo class that tests all the methods

# Chapter 5

## More Data Types and Operators

# Arrays

- An *array* is a collection of variables of the same type referred to by a common name.
- Each of the variables is specified by an index.
- One way to declare an array:  
*type[ ] arrayname = new type[size];*
- Example:

```
int[] grades = new int[30];
```

In the example, **grades** is the name of the collection of 30 integer variables.

```
// array reference of type int is declared below
```

- `int[] grades;`
- `// memory is allocate dynamically to accommodate 30 integers`
- `grades = new int[30];`

# Accessing Array Variables

- You must specify an index to access a variable.
- If the array has size 30, the indices are 0 to 29.
- Example:

```
int[] grades = new int[30];  
grades[0] = 100;  
grades[29] = 0;  
grades[1] = grades[0];  
grades[2] = grades[3*2+1];  
for(int i = 0; i < 30; i++)  
    grades[i] = i+70;
```

# Array Initializers

- To create and initialize an array at the same time, you can use this form:

*type[ ] arrayname = { val<sub>1</sub>, val<sub>2</sub>, ..., val<sub>N</sub> };*

- Example:

```
int[] fourVals = {3, 1, 4, 1};
```

- This example creates an array of length 4 storing the four values 3, 1, 4, and 1.

# Bubble Sort

- You can sort an array **nums** of length **size** as follows:

```
for(int a = 1; a < size; a++)  
    for(int b = size-1; b >= a; b--)  {  
        if(nums[b-1] > nums[b])  {  
            // if out of order exchange elements  
            int t = nums[b-1];  
            nums[b-1] = nums[b];  
            nums[b] = t;  
        }  
    }
```

# Multidimensional Arrays:

- Two-Dimensional Arrays :  
An array of one-dimensional arrays.
- Examples:

```
int[][] table = new int[3][4];
table[0][1] = 3;
for(int i = 0; i < 3; i++)
    for(int j = 0; j < 4; j++)
        table[i][j] = i+j;
```

```
int[][] newTable = {{1,2},{3,4},{5,6}};
```

# Irregular Two-dimensional Arrays

```
int[][] data = new int[3][];  
data[0] = new int[1];  
data[1] = new int[2];  
data[2] = new int[4];
```

```
int[][] moreData = {{1}, {2, 3}, {4, 5, 6}};  
System.out.println(moreData.length);  
// displays no. of rows 3
```

```
System.out.println(moreData[2].length);  
// displays no. of cols in 2nd row , i.e., 3
```

# Arrays of Three or More Dimensions

The general form of a multidimensional array declaration:

# Initializing Multidimensional Arrays

- Enclosing each dimension's initializer list within its own set of curly braces.

For example, the general form of array initialization for a two-dimensional array is shown here:

```
type-specifier array_name[ ][ ] = {  
{ val, val, val, ..., val },  
{ val, val, val, ..., val },  
}
```

Here, val indicates an initialization value.

# Alternative Array Declaration Syntax

- *type[ ] var-name;*

The square brackets follow the type specifier, not the name of the array variable

- int counter[] = new int[3];
- int[] counter = new int[3];

The following declarations are also equivalent:

- char table[][] = new char[3][4];
- char[][] table = new char[3][4];

# Alternative Array Declaration Syntax

- Convenient when declaring several arrays at the same time. For example, to create three arrays
- `int[] nums, nums2, nums3;`
- This creates three array variables of type **int**. It is the same as writing
- `int nums[], nums2[], nums3[];`
- also, create three arrays

# Alternative Array Declaration Syntax

- useful when specifying an array as a return type for a method. For example,
- `int[] someMeth( ) { ... }`
- This declares that **someMeth( )** returns an array of type **int**.

# Assigning Array References

- `int nums1[] = new int[10];`
- `int nums2[] = new int[10];`
- `for(i=0; i < 10; i++) nums1[i] = i;`
- `for(i=0; i < 10; i++) nums2[i] = -i;`
- Assign an array reference
- `nums2 = nums1; // now nums2 refers to nums1`

# Assigning Array References

- `nums1: 0 1 2 3 4 5 6 7 8 9`
- `nums2: 0 -1 -2 -3 -4 -5 -6 -7 -8 -9`
- `nums2 after assignment: 0 1 2 3 4 5 6 7 8 9`
- `nums1 after change through nums2: 0 1 2 3 4 5 6  
7 8 9`

# Using the length Member

- All arrays have a read-only instance variable called **length** that contains the number of elements that the array can hold.

# Using the length Member: An Example

```
int list[] = new int[10]; int nums[] = { 1, 2, 3 };  
// a variable-length table  
int table[][] = { {1, 2, 3},{4, 5}, {6, 7, 8, 9} };  
SOP("length of list is " + list.length);  
SOP("length of nums is " + nums.length);  
SOP("length of table is " + table.length);  
SOP("length of table[0] is " + table[0].length);  
SOP("length of table[1] is " + table[1].length);  
SOP("length of table[2] is " + table[2].length);
```

# Using the length Member: An Example

```
int list[] = new int[10]; int nums[] = { 1, 2, 3 };  
// a variable-length table  
int table[][] = { {1, 2, 3},{4, 5}, {6, 7, 8, 9} };  
SOP("length of list is " + list.length); //10  
SOP("length of nums is " + nums.length);//3  
SOP("length of table is " + table.length());//3  
SOP("length of table[0] is " + table[0].length());//3  
SOP("length of table[1] is " + table[1].length());//2  
SOP("length of table[2] is " + table[2].length());//4
```

# The **for-each** Style Loop

- General form:  
*for( type iterVar : collection ) statement-block*
- *type* specifies the type, and *iterVar* specifies the name of an *iteration variable* that will receive the elements from a collection, one at a time, from beginning to end.
- Its iteration variable is “read-only” as it relates to the underlying array.
- The contents of the array can’t be changed by assigning the iteration variable a new value.

# The **for-each** Style Loop

- Example:

```
int[] data = {3, 4, 5, 6};  
// these two loops do the same thing  
for(int i = 0; i < data.length; i++)  
    System.out.println(data[i]);  
  
for(int v : data)  
    System.out.println(v);  
// no subscript needed
```

# Use for-each style for to display and sum the values of an Array

```
int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = 0; //
for(int x : nums)
{ System.out.println("Value is: " + x);
  sum += x;
}
System.out.println("Summation: " + sum);
```

# The for-each Style Loop

```
//To add only the first 5 elements.  
int nums[] = { 1,2,3,4,5,6,7,8,9};  
for(int x : nums)  
{   SOP("Value is: " + x);  
    sum += x; // stop the loop when 5 is obtained  
    if(x == 5)  
        break;  
} System.out.println("Summation: " + sum);
```

# The for-each Style Loop

```
int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
for(int x : nums)
{ System.out.print(x + " ");
  x = x * 10;
// no effect on nums and doesnot change nums
}
```

## Use for-each style for on a two-dimensional array

```
int sum = 0; int nums[][] = new int[3][5];
for(int i = 0; i < 3; i++) // give nums some values
for(int j=0; j < 5; j++)
    nums[i][j] = (i+1)*(j+1);
// Use for-each for loop to display and sum the values
• for(int x[] : nums) //Notice how x is declared
{ for(int y : x)
    { System.out.println("Value is: " + y);
        sum += y;
    }
} System.out.println("Summation: " + sum);
```

# Applying the Enhanced for

- program using **for each** loop to search an unsorted array for a value. It stops if the value is found.

```
int nums[] = { 6, 8, 3, 7, 5, 6, 1, 4 }; int val = 5;  
boolean found = false;  
for(int x : nums)  
{   if(x == val)  
    { found = true; break;  }  
}  
if(found)  
    System.out.println("Value found!");
```

# Constructing Strings

- Java strings are objects of class **String**.
- They are constructed various ways.
- Example:

```
// all 3 statements create new String objects
String s1 = "hello";
String s2 = new String(" hai ");
String s3 = new String(s2);
```

boolean equals(str)	Returns true if the invoking string contains the same character sequence as str.
int length()	Returns the number of characters in the string.
char charAt(index)	Returns the character at the index specified by index.
int compareTo(str)	Returns a negative value if the invoking string is less than str, a positive value if the invoking string is greater than str, and zero if the strings are equal.
int indexOf(str)	Searches the invoking string for the substring specified by str. Returns the index of the first match or -1 on failure.
int lastIndexOf(str)	Searches the invoking string for the substring specified by str. Returns the index of the last match or -1 on failure.

# Examples Using String Operations

```
String s1 = "abcde";
System.out.println(s1.length()); // prints "5"
System.out.println(s1.charAt(2)); // prints "c"
if(s1.compareTo("xyz") < 0)
    System.out.println("Yes"); // prints "Yes"
System.out.println(s1.indexOf("bc")); // prints "1"
System.out.println(s1.indexOf("f")); // prints "-1"
System.out.println(s1.indexOf("ef")); // prints "-1"
```

# String Properties

- **Strings are immutable;**
- characters of a String can be accessed, but can't be changed
- In JDK 7, you can use a **String** to control a **switch statement**:

```
String temp="high";
switch(temp) {
    case "high":
        System.out.println("Switch off the heater");
        break;
    case "low":
        System.out.println("wait");
        break;
}
```

## **int indexOf(str)**

Searches the invoking string for the substring specified by str.

Returns the index of the first match or -1 on failure.

## **int lastIndexOf(str)**

Searches the invoking string for the substring specified by str.

Returns the index of the last match or -1 on failure.

String s1="abcdefbcf";

```
System.out.println(s1.indexOf("bc")) ; //1  
System.out.println(s1.lastIndexOf("bc")) ;  
//6
```

## **String substring(int start, int end);**

To extract substring from index start to end-1

```
System.out.print( "hai hello".substring(4,9)); // hello
```

# Arrays of Strings

```
String strs[] = { "This", "is", "a", "test." };
System.out.println("Original array: ");
for(String s : strs)
    System.out.print(s + " ");
System.out.println("\n"); // change a string
strs[1] = "was";      strs[3] = "test, too!";
System.out.println("Modified array: "); Output
for(String s : strs)
    System.out.print(s + " ");
```

Original array:  
This is a test.

Modified array:  
This was a test, too!

**String substring(int start);**

To extract substring from index start till the end

```
System.out.print( "hai hello how r u?".substring(4));  
// hello how r u?
```

# Command Line Arguments

```
// Display all command-line information.  
class CLDemo  
{  
    public static void main(String[] args)  
    {  
        System.out.println("There are " + args.length +  
                           " command-line arguments.");  
        System.out.println("They are: ");  
        for(int i=0; i<args.length; i++)  
            System.out.println("args[" + i + "]: " + args[i]);  
    }  
}
```

# Command Line Arguments

- java CLDemo first second third
  - Output:
  -
- 
- command line args

There are 3 command-line arguments.

They are:

args[0] : first

args[1] : second

args[2] : third

# Question

- Write a program to accept 4 integers from command line and to find and display the average of the numbers.
- Do you find any differences between conventional for loop and the enhanced for loop?

# Bitwise Operators

- Can be applied to long, int, short, char and byte datatypes  
Not with boolean, float, double or class types

Operator	Result
&	Bitwise AND
	Bitwise OR
^	Bitwise exclusive OR
>>	Shift right
>>>	Unsigned shift right
<<	Shift left
~	Bitwise NOT

bitwise AND : useful to determine whether the bit is on or off

To check whether the 4<sup>th</sup> bit of num is 0 or 1

e.g 1)

```
int num=15;// 15 in binary is 1111
```

```
    if( (num & 8) !=0) // 8 in binary is 1000  
        System.out.println( "yes");
```

```
    else
```

```
        System.out.println("no");
```

```
//lower to Uppercase conversion using bitwise &
class UpCase {
    public static void main(String[] args) {
        char ch;
        for(int i=0; i < 10; i++)
        {   ch = (char) ('a' + i);
            System.out.print(ch);
            // The below statement turns off the 6th bit.
            ch = (char) ((int) ch & 65503);
            // ch is now uppercase
            System.out.print(ch + " ");
        }
    }
}
```

```
class ShowBitsInByte
{
    public static void main(String[] args)
    {   int t;   byte val;   val = 123;
        for(t=128; t > 0; t = t/2)
        {   if((val & t) != 0)
            System.out.print("1 ");
            else
                System.out.print("0 ");
        }
    }
}
```

```
String msg = "This is a test";
String encMsg = ""; String decMsg = ""; int key = 88;
// encode the message
for(int i=0; i < msg.length(); i++)
    encMsg = encMsg+(char) (msg.charAt(i) ^ key);
System.out.print("Encoded message: "+encMsg);

// decode the message
for(int i=0; i < msg.length(); i++)
    decMsg = decMsg+(char) (encMsg.charAt(i) ^ key);
System.out.print("Decoded message: "+decMsg);
```

```
byte b = -34;
for(int t=128; t > 0; t = t/2)
{
    if((b & t) != 0)      System.out.print("1 ");
    else                  System.out.print("0 ");
}
System.out.println();
// reverse all bits
b = (byte) ~b;
for(int t=128; t > 0; t = t/2)
{
    if((b & t) != 0) System.out.print("1 ");
    else System.out.print("0 ");
}
```

```
// Demonstrate the bitwise NOT.  
byte b = -34;  
for(int t=128; t > 0; t = t/2)  
{    if((b & t) != 0) System.out.print("1 ");  
    else System.out.print("0 ");  
}  
System.out.println();  
// reverse all bits  
b = (byte) ~b;  
for(int t=128; t > 0; t = t/2)  
{    if((b & t) != 0) System.out.print("1 ");  
    else System.out.print("0 ");  
}
```

# Bit Actions

p	q	p & q	p   q	p ^ q	$\sim p$
0	0	0	0	0	1
1	0	0	1	1	0
0	1	0	1	1	1
1	1	1	1	0	0

# The ? Operator

- General form:  
*condition ? expression1 : expression2*
- This entire form is an expression whose value is the value of *expression1* if *condition* is true and whose value is the value of *expression2* if *condition* is false.
- Examples:

```
int x = (3 < 4 ? 5 : 6); // assigns 5 to x
int abs = (x < 0 ? -x : x); // abs is assigned the
                             // absolute value of x
```

```
// Prevent a division by zero using the ?.
class NoZeroDiv
{ public static void main(String[] args)
{ int result;
for(int i = -5; i < 6; i++)
{ result = i != 0 ? 100 / i : 0;
if(i != 0)
    System.out.println("100/" + i + "is" + result);
}
}
```

# Questions

- Can the **for** loops that perform sorting in the **Bubble** sort program be converted into for-each style loops? If not, why not?
- Write a Java program to sort an array of strings.

# Chapter 6

A Closer Look at Methods and  
Classes

# Introducing Access Control

- Java's access specifiers are **public**, **private**, **default** and **protected**
- **protected** applies only when inheritance is involved
- When a member of a class is modified by the **public** specifier, then that member can be accessed by any other code
- When a member of a class is specified as **private**, then that member can only be accessed by other members of its class

```
/* This program demonstrates the difference
   between public and private. */
class Test {
    int a;                  // default access
    public int b;            // public access
    private int c;           // private access
    // methods to access c
    void setc(int i) {      // set c's value
        c = i;
    }
    int getc() {             // get c's value
        return c;
    }
}
```

```
class AccessTest {  
    public static void main(String args[]) {  
        Test ob = new Test();  
        // These are OK, a and b may be accessed  
        directly  
        ob.a = 10;  
        ob.b = 20;  
        // This is not OK and will cause an error  
        //ob.c = 100;           // Error!  
        // You must access c through its methods  
        ob.setc(100);          // OK  
        System.out.println("a, b, and c: " + ob.a + " " +  
                           ob.b + " " +  
                           ob.getc());  
    }  
}
```

# Passing parameters

- There are two ways that a computer language can pass an argument to a subroutine
- Call by value
- Call by reference

# Passing Arguments to Methods

- In Java, arguments are passed using *call-by-value*.
- The value of the argument is copied into the parameter of the method.
- **If the argument is primitive**, then the primitive value is copied and so **changes made to the parameter do not affect the argument**.
- If the **argument is a reference type**, then the reference is copied and so the parameter and **the argument refer to the same object**.

## Call by value example:

- When you pass a primitive type to a method, it is passed by value

```
// Simple types are passed by value.
```

```
class Test {  
    void meth(int i, int j) {  
        i *= 2;  
        j /= 2;  
    }  
}
```

## Call by value example:

```
class CallByValue {  
    public static void main(String args[]) {  
        Test ob = new Test();  
        int a = 15, b = 20;  
        System.out.println("a and b before call: " + a + " "  
+ b);  
        ob.meth(a, b);  
        System.out.println("a and b after call: " + a + " " +  
b);  
    }  
}
```

The output from this program is shown here:

a and b before call: 15 20  
a and b after call: 15 20

## Call by reference example

- Objects are passed by reference
- Changes to the object inside the method *do* affect the object used as an argument

// Objects are passed by reference.

```
class Test {  
    int a, b;  
    Test(int i, int j) {  
        a = i;  
        b = j;  
    }  
}
```

```
// pass an object
void meth(Test o) {
    o.a *= 2;
    o.b /= 2;
}
}

class CallByRef {
    public static void main(String args[]) {
        Test ob = new Test(15, 20);
        System.out.println("ob.a and ob.b before call:
"+ob.a+" "+ob.b);
        ob.meth(ob);// Test o=ob;
        System.out.println("ob.a and ob.b after call: "+ob.a + "
"+ob.b);
    }
}
```

This program generates the following output:  
ob.a and ob.b before call: 15 20  
ob.a and ob.b after call: 30 10

# Method Overloading

- Two methods in a class can share a name, as long as they have different parameter declarations.
- Example:

```
class Overload {  
    void display() {  
        System.out.println("<nothing>");  
    }  
    void display(int x) {  
        System.out.println(x);  
    }  
}
```

# Overloading Methods

- In Java it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different
- When an overloaded method is invoked, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call
- The return type alone is insufficient to distinguish two versions of a method

```
class OverloadDemo
{
    void test()
    {
        System.out.println("No parameters");
    }
    // Overload test for one integer parameter.
    void test(int a)
    {
        System.out.println("a: " + a);
    }
    // Overload test for two integer parameters.
    void test(int a, int b)
    {
        System.out.println("a and b: " + a + " " + b);
    }
}
```

```
// overload test for a double parameter
double test(double a)
{
    System.out.println("double a: " + a);
    return a*a;
}
}
class Overload
{
    public static void main(String args[])
    {
        OverloadDemo ob = new OverloadDemo();
        double result;          // call all versions of test()
        ob.test();
        ob.test(10);
        ob.test(10, 20);
        result = ob.test(123.25);
        System.out.println("Result of ob.test(123.25): " + result);
    }
}
```

- In some cases Java's automatic type conversions can play a role in overload resolution
- Java will employ its automatic type conversions only if no exact match is found

```
// Automatic type conversions apply to
// overloading.
class OverloadDemo {
    void test() {
        System.out.println("No parameters");
    }
    // Overload test for two integer parameters.
    void test(int a, int b) {
        System.out.println("a and b: " + a + " " + b);
    }
    // overload test for a double parameter
    void test(double a) {
        System.out.println("Inside test(double) a: " + a);
    }
}
```

- double d=10;

```
class Overload {  
    public static void main(String args[]) {  
        OverloadDemo ob = new OverloadDemo();  
        int i = 88;  
        ob.test();  
        ob.test(10, 20);  
        ob.test(i); // this will invoke test(double)  
        ob.test(123.2); //this will invoke test(double)  
    }  
}
```

This program generates the following output:

No parameters

a and b: 10 20

Inside test(double) a: 88

Inside test(double) a: 123.2

# Constructor Overloading

- You can overload constructors.
- Example:

```
class Overload{  
    int data;  
    Overload(int x) {  
        data = x;  
    }  
    Overload(int x, int y) {  
        data = x+y;  
    }  
}
```

# Progress Check

1. Name Java's access modifiers
2. What private access specifier does?
3. What is the difference between call-by-value and call-by-reference?
4. How does Java pass primitive types? How does it pass objects?
5. Can a constructor take an object of its own class as a parameter?
6. Why might you want to provide overloaded constructors?

# Returning Objects

- A method can return any type of data, including class types that you create

```
// Returning an object.
```

```
class Test {  
    int a;  
    Test(int i) {  
        a = i;  
    }  
    Test incrByTen() {  
        Test temp = new Test(a+10);  
        return temp;  
    }  
}
```

```
class RetOb {  
    public static void main(String args[]) {  
        Test ob1 = new Test(2);  
        Test ob2;  
        ob2 = ob1.incrByTen();  
        System.out.println("ob1.a: " + ob1.a);  
        System.out.println("ob2.a: " + ob2.a);  
        ob2 = ob2.incrByTen();  
        System.out.println("ob2.a after second increase: " +  
ob2.a);  
    }  
}
```

The output generated by this program is shown here:

ob1.a: 2

ob2.a: 12

ob2.a after second increase: 22

# Understanding static

- When a member is declared **static**, it can be accessed before any objects of its class are created, and without reference to any object
- The most common example of a **static** member is **main()**
- **main()** is declared as **static** because it must be called before any objects exist
- Instance variables declared as **static** are, essentially, global variables

Methods declared as **static** have several restrictions:

- They can only call other **static** methods
- They must only access **static** data
- They cannot refer to **this** or **super** in any way
- We can declare a **static** block which gets executed exactly once, when the class is first loaded

```
// Demonstrate static variables, methods, and  
blocks.  
class UseStatic {  
    static int a = 3;  
    static int b;  
    static void meth(int x) {  
        System.out.println("x = " + x);  
        System.out.println("a = " + a);  
        System.out.println("b = " + b);  
    }  
    static {  
        System.out.println("Static block  
initialized.");  
        b = a * 4;  
    }  
}
```

```
public static void main(String args[ ]) {  
    meth(42);  
}  
}  
  
//output  
Static block initialized.
```

x = 42

a = 3

b = 12

- As soon as the **UseStatic** class is loaded, all of the **static** statements are run
- First, **a** is set to **3**, then the **static** block executes (printing a message), and finally, **b** is initialized to **a \* 4** or **12**
- Then **main( )** is called, which calls **meth( )**, passing **42** to **x**
- Output of the program :  
static block initialized.

**x=42**

**a=3**

**b=12**

- If you wish to call a **static** method from outside its class, you can do so using the following general form:

***classname.method( )***

- Here, *classname* is the name of the class in which the **static** method is declared

```
class StaticDemo {  
    static int a = 42;  
    static int b = 99;  
    static void callme() {  
        System.out.println("a = " + a);  
    }    }  
  
class StaticByName {  
    public static void main(String args[]) {  
        StaticDemo.callme();  
        System.out.println("b = " + StaticDemo.b);  
    }    }  
Here is the output of this program:  
a = 42  
b = 99
```

# Introducing final

- A variable can be declared as **final**
- Doing so prevents its contents from being modified
- We must initialize a **final** variable when it is declared
  - `final int FILE_NEW = 1;`
  - `final int FILE_OPEN = 2;`
  - `final double PI=3.14;`

- Variables declared as **final** do not occupy memory on a per-instance basis
- The keyword **final** can also be applied to methods, but its meaning is substantially different than when it is applied to variables

# Inner Classes

- A class declared within another class is called an *inner class*.
- Its scope is the enclosing class.
- It has access to all the variables and methods of the enclosing class.

# Example of an Inner Class

```
class Outer {  
    int x = 5;  
  
    class Inner {  
        void changeX() { x = 3; } // change Outer's x  
    }  
  
    void adjust() {  
        Inner inn = new Inner();  
        inn.changeX(); // Outer's x is now 3  
    }  
}
```

# Varargs

- Beginning with JDK 5, Java has included a feature that simplifies the creation of methods that need to take a variable number of arguments. This feature is called *varargs* and it is short for *variable-length arguments*.
- A method that takes a variable number of arguments is called a *variable-arity method*, or simply a *varargs method*.

# Varargs

- You can create methods with a variable number of arguments, using "..."
- Example:

```
class VATest {  
    static void vaMethod(int ... data) {  
        for(int x : data) System.out.print(x + " ");  
    }  
    public static void main(String[] args) {  
        vaMethod(1); // prints "1 "  
        vaMethod(1,2,3); // prints "1 2 3 "  
    }  
}
```

# Overloading Varargs Methods

- There must be only one varargs parameter.
- You can overload a method that takes a variable-length argument.

```
static void vaTest(int ... v) {  
    static void vaTest(boolean ... v) {  
        static void vaTest(String msg, int ... v) {
```

- *A varargs method can also be overloaded by a non-varargs method.*
- unexpected errors can result when overloading a method that takes a variable-length argument. These errors involve ambiguity because it is possible to create an ambiguous call to an overloaded varargs method.

# Overloading Varargs Methods

- Ambiguity can result in some cases.
- Example:

```
class VAOverload {  
    static void vaMethod(int ... data) { }  
    static void vaMethod(boolean ... data) { }  
}
```

- This is fine unless you call **vaMethod()** with no arguments, which is ambiguous.

# Exercise

Q1. Suppose you have a class MyClass with one instance variable x. What will be printed by the following code segment? Explain your answer.

```
MyClass c1 = new MyClass();
c1.x = 3;
MyClass c2 = c1;
c2.x = 4; System.out.println(c1.x);
```

Q2. Suppose that a class has an overloaded method named add with the following two implementations:

```
double add(int x, double y) { return x + y; }
```

```
double add(double x, int y) { return x + y + 1; }
```

What, if anything will be returned by the following method calls?

# Exercise

```
double add(int x, double y) { return x + y; } // v1  
double add(double x, int y) { return x + y + 1; } //v2
```

- A. add(3, 3.14) // v1
- B. add(3.14, 3) // v2
- C. add(3, 3) // ambiguous
- D. add(3.14, 3.14)// ambiguous

# Chapter 7

## Inheritance

# Inheritance

- **Inheritance** allows the creation of hierarchical classifications.
- Using inheritance, you can create a general class that defines traits common to a set of related items. This class can then be inherited by other, more specific classes, each adding those things that are unique to it.
- In the language of Java, a class that is inherited is called a ***superclass***. The class that does the inheriting is called a ***subclass***.

- A subclass is a specialized version of a superclass. It inherits all of the variables and methods defined by the superclass and adds its own, unique elements.
- A class (a "subclass") can inherit all the methods and variables of another class (a "superclass").
- Use the keyword **extends**.
- General form:  
*class subclass extends superclass { ... }*
- The subclass extends the superclass by adding behavior and data to the behavior and data provided by the superclass.

```
// A simple class hierarchy.  
  
// A class for two-dimensional objects.  
class TwoDShape {  
  
    double width;  
    double height;  
  
    void showDim() {  
        System.out.println("Width and height are " +  
                           width + " and " + height);  
    }  
}
```

```
// A subclass of TwoDShape for triangles.
```

```
class Triangle extends TwoDShape {
```

```
String style;
```

```
    double area() {
```

```
    return width * height / 2;
```

1

– Triangle inherits TwoDShape.

- Triangle can refer to the members of TwoDShape as if they were part of Triangle.

```
void showStyle() {
```

```
System.out.println("Triangle is " + style);
```

1

```
class Shapes {  
    public static void main(String args[]) {  
        Triangle t1 = new Triangle();  2 Subclass objects are created  
        Triangle t2 = new Triangle();  
  
        t1.width = 4.0;  
        t1.height = 4.0;  ←———— All members of Triangle are available to Triangle  
        t1.style = "filled";  objects, even those inherited from TwoDShape.  
  
        t2.width = 8.0;  
        t2.height = 12.0;  
        t2.style = "outlined";  
  
        System.out.println("Info for t1: ");  
        t1.showStyle();  
        t1.showDim();  
        System.out.println("Area is " + t1.area());
```

```
System.out.println();

System.out.println("Info for t1:");
t1.showStyle();
t1.showDim();
System.out.println("Area is " + t1.area());
}

}
```

The output from this program is shown here:

Info for t1:

Triangle is filled

Width and height are 4.0 and 4.0

Area is 8.0

Info for t2:

Triangle is outlined

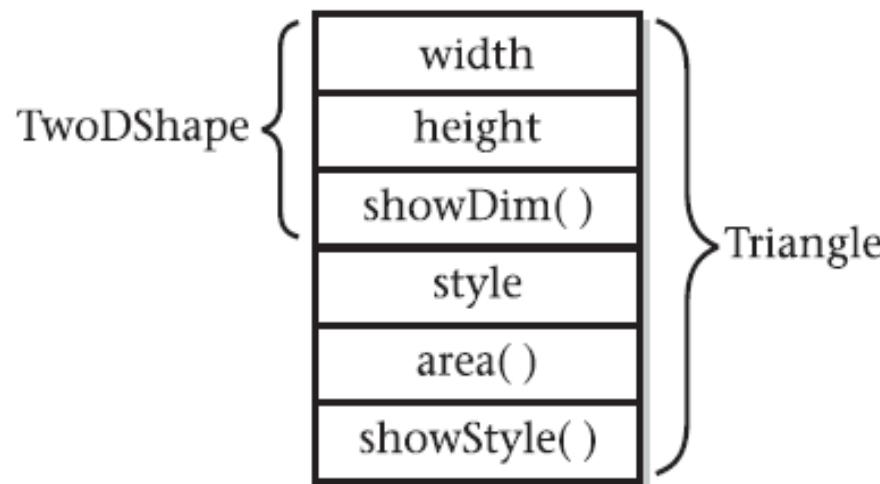
Width and height are 8.0 and 12.0

Area is 48.0

Even though **TwoDShape** is a superclass for **Triangle**, it is also a completely independent, stand-alone class. Being a superclass for a subclass does not mean that the superclass cannot be used by itself. For example, the following is perfectly valid:

```
TwoDShape shape = new TwoDShape();  
  
shape.width = 10;  
shape.height = 20;  
  
shape.showDim();
```

Of course, an object of **TwoDShape** has no knowledge of or access to any subclasses of **TwoDShape**.



**Figure 7-1** A conceptual depiction of the **Triangle** class

# Another Example

```
class OneDimPoint {  
    int x = 3;  
    int getX() { return x; }  
}  
class TwoDimPoint extends OneDimPoint {  
    int y = 4;  
    int getY() { return y; }  
}  
class TestInherit {  
    public static void main(String[] args) {  
        TwoDimPoint pt = new TwoDimPoint();  
        System.out.println(pt.getX() + "," + pt.getY());  
    }  
}
```

# Properties of Inheritance

- A subclass cannot access the private members of its superclass.
- Each class can have at most one superclass, but each superclass can have many subclasses.
- A subclass constructor can call a superclass constructor by use of **super( )**, before doing anything else.
- If you do not call a superclass constructor, the no-argument constructor is automatically called.

- Java does not support the inheritance of multiple superclasses into a single subclass
- You can create a hierarchy of inheritance in which a subclass becomes a superclass of another subclass
- A major advantage of inheritance is that once you have created a superclass that defines the attributes common to a set of objects, it can be used to create any number of more specific subclasses
- Each subclass can precisely tailor its own classification

```
// A subclass of TwoDShape for rectangles.  
class Rectangle extends TwoDShape {  
    boolean isSquare() {  
        if(width == height) return true;  
        return false;  
    }  
  
    double area() {  
        return width * height;  
    }  
}
```

The Rectangle class includes TwoDShape and adds the methods `isSquare()`, which determines if the rectangle is square, and `area()`, which computes the area of a rectangle.

## Member Access and Inheritance

- An instance variable of a class will be declared **private** to prevent its unauthorized use.
- Inheriting a class *does not* overrule the **private** access restriction.
- Thus, even though a subclass includes all of the members of its superclass, it cannot access those members of the superclass that have been declared **private**.
- For example, if, as shown here, **width** and **height** are made private in **TwoDShape**, then **Triangle** will not be able to access them:

```
// Private members are not inherited. This example will not compile.  
// A class for two-dimensional objects.  
class TwoDShape {  
    private double width; // these are  
    private double height; // now private  
    void showDim() {  
        System.out.println("Width and height are " +  
                           width + " and " + height);  
    }  
}  
// A subclass of TwoDShape for triangles.  
class Triangle extends TwoDShape {  
    String style;  
    double area() {  
        return width * height / 2; // Error! can't access  
    } // Can't access a private member of a superclass.  
    void showStyle() {  
        System.out.println("Triangle is " + style);  
    }  
}
```

```
// Use accessor methods to set and get private members.  
// A class for two-dimensional objects.  
class TwoDShape {  
    private double width; // these are  
    private double height; // now private  
    // Accessor methods for width and height.  
    double getWidth() { return width; }  
    double getHeight() { return height; }  
    void setWidth(double w) { width = w; }  
    void setHeight(double h) { height = h; }  
    void showDim() {  
        System.out.println("Width and height are " +width + " and " + height);  
    }  
}  
// A subclass of TwoDShape for triangles.  
class Triangle extends TwoDShape {  
    String style;  
    double area() {  
        return getWidth() * getHeight() / 2;  
    }  
}
```

```
void showStyle() {  
    System.out.println("Triangle is " + style);  
}  
}  
  
class Shapes2 {  
    public static void main(String args[]) {  
        Triangle t1 = new Triangle();  
        Triangle t2 = new Triangle();  
        t1.setWidth(4.0);  
        t1.setHeight(4.0);  
        t1.style = "filled";  
        t2.setWidth(8.0);  
        t2.setHeight(12.0);  
        t2.style = "outlined";  
        System.out.println("Info for t1: ");  
        t1.showStyle();  
        t1.showDim();  
        System.out.println("Area is " + t1.area());  
        System.out.println();  
        System.out.println("Info for t2: ");
```

```
t2.showStyle();
t2.showDim();
System.out.println("Area is " + t2.area());
}
}
```

## Constructors and Inheritance

- It is possible for both superclasses and subclasses to have their own constructors.
- What constructor is responsible for building an object of the subclass—the one in the superclass, the one in the subclass, or both?
- The constructor for the superclass constructs the superclass portion of the object, and the constructor for the subclass constructs the subclass part.
- The superclass has no knowledge of or access to any element in a subclass. Thus, their construction must be separate.
- in practice, most classes will have explicit constructors.
- When only the subclass defines a constructor, it constructs the subclass object.
- The superclass portion of the object is constructed automatically using its default constructor.

```
// Add a constructor to Triangle.  
// A class for two-dimensional objects.  
class TwoDShape {  
    private double width;    // these are  
    private double height;   // now private  
    // Accessor methods for width and height.  
    double getWidth() { return width; }  
    double getHeight() { return height; }  
    void setWidth(double w) { width = w; }  
    void setHeight(double h) { height = h; }  
    void showDim() {  
        System.out.println("Width and height are " +  
                           width + " and " + height);  
    }  
}  
// A subclass of TwoDShape for triangles.  
class Triangle extends TwoDShape {  
    private String style;  
    // Constructor
```

```
Triangle(String s, double w, double h) {  
    setWidth(w);  
    setHeight(h);  
    style = s;  
}  
double area() {  
    return getWidth() * getHeight() / 2;  
}  
void showStyle() {  
    System.out.println("Triangle is " + style);  
}  
}  
class Shapes3 {  
    public static void main(String args[]) {  
        Triangle t1 = new Triangle("filled", 4.0, 4.0);  
        Triangle t2 = new Triangle("outlined", 8.0, 12.0);  
        System.out.println("Info for t1: ");  
        t1.showStyle();  
        t1.showDim();  
        System.out.println("Area is " + t1.area());  
    }  
}
```

```
        System.out.println();
        System.out.println("Info for t2: ");
        t2.showStyle();
        t2.showDim();
        System.out.println("Area is " + t2.area());
    }
}
```

- When both the superclass and the subclass define constructors, both the superclass and subclass constructors must be executed.
- We must use Java's keyword, **super**, which has two general forms.
- **First one calls a superclass constructor.**
- **The second is used to access a member of the superclass that has been hidden by a member of a subclass.**
- A subclass can call a constructor defined by its superclass by use of the following form of super:

```
        super(parameter-list);
```

- Here, parameter-list specifies any parameters needed by the constructor in the superclass.
- super( ) must always be the first statement executed inside a subclass constructor.

```
// Add more constructors to TwoDShape.  
class TwoDShape {  
    private double width;  
    private double height;  
    // A default constructor.  
    TwoDShape() {  
        width = height = 0.0;  
    }  
    // Parameterized constructor.  
    TwoDShape(double w, double h) {  
        width = w;  
        height = h;  
    }  
    // Construct object with equal width and height.  
    TwoDShape(double x) {  
        width = height = x;  
    }  
    // Accessor methods for width and height.  
    double getWidth() { return width; }  
    double getHeight() { return height; }
```

```
void setWidth(double w) { width = w; }
void setHeight(double h) { height = h; }
void showDim() {
    System.out.println("Width and height are " + width + " and " + height);
}
// A subclass of TwoDShape for triangles.
class Triangle extends TwoDShape {
    private String style;
    // A default constructor.
    Triangle() {
        super();
        style = "none";
    }
    // Constructor
    Triangle(String s, double w, double h) {
        super(w, h); // call superclass constructor
        style = s;
    }
    // One argument constructor.
    Triangle(double x) {
        super(x); // call superclass constructor
```

```
style = "filled";
}
double area() {
return getWidth() * getHeight() / 2;
}
void showStyle() {
System.out.println("Triangle is " + style);
}
}
class Shapes5 {
public static void main(String args[]) {
Triangle t1 = new Triangle();
Triangle t2 = new Triangle("outlined", 8.0, 12.0);
Triangle t3 = new Triangle(4.0);
t1 = t2;
System.out.println("Info for t1: ");
t1.showStyle();
t1.showDim();
System.out.println("Area is " + t1.area());
System.out.println();
System.out.println("Info for t2: ");
t2.showStyle();
```

```
t2.showDim();
System.out.println("Area is " + t2.area());
System.out.println();
System.out.println("Info for t3: ");
t3.showStyle();
t3.showDim();
System.out.println("Area is " + t3.area());
}
}
```

Here is the output from this version:

Info for t1:

Triangle is outlined

Width and height are 8.0 and 12.0

Area is 48.0

Info for t2:

Triangle is outlined

Width and height are 8.0 and 12.0

Area is 48.0

Info for t3:

Triangle is filled

Width and height are 4.0 and 4.0

Area is 8.0

## Using super to Access Superclass Members

- There is a second form of super that refers to the superclass of the subclass in which it is used. This usage has the following general form:  
*super.member*

- Here, member can be either a method or an instance variable.
- This is applicable to situations in which member names of a subclass hide members by the same name in the superclass.

// Using super to overcome name hiding.

```
class A {  
    int i;  
}
```

// Create a subclass by extending class A.

```
class B extends A {  
    int i; // this i hides the i in A  
    B(int a, int b) {  
        super.i = a; // i in A  
        i = b; // i in B  
    }  
    void show() {  
        System.out.println("i in superclass: " + super.i);  
        System.out.println("i in subclass: " + i);  
    }  
}
```

```
class UseSuper {  
    public static void main(String args[]) {  
        B subOb = new B(1, 2);  
        subOb.show();  
    }  
}
```

This program displays the following:

i in superclass: 1

i in subclass: 2

## Creating a Multilevel Hierarchy

- You can build hierarchies that contain as many layers of inheritance as you like.
- It is perfectly acceptable to use a subclass as a superclass of another.
- Given three classes called A, B, and C, C can be a subclass of B, which is a subclass of A. C inherits all aspects of B and A.
- In the following, the subclass Triangle is used as a superclass to create the subclass called ColorTriangle.
- ColorTriangle inherits all of the traits of Triangle and TwoDShape and adds a field called color, which holds the color of the triangle.

```
// A multilevel hierarchy.
class TwoDShape {
    private double width;
    private double height;
    // A default constructor.
    TwoDShape() {
        width = height = 0.0;
    }
    // Parameterized constructor.
    TwoDShape(double w, double h) {
        width = w;
        height = h;
    }
    // Construct object with equal width and height.
    TwoDShape(double x) {
        width = height = x;
    }
    // Accessor methods for width and height.
    double getWidth() { return width; }
    double getHeight() { return height; }
```

```
void setWidth(double w) { width = w; }
void setHeight(double h) { height = h; }
void showDim() {
    System.out.println("Width and height are " +
                       width + " and " + height);
}
}
// Extend TwoDShape.
class Triangle extends TwoDShape {
    private String style;
    // A default constructor.
    Triangle() {
        super();
        style = "none";
    }
    Triangle(String s, double w, double h) {
        super(w, h); // call superclass constructor
        style = s;
    }
    // One argument constructor.
```

```
Triangle(double x) {  
    super(x); // call superclass constructor  
    style = "filled";  
}  
double area() {  
    return getWidth() * getHeight() / 2;  
}  
void showStyle() {  
    System.out.println("Triangle is " + style);  
}  
}  
}  
// Extend Triangle.  
class ColorTriangle extends Triangle {  
    private String color;  
    ColorTriangle(String c, String s, double w, double h) {  
        super(s, w, h);  
        color = c;  
    }
```

```
String getColor() { return color; }
void showColor() {
    System.out.println("Color is " + color);
}
}
class Shapes6 {
    public static void main(String args[]) {
        ColorTriangle t1 =
            new ColorTriangle("Blue", "outlined", 8.0, 12.0);
        ColorTriangle t2 =
            new ColorTriangle("Red", "filled", 2.0, 2.0);
        System.out.println("Info for t1: ");
        t1.showStyle();
        t1.showDim();
        t1.showColor();
        System.out.println("Area is " + t1.area());
        System.out.println();
        System.out.println("Info for t2: ");
        t2.showStyle();
        t2.showDim();
```

```
t2.showColor();
System.out.println("Area is " + t2.area());
}
}
```

The output of this program is shown here:

Info for t1:

Triangle is outlined

Width and height are 8.0 and 12.0

Color is Blue

Area is 48.0

Info for t2:

Triangle is filled

Width and height are 2.0 and 2.0

Color is Red

Area is 2.0

In a class hierarchy, constructors complete their execution in order of derivation, from superclass to subclass. Since super( ) must be the first statement executed in a subclass' constructor, this order is the same whether or not super( ) is used.

```
// Demonstrate when constructors are executed.  
// Create a super class.  
class A {  
    A() {  
        System.out.println("Constructing A.");  
    }  
}  
// Create a subclass by extending class A.  
class B extends A {  
    B() {  
        System.out.println("Constructing B.");  
    }  
}  
// Create another subclass by extending B.  
class C extends B {  
    C() {  
        System.out.println("Constructing C.");  
    }  
}
```

```
class OrderOfConstruction {  
    public static void main(String args[]) {  
        C c = new C();  
    }  
}
```

The output from this program is shown here:

Constructing A.

Constructing B.

Constructing C.

## Superclass References and Subclass Objects

- A reference variable for one class type cannot normally refer to an object of another class type.
- A reference variable of a superclass can be assigned a reference to an object of any subclass derived from that superclass.

// A superclass reference can refer to a subclass object.

```
class X {  
    int a;  
    X(int i) { a = i; }  
}
```

```
class Y extends X {  
    int b;  
    Y(int i, int j) {  
        super(j);  
        b = i;  
    }  
}  
class SupSubRef {  
    public static void main(String args[]) {  
        X x = new X(10);  
        X x2;  
        Y y = new Y(5, 6);  
        x2 = x;          // OK, both of same type  
        System.out.println("x2.a: " + x2.a);  
        x2 = y;          // still Ok because Y is derived from X  
        System.out.println("x2.a: " + x2.a);  
        // X references know only about X members  
        x2.a = 19;       // OK  
        // x2.b = 27;    // Error, X doesn't have a b member  
    }  
}
```

```
class TwoDShape { // a superclass reference can refer to a subclass object
    private double width;
    private double height;
    // Construct an object from an object.
    TwoDShape(TwoDShape ob) { //TwoDShape reference is pointing to
        width = ob.width;           // Triangle class object
        height = ob.height;
    }
}
// A subclass of TwoDShape for triangles.
class Triangle extends TwoDShape {
    private String style;
    // Construct an object from an object.
    Triangle(Triangle ob) {
        super(ob);           // pass object to TwoDShape constructor
        style = ob.style;
    }
}
class Shapes7 {
    public static void main(String args[]) {
        Triangle t1 = new Triangle("outlined", 8.0, 12.0);
        // make a copy of t1
        Triangle t2 = new Triangle(t1);
    }
}
```

# Method Overriding

- In a class hierarchy, when a method in a subclass has the same return type and signature as a method in its superclass, then the method in the subclass is said to override the method in the superclass.
- When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the subclass.
- The version of the method defined by the superclass will be hidden.
- If the signatures of the two methods are not identical then the two methods are simply overloaded.

// Method overriding.

```
class A {  
    int i, j;  
    A(int a, int b) {  
        i = a;  
        j = b;  
    }  
    // display i and j  
    void show() {  
        System.out.println("i and j: " + i + " " + j);  
    }  
}
```

```
class B extends A {  
    int k;  
    B(int a, int b, int c) {  
        super(a, b);  
        k = c;  
    }  
    void show() {      // display k – this overrides show() in A  
        System.out.println("overridden : " + k);  
    }  
    void show(String msg) {      // overload show()  
        System.out.println("msg :" + k); //since signature is different  
    }  
}  
class Override {  
    public static void main(String args[]) {  
        B subOb = new B(1, 2, 3);  
        subOb.show();    // this calls overridden show() in B  
        subOb.show("overload");    // this calls overloaded show  
    }  
} //The output produced by this program is shown here:      overridden: 3  
               overload : 3
```

## Overridden Methods Support Polymorphism

- *Dynamic method dispatch* is the mechanism by which a call to an overridden method is resolved at run time rather than compile time.
- Through Dynamic method dispatch, Java implements run-time polymorphism.
- A superclass reference variable can refer to a subclass object.
- Java uses this fact to resolve calls to overridden methods at run time.
- When an overridden method is called through a superclass reference, Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs.
- When different types of objects are referred to, different versions of an overridden method will be called.
- It is the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed.
- If a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through a superclass reference variable, different versions of the method are executed.

```
// Demonstrate dynamic method
dispatch.
class Sup {
    void who() {
        System.out.println("who() in Sup");
    }
}
class Sub1 extends Sup {
    void who() {
        System.out.println("who() in Sub1");
    }
}
class Sub2 extends Sup {
    void who() {
        System.out.println("who() in Sub2");
    }
}
class DynDispDemo {
    public static void main(String args[]) {
        Sup superOb = new Sup();
        Sub1 subOb1 = new Sub1();
```

```
        Sub2 subOb2 = new Sub2();
        Sup supRef;
        supRef = superOb;
        supRef.who();
        supRef = subOb1;
        supRef.who();
        supRef = subOb2;
        supRef.who();
    }
}
```

The output from the program is shown here:

```
who() in Sup
who() in Sub1
who() in Sub2
```

In each case, the version of **who( )** to call is determined at run time by the type of object being referred to.

# Why Overridden Methods?

- Overridden methods allow Java to support run-time polymorphism.
- Polymorphism allows a general class to specify methods that will be common to all of its derivatives, while allowing subclasses to define the specific implementation of some or all of those methods.
- Overridden methods allow Java implements the “one interface, multiple methods” aspect of polymorphism.
- The superclasses and subclasses form a hierarchy that moves from lesser to greater specialization.
- The superclass provides all elements that a subclass can use directly.
- It also defines those methods that the derived class must implement on its own.
- By combining inheritance with overridden methods, a superclass can define the general form of the methods that will be used by all of its subclasses.

# Abstract Classes

- Sometimes you want a class that is only partially implemented and you want to leave it to the subclasses to complete the implementation.
- In that case, use an *abstract* class with *abstract* methods.
- To declare a class or method as abstract, just add the keyword **abstract** in front of the class or method declaration.
- In the case of an abstract method, you must also leave off the body of the method.

# Example of an Abstract Class

```
abstract class Super {  
    int x;  
  
    int getX() { return x; }  
  
    abstract void setX(int newX); // no body  
}  
  
class Sub extends Super {  
  
    void setX(int newX) { x = newX; }  
}
```

# The Keyword **final**

- If you do not want a class to be subclassed, precede the class declaration with the keyword **final**.
- If you do not want a method to be overridden by a subclass, precede the method declaration with the keyword **final**.
- If you want a variable to be read-only (that is, a constant), precede it with the keyword **final**.

# Example Using **final**

```
//class final, prevents inheritance
final class MyClass {
    //variable final, read only
    final int x = 3;
    public static final double PI = 3.14159;
    //method final, can not be overridden
    final double getPI() { return PI; }
}
```

# The Object Class

- Java defines a special class called **Object** that is an implicit superclass of all other classes.
- Therefore, all classes inherit the methods in the **Object** class.
- A variable of type **Object** can refer to an object of any other class, including an array.

# Some Methods in the Object Class

Method	Purpose
Object clone( )	Creates a copy of this object
boolean equals(Object <i>obj</i> )	Tests whether two objects are equal
void finalize( )	Called before recycling the object
Class<?> getClass( )	Returns the class of the object
int hashCode( )	Returns the hash code of the object
void notify( )	Resumes execution of a thread waiting on the object
void notifyAll( )	Resumes execution of all threads waiting on the object
String toString( )	Returns a string describing the object
void wait( )	Waits on another thread of execution

# Chapter 8

# Interfaces

# Interfaces

- **Interface** is an abstract way of defining **what** a class must do, but **not how** to do it.
- Interfaces are syntactically similar to classes, but they lack instance variables and concrete methods.
- All the methods are abstract methods declared without any body, and all the variables are public final static
- Interfaces don't make assumptions about how they are implemented.

# Interfaces

- To implement an interface, a class must **provide implementations** for the methods described by the interface.
- Each class is free to determine the **details** of its own implementation.
- Using **interfaces**, Java allows to fully utilize the “**one interface, multiple methods**” aspect of polymorphism.
- Interfaces are designed to support **dynamic method dispatch (at run time)**

# Declaring an Interface

An interface is declared much like a class.

*access interface name*

```
{  
    return-type method-name1(parameter-list);  
    return-type method-name2(parameter-list);  
    return-type method-nameN(parameter-list);
```

*type final-varname1 = value;*

*type final-varname2 = value;*

*// ...*

*type final-varnameN = value;*

```
}
```

*name* is the name of the interface, and can be any valid identifier.

- The methods which are declared have no bodies.
- They end with a semicolon after the parameter list.
- They are abstract methods

- There can be no default implementation of any method specified within an interface.
- Each class that implements an interface must implement all of the methods.
- *Access is either **public** or not used(default).*
- *When no access specifier is included, then* default access results, and the interface is only available to other members of the package in which it is declared.
- When it is declared as **public**, the interface can be used by any other package also.

- Variables can be declared inside of interface declarations.
- They are implicitly **final** and **static**, meaning they cannot be changed by the implementing class.
- They must also be initialized with a constant value. All methods and variables are implicitly **public** if the interface, itself, is declared as **public**.

## Series.java: Number Series Generator

```
public interface Series{  
    int getNext(); //return next no. in series  
    void reset(); //restart  
    void setStart(); //set starting value  
}
```

Example Series: Even numbers starting with 2  
Prime numbers

# Implementing Interfaces

- Once an **interface** has been defined, one or more classes can implement that interface.
- To implement an interface,
  - Include the **implements** clause in a class definition
  - Inside the class, implement the methods defined by the interface.
- The general form of a class that includes the implements clause looks like this:

```
access class classname [extends superclass]  
[implements interface [,interface...]] {  
    // class-body  
}
```

- Here, *access* is either **public** or **not used**.

- If a class implements more than one interface, **the interfaces are separated with a comma.**
- If a class implements two interfaces that declare the same method, then the same method will be used by clients of either interface.
- The **methods** that implement an interface must be declared **public**.
- Type signature of the implementing method must match exactly **the type signature specified in the interface declaration.**
- It is both permissible and common for classes that implement interfaces to define **additional members of their own.**

## **EXAMPLE:**

**ByTwos Series...**

**ByThrees Series...**

```
class ByTwos implements Series {  
    //Implements interface  
    int start;  
    int val;  
  
    ByTwos() {  
        start = 0;  
        val = 0;  
    }  
  
    public void setStart(int x) {  
        start = x;  
        val = x;  
    }  
  
    public int getNext() {  
        val += 2;  
        return val;  
    }  
  
    public void reset() {  
        val = start;  
    }  
}
```

```
interface Series {  
    int getNext(); //return next no. in series  
    void reset(); //restart  
    void setStart(int x); //set starting value  
}
```

```
public class ByTwosSeries {  
  
    public static void main(String[] args) {  
        ByTwos ob = new ByTwos();  
  
        System.out.println("Series: ");  
        System.out.println("Starting at: "+ ob.val);  
        for(int i = 0; i<5; i++)  
            System.out.println("Next value: "+ ob.getNext());  
  
        System.out.println("Series: ");  
        ob.reset();  
        System.out.println("Starting at: "+ ob.val);  
        for(int i = 0; i<5; i++)  
            System.out.println("Next value: "+ ob.getNext());  
  
        System.out.println("Series: ");  
        ob.setStart(100);  
        System.out.println("Starting at: "+ ob.val);  
        for(int i = 0; i<5; i++)  
            System.out.println("Next value: "+ ob.getNext());  
    }  
}
```

```
class ByThrees implements Series {  
    //Implements interface  
    int start;  
    int val;  
    ByThrees() {  
        start = 0;  
        val = 0;  
    }  
    public void setStart(int x) {  
        start = x;  
        val = x;  
    }  
    public int getNext() {  
        val += 3;  
        return val;  
    }  
    public void reset() {  
        val = start;  
    }  
}
```

```
public class ByTwoThreesSeries {  
    public static void main(String[] args) {  
        ByTwos ob2 = new ByTwos();  
        ByThrees ob3 = new ByThrees();  
  
        System.out.println("ByTwo Series: ");  
        System.out.println("Starting at: "+ ob2.val);  
        for(int i = 0; i<5; i++)  
            System.out.println("Next value: "+ ob2.getNext());  
  
        System.out.println("ByThree Series: ");  
        ob3.reset();  
        System.out.println("Starting at: "+ ob3.val);  
        for(int i = 0; i<5; i++)  
            System.out.println("Next value: "+ ob3.getNext());  
  
        System.out.println("ByTwo Series: ");  
        ob2.setStart(100);  
        System.out.println("Starting at: "+ ob2.val);  
        for(int i = 0; i<5; i++)  
            System.out.println("Next value: "+ ob2.getNext());  
    }  
}
```

# Progress Check

- What is an interface? What keyword is used to declare one?
- What is **implements** for?

# Using Interface References

- Similar to class declaration, **interface declaration also creates a new reference type.**
- Any instance of any class that implements the declared interface **can be referred to by such a reference variable.**
- When you call a method through one of these references, **the correct version will be called based on the actual instance of the interface being referred to.**

# Interface References: Example

```
public class InterfaceRef {  
  
    public static void main(String[] args) {  
        ByTwos ob2 = new ByTwos();  
        ByThrees ob3 = new ByThrees();
```

## **Series iRef; //Interface Reference**

```
System.out.println("Series.....");  
  
for(int i = 0; i<5; i++){  
    iRef = ob2; //Refers to ByTwos object  
    System.out.println("ByTwos Next value: "+ iRef.getNext());  
    iRef = ob3; //Refers to ByThrees object  
    System.out.println("ByThrees Next value: "+  
    iRef.getNext());  
}
```

# Interface References: Polymorphism

```
class Simulation{  
  
    Series numSeq;  
  
    Simulation(Series s) {  
        numSeq = s;  
    }  
    }  
    .....  
    .....  
}
```

```
Simulation sim2 = new Simulation(new ByTwos());  
Simulation sim3 = new Simulation(new ByThrees());
```

- Calling interface methods through an interface reference helps to **fully realize the benefit of “one interface, multiple methods” philosophy.**

# Implementing multiple Interfaces

```
interface ifA{  
    void doSomething();  
}  
  
interface ifB {  
    void doSomethingElse();  
}  
  
class MyClass implements ifA, ifB {  
    public void doSomething(){  
        System.out.println("Doing Something");  
    }  
    public void doSomethingElse(){  
        System.out.println("Doing Something else");  
    }  
}
```

- In real-world applications, it is common for a class to implement more than one interface.

- *Because dynamic lookup of a method at run time incurs a significant overhead when compared with the normal method invocation in Java, you should be careful not to use interfaces casually in performance-critical code.*

# Progress Check

- Can an interface reference variable refer to an object that implements that interface?
- A class can implement only one interface. T/F?
- When implementing multiple interfaces, what happens if both interfaces declare the same method?

# Stack Interface

- The interface that defines an integer stack in a file called **IntStack.java**

```
// Define an integer stack interface.
```

```
interface IntStack
```

```
{
```

```
    void push(int item); // store an item
```

```
    int pop(); // retrieve an item
```

```
    boolean isEmpty(); //return true if stack is empty
```

```
    boolean isFull(); // return true if stack is full
```

```
}
```

```
class FixedStack implements IntStack
{
    private int stck[];
    private int tos;
    // allocate and initialize stack
    FixedStack(int size)
    {
        stck = new int[size];
        tos = -1;
    }
    public boolean isFull()  { return(tos==stck.length-1);}
    Public boolean isEmpty() { return(tos===-1);}

    public void push(int item)
    {
        if (isFull() )
            System.out.println("Stack is full.");
        else
            1/9/2022
            stck[++tos] = item;
    }
}
```

```
public int pop()
{
    if(isEmpty() )
    {
        System.out.println("Stack underflow.");
        return -1;
    }
    else
        return stck[tos--];
}
void display()
{
    if(isEmpty() )
        System.out.println("Empty Stack.");
    else
        for (int i=0;i<=tos;i++)
            System.out.println(stck[i]+ " ");
}
```

```
class IFTest
{
    public static void main(String args[])
    {
        FixedStack mystack1 = new FixedStack(5);

        // push some numbers onto the stack
        for(int i=0; i<5; i++) mystack1.push(i);

        // pop those numbers off the stack
        System.out.println("Stack in mystack1:");
        for(int i=0; i<5; i++)
            System.out.println(mystack1.pop());

    }
}
```

```
// Implement a "growable" stack.  
class DynStack implements IntStack  
{  
    // all the variables and methods are same except push  
    // Push an item onto the stack  
    public void push(int item)  
    {  
        // if stack is full, allocate a larger stack  
        if(isFull())  
        {  
            int temp[] = new int[stck.length * 2]; // double size  
            for(int i=0; i<stck.length; i++) temp[i] = stck[i];  
            stck = temp;  
            stck[++tos] = item;  
        }  
        else  
            stck[++tos] = item;  
    }  
}
```

```
class IFTest2
{
    public static void main(String args[])
    {
        DynStack mystack1 = new DynStack(5);

        // these loops cause each stack to grow
        for(int i=0; i<12; i++)
            mystack1.push(i);

        System.out.println("Stack in mystack1:");
        for(int i=0; i<12; i++)
            System.out.println(mystack1.pop());
    }
}
```

```
class IFTest3
{
    public static void main(String args[])
    {
        IntStack mystack; // create an interface ref var
        DynStack ds = new DynStack(5);
        FixedStack fs = new FixedStack(8);
        mystack = ds; // load dynamic stack
        // push some numbers onto the stack
        for(int i=0; i<12; i++) mystack.push(i);
        mystack = fs; // load fixed stack
        for(int i=0; i<8; i++) mystack.push(i);
        mystack = ds;
        System.out.println("Values in dynamic stack:");
        for(int i=0; i<12; i++)
            System.out.println(mystack.pop());
        mystack = fs;
        System.out.println("Values in fixed stack:");
        for(int i=0; i<8; i++)
            System.out.println(mystack.pop());
    }
}
```

- Accessing multiple implementations of an interface through an interface reference variable is the **most powerful way that Java achieves runtime polymorphism**.

# Constants in Interfaces

- Interfaces can also include data members, but they are implicitly **public final static** variables and must be initialized.
- Thus they are essentially constants.
- These constants are accessed like other static variables.
- Example:

```
interface MathConstants {  
    double PI = 3.14159;  
    double E = 2.71828;  
}
```

```
interface Iconst{  
    int MIN = 0;  
    int MAX = 10;  
    String ERRORMSG = "Boundary Error";  
}
```

```
public class IfConstants implements Iconst {  
    public static void main(String[] args) {  
        int [] nums = new int[MAX];  
        for(int i =MIN; i<= MAX; i++){  
            if (i>=MAX)  
                System.out.println(ERRORMSG);  
            else {  
                nums[i] = i;  
                System.out.print(nums[i]+ " ");  
            }  
        }  
    }  
}
```

0 1 2 3 4 5 6 7 8 9 Boundary Error

# Interfaces Can Be Extended

- Interface can inherit another by use of the keyword **extends**.
- The syntax is the same as for **inheriting classes**.
- When a class implements an interface that inherits another interface, **it must provide implementations for all methods defined within the interface inheritance chain**.

## **interface A**

```
{  
    void meth1();  
    void meth2();  
}
```

// B now includes meth1() and meth2() -- it adds meth3().

## **interface B extends A**

```
{  
    void meth3();  
}
```

// This class must implement all of A and B

## **class MyClass implements B**

```
{  
    public void meth1() {  
        System.out.println("Implement meth1().");  
    }  
    public void meth2() {  
        System.out.println("Implement meth2().");  
    }  
    public void meth3() {  
        System.out.println("Implement meth3().");  
    }  
}
```

```
class IFExtend
{
    public static void main(String arg[])
    {
        MyClass ob = new MyClass();
        ob.meth1();
        ob.meth2();
        ob.meth3();
    }
}
```

# More about interface

- They have no instance variables
- The variables declared inside interface are static,final by default
- Public /no modifier can be applied to interface as access
- All variables and methods are public if interface itself is declared public

# Interfaces vs. Classes

- All methods in an interface are abstract—  
it has no implementation
- All methods in an interface are automatically public
- An interface doesn't have instance variables
- Any data member declared in interface is public and final  
and is also static type.
- In a class all methods are concrete /complete

# Abstract class Vs interfaces

- Abstract classes can have complete methods unlike interface.
- Abstract classes can have fields unlike interfaces
- Abstract classes have to be extended unlike interface where these are implemented.
- Abstract methods of an abstract class can have different access unlike interfaces where every method defined is public.

# Nested Interfaces

- An interface can be declared a member of another interface or of a class.
- In that case, it is called a *member* or *nested interface*.
- Member interface of a class can be declared public, private, or protected.
- Member interface of an interface is implicitly public

```
interface A {  
    public interface NestedIF {  
        boolean isNotNegative(int x);  
    }  
    void doSomething();  
}  
  
class B implements A.NestedIF {  
    public boolean isNotNegative(int x){  
        return x<0? false: true;  
    }  
}  
  
public class NestedIF {  
    public static void main(String [] args){  
        A.NestedIF nif = new B();  
        if(nif.isNotNegative(10))  
            System.out.println("10 is not negative");  
        if(nif.isNotNegative(-12))  
            System.out.println("This won't be displayed");  
    }  
}
```

# Progress Check

- A variable declared in an interface creates a constant because it is implicitly \_\_\_\_\_, \_\_\_\_\_ and \_\_\_\_\_.
- Must an interface constant be initialized?

```
public interface SomethingIsWrong { void aMethod(int aValue) { System.out.println("Hi Mom"); } }
```

# Exercise

1. How many classes can implement an interface?
2. How many interfaces can a class implement?
3. Can interfaces be extended?
4. Can one interface be a member of other?
5. Given two interfaces called Alpha and Beta, show how a class called MyClass specifies that it implements each.
6. What is wrong with the following interface? Fix it.

```
public interface SomethingIsWrong {  
    void aMethod(int aValue) {  
        System.out.println("Hi Mom");  
    }  
}
```

```
public interface SomethingIsWrong { void aMethod(int aValue) { System.out.println("Hi Mom"); } }
```

# Exercise..

7. Suppose that a class Class1 extends a class Class2 and implements an interface Interface1 that extends an interface Interface2. Also assume Class1 has a no-argument constructor. Which of the following statements are legal?

- A. Class1 x = new Class1();
- B. Class2 x = new Class1();
- C. Interface1 x = new Class1();
- D. Interface2 x = new Class1();
- E. Object x = new Class1();

# Chapter 9

## Packages

# Package Fundamentals

- Package is Java's way of grouping a variety of classes and / interfaces together - **containers** for classes.
- Packages provide a way of organizing your code and of controlling access to the code.
- Classes defined within a package must be accessed through their package name.
- When no package is specified, the global (default) no-name package is used.

# Package Fundamentals

- To create **package** – include a **package** command as the first statement in the Java source file.
- The **package** statement defines a name space in which classes are stored.
- The standard Java library is distributed over a number of packages – java.lang, util, net and so on

# Package Fundamentals

The benefits of packages are –

- Classes in packages can be easily reused
- Two different packages can contain classes with same name
- Provide a way to ‘hide’ classes thus preventing other programs or packages from accessing classes that are meant for internal use only

# Defining a Package

- General form:  
`package pkgnname;`
- This statement must be at the top of a Java source file.
- Usually lower case is used for the package name.
- In the file system, each package is stored in its own directory that has the same name as the package.

# Defining a Package

- You can define packages inside of packages.
- To put a class in a package **pkg2** which is nested inside a **pkg1**, use the statement:

```
package pkg1.pkg2;
```

- To compile **MyClass.java** that is in the package **pkg1.pkg2**, use

```
javac pgk1/pgk2/MyClass.java
```

- To run **MyClass**, use

```
java pgk1.pgk2.MyClass
```

# Finding Packages and Classpath

- How does java run time system know where to look for packages that you create?
  1. By default the run time system uses the current working directory as its starting point
  2. If the above is not met, you can specify a directory path or paths by setting the CLASSPATH environmental variable
  3. Else you can use the **-classpath option** with java and javac to specify the path to your classes

# Finding Packages and Classpath

1. The easiest way to try the examples is to create the package directories below your current working directory, put the .class files into appropriate directories and then execute the programs from the current working directory

# Packages and Member Access

	Private	Default	Protected	Public
<b>Visible within same class</b>	Yes	Yes	Yes	Yes
<b>Visible within the same package by subclass</b>	No	Yes	Yes	Yes
<b>Visible within the same package by non-subclass</b>	No	Yes	Yes	Yes
<b>Visible within different packages by subclass</b>	No	No	Yes	Yes
<b>Visible within different packages by non-subclass</b>	No	No	No	Yes

## An access example:

- This has two packages(p1 and p2) and five classes(Protection, Derived, SamePackage, Protection2, OtherPackage) and stored in two different directories p1 and p2.

```
package p1;          // This is file Protection.java
public class Protection {
    int n = 1;
    private int n_pri = 2;
    protected int n_pro = 3;
    public int n_pub = 4;
    public Protection() {
        System.out.println("base constructor");
        System.out.println("n = " + n);
        System.out.println("n_pri = " + n_pri);
        System.out.println("n_pro = " + n_pro);
        System.out.println("n_pub = " + n_pub);
    }
}
```

```
//This is file Derived.java
package p1;
class Derived extends Protection {
    Derived() {
        System.out.println("derived constructor");
        System.out.println("n = " + n);
    }
    // class only but not here
    // System.out.println("n_pri = " + n_pri);
    System.out.println("n_pro = " + n_pro);
    System.out.println("n_pub = " + n_pub);
}
```

```
//This is file SamplePackage.java
package p1;
class SamePackage {
    SamePackage() {
        Protection p = new Protection();
        System.out.println("same package constructor");
        System.out.println("n = " + p.n);

        // System.out.println("n_pri = " + p.n_pri); // class
        // only

        System.out.println("n_pro = " + p.n_pro);
        System.out.println("n_pub = " + p.n_pub);
    }
}
```

```
//This is file Protection2.java
package p2;
class Protection2 extends p1.Protection {
    Protection2() {
        System.out.println("derived other package
constructor");
        // class or package only
        // System.out.println("n = " + n);
        // class only
        // System.out.println("n_pri = " + n_pri);
        System.out.println("n_pro = " + n_pro);
        System.out.println("n_pub = " + n_pub);
    }
}
```

```
//This file is OtherPackage.java
package p2;
class OtherPackage {
    OtherPackage() {
        p1.Protection p = new p1.Protection();
        System.out.println("other package
constructor");
        // class or package only
        // System.out.println("n = " + p.n);
        // class only
        // System.out.println("n_pri = " + p.n_pri);
        // class, subclass or package only
        // System.out.println("n_pro = " + p.n_pro);
        System.out.println("n_pub = " + p.n_pub);
    }
}
```

```
//Test files to use the above packages:  
package p1;    // Demo package p1.  
public class Demo { // Instantiate the various classes in  
    p1.  
    public static void main(String args[]) {  
        Protection ob1 = new Protection();  
        Derived ob2 = new Derived();  
        SamePackage ob3 = new SamePackage();  
    } }  
package p2;    // Demo package p2.  
public class Demo { // Instantiate the various classes in p2.  
    public static void main(String args[]) {  
        Protection2 ob1 = new Protection2();  
        OtherPackage ob2 = new OtherPackage();  
    } }
```

# Importing Packages:

- Java includes the **import** statement to bring certain classes, or entire packages, into visibility. Once imported, a class can be referred to directly, using only its name.
- The **import** statement is a convenience to the programmer and is not technically needed to write a complete Java program.
- If you are going to refer to a few dozen classes in your application, however, the **import** statement will save a lot of typing.

# Importing Packages

- To use **MyClass**, you can use the fully qualified name, as follows:

```
pkg1.pkg2.MyClass v = new pkg1.pkg2.MyClass();
```

- Or use the **import** statement with general form:  
*import pkg.classname;*
- Use an asterisk (\*) for the classname to import the entire contents of a package.
- The **import** statement goes after the **package** statement and before any class definitions.
- With imports, you need not qualify **MyClass**.

# Commonly Used Standard Packages

Package	Description
java.lang	Contains a large number of general-purpose classes
java.io	Contains the I/O classes
java.net	Contains those classes that support networking
java.applet	Contains classes for creating applets
java.awt	Contains classes that support the Abstract Window Toolkit
java.util	Contains various utility classes, plus the Collections Framework

The **java.lang** package is imported automatically in every Java program. All others need to be explicitly imported.

## Import static

- The use of **import** followed by **static** allows you to import static members of a class.
- Those static members can then be referred to directly by their names, without having to qualify them with the name of their class.
- For example, to use the **Math** function **sqrt( )** in the form "sqrt( )" instead of "Math.sqrt( )", add one of the **import** statements

```
import static Math.sqrt;  
import static Math.*;
```

## Import static

- The static import declaration is analogous to the normal import declaration.
- Where the normal import declaration imports classes from packages, allowing them to be used without package qualification, the static import declaration imports static members from classes, allowing them to be used without class qualification.

```
import static java.lang.Math.pow;
import static java.lang.Math.sqrt;
public class Hypot {
    public static void main(String args[]) {
        double side1, side2;
        double hypot;
        side1 = 3.0;
        side2 = 4.0;
        hypot = sqrt(pow(side1, 2) + pow(side2, 2));
        System.out.println("Given sides of lengths " + side1
        + " and " + side2 + " the hypotenuse is "
        + hypot);
    }
}
```

# To compile and run java package

For example consider the following package program:

**//Simple example of java package**

**//package keyword** is used to create a package in java.

//save as Simple.java

**package mypack;**

**public class Simple**

{

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

{ System.out.println("Welcome to package");}

# To compile java package

- Give the command with the following format in the terminal:
- `javac -d directory javafilename`
- For example
- `javac -d . Simple.java`
- The `-d` switch specifies the destination where to put the generated class file. Any directory name like `/home` (in case of Linux), `d:/abc` (in case of windows) etc. can be used. To keep the package within the same directory, use `.` (dot).

# To run java package

- Use fully qualified name e.g.  
java mypack.Simple to run the class.

- **Output:**

Welcome to package

# Progress Check

- What is package? Show how to declare a package called mypackage.
- How do you include another package in a source file?
- Do you need to include **java.lang** explicitly?
- Show how to include all of the classes in a package called mypackage
- What does protected do?
- If a class member has default access inside a package, is that member accessible by other packages?

# Chapter 10

## **EXCEPTION HANDLING**

# Exception

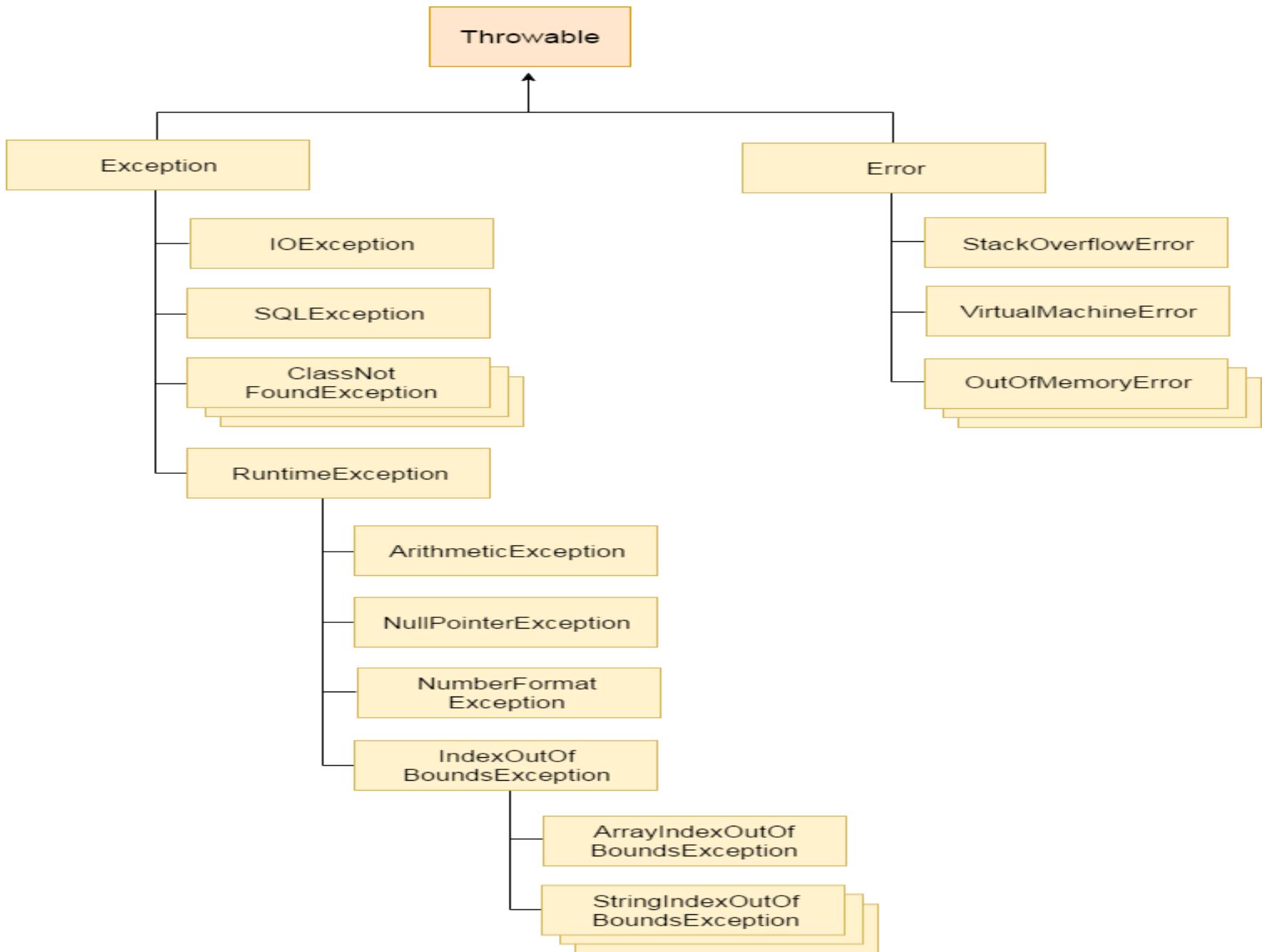
An exception is an abnormal condition that arises in a code sequence at run time, it may cause system to crash and affect other programs running in parallel.

How is an Exception represented in Java?

- A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code
- When an exceptional condition arises, an object representing that exception is created and *thrown* in the method that caused the error
- That method may choose to
  - a) handle it self
  - or
  - b) pass it on.
- At some point the exception is *caught* and processed

# The Exception Hierarchy

- In Java, all exceptions are represented by classes. All exception classes are derived from a class called **Throwable**. Thus, when an exception occurs in a program, an object of some type of exception class is generated.
- There are two direct subclasses of **Throwable**: **Exception** and **Error**.
- Exceptions of type **Error**:
  - Errors that occur in the Java virtual machine itself, and not in the program.
  - Beyond the user control, and the program will not usually deal with them.
- Errors that result from program activity are represented by subclasses of **Exception**.
- For example, divide-by-zero, array boundary, and file errors fall into this category. The program should handle exceptions of these types. An important subclass of **Exception** is **RuntimeException**, which is used to represent various common types of run-time errors.



# Exception Handling Fundamentals

- Java exception handling is managed via five keywords:
- **try, catch, throw, throws, and finally**

# Using try and catch

At the core of exception handling are **try** and **catch**. These keywords work together; you have a **catch** without a **try**. Here is the general form of the **try/catch** exception handling

```
try {  
    // block of code to monitor for errors  
}  
  
catch (ExcepType1 exOb) {  
    // handler for ExcepType1  
}  
  
catch (ExcepType2 exOb) {  
    // handler for ExcepType2  
}
```

# A Simple Exception Example Using try and catch blocks

```
class ExcDemo1 {  
    public static void main(String args[]) {  
        int nums[] = new int[4];  
  
        try { ← Create a try block.  
            System.out.println("Before exception is generated.");  
  
            // Generate an index out-of-bounds exception.  
            nums[7] = 10; ← Attempt to index past  
            System.out.println("this won't be displayed");  
        }  
        catch (ArrayIndexOutOfBoundsException exc) { ← Catch array boundary  
            // catch the exception  
            System.out.println("Index out-of-bounds!");  
        }  
        System.out.println("After catch statement.");  
    }  
}
```

This program displays the following output:

```
Before exception is generated.  
Index out-of-bounds!  
After catch statement.
```

bounds index will never execute. After the **catch** statement executes, program control continues with the statements following the **catch**. Thus, it is the job of your exception handler to remedy the problem that caused the exception so that program execution can continue normally.

Remember, if no exception is thrown by a **try** block, no **catch** statements will be executed and program control resumes after the **catch** statement. To confirm this, in the preceding program, change the line

```
nums[7] = 10;
```

to

```
nums[0] = 10;
```

Now, no exception is generated, and the **catch** block is not executed.

```
/* An exception can be generated by one
   method and caught by another. */

class ExcTest {
    // Generate an exception.

    static void genException() {
        int nums[] = new int[4];

        System.out.println("Before exception is generated.");

        // generate an index out-of-bounds exception
        nums[7] = 10; ← Exception generated here.
        System.out.println("this won't be displayed");
    }
}
```

```
class ExcDemo2 {  
    public static void main(String args[]) {  
  
        try {  
            ExcTest.genException();  
        } catch (ArrayIndexOutOfBoundsException exc) {  
            // catch the exception  
            System.out.println("Index out-of-bounds!");  
        }  
        System.out.println("After catch statement.");  
    }  
}
```

Exception caught here.

Before exception is generated.  
Index out-of-bounds!  
After catch statement.

# The Consequences of an Uncaught Exception

```
// Let JVM handle the error.  
class NotHandled {  
    public static void main(String args[]) {  
        int nums[] = new int[4];  
  
        System.out.println("Before exception is generated.");  
  
        // generate an index out-of-bounds exception  
        nums[7] = 10;  
    }  
}
```

If your program does not catch an exception, then it will be caught by the JVM. The JVM's default exception handler terminates execution and displays a stack trace and error message

When the array index error occurs, execution is halted, and the following error message is displayed.

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 7  
at NotHandled.main(NotHandled.java:9)
```

```
// This won't work!
class ExcTypeMismatch {
    public static void main(String args[]) {
        int nums[] = new int[4];
        System.out.println("Before exception is generated.");
        //generate an index out-of-bounds exception
        nums[7] = 10; ←
        System.out.println("this won't be displayed");
    }
}

/* Can't catch an array boundary error with an
   ArithmeticException. */

```

This throws an  
ArrayIndexOutOfBoundsException.

```
catch (ArithmeticException exc) { ← This tries to catch it with an
    // catch the exception
    System.out.println("Index out-of-bounds!");
}
System.out.println("After catch statement.");
}
```

The output is shown here.

Before exception is generated.

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 7
at ExcTypeMismatch.main(ExcTypeMismatch.java:10)
```

As the output demonstrates, a catch for **Arithm~~etic~~Exception** won't catch an **ArrayIndexOutOfBoundsException**.

# Exceptions Enable You to Handle Errors Gracefully

```
// Handle error gracefully and continue.
class ExcDemo3 {
    public static void main(String args[]) {
        int numer[] = { 4, 8, 16, 32, 64, 128 };
        int denom[] = { 2, 0, 4, 4, 0, 8 };

        for(int i=0; i<numer.length; i++) {
            try {
                System.out.println(numer[i] + " / " +
                                   denom[i] + " is " +
                                   numer[i]/denom[i]);
            }
            catch (ArithmaticException exc) {
                // catch the exception
                System.out.println("Can't divide by Zero!");
            }
        }
    }
}
```

The output from the program is shown here:

```
4 / 2 is 2
Can't divide by Zero!
16 / 4 is 4
32 / 4 is 8
Can't divide by Zero!
128 / 8 is 16|
```

# Using Multiple catch Statements

```
// Use multiple catch statements.
class ExcDemo4 {
    public static void main(String args[]) {
        // Here, numer is longer than denom.
        int numer[] = { 4, 8, 16, 32, 64, 128, 256, 512 };
        int denom[] = { 2, 0, 4, 4, 0, 8 };

        for(int i=0; i<numer.length; i++) {
            try {
                System.out.println(numer[i] + " / " +
                                   denom[i] + " is " +
                                   numer[i]/denom[i]);
            }
            catch (ArithmetricException exc) { ←———— Multiple catch statements
                // catch the exception
                System.out.println("Can't divide by Zero!");
            }
        }
    }
}
```

```
        catch (ArrayIndexOutOfBoundsException exc) { ←
            // catch the exception
            System.out.println("No matching element found.");
        }
    }
}
}
```

This program produces the following output:

4 / 2 is 2

Can't divide by Zero!

16 / 4 is 4

32 / 4 is 8

Can't divide by Zero!

128 / 8 is 16

No matching element found.

No matching element found.

# Catching Subclass Exceptions

- If you want to catch exceptions of both a superclass type and a subclass type, put the subclass first in the **catch** sequence.
- Putting the superclass first causes unreachable code to be created, since the subclass **catch** clause can never execute

```
class ExcDemo5 {  
    public static void main(String args[]) {  
        // Here, numer is longer than denom.  
        int numer[] = { 4, 8, 16, 32, 64, 128, 256, 512 };  
        int denom[] = { 2, 0, 4, 4, 0, 8 };  
  
        for(int i=0; i<numer.length; i++) {  
            try {  
                System.out.println(numer[i] + " / " +  
                                denom[i] + " is " +  
                                numer[i]/denom[i]);  
            }  
            catch (ArrayIndexOutOfBoundsException exc) { ← Catch subclass  
                // catch the exception  
                System.out.println("No matching element found.");  
            }  
        }  
    }  
}
```

```
    catch (Throwable exc) { ←————— Catch superclass
        System.out.println("Some exception occurred.");
    }
}
}
}
```

The output from the program is shown here:

4 / 2 is 2

Some exception occurred.

16 / 4 is 4

32 / 4 is 8

Some exception occurred.

128 / 8 is 16

No matching element found.

No matching element found.

# Nested Try

- A **try** statement can be inside the block of another **try**
- Each time a **try** statement is entered, the context of that exception is pushed on the stack.
- If an inner **try** statement does not have a **catch** handler for a particular exception, the stack is unwound and the next **try** statement's **catch** handlers are inspected for a match
- This continues until one of the **catch** statements succeeds, or until all of the nested **try** statements are exhausted
- If no **catch** statement matches, then the Java run-time system will handle the exception

# Try Blocks Can Be Nested

```
// Use a nested try block.
class NestTrys {
    public static void main(String args[]) {
        // Here, numer is longer than denom.
        int numer[] = { 4, 8, 16, 32, 64, 128, 256, 512 };
        int denom[] = { 2, 0, 4, 4, 0, 8 };

        try { // outer try ← Nested try blocks
            for(int i=0; i<numer.length; i++) {
                try { // nested try ←
                    System.out.println(numer[i] + " / " +
                        denom[i] + " is " +
                        numer[i]/denom[i]);
                }
                catch (ArithmetcException exc) {
                    // catch the exception
                    System.out.println("Can't divide by Zero!");
                }
            }
        }
    }
}
```

```
        System.out.println(numer[i] + " / " +
                            denom[i] + " is " +
                            numer[i]/denom[i]);
    }
    catch (ArithmaticException exc) {
        // catch the exception
        System.out.println("Can't divide by Zero!");
    }
}
}
catch (ArrayIndexOutOfBoundsException exc) {
    // catch the exception
    System.out.println("No matching element found.");
    System.out.println("Fatal error - program terminated.")
}
}
```

The output from the program is shown here:

```
4 / 2 is 2
Can't divide by Zero!
16 / 4 is 4
32 / 4 is 8
Can't divide by Zero!
128 / 8 is 16
No matching element found.
Fatal error - program terminated.
```

In this example, an exception that can be handled by the inner try—in this case, a divide-by-zero error—allows the program to continue. However, an array boundary error is caught by the outer try, which causes the program to terminate.

# Throwing an Exception

- It is possible to manually throw an exception by using the **throw** statement. Its general form is shown here:
- `throw exceptOb;`
- Here, *exceptOb* must be an object of an exception class derived from **Throwable**.
- Here is an example that illustrates the **throw** statement by manually throwing an **ArithmeticException**:

# throw Demo

```
// Manually throw an exception.  
class ThrowDemo {  
    public static void main(String args[]) {  
        try {  
            System.out.println("Before throw.");  
            throw new ArithmeticException(); ←———— Throw an exception.  
        }  
        catch (ArithmeticException exc) {  
            // catch the exception  
            System.out.println("Exception caught.");  
        }  
        System.out.println("After try/catch block.");  
    }  
}
```

The output from the program is shown here:

```
Before throw.  
Exception caught.  
After try/catch block.
```

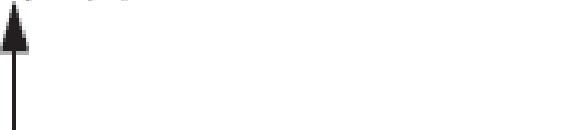
# Rethrowing an Exception

An exception caught by one catch statement can be rethrown so that it can be caught by an outer catch. The most likely reason for rethrowing this way is to allow multiple handlers access to the exception. For example, perhaps one exception handler manages one aspect of an exception, and a second handler copes with another aspect. Remember, when you rethrow an exception, it will not be recaught by the same catch statement. It will propagate to the next catch statement. The following program illustrates rethrowing an exception:

```
// Rethrow an exception.

class Rethrow {
    public static void genException() {
        // here, numer is longer than denom
        int numer[] = { 4, 8, 16, 32, 64, 128, 256, 512 };
        int denom[] = { 2, 0, 4, 4, 0, 8 };
```

```
for(int i=0; i<numer.length; i++) {  
    try {  
        System.out.println(numer[i] + " / " +  
                            denom[i] + " is " +  
                            numer[i]/denom[i]);  
    }  
    catch (ArithmeticException exc) {  
        // catch the exception  
        System.out.println("Can't divide by Zero!");  
    }  
    catch (ArrayIndexOutOfBoundsException exc) {  
        // catch the exception  
        System.out.println("No matching element found.");  
        throw exc; // rethrow the exception  
    }  
}
```



Rethrow the exception.

```
}

class RethrowDemo {
    public static void main(String args[]) {
        try {
            Rethrow.genException();
        }
        catch(ArrayIndexOutOfBoundsException exc) { ← Catch rethrown exception.
            // recatch exception
            System.out.println("Fatal error - " +
                "program terminated.");
        }
    }
}
```

In this program, divide-by-zero errors are handled locally, by `genException()`, but an array boundary error is rethrown. In this case, it is caught by `main()`.

Method	Description
<code>Throwable fillInStackTrace()</code>	Returns a <code>Throwable</code> object that contains a completed stack trace. This object can be rethrown.
<code>String getLocalizedMessage()</code>	Returns a localized description of the exception.
<code>String getMessage()</code>	Returns a description of the exception.
<code>void printStackTrace()</code>	Displays the stack trace.
<code>void printStackTrace(PrintStream stream)</code>	Sends the stack trace to the specified stream.
<code>void printStackTrace(PrintWriter stream)</code>	Sends the stack trace to the specified stream.
<code>String toString()</code>	Returns a <code>String</code> object containing a complete description of the exception. This method is called by <code>println()</code> when outputting a <code>Throwable</code> object.

Table 9-1 Commonly Used Methods Defined by `Throwable`

Of the methods defined by `Throwable`, two of the most interesting are `printStackTrace()` and `toString()`. You can display the standard error message plus a record of the method calls that lead up to the exception by calling `printStackTrace()`. You can use `toString()` to retrieve the standard error message. The `toString()` method is also called when an exception is used as an argument to `println()`. The following program demonstrates these methods:

```
// Using the Throwable methods.
```

```
class ExcTest {  
    static void genException() {  
        int nums[] = new int[4];  
  
        System.out.println("Before exception is generated.");  
  
        // generate an index out-of-bounds exception  
    }  
}
```

```
// generate an index out-of-bounds exception
nums[7] = 10;
System.out.println("this won't be displayed");
}
}

class UseThrowableMethods {
    public static void main(String args[]) {
        try {
            ExcTest.genException();
        }
        catch (ArrayIndexOutOfBoundsException exc) {
            // catch the exception
            System.out.println("Standard message is: ");
            System.out.println(exc);
            System.out.println("\nStack trace: ");
        }
    }
}
```

```
    exc.printStackTrace();
}
System.out.println("After catch statement.");
}
}
```

The output from this program is shown here:

Before exception is generated.

Standard message is:

java.lang.ArrayIndexOutOfBoundsException: 7

Stack trace:

java.lang.ArrayIndexOutOfBoundsException: 7

at ExcTest.genException(UseThrowableMethods.java:10)

at UseThrowableMethods.main(UseThrowableMethods.java:19)

After catch statement.

# Using finally

Sometimes you will want to define a block of code that will execute when a try/catch block is left. For example, an exception might cause an error that terminates the current method, causing its premature return. However, that method may have opened a file or a network connection that needs to be closed. Such types of circumstances are common in programming, and Java provides a convenient way to handle them: **finally**.

To specify a block of code to execute when a try/catch block is exited, include a **finally** block at the end of a try/catch sequence. The general form of a try/catch that includes finally is shown here.

```
try {  
    // block of code to monitor for errors  
}  
  
catch (ExcepType1 exOb) {  
    // handler for ExcepType1  
}  
  
}
```

```
    catch (ExcepType2 exOb) {  
        // handler for ExcepType2  
    }  
    //...  
    finally {  
        // finally code  
    }
```

The `finally` block will be executed whenever execution leaves a try/catch block, no matter what conditions cause it. That is, whether the `try` block ends normally, or because of an exception, the last code executed is that defined by `finally`. The `finally` block is also executed if any code within the `try` block or any of its `catch` statements return from the method.

Here is an example of `finally`:

```
// Use finally.  
class UseFinally {  
    public static void genException(int what) {  
        int t;  
        int nums [] = new int [2];  
  
        System.out.println("Receiving " + what);  
    }  
}
```

```
try {
    switch(what) {
        case 0:
            t = 10 / what; // generate div-by-zero error
            break;
        case 1:
            nums[4] = 4; // generate array index error.
            break;
        case 2:
            return; // return from try block
    }
}
catch (ArithmeticException exc) {
    // catch the exception
    System.out.println("Can't divide by Zero!");
    return; // return from catch
}
```

```
    catch (ArrayIndexOutOfBoundsException exc) {  
        // catch the exception  
        System.out.println("No matching element found.");  
    }  
    finally { ←—————  
        System.out.println("Leaving try.");  
    }  
}  
}
```

This is executed on the way  
out of try/catch blocks.

```
class FinallyDemo {  
    public static void main(String args[]) {  
        for(int i=0; i < 3; i++) {  
            UseFinally.genException(i);  
            System.out.println();  
        }  
    }  
}
```

Here is the output produced by the program:

```
Receiving 0
Can't divide by Zero!
Leaving try.
```

```
Receiving 1
No matching element found.
Leaving try.
```

```
Receiving 2
Leaving try.
```

As the output shows, no matter how the try block is exited, the finally block is executed.

# Using throws

- In some cases, if a method generates an exception that it does not handle, it must declare that exception in a **throws** clause. Here is the general form of a method that includes a **throws** clause:
  - *ret-type methName(param-list) throws except-list*
  - { // body }
- Here, *except-list* is a comma-separated list of exceptions that the method might throw outside of itself.
- Exceptions that are subclasses of **Error** or **RuntimeException** don't need to be specified in a **throws** list. Java simply assumes that a method may throw one. All other types of exceptions *do* need to be declared. Failure to do so causes a compile-time error.

```
// Use throws.  
class ThrowsDemo {  
    public static char prompt(String str)  
        throws java.io.IOException { ← Notice the throws clause.  
        System.out.print(str + ": ");  
        return (char) System.in.read();  
    }
```

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

```
    try {  
        ch = prompt("Enter a letter"); ←  
    }
```

```
    catch(java.io.IOException exc) {
```

Since `prompt()` might throw an exception, a call to it must be enclosed within a `try` block.

```
    catch(java.io.IOException exc) {  
        System.out.println("I/O exception occurred.");  
        ch = 'X';  
    }  
  
    System.out.println("You pressed " + ch);  
}  
}
```

On a related point, notice that **IOException** is fully qualified by its package name **java.io**. As you will learn in Chapter 10, Java's I/O system is contained in the **java.io** package. Thus, the **IOException** is also contained there. It would also have been possible to import **java.io** and then refer to **IOException** directly.

# Three Recently Added Exception Features

- The first supports *automatic resource management*, which automates the process of releasing a resource, such as a file, when it is no longer needed. It is based on an expanded form of **try**, called the ***try-with-resources*** statement.
- The second is multi-catch allows two or more exceptions to be caught by the same **catch** clause.
- The third is  
*final rethrow or more precise rethrow*

# Multi catch

```
// Use the multi-catch feature. Note: This code requires JDK 7 or
// later to compile.
class MultiCatch {
    public static void main(String args[]) {
        int a=88, b=0;
        int result;
        char chrs[] = { 'A', 'B', 'C' };

        for(int i=0; i < 2; i++) {
            try {
                if(i == 0)
                    result = a / b; // generate an ArithmeticException
                else
                    chrs[5] = 'X'; // generate an ArrayIndexOutOfBoundsException

                // This catch clause catches both exceptions.
            }
            catch(ArithmeticException | ArrayIndexOutOfBoundsException e) {
                System.out.println("Exception caught: " + e);
            }
        }

        System.out.println("After multi-catch.");
    }
}
```

# More precise rethrow

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

# Java's Built-in Exceptions

## *Unchecked exceptions:*

- The compiler does not check to see if a method handles or throws these exceptions.
- They need not be included in any method's **throws** list

## *Checked exceptions:*

- Must be included in a method's **throws** list if that method can generate one of these exceptions and does not handle it itself.

Exception	Meaning
ArithmaticException	Arithmatic error, such as integer divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
EnumConstantNotPresentException	An attempt is made to use an undefined enumeration value.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security.
StringIndexOutOfBoundsException	Attempt to index outside the bounds of a string.
TypeNotPresentException	Type not found.
UnsupportedOperationException	An unsupported operation was encountered.

Table 9-2 The Unchecked Exceptions Defined in `java.lang`

Exception	Meaning
ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object that does not implement the Cloneable interface.
IllegalAccessException	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.
ReflectiveOperationException	Superclass of reflection-related exceptions.

Table 9-3 The Checked Exceptions Defined in `java.lang`

# Creating Exception Subclasses

```
// Use a custom exception.  
  
// Create an exception.  
class NonIntResultException extends Exception {  
    int n;  
    int d;  
  
    NonIntResultException(int i, int j) {  
        n = i;  
        d = j;  
    }  
  
    public String toString() {  
        return "Result of " + n + " / " + d +  
            " is non-integer.";  
    }  
}
```

```
class CustomExceptDemo {  
    public static void main(String args[]) {  
  
        // Here, numer contains some odd values.  
        int numer[] = { 4, 8, 15, 32, 64, 127, 256, 512 };  
        int denom[] = { 2, 0, 4, 4, 0, 8 };  
  
        for(int i=0; i<numer.length; i++) {  
            try {  
                if( (numer[i] % 2) != 0)  
                    throw new  
                        NonIntResultException(numer[i], denom[i]);  
  
                System.out.println(numer[i] + " / " +  
                                denom[i] + " is " +  
                                numer[i]/denom[i]);  
            }  
        }  
    }  
}
```

```
        catch (ArithmetricException exc) {  
            // catch the exception  
            System.out.println("Can't divide by Zero!");  
        }  
        catch (ArrayIndexOutOfBoundsException exc) {  
            // catch the exception  
            System.out.println("No matching element found.");  
        }  
        catch (NonIntResultException exc) {  
            System.out.println(exc);  
        }  
    }  
}
```

The output from the program is shown here:

```
4 / 2 is 2  
Can't divide by Zero!  
Result of 15 / 4 is non-integer.  
32 / 4 is 8  
Can't divide by Zero!  
Result of 127 / 8 is non-integer.  
No matching element found.  
No matching element found.
```

# Questions

- What class is at the top of the exception hierarchy?
- What are checked and unchecked exceptions?
- Differentiate between throw and throws keywords.
- What is the significance of finally keyword?  
Explain.

- throw throws
- throw ExceptionObject throws comma separated list of checked Exception classes
- void fn()
- { ...
- ....
- throw e;
- }
- void fn() throws IOException, SQLException
- { ...
- ....
- throw e;
- }

# Chapter 11

# Multithreaded Programming

# Multithreading Fundamentals

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

**Process-based multitasking** is the feature that allows your computer to run two or more programs concurrently.

For example, it is process-based multitasking that allows you to run the Java compiler at the same time you are using a text editor or browsing the Internet. In process-based multitasking, a program is the smallest unit of code that can be dispatched by the scheduler.

In a **thread-based multitasking** environment, the thread is the smallest unit of dispatchable code. This means that a single program can perform two or more tasks at once. For instance, a text editor can be formatting text at the same time that it is printing, as long as these two actions are being performed by two separate threads.

# Multithreading Fundamentals...

A principal advantage of multithreading is that it enables you to write very efficient programs because **it lets you utilize the idle time that is present in most programs.**

As you probably know, most I/O devices, whether they be network ports, disk drives, or the keyboard, are much slower than the CPU. Thus, a program will often spend a majority of its execution time waiting to send or receive information to or from a device.

By using multithreading, your program can execute another task during this idle time. For example, while one part of your program is sending a file over the Internet, another part can be reading keyboard input, and still another can be buffering the next block of data to send.

# Multithreading Fundamentals...

Java's multithreading features work in both types of systems.

In a **single-core system**, concurrently executing threads share the CPU, with each thread receiving a slice of CPU time.

Therefore, in a single-core system, two or more threads do not actually run at the same time, but idle CPU time is utilized.

However, in **multiprocessor/multicore systems**, it is possible for two or more threads to actually execute simultaneously. In many cases, this can further improve program efficiency and increase the speed of certain operations.

# Multithreading Fundamentals...

A thread can be in one of several states. It can be *running*. It can be *ready to run* as soon as it gets CPU time. A running thread can be *suspended*, which is a temporary halt to its execution. It can later be *resumed*. A thread can be *blocked* when waiting for a resource. A thread can be *terminated*, in which case its execution ends and cannot be resumed.

Along with thread-based multitasking comes the need for a special type of feature called *synchronization*, which allows the execution of threads to be coordinated in certain well-defined ways. Java has a complete subsystem devoted to synchronization

# The Thread Class and Runnable Interface

Java's multithreading system is built upon the **Thread** class and its companion interface, **Runnable**. Both are packaged in `java.lang`. **Thread** encapsulates a thread of execution. To create a new thread, your program will either extend **Thread** or implement the **Runnable** interface.

The **Thread** class defines several methods that help manage threads. Here are some of the more commonly used ones (we will be looking at these more closely as they are used):

Method	Meaning
<code>final String getName()</code>	Obtains a thread's name.
<code>final int getPriority()</code>	Obtains a thread's priority.
<code>final boolean isAlive()</code>	Determines whether a thread is still running.
<code>final void join()</code>	Waits for a thread to terminate.
<code>void run()</code>	Entry point for the thread.
<code>static void sleep(long milliseconds)</code>	Suspends a thread for a specified period of milliseconds.
<code>void start()</code>	Starts a thread by calling its <code>run()</code> method.

All processes have at least one thread of execution, which is usually called the *main thread*, because it is the one that is executed when your program begins. Thus, the main thread is the thread that all of the preceding example programs in the book have been using. From the main thread, you can create other threads.

## Creating a Thread

You create a thread by instantiating an object of type `Thread`. The `Thread` class encapsulates an object that is runnable. As mentioned, Java defines two ways in which you can create a runnable object:

- You can implement the `Runnable` interface.
- You can extend the `Thread` class.

The **Runnable** interface abstracts a unit of executable code. You can construct a thread on any object that implements the **Runnable** interface. **Runnable** defines only one method called **run()**, which is declared like this:

```
public void run( )
```

Inside **run()**, you will define the code that constitutes the new thread. It is important to understand that **run()** can call other methods, use other classes, and declare variables just like the main thread. The only difference is that **run()** establishes the entry point for another, concurrent thread of execution within your program. This thread will end when **run()** returns.

After you have created a class that implements **Runnable**, you will instantiate an object of type **Thread** on an object of that class. **Thread** defines several constructors. The one that we will use first is shown here:

```
Thread(Runnable threadOb)
```

In this constructor, *threadOb* is an instance of a class that implements the **Runnable** interface. This defines where execution of the thread will begin.

Once created, the new thread will not start running until you call its *start()* method, which is declared within **Thread**. In essence, *start()* executes a call to *run()*. The *start()* method is shown here:

```
void start()
```

Here is an example that creates a new thread and starts it running:

```
// Create a thread by implementing Runnable.
```

```
class MyThread implements Runnable {  
    String thrdName;
```

Objects of **MyThread** can be run in their own threads because **MyThread** implements **Runnable**.

```
MyThread(String name) {  
    thrdName = name;  
}  
  
// Entry point of thread.  
public void run() { ←———— Threads start executing here.  
    System.out.println(thrdName + " starting.");  
    try {  
        for(int count=0; count < 10; count++) {  
            Thread.sleep(400);  
            System.out.println("In " + thrdName +  
                ", count is " + count);  
        }  
    }  
    catch(InterruptedException exc) {  
        System.out.println(thrdName + " interrupted.");  
    }  
    System.out.println(thrdName + " terminating.");  
}
```

```
class UseThreads {  
    public static void main(String args[]) {  
        System.out.println("Main thread starting.");  
  
        // First, construct a MyThread object.  
        MyThread mt = new MyThread("Child #1"); ← Create a runnable object.  
  
        // Next, construct a thread from that object.  
        Thread newThrd = new Thread(mt); ← Construct a thread on that object.  
  
        // Finally, start execution of the thread.  
        newThrd.start(); ← Start running the thread.  
  
        for(int i=0; i<50; i++) {  
            System.out.print(".");  
            try {  
                Thread.sleep(100);  
            }  
            catch(InterruptedException exc) {  
                System.out.println("Main thread interrupted.");  
            }  
        }  
  
        System.out.println("Main thread ending.");
```

- The **java.lang.Thread.sleep(long millis)** method causes the currently executing thread to sleep for the specified number of milliseconds, subject to the precision and accuracy of system timers and schedulers
- Following is the declaration for `java.lang.Thread.sleep()` method

**public static void sleep(long millis) throws InterruptedException**

millis – This is the length of time to sleep in milliseconds.

```
Main thread starting.  
.Child #1 starting.  
...In Child #1, count is 0  
....In Child #1, count is 1  
....In Child #1, count is 2  
...In Child #1, count is 3  
....In Child #1, count is 4  
....In Child #1, count is 5  
....In Child #1, count is 6  
...In Child #1, count is 7  
....In Child #1, count is 8  
....In Child #1, count is 9  
child #1 terminating.  
.....Main thread ending.
```

One other point: In a multithreaded program, you often will want the main thread to be the last thread to finish running. As a general rule, a program continues to run until all of its threads have ended. Thus, having the main thread finish last is not a requirement. It is, however, often a good practice to follow—especially when you are first learning about threads.

```
// Improved MyThread.

class MyThread implements Runnable {
    Thread thrd; ← A reference to the thread is stored in thrd.

    // Construct a new thread.
    MyThread(String name) {
        thrd = new Thread(this, name); ← The thread is named when it is created.
        thrd.start(); // start the thread ← Begin executing the thread.
    }

    // Begin execution of new thread.
    public void run() {
        System.out.println(thrd.getName() + " starting.");
        try {
            for(int count=0; count<10; count++) {
                Thread.sleep(400);
                System.out.println("In " + thrd.getName() +
                                   ", count is " + count);
            }
        }
    }
}
```

```
catch(InterruptedException exc) {  
    System.out.println(thrd.getName() + " interrupted.");  
}  
System.out.println(thrd.getName() + " terminating.");  
}  
  
}  
  
class UseThreadsImproved {  
    public static void main(String args[]) {  
        System.out.println("Main thread starting.");  
  
        MyThread mt = new MyThread("Child #1");  
  
        for(int i=0; i < 50; i++) {  
            System.out.print(".");  
            try {  
                Thread.sleep(100);  
            }  
            catch(InterruptedException exc) {  
                System.out.println("Main thread interrupted.");  
            }  
        }  
  
        System.out.println("Main thread ending.");  
    }  
}
```

Now the thread starts when it is created.

# Extending Thread

Implementing **Runnable** is one way to create a class that can instantiate thread objects. Extending **Thread** is the other. In this project, you will see how to extend **Thread** by creating a program functionally identical to the **UseThreadsImproved** program.

When a class extends **Thread**, it must override the **run()** method, which is the entry point for the new thread. It must also call **start()** to begin execution of the new thread. It is possible to override other **Thread** methods, but doing so is not required.

```
Extend Thread.  
*/  
class MyThread extends Thread {  
  
    // Construct a new thread.  
    MyThread(String name) {  
        super(name); // name thread  
        start(); // start the thread  
    }  
}
```

```
// Begin execution of new thread.  
public void run() {  
    System.out.println(getName() + " starting.");  
    try {  
        for(int count=0; count < 10; count++) {  
            Thread.sleep(400);  
            System.out.println("In " + getName() +  
                               ", count is " + count);  
        }  
    }  
    catch(InterruptedException exc) {  
        System.out.println(getName() + " interrupted.");  
    }  
    System.out.println(getName() + " terminating.");  
}
```

```
class ExtendThread {  
    public static void main(String args[]) {  
        System.out.println("Main thread starting.");  
  
        MyThread mt = new MyThread("Child #1");  
  
        for(int i=0; i < 50; i++) {  
            System.out.print(".");  
            try {  
                Thread.sleep(100);  
            }  
            catch(InterruptedException exc) {  
                System.out.println("Main thread interrupted.");  
            }  
        }  
  
        System.out.println("Main thread ending.");  
    }  
}
```

```
// Create multiple threads.  
  
class MyThread implements Runnable {  
    Thread thrd;  
  
    // Construct a new thread.  
    MyThread(String name) {  
        thrd = new Thread(this, name);  
  
        thrd.start(); // start the thread  
    }  
  
    // Begin execution of new thread.  
    public void run() {  
        System.out.println(thrd.getName() + " starting.");  
    }  
}
```

```
try {
    for(int count=0; count < 10; count++) {
        Thread.sleep(400);
        System.out.println("In " + thrd.getName() +
                           ", count is " + count);
    }
}
catch(InterruptedException exc) {
    System.out.println(thrd.getName() + " interrupted.");
}
System.out.println(thrd.getName() + " terminating.");
}
```

```
class MoreThreads {
    public static void main(String args[]) {
        System.out.println("Main thread starting.");
```

```
MyThread mt1 = new MyThread("Child #1");
MyThread mt2 = new MyThread("Child #2"); ← Create and start
MyThread mt3 = new MyThread("Child #3"); executing three threads.
```

```
for(int i=0; i < 50; i++) {
    System.out.print(".");
    try {
        Thread.sleep(100);
    }
    catch(InterruptedException exc) {
        System.out.println("Main thread interrupted.");
    }
}
System.out.println("Main thread ending.");
}
```

Main thread starting.

Child #1 starting.

. Child #2 starting.

Child #3 starting.

... In Child #3, count is 0

In Child #2, count is 0

In Child #1, count is 0

.... In Child #1, count is 1

In Child #2, count is 1

In Child #3, count is 1

.... In Child #2, count is 2

In Child #3, count is 2

In Child #1, count is 2

... In Child #1, count is 3

In Child #2, count is 3

In Child #3, count is 3

.... In Child #1, count is 4

In Child #3, count is 4

In Child #2, count is 4

.... In Child #1, count is 5

In Child #3, count is 5

In Child #2, count is 5

... In Child #3, count is 6

. In Child #2, count is 6

In Child #1, count is 6

... In Child #3, count is 7

In Child #1, count is 7

In Child #2, count is 7

.... In Child #2, count is 8

In Child #1, count is 8

In Child #3, count is 8

.... In Child #1, count is 9

Child #1 terminating.

In Child #2, count is 9

Child #2 terminating.

In Child #3, count is 9

Child #3 terminating.

..... Main thread ending.

## Determining When a Thread Ends

Fortunately, **Thread** provides two means by which you can determine if a thread has ended. First, you can call **isAlive( )** on the thread. Its general form is shown here:

```
final boolean isAlive()
```

The **isAlive( )** method returns **true** if the thread upon which it is called is still running. It returns **false** otherwise. To try **isAlive( )**, substitute this version of **MoreThreads** for the one shown in the preceding program:

```
// Use isAlive().  
class MoreThreads {  
    public static void main(String args[]) {  
        System.out.println("Main thread starting.");  
  
        MyThread mt1 = new MyThread("Child #1");  
        mt1.start();  
        System.out.println("Waiting for child...");  
        if (mt1.isAlive())  
            System.out.println("Child is still alive");  
        else  
            System.out.println("Child has ended");  
    }  
}
```

```
MyThread mt2 = new MyThread("Child #2");
MyThread mt3 = new MyThread("Child #3");
do {
    System.out.print(".");
    try {
        Thread.sleep(100);
    }
    catch(InterruptedException exc) {
        System.out.println("Main thread interrupted.");
    }
} while (mt1.thrd.isAlive() ||
         mt2.thrd.isAlive() || ← This waits until all threads terminate.
         mt3.thrd.isAlive());
System.out.println("Main thread ending.");
}
```

This version produces output that is similar to the previous version, except that `main()` ends as soon as the other threads finish. The difference is that it uses `isAlive()` to wait for the child threads to terminate. Another way to wait for a thread to finish is to call `join()`, shown here:

```
final void join() throws InterruptedException
```

This method waits until the thread on which it is called terminates. Its name comes from the concept of the calling thread waiting until the specified thread *joins* it. Additional forms of `join()` allow you to specify a maximum amount of time that you want to wait for the specified thread to terminate.

Here is a program that uses `join()` to ensure that the main thread is the last to stop:

```
// Use join().  
  
class MyThread implements Runnable {  
    Thread thrd;
```

```
// Construct a new thread.  
MyThread(String name) {  
    thrd = new Thread(this, name);  
    thrd.start(); // start the thread  
}  
  
// Begin execution of new thread.  
public void run() {  
    System.out.println(thrd.getName() + " starting.");  
    try {  
        for(int count=0; count < 10; count++) {  
            Thread.sleep(400);  
            System.out.println("In " + thrd.getName() +  
                ", count is " + count);  
        }  
    }  
}
```

```
        catch(InterruptedException exc) {
            System.out.println(thrd.getName() + " interrupted.");
        }
        System.out.println(thrd.getName() + " terminating.");
    }
}
```

```
class JoinThreads {
    public static void main(String args[]) {
        System.out.println("Main thread starting.");

        MyThread mt1 = new MyThread("Child #1");
        MyThread mt2 = new MyThread("Child #2");
        MyThread mt3 = new MyThread("Child #3");
    }
}
```

```
try {
    mt1.thrd.join(); ←
    System.out.println("Child #1 joined.");
    mt2.thrd.join(); ←
    System.out.println("Child #2 joined.");
    mt3.thrd.join(); ←
    System.out.println("Child #3 joined.");
}
catch(InterruptedException exc) {
    System.out.println("Main thread interrupted.");
}
System.out.println("Main thread ending.");
}
```

Wait until the specified thread ends.

Main thread starting.  
Child #1 starting.  
Child #2 starting.  
Child #3 starting.  
In Child #2, count is 0  
In Child #1, count is 0  
In Child #3, count is 0  
In Child #2, count is 1  
In Child #3, count is 1  
In Child #1, count is 1  
In Child #2, count is 2  
In Child #1, count is 2  
In Child #3, count is 2  
In Child #2, count is 3  
In Child #3, count is 3  
In Child #1, count is 3  
In Child #2, count is 4  
In Child #3, count is 4  
In Child #1, count is 4  
In Child #3, count is 5  
In Child #1, count is 5  
In Child #2, count is 5  
In Child #3, count is 6  
In Child #2, count is 6  
In Child #1, count is 6  
In Child #3, count is 7  
In Child #1, count is 7  
In Child #2, count is 7  
In Child #3, count is 8  
In Child #2, count is 8  
In Child #1, count is 8  
In Child #3, count is 9  
Child #3 terminating.  
In Child #2, count is 9  
Child #2 terminating.  
In Child #1, count is 9  
Child #1 terminating.  
Child #1 joined.  
Child #2 joined.  
Child #3 joined.  
Main thread ending.

# Thread Priorities

Each thread has associated with it a priority setting. A thread's priority determines, in part, how much CPU time a thread receives relative to the other active threads. In general, over a given period of time, low-priority threads receive little. High-priority threads receive a lot. As you might expect, how much CPU time a thread receives has profound impact on its execution characteristics and its interaction with other threads currently executing in the system.

It is important to understand that factors other than a thread's priority also affect how much CPU time a thread receives. For example, if a high-priority thread is waiting on some resource, perhaps for keyboard input, then it will be blocked, and a lower priority thread will run. However, when that high-priority thread gains access to the resource, it can preempt the low-priority thread and resume execution. Another factor that affects the scheduling of threads is the way the operating system implements multitasking. (See "Ask the Expert" at the end of this section.) Thus, just because you give one thread a high priority and another a low priority does not necessarily mean that one thread will run faster or more often than the other. It's just that the high-priority thread has greater potential access to the CPU.

When a child thread is started, its priority setting is equal to that of its parent thread. You can change a thread's priority by calling `setPriority()`, which is a member of `Thread`. This is its general form:

```
final void setPriority(int level)
```

Here, `level` specifies the new priority setting for the calling thread. The value of `level` must be within the range `MIN_PRIORITY` and `MAX_PRIORITY`. Currently, these values are 1 and 10, respectively. To return a thread to default priority, specify `NORM_PRIORITY`, which is currently 5. These priorities are defined as `static final` variables within `Thread`.

You can obtain the current priority setting by calling the `getPriority()` method of `Thread`, shown here:

```
final int getPriority()
```

# Synchronization

When using multiple threads, it is sometimes necessary to coordinate the activities of two or more. The process by which this is achieved is called *synchronization*. The most common reason for synchronization is when two or more threads need access to a shared resource that can be used by only one thread at a time. For example, when one thread is writing to a file, a second thread must be prevented from doing so at the same time. Another reason for synchronization is when one thread is waiting for an event that is caused by another thread. In this case, there must be some means by which the first thread is held in a suspended state until the event has occurred. Then, the waiting thread must resume execution.

Key to synchronization in Java is the concept of the *monitor*, which controls access to an object. A monitor works by implementing the concept of a *lock*. When an object is locked by one thread, no other thread can gain access to the object. When the thread exits, the object is unlocked and is available for use by another thread.

All objects in Java have a monitor. This feature is built into the Java language, itself. Thus, all objects can be synchronized. Synchronization is supported by the keyword **synchronized** and a few well-defined methods that all objects have. Since synchronization was designed into Java from the start, it is much easier to use than you might first expect. In fact, for many programs, the synchronization of objects is almost transparent.

There are two ways that you can synchronize your code. Both involve the use of the **synchronized** keyword, and both are examined here.

## Using Synchronized Methods

You can synchronize access to a method by modifying it with the **synchronized** keyword. When that method is called, the calling thread enters the object's monitor, which then locks the object. While locked, no other thread can enter the method, or enter any other synchronized method defined by the object's class. When the thread returns from the method, the monitor unlocks the object, allowing it to be used by the next thread. Thus, synchronization is achieved with virtually no programming effort on your part.

```
// Use synchronize to control access.
```

```
class SumArray {  
    private int sum;
```

```
synchronized int sumArray(int nums[]) { ←———— sumArray() is synchronized.  
    sum = 0; // reset sum
```

```
    for(int i=0; i<nums.length; i++) {  
        sum += nums[i];  
        System.out.println("Running total for " +  
            Thread.currentThread().getName() +  
                " is " + sum);
```

```
    try {  
        Thread.sleep(10); // allow task-switch  
    }
```

```
        catch(InterruptedException exc) {
            System.out.println("Thread interrupted.");
        }
    }
    return sum;
}
}
```

```
class MyThread implements Runnable {
    Thread thrd;
    static SumArray sa = new SumArray();
    int a[];
    int answer;

    // Construct a new thread.
    MyThread(String name, int nums[]) {
        thrd = new Thread(this, name);
```

```
a = nums;
thrd.start(); // start the thread
}

// Begin execution of new thread.
public void run() {
    int sum;

    System.out.println(thrd.getName() + " starting.");

    answer = sa.sumArray(a);
    System.out.println("Sum for " + thrd.getName() +
                       " is " + answer);

    System.out.println(thrd.getName() + " terminating.");
}
```

```
class Sync {  
    public static void main(String args[]) {  
        int a[] = {1, 2, 3, 4, 5};  
  
        MyThread mt1 = new MyThread("Child #1", a);  
        MyThread mt2 = new MyThread("Child #2", a);  
  
        try {  
            mt1.thrd.join();  
            mt2.thrd.join();  
        }  
        catch(InterruptedException exc) {  
            System.out.println("Main thread interrupted.");  
        }  
    }  
}
```

Child #1 starting.

Running total for Child #1 is 1

Child #2 starting.

Running total for Child #1 is 3

Running total for Child #1 is 6

Running total for Child #1 is 10

Running total for Child #1 is 15

Sum for Child #1 is 15

Child #1 terminating.

Running total for Child #2 is 1

Running total for Child #2 is 3

Running total for Child #2 is 6

Running total for Child #2 is 10

Running total for Child #2 is 15

Sum for Child #2 is 15

Child #2 terminating.

- A synchronized method is created by preceding its declaration with **synchronized**.
- For any given object, once a synchronized method has been called, the object is locked and no synchronized methods on the same object can be used by another thread of execution.
- Other threads trying to call an in-use synchronized object will enter a wait state until the object is unlocked.
- When a thread leaves the synchronized method, the object is unlocked.

# The synchronized Statement

Although creating **synchronized** methods within classes that you create is an easy and effective means of achieving synchronization, it will not work in all cases. For example, you might want to synchronize access to some method that is not modified by **synchronized**. This can occur because you want to use a class that was not created by you but by a third party, and you do not have access to the source code. Thus, it is not possible for you to add **synchronized** to the appropriate methods within the class. How can access to an object of this class be synchronized? Fortunately, the solution to this problem is quite easy: You simply put calls to the methods defined by this class inside a **synchronized** block.

This is the general form of a **synchronized** block:

```
synchronized(objref) {  
    // statements to be synchronized  
}
```

```
// Use a synchronized block to control access to SumArray
class SumArray {
    private int sum;

    int sumArray(int nums[]) { ← Here, sumArray()
        sum = 0; // reset sum                         is not synchronized.

        for(int i=0; i<nums.length; i++) {
            sum += nums[i];
            System.out.println("Running total for " +
                Thread.currentThread().getName() +
                " is " + sum);
        try {
            Thread.sleep(10); // allow task-switch
        }
        catch(InterruptedException exc) {
            System.out.println("Thread interrupted.");
        }
    }
}
```

```
        }
        return sum;
    }
}

class MyThread implements Runnable {
    Thread thrd;
    static SumArray sa = new SumArray();
    int a[];
    int answer;

    // Construct a new thread.
    MyThread(String name, int nums[]) {
        thrd = new Thread(this, name);
        a = nums;
        thrd.start(); // start the thread
    }

    // Begin execution of new thread.
    public void run() {
        int sum;
```

```
System.out.println(thrd.getName() + " starting.");  
  
    // synchronize calls to sumArray()  
    synchronized(sa) { ← Here, calls to sumArray()  
        answer = sa.sumArray(a);          on sa are synchronized.  
    }  
    System.out.println("Sum for " + thrd.getName() +  
                      " is " + answer);  
  
    System.out.println(thrd.getName() + " terminating.");  
}  
}  
  
class Sync {  
    public static void main(String args[]) {  
        int a[] = {1, 2, 3, 4, 5};
```

```
MyThread mt1 = new MyThread("Child #1", a);
MyThread mt2 = new MyThread("Child #2", a);

try {
    mt1.thrd.join();
    mt2.thrd.join();
} catch(InterruptedException exc) {
    System.out.println("Main thread interrupted.");
}
}
```

This version produces the same, correct output as the one shown earlier that uses a synchronized method.

```
MyThread(String name, int nums[]) {  
    thrd = new Thread(this, name);  
    a = nums;  
    thrd.start(); // start the thread  
}
```

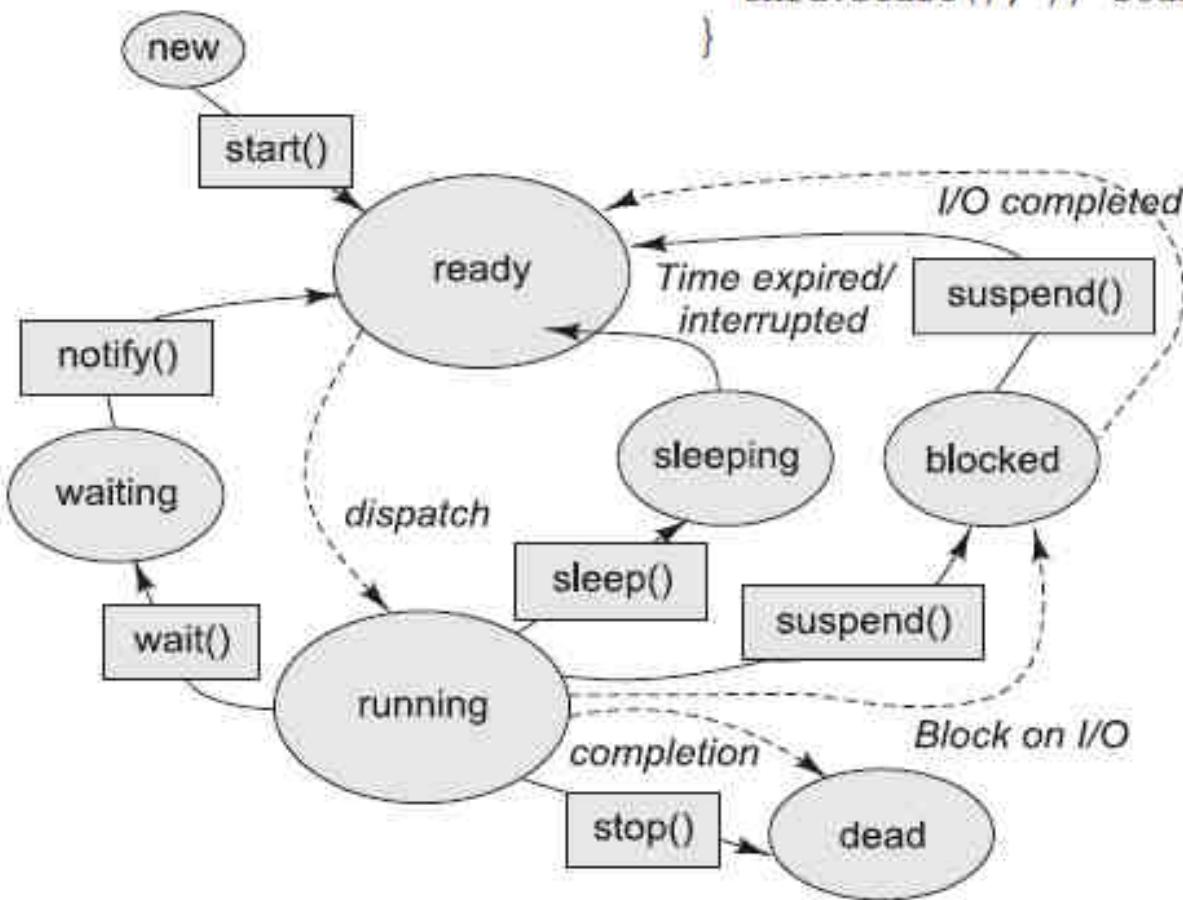
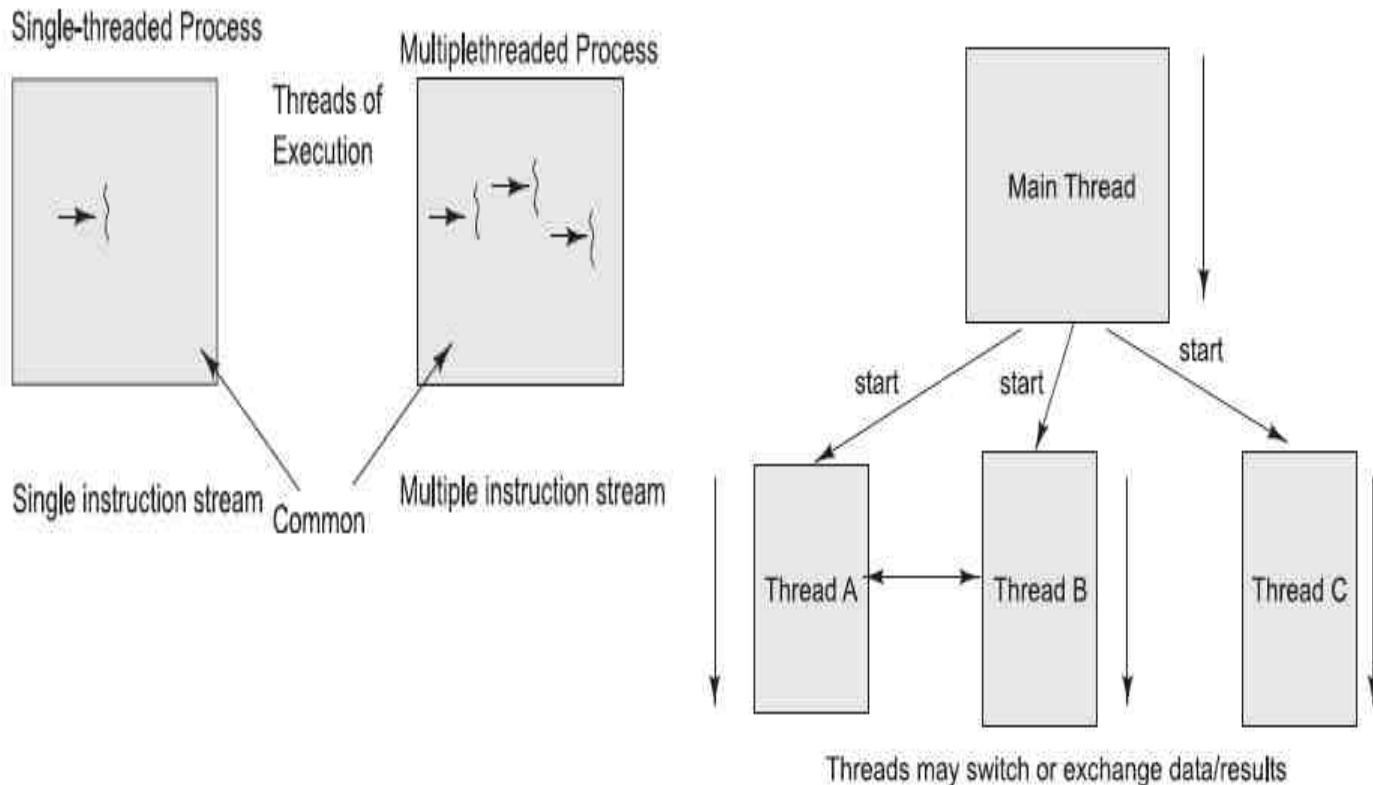


Fig. 14.4 Life cycle of Java threads



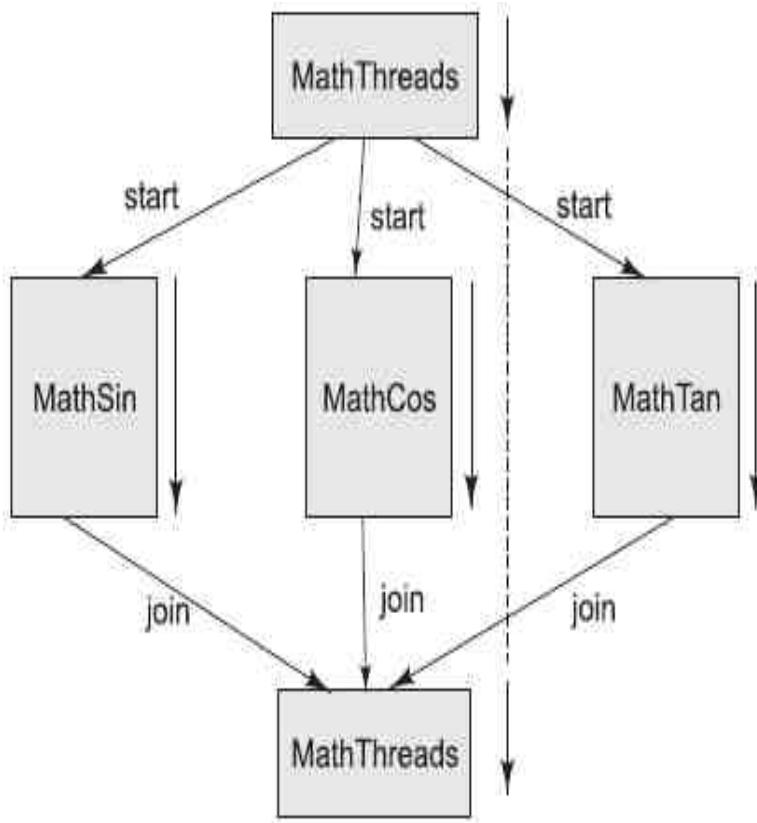


Fig. 14.5 Flow of control in a master and multiple workers threads application

# Thread Communication Using `notify()`, `wait()`, and `notifyAll()`

Consider the following situation. A thread called T is executing inside a synchronized method and needs access to a resource called R that is temporarily unavailable. What should T do? If T enters some form of polling loop that waits for R, T ties up the object, preventing other threads' access to it. This is a less than optimal solution because it partially defeats the advantages of programming for a multithreaded environment. A better solution is to have T temporarily relinquish control of the object, allowing another thread to run. When R becomes available, T can be notified and resume execution. Such an approach relies upon some form of interthread communication in which one thread can notify another that it is blocked and be notified that it can resume execution. Java supports interthread communication with the `wait()`, `notify()`, and `notifyAll()` methods.

The `wait()`, `notify()`, and `notifyAll()` methods are part of all objects because they are implemented by the `Object` class. These methods should be called only from within a `synchronized` context. Here is how they are used. When a thread is temporarily blocked from running, it calls `wait()`. This causes the thread to go to sleep and the monitor for that object to be released, allowing another thread to use the object. At a later point, the sleeping thread is awakened when some other thread enters the same monitor and calls `notify()`, or `notifyAll()`.

Following are the various forms of `wait()` defined by `Object`:

`final void wait() throws InterruptedException`

`final void wait(long millis) throws InterruptedException`

`final void wait(long millis, int nanos) throws InterruptedException`

The first form waits until notified. The second form waits until notified or until the specified period of milliseconds has expired. The third form allows you to specify the wait period in terms of nanoseconds.

Here are the general forms for **notify( )** and **notifyAll( )**:

```
final void notify()
```

```
final void notifyAll()
```

A call to **notify( )** resumes one waiting thread. A call to **notifyAll( )** notifies all threads, with the highest priority thread gaining access to the object.

Before looking at an example that uses **wait( )**, an important point needs to be made. Although **wait( )** normally waits until **notify( )** or **notifyAll( )** is called, there is a possibility that in very rare cases the waiting thread could be awakened due to a *spurious wakeup*. The conditions that

```
// A correct implementation of a producer and consumer.
class Q {
    int n;
    boolean valueSet = false;

    synchronized int get() {
        while(!valueSet)
            try {
                wait();
            } catch(InterruptedException e) {
                System.out.println("InterruptedException caught");
            }

        System.out.println("Got: " + n);
        valueSet = false;
        notify();
        return n;
    }
}
```

```
synchronized void put(int n) {
    while(valueSet)
        try {
            wait();
        } catch(InterruptedException e) {
            System.out.println("InterruptedException caught");
        }

    this.n = n;
    valueSet = true;
    System.out.println("Put: " + n);
    notify();
}
}
```

```
class Producer implements Runnable {
    Q q;
    Producer(Q q) {
        this.q = q;
        new Thread(this, "Producer").start();
    }
    public void run() {
        int i = 0;
        while(true) {
            q.put(i++);
        }
    }
}
class Consumer implements Runnable {
    Q q;
    Consumer(Q q) {
        this.q = q;
        new Thread(this, "Consumer").start();
    }
}
```

```
public void run() {
    while(true) {
        q.get();
    }
}

class PCFixed {
    public static void main(String args[]) {
        Q q = new Q();
        new Producer(q);
        new Consumer(q);

        System.out.println("Press Control-C to stop.");
    }
}
```

## An Example That Uses `wait()` and `notify()`

To understand the need for and the application of `wait()` and `notify()`, we will create a program that simulates the ticking of a clock by displaying the words Tick and Tock on the screen. To accomplish this, we will create a class called `TickTock` that contains two methods: `tick()` and `tock()`. The `tick()` method displays the word "Tick", and `tock()` displays "Tock". To run the clock, two threads are created, one that calls `tick()` and one that calls `tock()`. The goal is to make the two threads execute in a way that the output from the program displays a consistent "Tick Tock"—that is, a repeated pattern of one tick followed by one tock.

```
// Use wait() and notify() to create a ticking clock.  
  
class TickTock {  
  
    String state; // contains the state of the clock
```

```
synchronized void tick(boolean running) {  
    if(!running) { // stop the clock  
        state = "ticked";  
        notify(); // notify any waiting threads  
        return;  
    }  
  
    System.out.print("Tick ");  
  
    state = "ticked"; // set the current state to ticked  
  
    notify(); // let tock() run ← tick() notifies tock().  
    try {  
        while(!state.equals("tocked"))  
            wait(); // wait for tock() to complete ← tick() waits for tock()  
    }  
    catch(InterruptedException exc) {  
        System.out.println("Thread interrupted.");  
    }  
}
```

```
        catch(InterruptedException exc) {  
            System.out.println("Thread interrupted.");  
        }  
    }  
  
    synchronized void tock(boolean running) {  
        if(!running) { // stop the clock  
            state = "tocked";  
            notify(); // notify any waiting threads  
            return;  
        }  
  
        System.out.println("Tock");  
  
        state = "tocked"; // set the current state to tocked  
        notify(); // let tick() run ←———— tock() notifies tick().
```

```
try {
    while(!state.equals("ticked"))
        wait(); // wait for tick to complete ← tock( ) waits for tick( ).}
}
catch(InterruptedException exc) {
    System.out.println("Thread interrupted.");
}
}

class MyThread implements Runnable {
    Thread thrd;
    TickTock tt0b;

    // Construct a new thread.
    MyThread(String name, TickTock tt) {
        thrd = new Thread(this, name);
        tt0b = tt;
        thrd.start(); // start the thread
    }
}
```

```
// Begin execution of new thread.
public void run() {

    if(thrd.getName().compareTo("Tick") == 0) {
        for(int i=0; i<5; i++) ttOb.tick(true);
        ttOb.tick(false);
    }
    else {
        for(int i=0; i<5; i++) ttOb.tock(true);
        ttOb.tock(false);
    }
}

class ThreadCom {
    public static void main(String args[]) {
        TickTock tt = new TickTock();
        MyThread mt1 = new MyThread("Tick", tt);
        MyThread mt2 = new MyThread("Tock", tt);
```

```
try {
    mt1.thrd.join();
    mt2.thrd.join();
} catch(InterruptedException exc) {
    System.out.println("Main thread interrupted.");
}
}
```

Here is the output produced by the program:

Tick Tock  
Tick Tock  
Tick Tock  
Tick Tock  
Tick Tock

# Suspending, Resuming, and Stopping Threads

It is sometimes useful to suspend execution of a thread. For example, a separate thread can be used to display the time of day. If the user does not desire a clock, then its thread can be suspended. Whatever the case, it is a simple matter to suspend a thread. Once suspended, it is also a simple matter to restart the thread.

The mechanisms to suspend, stop, and resume threads differ between early versions of Java and more modern versions, beginning with Java 2. Prior to Java 2, a program used **suspend()**, **resume()**, and **stop()**, which are methods defined by **Thread**, to pause, restart, and stop the execution of a thread. They have the following forms:

`final void resume()`

`final void suspend()`

`final void stop()`

While these methods seem to be a perfectly reasonable and convenient approach to managing the execution of threads, they must no longer be used. Here's why. The **suspend( )** method of the **Thread** class was deprecated by Java 2. This was done because **suspend( )** can sometimes cause serious problems that involve deadlock. The **resume( )** method is also deprecated. It does not cause problems but cannot be used without the **suspend( )** method as its counterpart. The **stop( )** method of the **Thread** class was also deprecated by Java 2. This was done because this method too can sometimes cause serious problems.

Since you cannot now use the **suspend( )**, **resume( )**, or **stop( )** methods to control a thread, you might at first be thinking that there is no way to pause, restart, or terminate a thread. But, fortunately, this is not true. Instead, a thread must be designed so that the **run( )** method periodically checks to determine if that thread should suspend, resume, or stop its own execution. Typically, this is accomplished by establishing two flag variables: one for suspend and resume, and one for stop. For suspend and resume, as long as the flag is set to "running," the **run( )** method must continue to let the thread execute. If this variable is set to "suspend," the thread must pause. For the stop flag, if it is set to "stop," the thread must terminate.

```
// Suspending, resuming, and stopping a thread

class MyThread implements Runnable {
    Thread thrd;

    boolean suspended; ← Suspends thread when true.
    boolean stopped; ← Stops thread when true.

    MyThread(String name) {
        thrd = new Thread(this, name);
        suspended = false;
        stopped = false;
        thrd.start();
    }

    // This is the entry point for thread.
```

```
public void run() {  
    System.out.println(thrd.getName() + " starting.");  
    try {  
        for(int i = 1; i < 1000; i++) {  
            System.out.print(i + " ");  
            if((i%10)==0) {  
                System.out.println();  
                Thread.sleep(250);  
            }  
  
            // Use synchronized block to check suspended and stopped.  
            synchronized(this) { ←  
                while(suspended) {  
                    wait();  
                }  
                if(stopped) break;  
            }  
        }  
    }  
}
```

This synchronized block checks  
suspended and stopped.

```
        }
    }
} catch (InterruptedException exc) {
    System.out.println(thrd.getName() + " interrupted.");
}
System.out.println(thrd.getName() + " exiting.");
}

// Stop the thread.
synchronized void mystop() {
    stopped = true;
    // The following ensures that a suspended thread can be stopped.
    suspended = false;
    notify();
}
```

```
// Suspend the thread.  
synchronized void mysuspend() {  
    suspended = true;  
}  
  
// Resume the thread.  
synchronized void myresume() {  
    suspended = false;  
    notify();  
}  
}  
  
class Suspend {  
    public static void main(String args[]) {  
        MyThread ob1 = new MyThread("My Thread");  
        ob1.start();  
        try {  
            Thread.sleep(1000);  
        } catch (InterruptedException e) {}  
        ob1.mysuspend();  
        System.out.println("Resuming the thread");  
        ob1.myresume();  
        ob1.join();  
    }  
}
```

```
try {  
    Thread.sleep(1000); // let ob1 thread start executing  
  
    ob1.mysuspend();  
    System.out.println("Suspending thread.");  
    Thread.sleep(1000);  
  
    ob1.myresume();  
    System.out.println("Resuming thread.");  
    Thread.sleep(1000);  
  
    ob1.mysuspend();  
    System.out.println("Suspending thread.");  
    Thread.sleep(1000);
```

```
    ob1.myresume();
    System.out.println("Resuming thread.");
    Thread.sleep(1000);

    ob1.mysuspend();
    System.out.println("Stopping thread.");
    ob1.mystop();
} catch (InterruptedException e) {
    System.out.println("Main thread Interrupted");
}
// wait for thread to finish
try {
    ob1.thrd.join();
} catch (InterruptedException e) {
    System.out.println("Main thread Interrupted");
}

System.out.println("Main thread exiting.");
```

My Thread starting.

```
1 2 3 4 5 6 7 8 9 10  
11 12 13 14 15 16 17 18 19 20  
21 22 23 24 25 26 27 28 29 30  
31 32 33 34 35 36 37 38 39 40
```

Suspending thread.

Resuming thread.

```
41 42 43 44 45 46 47 48 49 50  
51 52 53 54 55 56 57 58 59 60  
61 62 63 64 65 66 67 68 69 70  
71 72 73 74 75 76 77 78 79 80
```

Suspending thread.

Resuming thread.

```
81 82 83 84 85 86 87 88 89 90  
91 92 93 94 95 96 97 98 99 100  
101 102 103 104 105 106 107 108 109 110  
111 112| 113 114 115 116 117 118 119 120
```

Stopping thread.

My Thread exiting.

Main thread exiting.

```
Controlling the main thread.  
*/  
  
class UseMain {  
    public static void main(String args[]) {  
        Thread thrd;  
        // Get the main thread.  
        thrd = Thread.currentThread();  
  
        // Display main thread's name.  
        System.out.println("Main thread is called: " +  
                           thrd.getName());  
  
        // Display main thread's priority.  
        System.out.println("Priority: " +  
                           thrd.getPriority());
```

```
System.out.println();

// Set the name and priority.
System.out.println("Setting name and priority.\n");
thrd.setName("Thread #1");
thrd.setPriority(Thread.NORM_PRIORITY+3);

System.out.println("Main thread is now called: " +
                   thrd.getName());

System.out.println("Priority is now: " +
                   thrd.getPriority());
}

}
```

# Chapter 14

## Generics

# Generics (Parameterized Types)

- Parameterized types enable you to create classes, interfaces, and methods in which the **type of data on which they operate is specified as a parameter**.
- The **advantage** of Generics is it provide **high level of reusability**
- In old code, many classes and methods used **Object** as the type of data they operated on.
- The disadvantage of the old approach is that explicit conversion with typecasting is often necessary to convert to the actual type of the data.

# Example Without Generics

```
class KVPair{  
    Object key, value;  
  
    Object getKey() { return key; }  
    Object getValue() { return value; }  
    void setKey(Object ob) { key = ob; }  
    void setValue(Object ob) { value = ob; }  
}
```

Usage: KVPair pair = new KVPair();  
 pair.setValue("Address");  
 String v = (String) pair.getValue(); // typecast

# Example with Generics

```
class KVPair<T>{
    T key, value;

    T getKey() { return key; }
    T getValue() { return value; }
    void setKey(T ob) { key = ob; }
    void setValue(T ob) { value = ob; }
}
```

Usage: KVPair<String> pair = new KVPair<String>();
pair.setValue("Address");
String v = pair.getValue(); // no typecast

# Example Explained

- The **<T>** syntax indicates that **T** is a type parameter.
- **T** is a placeholder for the actual type, which is provided when a **KVPair** object is created.
- Use of generics allows us to avoid the type casting.
- That is, generics **makes type casts automatic and implicit, and therefore add type safety.**

# Another Generics Example

```
// A simple generic class.  
// Here, T is a type parameter that  
// will be replaced by a real type  
// when an object of type Gen is created.  
class Gen<T> | ←  
T ob; // declare an object of type T
```

Declare a generic class. T is the generic type parameter.

```
// Pass the constructor a reference to
// an object of type T.
Gen(T o) {
    ob = o;
}

// Return ob.
T getob() {
    return ob;
}

// Show type of T.
void showType() {
    System.out.println("Type of T is " +
                       ob.getClass().getName());
}
```

```
// Demonstrate the generic class.  
class GenDemo {  
    public static void main(String args[]) {  
        // Create a Gen reference for Integers.  
        Gen<Integer> iOb; ← Create a reference  
                                to an object of type  
                                Gen<Integer>.  
        // Create a Gen<Integer> object and assign its  
        // reference to iOb. Notice the use of autoboxing  
        // to encapsulate the value 88 within an Integer object.  
        iOb = new Gen<Integer>(88); ← Instantiate an object of type  
                                Gen<Integer>.  
        // Show the type of data used by iOb.  
        iOb.showType();  
  
        // Get the value in iOb. Notice that  
        // no cast is needed.  
        int v = iOb.getob();  
        System.out.println("value: " + v);  
        System.out.println();  
  
        // Create a Gen object for Strings.  
        Gen<String> strOb = new Gen<String>("Generics Test"); ← Create a reference and an  
                                object of type Gen<String>.  
        // Show the type of data used by strOb.  
        strOb.showType();
```

```
// Get the value of strOb. Again, notice  
// that no cast is needed.  
String str = strOb.getOb();  
System.out.println("value: " + str);  
}  
}
```

The output produced by the program is shown here:

```
Type of T is java.lang.Integer  
value: 66
```

```
Type of T is java.lang.String  
value: Generics Test
```

# Generics Work Only with Reference Types

- ```
Gen<int> intOb = new Gen<int>(53);
```

```
// Error, can't use primitive type
```

# Generic Types Differ Based on Their Type Arguments

- `iOb = strOb; // Wrong!`
- Even though both **iOb** and **strOb** are of type **Gen<T>**, they are references to different types because their type arguments differ. This is part of the way that generics add type safety and prevent errors.

# Generics with two Type Parameters

- You can use two or more type parameters:

```
class KVPair<K, V> {  
    K key;  
    V value;  
  
    K getKey() { return key; }  
    V getValue() { return value; }  
    void setKey(K ob) { key = ob; }  
    void setValue(V ob) { value = ob; }  
}
```

**USAGE:** KVPair<String, Integer> pair =  
new KVPair<String, Integer>();

# The General Form of a Generic Class

- class *class-name*<*type-param-list*> { // ...
- Here is the full syntax for declaring a reference to a generic class and creating a generic instance:
- *class-name*<*type-arg-list*> *var-name* = new *class-name*<*type-arg-list*>(*cons-arg-list*);

# **Exercise:**

## **Demonstrate the working of Generic Stack Class**

```
class Stack<T>
{
    private int size;
    public T[] stackAr;
    private int top; // top of stack
    public Stack(int size, T [] arr) {
        this.size = size;
        stackAr =arr;
        top = -1; // initialize Stack to with -1
    }
    public void push(T value)
    {
        if(isFull())
            throw new StackFullException("Cannot push "+value+", Stack is full");
        stackAr[++top] = value;
    }
}
```

# Exercise:

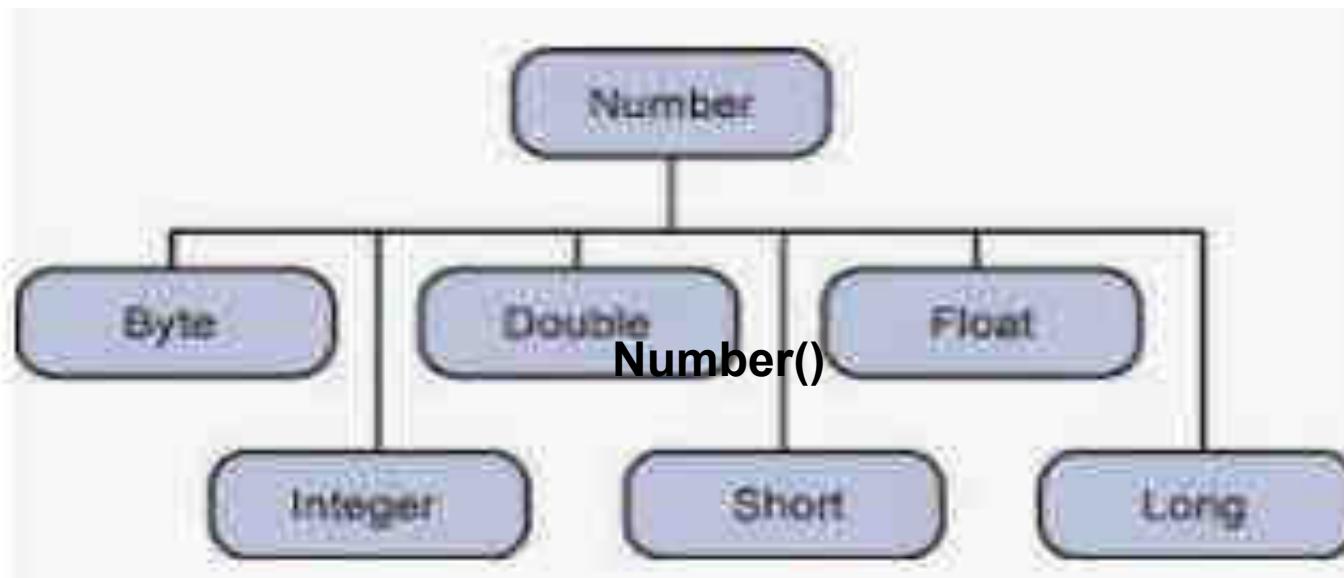
## Demonstrate the working of Generic Stack Class

```
public T pop() {  
    if(isEmpty()){  
        throw new StackEmptyException("Stack is empty");  
    }  
    return stackAr[top--]; // remove item and decrement top as well.  
}  
int retsize()  
{ return size;}  
public boolean isEmpty(){  
    return (top == -1);  
}  
public boolean isFull(){  
    return (top == size - 1);  }  
}
```

# Bounded Types

- Used to restrict the types that can be used as type arguments in a parameterized type.
- For example, a class/method that operates on numbers might only want to accept instances of Number or its subclasses.

# Class Number hierarchy



**Single constructor  
Number()**

# Number class methods

- 1. byte byteValue()
  - This method returns the value of the specified number as a byte.
- 2 abstract double doubleValue()
  - This method returns the value of the specified number as a double.
- 3 abstract float floatValue()
  - This method returns the value of the specified number as a float.
- 4 abstract int intValue()
  - This method returns the value of the specified number as a int.
- 5 abstract long longValue()
  - This method returns the value of the specified number as a long.

# doubleValue() method of Double class

```
Double a = 10.5;// Wrapper class Double  
  
// wrapping the Double value  
// in the wrapper class Double  
Double b = new Double(a);  
  
// doubleValue of the Double Object  
double res = b.doubleValue();  
  
// prdoubleing the output  
System.out.println("Double value of "  
+ b + " is : " + res);
```

## Output:

Double value of 17.47 is : 17.47

```
// NumericFns attempts (unsuccessfully) to create
// a generic class that can compute various
// numeric functions, such as the reciprocal or the
// fractional component, given any type of number.
class NumericFns<T> {
    T num;

    // Pass the constructor a reference to
    // a numeric object.
    NumericFns(T n) {
        num = n;
    }

    // Return the reciprocal.
    double reciprocal() {
        return 1 / num.doubleValue(); // Error!
    }
}
```

```
// Return the fractional component.  
double fraction() {  
    return num.doubleValue() - num.intValue(); // Error!  
}  
  
// ...  
}
```

Unfortunately, `NumericFns` will not compile as written because both methods will generate compile-time errors. First, examine the `reciprocal()` method, which attempts to return the reciprocal of `num`. To do this, it must divide 1 by the value of `num`. The value of `num` is obtained by calling `doubleValue()`, which obtains the `double` version of the numeric object stored in `num`. Because all numeric classes, such as `Integer` and `Double`, are subclasses of `Number`, and `Number` defines the `doubleValue()` method, this method is available to all numeric wrapper classes. The trouble is that the compiler has no way to know that you are intending to create `NumericFns` objects using only numeric types. Thus, when you try to compile `NumericFns`, an error is reported that indicates that the `doubleValue()` method is unknown. The same type of error occurs twice in `fraction()`, which needs to call both

# Bounded Types

- To limit the type parameter, Java provides *bounded types*. When specifying a type parameter, **you can create an upper bound that declares the superclass** from which all type arguments must be derived. This is accomplished through the use of an **extends** clause when specifying the type parameter, as shown here:
- $\langle T \text{ extends } \textit{superclass} \rangle$
- This specifies that  $T$  can be replaced only by *superclass*, or subclasses of *superclass*. Thus, *superclass* defines an inclusive, **upper limit**.

# Bounded Types Explained

- The notation **T extends Number** means that **T** must be a **Number** or a subclass of **Number**.
- Byte, Double, Float, Integer, Long, and Short are subclasses of Number
- Other option:
  - **<T, K extends T>** means that the second type parameter must be the same as the first parameter or a subclass of the first parameter.

```
// In this version of NumericFns, the type argument
// for T must be either Number, or a class derived
// from Number.
```

```
class NumericFns<T extends Number> { ←
```

```
    T num;
```

```
    // Pass the constructor a reference to
    // a numeric object.
```

```
    NumericFns(T n) {
```

```
        num = n;
```

```
}
```

```
    // Return the reciprocal.
```

```
    double reciprocal() {
```

```
        return 1 / num.doubleValue();
```

```
}
```

In this case, the type argument  
must be either **Number** or  
a subclass of **Number**.

```
// Return the fractional component.  
double fraction() {  
    return num.doubleValue() - num.intValue();  
}  
  
// ...  
}  
  
// Demonstrate NumericFns.  
class BoundsDemo {  
    public static void main(String args[]) {
```

```
NumericFns<Integer> i0b = ←  
    new NumericFns<Integer>(5);
```

Integer is OK because it  
is a subclass of Number.

```
System.out.println("Reciprocal of i0b is " +  
    i0b.reciprocal());  
  
System.out.println("Fractional component of i0b is " +  
    i0b.fraction());  
  
System.out.println();
```

```
NumericFns<Double> d0b = ← Double is also OK.  
    new NumericFns<Double>(5.25);
```

```
System.out.println("Reciprocal of d0b is " +  
    d0b.reciprocal());  
  
System.out.println("Fractional component of d0b is " +  
    d0b.fraction());
```

```
// This won't compile because String is not a
// subclass of Number.
// NumericFns<String> str0b = new NumericFns<String>("Error"); ←
}
}
```

String is illegal because it is  
not a subclass of Number.

The output is shown here:

Reciprocal of i0b is 0.2

Fractional component of i0b is 0.0

Reciprocal of d0b is 0.19047619047619047

Fractional component of d0b is 0.25

Bounded types are especially useful when you need to ensure that one type parameter is compatible with another. For example, consider the following class called Pair, which stores two objects that must be compatible with each other:

```
class Pair<T, V extends T> { ←  
    T first;  
    V second;  
  
    Pair(T a, V b) {  
        first = a;  
        second = b;  
    }  
    // ...  
}
```

Here, V must be either the same type as T, or a subclass of T.

Notice that `Pair` uses two type parameters, `T` and `V`, and that `V` extends `T`. This means that `V` will either be the same as `T` or a subclass of `T`. This ensures that the two arguments to `Pair`'s constructor will be objects of the same type or of related types. For example, the following constructions are valid:

```
// This is OK because both T and V are Integer.  
Pair<Integer, Integer> x = new Pair<Integer, Integer>(1, 2);
```

```
// This is OK because Integer is a subclass of Number.  
Pair<Number, Integer> y = new Pair<Number, Integer>(10.4, 12);
```

However, the following is invalid:

```
// This causes an error because String is not  
// a subclass of Number  
Pair<Number, String> z = new Pair<Number, String>(10.4, "12");
```

In this case, String is not a subclass of Number, which violates the bound specified by Pair.

# Using Wildcard Arguments

- For example, if one object contains the **Double** value 1.25 and the other object contains the **Float** value –1.25, then **absEqual( )** on these objects would return true.
- One way to implement **absEqual( )** is to pass it a **NumericFns** argument, and then compare the absolute value of that argument against the absolute value of the invoking object, returning true only if the values are the same.

# Using Wildcard Arguments

```
NumericFns<Double> dOb = new  
NumericFns<Double>(1.25);  
NumericFns<Float> fOb = new NumericFns<Float>(-  
1.25);  
if(dOb.absEqual(fOb))  
    System.out.println("Absolute values are the  
                      same.");  
else  
    System.out.println("Absolute values differ.");
```

# Using Wildcard Arguments

- The trouble with this attempt is that it will work only with other NumericFns **objects whose type is the same as the invoking object.**
- For example, if the invoking object is of type NumericFns, then the parameter ob must also be of type NumericFns. It can't be used to compare an object of type NumericFns, for example. Therefore, this approach does not yield a general (i.e., generic) solution

```
// This won't work!
// Determine if the absolute values of two objects are the same.
boolean absEqual(NumericFns<T> ob) {
    if(Math.abs(num.doubleValue()) ==
        Math.abs(ob.num.doubleValue())) return true;

    return false;
}
```

# Using Wildcard Arguments

To create a generic `absEqual()` method, you must use another feature of Java generics: the *wildcard argument*. The wildcard argument is specified by the `?`, and it represents an unknown type. Using a wildcard, here is one way to write the `absEqual()` method:

```
// Determine if the absolute values of two
// objects are the same.
boolean absEqual(NumericFns<?> ob) { ←———— Notice the wildcard.
    if(Math.abs(num.doubleValue()) ==
        Math.abs(ob.num.doubleValue())) return true;
    return false;
}
```

```
// Use a wildcard.
class NumericFns<T extends Number> {
    T num;

    // Pass the constructor a reference to
    // a numeric object.
    NumericFns(T n) {
        num = n;
    }

    // Return the reciprocal.
    double reciprocal() {
        return 1 / num.doubleValue();
    }

    // Return the fractional component.
    double fraction() {
        return num.doubleValue() - num.intValue();
    }
}
```

```
// Determine if the absolute values of two
// objects are the same.
boolean absEqual(NumericFns<?> ob) {
    if(Math.abs(num.doubleValue()) ==
        Math.abs(ob.num.doubleValue())) return true;

    return false;
}

// ...
}

// Demonstrate a wildcard.
class WildcardDemo {
    public static void main(String args[]) {
        NumericFns<Integer> iOb =
            new NumericFns<Integer>(6);
```

```
NumericFns<Double> d0b =
    new NumericFns<Double>(-6.0);

NumericFns<Long> l0b =
    new NumericFns<Long>(5L);           In this call, the wildcard
   type matches Double.

System.out.println("Testing i0b and d0b.");
if(i0b.absEqual(d0b))               ←
    System.out.println("Absolute values are equal.");
else
    System.out.println("Absolute values differ.");

System.out.println();                In this call, the wildcard
   matches Long.

System.out.println("Testing i0b and l0b.");
if(i0b.absEqual(l0b))               ←
    System.out.println("Absolute values are equal.");
else
    System.out.println("Absolute values differ."); }
```

The output is shown here:

Testing i0b and d0b.

Absolute values are equal.

Testing i0b and l0b.

Absolute values differ.

In the program, notice these two calls to `absEqual()`:

```
if(i0b.absEqual(d0b))
```

```
if(i0b.absEqual(l0b))
```

In the first call, `i0b` is an object of type `NumericFns<Integer>` and `d0b` is an object of type `NumericFns<Double>`. However, through the use of a wildcard, it is possible for `i0b` to pass `d0b` in the call to `absEqual()`. The same applies to the second call, in which an object of type `NumericFns<Long>` is passed.

One last point: It is important to understand that the wildcard does not affect what type of `NumericFns` objects can be created. This is governed by the `extends` clause in the `NumericFns` declaration. The wildcard simply matches any *valid* `NumericFns` object.

# Bounded Wildcards

Wildcard arguments can be bounded in much the same way that a type parameter can be bounded. A bounded wildcard is especially important when you are creating a method that is designed to operate only on objects that are subclasses of a specific superclass. To understand why, let's work through a simple example. Consider the following set of classes:

```
class A {  
    // ...  
}
```

```
class B extends A {  
    // ...  
}
```

```
class C extends A {  
    // ...  
}
```

```
// Note that D does NOT extend A.  
class D {  
    // ...  
}
```

Here, class A is extended by classes B and C, but not by D.

```
class UseBoundedWildcard {  
    // Here, the ? will match A or any class type  
    // that extends A.  
    static void test(Gen<? extends A> o) { ← Use a bounded wildcard.  
        // ...  
    }  
}
```

```
public static void main(String args[]) {  
    A a = new A();  
    B b = new B();  
    C c = new C();  
    D d = new D();  
  
    Gen<A> w = new Gen<A>(a);  
    Gen<B> w2 = new Gen<B>(b);  
    Gen<C> w3 = new Gen<C>(c);  
    Gen<D> w4 = new Gen<D>(d);
```

```
// These calls to test() are OK.  
test(w);  
test(w2);  
test(w3);  
  
// Can't call test() with w4 because  
// it is not an object of a class that  
// inherits A.  
// test(w4); // Error! ← This is illegal because w4 is not a subclass of A.  
}  
}
```

In general, to establish an upper bound for a wildcard, use the following type of wildcard expression:

<? extends superclass>

# Generic Methods

- You can write a single generic method declaration that can be called with arguments of different types.
- Based on the types of the arguments passed to the generic method, the compiler handles each method call appropriately.
- Following are the rules to define Generic Methods –
  - All generic method declarations have a type parameter section delimited by angle brackets (< and >) that precedes the method's return type ( < E > in the next example).
  - Each type parameter section contains one or more type parameters separated by commas. A type parameter, also known as a type variable, is an identifier that specifies a generic type name.
  - The type parameters can be used to declare the return type and act as placeholders for the types of the arguments passed to the generic method, which are known as actual type arguments.

Following example illustrates how we can print an array of different type using a single Generic method –

```
public class GenericMethodTest {  
    // generic method printArray  
    public static < E > void printArray( E[ ] inputArray ) {  
        // Display array elements  
        for(E element : inputArray) {  
            System.out.print(" "+ element);  
        }  
        System.out.println();  
    }  
}
```

```
public static void main(String args[]) {  
    // Create arrays of Integer, Double and Character  
    Integer[] intArray = { 1, 2, 3, 4, 5 };  
    Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4 };  
    Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };
```

```
    System.out.println("Array integerArray contains:");  
    printArray(intArray); // pass an Integer array
```

```
    System.out.println("\nArray doubleArray contains:");  
    printArray(doubleArray); // pass a Double array
```

```
    System.out.println("\nArray characterArray contains:");  
    printArray(charArray); // pass a Character array
```

```
}
```

# Generic Constructor

- A constructor can be generic, even if its class is not. For example, in the following program, the class Summation is not generic, but its constructor is.
- The Summation class computes and encapsulates the summation of the numeric value passed to its constructor. Recall that the summation of N is the sum of all the whole numbers between 0 and N. Because Summation( ) specifies a type parameter that is bounded by Number, a Summation object can be constructed using any numeric type, including Integer, Float, or Double.
- No matter what numeric type is used, its value is converted to Integer by calling intValue( ), and the summation is computed. Therefore, it is not necessary for the class Summation to be generic; only a generic constructor is needed.

# Generic Constructor

```
// Use a generic constructor.
class Summation {
    private int sum;

    <T extends Number> Summation(T arg) { ←———— A generic constructor
        sum = 0;

        for(int i=0; i <= arg.intValue(); i++)
            sum += i;
    }

    int getSum() {
        return sum;
    }
}

class GenConsDemo {
    public static void main(String args[]) {
        Summation ob = new Summation(4.0);

        System.out.println("Summation of 4.0 is " +
                           ob.getSum());
    }
}
```

# Inheritance and Generics

- Generic classes can be extended by generic subclasses.
- The subclasses must pass type arguments up to the superclass that needs them.
- Example:

```
class Sub<T, V> extends Sup<T> {  
    . . .  
}
```

# Interfaces and Generics

- Interfaces can also be generic.
- Any class that implements a generic interface must also be generic unless it implements a specific type of the interface.
- Examples:

```
class C<T> implements I<T> {  
... }
```

```
class D implements I<String> {  
... }
```

# Interfaces and Generics

- In an example program given in next slide, the standard **interface Comparable** is used to ensure that how exactly elements of two arrays could be compared.
- Generic interfaces are specified just like generic classes. Here is an example. It creates an interface called Containment, which can be implemented by classes that store one or more values. It declares a method called contains( ) that determines if a specified value is contained by the invoking object.

# Interfaces and Generics

```
// A generic interface example.  
  
// A generic containment interface.  
// This interface implies that an implementing  
// class contains one or more values.  
interface Containment<T> { ← A generic interface  
    // The contains() method tests if a  
    // specific item is contained within  
    // an object that implements Containment.  
    boolean contains(T o);  
}  
  
// Implement Containment using an array to  
// hold the values.  
class MyClass<T> implements Containment<T> { ← Any class that implements  
    T[] arrayRef;  
  
    MyClass(T[] o) {  
        arrayRef = o;  
    }  
  
    // Implement contains()  
    public boolean contains(T o) {  
        for(T x : arrayRef)  
            if(x.equals(o)) return true;  
        return false;  
    }
```

# Interfaces and Generics

```
class GenIFDemo {  
    public static void main(String args[]) {  
        Integer x[] = { 1, 2, 3 };  
  
        MyClass<Integer> ob = new MyClass<Integer>(x);  
        if(ob.contains(2))  
            System.out.println("2 is in ob");  
        else  
            System.out.println("2 is NOT in ob");  
  
        if(ob.contains(5))  
            System.out.println("5 is in ob");  
        else  
            System.out.println("5 is NOT in ob");  
  
        // The following is illegal because ob  
        // is an Integer Containment and 9.25 is  
        // a Double value.  
        // if(ob.contains(9.25)) // Illegal!  
        //     System.out.println("9.25 is in ob");  
    }  
}
```

The output is shown here:

# Interfaces and Generics

In general, a generic interface is declared in the same way as a generic class. In this case, the type parameter **T** specifies the type of objects that are contained.

Next, **Containment** is implemented by **MyClass**. Notice the declaration of **MyClass**, shown here:

```
class MyClass<T> implements Containment<T> {
```

In general, if a class implements a generic interface, then that class must also be generic, at least to the extent that it takes a type parameter that is passed to the interface. For example, the following attempt to declare **MyClass** is in error:

```
class MyClass implements Containment<T> { // Wrong!
```

This declaration is wrong because **MyClass** does not declare a type parameter, which means that there is no way to pass one to **Containment**. In this case, the identifier **T** is simply unknown and the compiler reports an error. Of course, if a class implements a *specific type* of generic interface, such as shown here:

```
class MyClass implements Containment<Double> { // OK
```

then the implementing class does not need to be generic.

# Interfaces and Generics

As you might expect, the type parameter(s) specified by a generic interface can be bounded. This lets you limit the type of data for which the interface can be implemented. For example, if you wanted to limit **Containment** to numeric types, then you could declare it like this:

```
interface Containment<T extends Number> {
```

Now, any implementing class must pass to **Containment** a type argument also having the same bound. For example, now **MyClass** must be declared as shown here:

```
class MyClass<T extends Number> implements Containment<T> {
```

Pay special attention to the way the type parameter **T** is declared by **MyClass** and then passed to **Containment**. Because **Containment** now requires a type that extends **Number**, the implementing class (**MyClass** in this case) must specify the same bound. Furthermore, once this bound has been established, there is no need to specify it again in the **implements** clause. In fact, it would be wrong to do so. For example, this declaration is incorrect and won't compile:

```
// This is wrong!
class MyClass<T extends Number>
    implements Containment<T extends Number> { // Wrong!
```

# Generic Restrictions

(i) **Static variables and methods cannot use the type parameters** declared by their generic class (but you can have static generic methods).

- **Example:**

```
class C<T> {  
    static T x; // illegal  
    static T getX() { return x; } // illegal  
    static <W> void setX(W w) { // legal  
        ...  
    }  
}
```

# More Generic Restrictions

ii) You can't create an instance of a type parameter.

- Example:

```
class C<T>
{
    T x;
    C()
    { x= new T(); // illegal
    }
}
```

(iii) You can't create an array whose element type is the type parameter nor an array of type-specific generic references.

- Example:

```
class C<T> {
    T[] x ;
    x = new T[10]; // illegal
    C<String>[] data= new C<String>[10]; // illegal
    C<?>[] data = new C<?>[10]; // legal
```

# More Generic Restrictions

## (iv) Generic Exception Restriction:

A generic class cannot extend Throwable. This means that you cannot create generic exception classes.

## Exercise: Demonstrate the working of Generic Singly Linked List Class

```
class Node {  
    int data; // data in Node.  
    Node next; // points to next Node in list.  
    //Constructor  
    Node(int data)  
    {  
        this.data = data;  
    }  
  
    // Display Node's data  
    void displayNode() {  
        System.out.print( data + " ");  
    }  
}
```

## **Exercise: Demonstrate the working of Generic Singly Linked List Class**

// generic Single LinkedList class (Generic implementation)

```
class LinkedList {
```

```
    private Node first; // ref to first link on list
```

```
    LinkedList(){
```

```
        first = null;
```

```
}
```

```
    void insertFirst(int data) {
```

```
        Node newNode = new Node(data); //Creation of New Node.
```

```
        newNode.next = first; //newLink ---> old first
```

```
        first = newNode; //first ---> newNode
```

```
}
```

```
    Node deleteFirst()
```

```
{
```

```
    if(first==null){ //means LinkedList in empty, throw exception.
```

```
        System.out.println("LinkedList is empty.");
```

```
}
```

Node tempNode = first; // save reference to first Node in tempNode- so that we could return saved reference.

```
    first = first.next; // delete first Node (make first point to second node)
```

```
    return tempNode; // return tempNode (i.e. deleted Node)
```

```
}
```

## **Exercise: Demonstrate the working of Generic Singly Linked List Class**

// Display generic Single LinkedList

```
void displayLinkedList() {  
  
    System.out.print("Displaying LinkedList [first--->last]: ");  
  
    Node tempDisplay = first; // start at the beginning of linkedList  
  
    while (tempDisplay != null){ // Executes until we don't find end of list.  
  
        tempDisplay.displayNode();  
  
        tempDisplay = tempDisplay.next; // move to next Node  
  
    }  
  
    System.out.println();  
  
}
```

### Exercise: Demonstrate the working of Generic Singly Linked List Class

// Main class - To test generic Single LinkedList .

```
public class SinglyLL
```

```
{
```

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

```
    {
```

```
        LinkedList linkedList = new LinkedList(); // creation of Linked List
```

```
        linkedList.insertFirst(11);
```

```
        linkedList.insertFirst(21);
```

```
        linkedList.insertFirst(59);
```

```
        linkedList.insertFirst(14);
```

```
        linkedList.insertFirst(39);
```

```
        linkedList.displayLinkedList(); // display LinkedList
```

```
        System.out.print("Deleted Nodes: ");
```

```
        Node deletedNode = linkedList.deleteFirst(); //delete Node
```

```
        deletedNode.displayNode(); //display deleted Node.
```

```
        deletedNode = linkedList.deleteFirst(); //delete Node.
```

```
        deletedNode.displayNode(); //display deleted Node.
```

```
        System.out.println();// to format output
```

```
        linkedList.displayLinkedList(); //Again display LinkedList
```

```
}
```

```
}
```

# Self Test

1. Can a primitive type be used as a type argument in Generics?
2. Show how to declare a class called FlightSched that takes two generic parameters.
3. Beginning with your answer to question 2, change FlightSched's second type parameter so that it must extend Thread.
4. Now, change FlightSched so that its second type parameter must be a subclass of its first type parameter.
5. As it relates to generics, what is the ? and what does it do?
6. Can the wildcard argument be bounded?
7. A generic method called MyGen( ) has one type parameter. Furthermore, MyGen( ) has one parameter whose type is that of the type parameter. It also returns an object of that type parameter. Show how to declare MyGen( ).

# Chapter 34

## Introduction to JavaFX GUI Programming

Creation of GUI

Event driven programming

# AWT → SWINGS → JavaFX

- AWT
  - Original Framework. Has Several Limitations
- SWINGS
  - Successful for two decades. Suitable for Enterprise Applications
- JavaFX
  - Mobile Apps
  - Java's next generation client platform and GUI framework.
  - provides a powerful, streamlined, flexible framework that simplifies the creation of modern, visually exciting GUIs

# JavaFX Basic Concepts

- JavaFX facilitates a more visually dynamic approach to GUIs

## JavaFX Packages

- The JavaFX elements are contained in packages that begin with the javafx prefix.
- Examples: **javafx.application**, **javafx.stage**, **javafx.scene**, and **javafx.scene.layout**.
- Beginning with JDK 9, the JavaFX packages are organized into modules, such as **javafx.base**, **javafx.graphics**, and **javafx.controls**.

# JavaFX Basic Concepts..

## The Stage and Scene Classes

To create a JavaFX application, you will, at minimum, add at least one **Scene** object to a **Stage**.

A stage is a container for scenes and a scene is a container for the items that comprise the scene. These elements are encapsulated in the JavaFX API by the **Stage** and **Scene** classes.

**Stage** is a top-level container. All JavaFX applications automatically have access to one **Stage**, called the *primary stage*. The primary stage is supplied by the run-time system when a JavaFX application is started. Although you can create other stages, for many applications, the primary stage will be the only one required.

As mentioned, **Scene** is a container for the items that comprise the scene. These can consist of controls, such as push buttons and check boxes, text, and graphics. To create a scene, you will add those elements to an instance of **Scene**.

# JavaFX Basic Concepts..

## Nodes and Scene Graphs

The individual elements of a scene are called *nodes*. For example, a push button control is a node.

However, nodes can also consist of groups of nodes. Furthermore, a node can have a child node. In this case, a node with a child is called a *parent node* or *branch node*. Nodes without children are terminal nodes and are called leaves.

The collection of all nodes in a scene creates what is referred to as a *scene graph*, which comprises a *tree*.

There is one special type of node in the scene graph, called the *root node*. This is the top-level node and is the only node in the scene graph that does not have a parent. Thus, with the exception of the root node, all other nodes have parents, and all nodes either directly or indirectly descend from the root node.

The base class for all nodes is **Node**. There are several other classes that are, either directly or indirectly, subclasses of **Node**. These include **Parent**, **Group**, **Region**, and **Control**, to name a few.

# JavaFX Basic Concepts..

## Layouts

- JavaFX provides several layout panes that manage the process of placing elements in a scene.
- For example, the **FlowPane** class provides a flow layout and the **GridPane** class supports a row/column grid-based layout.
- Several other layouts, such as **BorderPane** are available. The layout panes are packaged in **javafx.scene.layout**.

# JavaFX Basic Concepts..

## The Application Class and the Life-Cycle Methods

A JavaFX application must be a subclass of the **Application** class, which is packaged in **javafx.application**. Thus, your application class will extend **Application**.

The **Application** class defines three life-cycle methods that your application can override. These are called **init( )**, **start( )**, and **stop( )**, and are shown here, in the order in which they are called:

**void init( )**  
**abstract void start(Stage primaryStage)**  
**void stop( )**

The **init( )** method is called when the application begins execution. It is used to perform various initializations. As will be explained, however, it *cannot* be used to create a stage or build a scene. If no initializations are required, this method need not be overridden because an empty, default version is provided.

# JavaFX Basic Concepts..

## The Application Class and the Life-Cycle Methods..

The **start( )** method is called after **init( )**. This is where your application begins and it *can* be used to construct and set the scene. Notice that it is passed a reference to a **Stage** object. This is the stage provided by the run-time system and is the primary stage. (You can also create other stages, but you won't need to for simple applications.) Notice that this method is abstract. Thus, it must be overridden by your application.

When your application is terminated, the **stop( )** method is called. It is here that you can handle any cleanup or shutdown chores. In cases in which no such actions are needed, an empty, default version is provided.

One other point: For a modular JavaFX application, the package that contains your main application class (that is, your subclass of **Application**) must be exported by its module so that it can be found by the **javafx.graphics** module.

# JavaFX Basic Concepts..

## Launching a JavaFX Application

To start a free-standing JavaFX application, you must call the **launch( )** method defined by **Application**. It has two forms. Here is the one used in this chapter:

```
public static void launch(String ... args)
```

Here, *args* is a possibly empty list of strings that typically specify command-line arguments. When called, **launch( )** causes the application to be constructed, followed by calls to **init( )** and **start( )**. The **launch( )** method will not return until after the application has terminated. This version of **launch( )** starts the subclass of **Application** from which **launch( )** is called. The second form of **launch( )** lets you specify a class other than the enclosing class to start.

Before moving on, it is necessary to make an important point: JavaFX applications that have been packaged by using the **javafxpackager** tool (or its equivalent in an IDE) do not need to include a call to **launch( )**. However, its inclusion often simplifies the test/debug cycle, and it lets the program be used without the creation of a JAR file.

# JavaFX Application Skeleton

All JavaFX applications share the same basic skeleton. Therefore, before looking at any more JavaFX features, it will be useful to see what that skeleton looks like.

In addition to showing the general form of a JavaFX application, the skeleton also illustrates how to launch the application and demonstrates when the life-cycle methods are called.

A message noting when each life-cycle method is called is displayed on the console. The complete skeleton is shown here:

```
// A JavaFX application skeleton.

import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;

public class JavaFXSkel extends Application {

    public static void main(String[] args) {
        System.out.println("Launching JavaFX application.");

        // Start the JavaFX application by calling launch().
        launch(args);
    }
}
```

```
// Override the init() method.
public void init() {
    System.out.println("Inside the init() method.");
}

// Override the start() method.
public void start(Stage myStage) {

    System.out.println("Inside the start() method.");

    // Give the stage a title.
    myStage.setTitle("JavaFX Skeleton.");

    // Create a root node. In this case, a flow layout pan
    // is used, but several alternatives exist.
    FlowPane rootNode = new FlowPane();

    // Create a scene.
    Scene myScene = new Scene(rootNode, 300, 200);

    // Set the scene on the stage.
    myStage.setScene(myScene);

    // Show the stage and its scene.
    myStage.show();
}

// Override the stop() method.
public void stop() {
    System.out.println("Inside the stop() method.");
}
```



It also produces the following output on the console:

Launching JavaFX application.

Inside the init() method.

Inside the start() method.

When you close the window, this message is displayed on the console:

Inside the stop() method.

# The Application Thread

**init( )** method cannot be used to construct a stage or scene. You also cannot create these items inside the application's constructor. The reason is that a stage or scene must be constructed on the *application thread*.

However, the application's constructor and the **init( )** method are called on the main thread, also called the *launcher thread*. Thus, they can't be used to construct a stage or scene.

Instead, you must use the **start( )** method, as the skeleton demonstrates, to create the initial GUI because **start( )** is called on the application thread.

Furthermore, any changes to the GUI currently displayed must be made from the application thread. Fortunately, in JavaFX, events are sent to your program on the application thread. Therefore, event handlers can be used to interact with the GUI.

The **stop( )** method is also called on the application thread.

# A Simple JavaFX Control: Label

The primary ingredient in most user interfaces is the control because a control enables the user to interact with the application. As you would expect, JavaFX supplies a rich assortment of controls.

The simplest control is the label because it just displays a message, which, in this example, is text. Although quite easy to use, the label is a good way to introduce the techniques needed to begin building a scene graph.

The JavaFX label is an instance of the **Label** class, which is packaged in **javafx.scene.control**. **Label** inherits **Labeled** and **Control**, among other classes.

The **Labeled** class defines several features that are common to all labeled elements (that is, those that can contain text), and **Control** defines features related to all controls.

# A Simple JavaFX Control: Label

**Label** defines three constructors. The one we will use here is

`Label(String str)`

Here, *str* is the string that is displayed.

Once you have created a label (or any other control), it must be added to the scene's content, which means adding it to the scene graph. To do this, you will first call `getChildren()` on the root node of the scene graph. It returns a list of the child nodes in the form of an **ObservableList<Node>**.

**ObservableList** is packaged in **javafx.collections**, and it inherits **java.util.List**, which means that it supports all of the features available to a list as defined by the Collections Framework. Using the returned list of child nodes, you can add the label to the list by calling `add()`, passing in a reference to the label.

```
// Demonstrate a JavaFX label.

import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;

public class JavaFXLabelDemo extends Application {

    public static void main(String[] args) {

        // Start the JavaFX application by calling launch().
        launch(args);
    }

    // Override the start() method.
    public void start(Stage myStage) {

        // Give the stage a title.
        myStage.setTitle("Demonstrate a JavaFX label.");
    }
}
```

```
// Use a FlowPane for the root node.  
FlowPane rootNode = new FlowPane();  
  
// Create a scene.  
Scene myScene = new Scene(rootNode, 300, 200);  
  
// Set the scene on the stage.  
myStage.setScene(myScene);  
  
// Create a label.  
Label myLabel = new Label("This is a JavaFX label");  
  
// Add the label to the scene graph.  
rootNode.getChildren().add(myLabel);  
  
// Show the stage and its scene.  
myStage.show();  
}  
}
```

# Output:



In the program, pay special attention to this line:

```
rootNode.getChildren().add(myLabel);
```

It adds the label to the list of children for which **rootNode** is the parent. Although this line could be separated into its individual pieces if necessary, you will often see it as shown here.

Before moving on, it is useful to point out that **ObservableList** provides a method called **addAll( )** that can be used to add two or more children to the scene graph in a single call. (You will see an example of this shortly.)

To remove a control from the scene graph, call **remove( )** on the **ObservableList**. For example,

```
rootNode.getChildren().remove(myLabel);
```

removes **myLabel** from the scene.

# TextField

- JavaFX includes several text-based controls
- TextField allows one line of text to be entered
- useful for obtaining names, ID strings, addresses etc.
- TextField inherits TextInputControl which defines much of its functionality
- TextField defines two constructors
  1. The first is the default constructor, which creates an empty text field that has the default size.
  2. The second lets you specify the initial contents of the field.
- Although the default size of a TextField is sometimes adequate, often you will want to specify its size, use the method below.

```
final void setPrefColumnCount(int columns)
```

The columns value is used by TextField to determine its size.

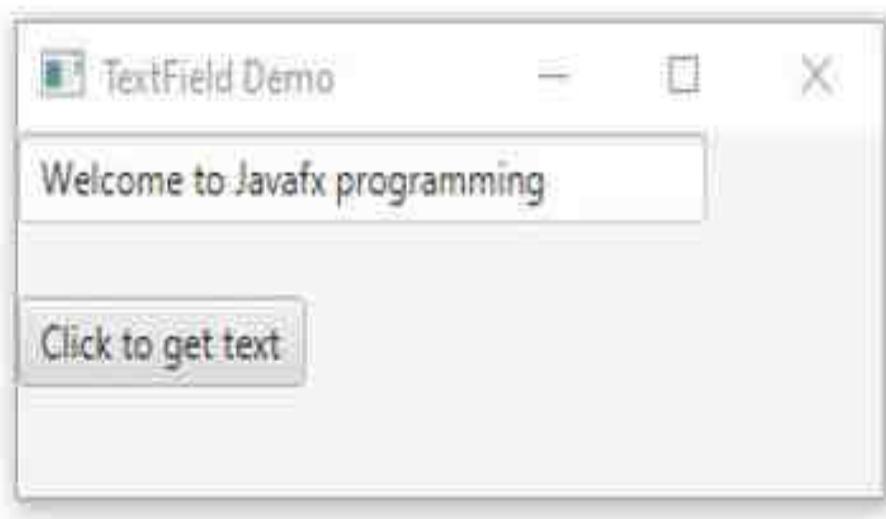
# TextField

- set the text in a text field by calling `setText( )`.
- obtain the current text by calling `getText( )`
- To set a prompting message inside the text field when the user attempts to use a blank field use method below
  - `final void setPromptText(String str)`
- When the user presses enter while inside a `TextField`, an action event is generated

# TextField

- `TextField tf;`
- `// Create a text field.`
- `tf = new TextField();`
- `// Set the prompt.`
- `tf.setPromptText("Enter a name.");`
- `// Set preferred column count.`
- `tf.setPrefColumnCount(15);`

# TextField Demo program



Program to display the contents of textfield at the terminal on Button click

```
public class TextFieldDemo extends Application
{
    public void start(Stage primaryStage)
    {
        primaryStage.setTitle("TextField Demo");
        FlowPane root=new FlowPane();
        Scene scene = new Scene(root, 300, 100);

        TextField tf = new TextField();
        tf.setPrefColumnCount(20);
        root.setHgap(10);
        root.setVgap(20);
        Button button = new Button("Click to get text");
        root.getChildren().addAll(tf,button);
```

```
button.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent ae) {
        System.out.println(tf.getText());
    });
}

primaryStage.setScene(scene);
primaryStage.show();

}

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

# Using Button and Events

Although the program in the preceding section presents a simple example of using a JavaFX control and constructing a scene graph, it does not show how to handle **events**.

As you know, most GUI controls generate events that are handled by your program. For example, buttons, check boxes, and lists all generate events when they are used.

One commonly used control is the button. This makes button events one of the most frequently handled. Therefore, a button is a good way to demonstrate the fundamentals of event handling in JavaFX. For this reason, the fundamentals of event handling and the button are introduced together.

# Using Buttons and Events..

## Event Basics

The base class for JavaFX events is the **Event** class, which is packaged in **javafx.event**. **Event** inherits **java.util.EventObject**, which means that JavaFX events share the same basic functionality as other Java events.

Several subclasses of **Event** are defined. The one that we will use here is **ActionEvent**. It handles action events generated by a button.

In general, JavaFX uses what is, in essence, the delegation event model approach to event handling. To handle an event, you must first register the handler that acts as a listener for the event. When the event occurs, the listener is called. It must then respond to the event and return.

Events are handled by implementing the **EventHandler** interface, which is also in **javafx.event**. It is a generic interface with the following form:

```
interface EventHandler<T extends Event>
```

Here, **T** specifies the type of event that the handler will handle. It defines one method, called **handle( )**, which receives the event object as a parameter. It is shown

# Using Buttons and Events..

## Event Basics...

It is shown here:

```
void handle(T eventObj)
```

Here, *eventObj* is the event that was generated. Typically, event handlers are implemented through anonymous inner classes or lambda expressions, but you can use stand-alone classes for this purpose if it is more appropriate to your application

(for example, if one event handler will handle events from more than one source).

Although not required by the examples in this chapter, it is sometimes useful to know the source of an event. This is especially true if you are using one handler to handle events from different sources. You can obtain the source of the event by calling **getSource( )**, which is inherited from **java.util.EventObject**. It is shown here:

```
Object getSource()
```

# Using Buttons and Events..

## Event Basics...

Other methods in **Event** let you obtain the event type, determine if the event has been consumed, consume an event, fire an event, and obtain the target of the event. When an event is consumed, it stops the event from being passed to a parent handler.

One last point: In JavaFX, events are processed via an *event dispatch chain*.

When an event is generated, it is passed to the root node of the chain. The event is then passed down the chain to the target of the event. After the target node processes the event, the event is passed back up the chain, thus allowing parent nodes a chance to process the event, if necessary. This is called *event bubbling*. It is possible for a node in the chain to consume an event, which prevents it from being further processed.

# Using Buttons and Events..

## Introducing Button Control

In JavaFX, the push button control is provided by the **Button** class, which is in

**javafx.scene.control**. **Button** inherits a fairly long list of base classes that include **ButtonBase**, **Labeled**, **Region**, **Control**, **Parent**, and **Node**. If you examine the API documentation for **Button**, you will see that much of its functionality comes from its base classes. Furthermore, it supports a wide array of options.

However, here we will use its default form. Buttons can contain text, graphics, or both. In this chapter, we will use text-based buttons. An example of a graphics-based button is shown in the next chapter.

**Button** defines three constructors. The one we will use is shown here:

**Button(String str)**

In this case, *str* is the message that is displayed in the button.

# Using Buttons and Events..

## Introducing Button Control..

When a button is pressed, an **ActionEvent** is generated.

**ActionEvent** is packaged in **javafx.event**. You can register a listener for this event by using **setOnAction( )**, which has this general form:

```
final void setOnAction(EventHandler<ActionEvent> handler)
```

Here, *handler* is the handler being registered. As mentioned, often you will use an anonymous inner class or lambda expression for the handler. The **setOnAction( )** method sets the property **onAction**, which stores a reference to the handler.

As with all other Java event handling, your handler must respond to the event as fast as possible and then return. If your handler consumes too much time, it will noticeably slow down the application. For lengthy operations, you must use a separate thread of execution.

```
import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.event.*;
import javafx.geometry.*;

public class JavaFXEventDemo extends Application {

    Label response;

    public static void main(String[] args) {

        // Start the JavaFX application by calling launch().
        launch(args);
    }

    // Override the start() method.
    public void start(Stage myStage) {

        // Give the stage a title.
        myStage.setTitle("Demonstrate JavaFX Buttons and Events.");
    }
}
```

```
// Use a FlowPane for the root node. In this case,  
// vertical and horizontal gaps of 10.  
FlowPane rootNode = new FlowPane(10, 10);  
  
// Center the controls in the scene.  
rootNode.setAlignment(Pos.CENTER);  
  
// Create a scene.  
Scene myScene = new Scene(rootNode, 300, 100);  
  
// Set the scene on the stage.  
myStage.setScene(myScene);  
  
// Create a label.  
response = new Label("Push a Button");  
  
// Create two push buttons.  
Button btnAlpha = new Button("Alpha");  
Button btnBeta = new Button("Beta");
```

```
// Handle the action events for the Alpha button.  
btnAlpha.setOnAction(new EventHandler<ActionEvent>() {  
    public void handle(ActionEvent ae) {  
        response.setText("Alpha was pressed.");  
    }  
});  
  
// Handle the action events for the Beta button.  
btnBeta.setOnAction(new EventHandler<ActionEvent>() {  
    public void handle(ActionEvent ae) {  
        response.setText("Beta was pressed.");  
    }  
});  
  
// Add the label and buttons to the scene graph.  
rootNode.getChildren().addAll(btnAlpha, btnBeta, response);  
  
// Show the stage and its scene.  
myStage.show();  
}  
}
```

- Sample Output



Let's examine a few key portions of this program. First, notice how buttons are created by these two lines:

```
Button btnAlpha = new Button("Alpha");  
Button btnBeta = new Button("Beta");
```

This creates two text-based buttons. The first displays the string Alpha; the second displays Beta.

Next, an action event handler is set for each of these buttons. The sequence for the Alpha button is shown here:

```
// Handle the action events for the Alpha button.  
btnAlpha.setOnAction(new EventHandler<ActionEvent>() {  
    public void handle(ActionEvent ae) {  
        response.setText("Alpha was pressed.");  
    }  
});
```

As explained, buttons respond to events of type **ActionEvent**. To register a handler for these events, the **setOnAction( )** method is called on the button. It uses an anonymous inner class to implement the **EventHandler** interface. (Recall that **EventHandler** defines only the **handle( )** method.) Inside **handle( )**, the text in the **response** label is set to reflect the fact that the Alpha button was pressed. Notice that this is done by calling the **setText( )** method on the label. Events are handled by the Beta button in the same way.

After the event handlers have been set, the **response** label and the buttons **btnAlpha** and **btnBeta** are added to the scene graph by using a call to **addAll( )**:

```
rootNode.getChildren().addAll(btnAlpha, btnBeta, response);
```

The **addAll( )** method adds a list of nodes to the invoking parent node. Of course, these nodes could have been added by three separate calls to **add( )**, but the **addAll( )** method is more convenient to use in this situation.

There are two other things of interest in this program that relate to the way the controls are displayed in the window. First, when the root node is created, this statement is used:

```
FlowPane rootNode = new FlowPane(10, 10);
```

Here, the **FlowPane** constructor is passed two values. These specify the horizontal and vertical gap that will be left around elements in the scene. If these gaps are not specified, then two elements (such as two buttons) would be positioned in such a way that no space is between them. Thus, the controls would run together, creating a very unappealing user interface. Specifying gaps prevents this.

The second point of interest is the following line, which sets the alignment of the elements in the **FlowPane**:

```
rootNode.setAlignment(Pos.CENTER);
```

Here, the alignment of the elements is centered. This is done by calling **setAlignment()** on the **FlowPane**. The value **Pos.CENTER** specifies that both a vertical and horizontal center will be used. Other alignments are possible. **Pos** is an enumeration that specifies alignment constants. It is packaged in **javafx.geometry**.

Before moving on, one more point needs to be made. The preceding program used anonymous inner classes to handle button events. However, because the **EventHandler** interface defines only one abstract method, **handle()**, a lambda expression could have passed to **setOnAction()**, instead. In this case, the parameter type of **setOnAction()** would supply the target context for the lambda expression. For example, here is the handler for the Alpha button, rewritten to use a lambda:

```
btnAlpha.setOnAction( ae) ->
    response.setText("Alpha was pressed.")
);
```

# Drawing Directly on a Canvas

One place that JavaFX's approach to rendering is especially helpful is when displaying graphics objects, such as lines, circles, and rectangles. JavaFX's graphics methods are found in the **GraphicsContext** class, which is part of **javafx.scene.canvas**. These methods can be used to draw directly on the surface of a canvas, which is encapsulated by the **Canvas** class in **javafx.scene.canvas**. When you draw something, such as a line, on a canvas, JavaFX automatically renders it whenever it needs to be redisplayed.

Before you can draw on a canvas, you must perform two steps. First, you must create a **Canvas** instance. Second, you must obtain a **GraphicsContext** object that refers to that canvas. You can then use the **GraphicsContext** to draw output on the canvas.

The **Canvas** class is derived from **Node**; thus it can be used as a node in a scene graph. **Canvas** defines two constructors. One is the default constructor, and the other is the one shown here:

`Canvas(double width, double height)`

Here, *width* and *height* specify the dimensions of the canvas.

To obtain a **GraphicsContext** that refers to a canvas, call **getGraphicsContext2D()**. Here is its general form:

`GraphicsContext getGraphicsContext2D()`

The graphics context for the canvas is returned.

**GraphicsContext** defines a large number of methods that draw shapes, text, and images, and support effects and transforms.

You can draw a line using **strokeLine( )**, shown here:

```
void strokeLine(double startX, double startY, double endX, double endY)
```

It draws a line from *startX*,*startY* to *endX*,*endY*, using the current stroke, which can be a solid color or some more complex style.

To draw a rectangle, use either **strokeRect( )** or **fillRect( )**, shown here:

```
void strokeRect(double topX, double topY, double width, double height)
```

```
void fillRect(double topX, double topY, double width, double height)
```

The upper-left corner of the rectangle is at *topX*,*topY*. The *width* and *height* parameters specify its width and height. The **strokeRect( )** method draws the outline of a rectangle using the current stroke, and **fillRect( )** fills the rectangle with the current fill. The current fill can be as simple as a solid color or something more complex.

To draw an ellipse, use either **strokeOval( )** or **fillOval( )**, shown next:

```
void strokeOval(double topX, double topY, double width, double height)
void fillOval(double topX, double topY, double width, double height)
```

The upper-left corner of the rectangle that bounds the ellipse is at *topX*,*topY*. The *width* and *height* parameters specify its width and height. The **strokeOval( )** method draws the outline of an ellipse using the current stroke, and **fillOval( )** fills the oval with the current fill. To draw a circle, pass the same value for *width* and *height*.

You can draw text on a canvas by using the **strokeText( )** and **fillText( )** methods. We will use this version of **fillText( )**:

```
void fillText(String str, double topX, double topY)
```

It displays *str* starting at the location specified by *topX*,*topY*, filling the text with the current fill.

You can set the font and font size of the text being displayed by using `setFont()`. You can obtain the font used by the canvas by calling `getFont()`. By default, the system font is used. You can create a new font by constructing a `Font` object. `Font` is packaged in `javafx.scene.text`. For example, you can create a default font of a specified size by using this constructor:

```
Font(double fontSize)
```

Here, `fontSize` specifies the size of the font.

You can specify the fill and stroke using these two methods defined by `GrahpicsContext`:

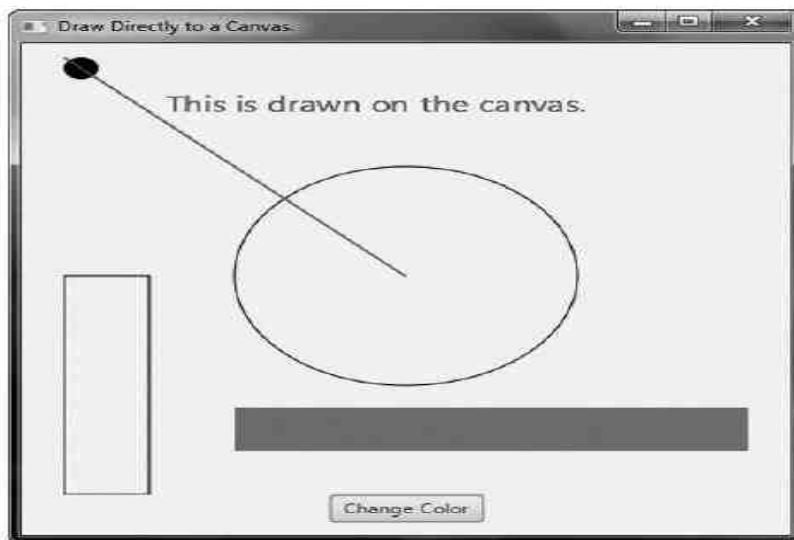
```
void setFill(Paint newFill)
```

```
void setStroke(Paint newStroke)
```

Notice that the parameter of both methods is of type `Paint`. This is an abstract class packaged in `javafx.scene.paint`. Its subclasses define fills and strokes. The one we will use is `Color`, which simply describes a solid color. `Color` defines several static fields that specify a wide array of colors, such as `Color.BLUE`, `Color.RED`, `Color.GREEN`, and so on.

# Example program to demonstrate Drawing

The following program uses the aforementioned methods to demonstrate drawing on a canvas. It first displays a few graphic objects on the canvas. Then, each time the Change Color button is pressed, the color of three of the objects changes color. If you run the program, you will see that the shapes whose color is not changed are unaffected by the change in color of the other objects. Furthermore, if you try covering and then uncovering the window, you will see that the canvas is automatically repainted, without any other actions on the part of your program. Sample output is shown here:



```
// Demonstrate drawing.

import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.event.*;
import javafx.geometry.*;
import javafx.scene.shape.*;
import javafx.scene.canvas.*;
import javafx.scene.paint.*;
import javafx.scene.text.*;

public class DirectDrawDemo extends Application {
    GraphicsContext gc;

    Color[] colors = { Color.RED, Color.BLUE, Color.GREEN, Color.BLACK };
    int colorIdx = 0;
```

```
public static void main(String[] args) {  
  
    // Start the JavaFX application by calling launch().  
    launch(args);  
}  
  
// Override the start() method.  
public void start(Stage myStage) {  
  
    // Give the stage a title.  
    myStage.setTitle("Draw Directly to a Canvas.");  
  
    // Use a FlowPane for the root node.  
    FlowPane rootNode = new FlowPane();  
  
    // Center the nodes in the scene.  
    rootNode.setAlignment(Pos.CENTER);  
  
    // Create a scene.  
    Scene myScene = new Scene(rootNode, 450, 450);  
  
    // Set the scene on the stage.  
    myStage.setScene(myScene);
```

```
// Create a canvas.  
Canvas myCanvas = new Canvas(400, 400);  
  
// Get the graphics context for the canvas.  
gc = myCanvas.getGraphicsContext2D();  
  
// Create a push button.  
Button btnChangeColor = new Button("Change Color");  
  
// Handle the action events for the Change Color button.  
btnChangeColor.setOnAction(new EventHandler<ActionEvent>() {  
    public void handle(ActionEvent ae) {  
  
        // Set the stroke and fill color.  
        gc.setStroke(colors[colorIdx]);  
        gc.setFill(colors[colorIdx]);  
  
        // Redraw the line, text, and filled rectangle in the  
        // new color. This leaves the color of the other nodes  
        // unchanged.  
        gc.strokeLine(0, 0, 200, 200);  
        gc.fillText("This is drawn on the canvas.", 60, 50);  
        ac.fillRect(100, 320, 300, 40);  
    }  
});
```

```
// Change the color.  
colorIdx++;  
if(colorIdx == colors.length) colorIdx= 0;  
}  
});  
  
// Draw initial output on the canvas.  
gc.strokeLine(0, 0, 200, 200);  
gc.strokeOval(100, 100, 200, 200);  
gc.strokeRect(0, 200, 50, 200);  
gc.fillOval(0, 0, 20, 20);  
gc.fillRect(100, 320, 300, 40);  
  
// Set the font size to 20 and draw text.  
gc.setFont(new Font(20));  
gc.fillText("This is drawn on the canvas.", 60, 50);  
  
// Add the canvas and button to the scene graph.  
rootNode.getChildren().addAll(myCanvas, btnChangeColor);  
  
// Show the stage and its scene.  
myStage.show();  
}
```

# **ToggleButton**

A toggle button looks just like a push button, but it acts differently because it has two states: **pushed and released**.

That is, when you press a toggle button, it stays pressed rather than popping back up as a regular push button does. When you press the toggle button a second time, it releases (pops up).

Therefore, each time a toggle button is pushed, it toggles between these two states.

In JavaFX, a toggle button is encapsulated in the **ToggleButton** class. Like **Button**, **ToggleButton** is also derived from **ButtonBase**. It implements the **Toggle** interface, which defines functionality common to all types of two-state buttons.

## **ToggleButton** constructors:

`public ToggleButton()`

Creates a toggle button with an empty string for its label

`public ToggleButton(String text)`

Creates a toggle button with the specified text as its label.

Parameters:

`text` - A text string for its label (the text displayed on the button)

When the button is pressed, the option is selected.

When the button is released, the option is deselected.

For this reason, a program usually needs to determine the toggle button's state.

To do this, use the **isSelected( )** method, shown here:

*final boolean isSelected( )*

It returns **true** if the button is pressed and **false** otherwise.

```
// Demonstrate a toggle button.  
import javafx.application.*;  
import javafx.scene.*;  
import javafx.stage.*;  
import javafx.scene.layout.*;  
import javafx.scene.control.*;  
import javafx.event.*;  
import javafx.geometry.*;  
public class ToggleButtonDemo extends Application {  
    ToggleButton tbOnOff;  
    Label response;  
    public static void main(String[] args) {  
        // Start the JavaFX application by calling launch().  
        launch(args);  
    }  
    // Override the start() method.
```

```
public void start(Stage myStage) {  
    // Give the stage a title.  
    myStage.setTitle("Demonstrate a Toggle Button");  
    // Use a FlowPane for the root node. In this case,  
    // vertical and horizontal gaps of 10.  
    FlowPane rootNode = new FlowPane(10, 10);  
    // Center the controls in the scene.  
    rootNode.setAlignment(Pos.CENTER);  
    // Create a scene.  
    Scene myScene = new Scene(rootNode, 220, 120);  
    // Set the scene on the stage.  
    myStage.setScene(myScene);  
    // Create a label.  
    response = new Label("Push the Button.");  
    // Create the toggle button.  
    tbOnOff = new ToggleButton("On/Off");
```

```
// Handle action events for the toggle button.  
tbOnOff.setOnAction(new EventHandler<ActionEvent>() {  
    public void handle(ActionEvent ae) {  
        if(tbOnOff.isSelected()) response.setText("Button is on.");  
        else response.setText("Button is off.");  
    }  
});  
// Add the label and buttons to the scene graph.  
rootNode.getChildren().addAll(tbOnOff, response);  
// Show the stage and its scene.  
myStage.show();  
}  
}
```

When the button is pressed, **isSelected( )** returns **true**.  
When the button is released, **isSelected( )** returns **false**.



# RadioButton

Another type of button provided by JavaFX is the *radio button*.

Radio buttons are a group of mutually exclusive buttons, in which only one button can be selected at any one time. They are supported by the **RadioButton** class, which extends both **ButtonBase** and **ToggleButton**.

It also implements the **Toggle** interface.

**Thus, a radio button is a specialized form of a toggle button.**

They are the primary control employed when the user must select only one option among several alternatives.

To create a radio button, we will use the following constructor:

*RadioButton(String str)*

Here, *str* is the label for the button.

Like other buttons, when a **RadioButton** is used, an action event is generated.

Radio buttons must be configured into a group. Only one of the buttons in the group can be selected at any time.

A button group is created by the **ToggleGroup** class, which is packaged in **javafx.scene.control**. **ToggleGroup** provides only a default constructor.

Radio buttons are added to the toggle group by calling the **setToggleGroup( )** method, defined by **ToggleButton**, on the button. It is shown here:

*final void setToggleGroup(ToggleGroup tg)*

Here, *tg* is a reference to the toggle button group to which the button is added.

In general, when radio buttons are used in a group, one of the buttons is selected when the group is first displayed in the GUI. Here are two ways to do this.

First, you can call  **setSelected( )** on the button that you want to select. It is defined by **ToggleButton** (which is a superclass of **RadioButton**). It is shown here:

*final void setSelected(boolean state)*

If *state* is **true**, the button is selected. Otherwise, it is deselected.

Although the button is selected, no action event is generated.

A second way to initially select a radio button is to call  **fire( )** on the button. It is shown here:      *void fire( )*

This method results in an action event being generated for the button if the button was previously not selected.

```
// This program responds to the action events generated
// by a radio button selection. It also shows how to
// fire the button under program control.
import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.event.*;
import javafx.geometry.*;
public class RadioButtonDemo extends Application {
    Label response;
    public static void main(String[] args) {
        // Start the JavaFX application by calling launch().
        launch(args);
    }
    // Override the start() method.
```

```
public void start(Stage myStage) {  
    // Give the stage a title.  
    myStage.setTitle("Demonstrate Radio Buttons");  
    // Use a FlowPane for the root node. In this case,  
    // vertical and horizontal gaps of 10.  
    FlowPane rootNode = new FlowPane(10, 10);  
    // Center the controls in the scene.  
    rootNode.setAlignment(Pos.CENTER);  
    // Create a scene.  
    Scene myScene = new Scene(rootNode, 220, 120);  
    // Set the scene on the stage.  
    myStage.setScene(myScene);  
    // Create a label that will report the selection.  
    response = new Label("");  
    // Create the radio buttons.  
    RadioButton rbTrain = new RadioButton("Train");  
    RadioButton rbCar = new RadioButton("Car");  
    RadioButton rbPlane = new RadioButton("Airplane");
```

```
// Create a toggle group.  
ToggleGroup tg = new ToggleGroup();  
// Add each button to a toggle group.  
rbTrain.setToggleGroup(tg);  
rbCar.setToggleGroup(tg);  
rbPlane.setToggleGroup(tg);  
// Handle action events for the radio buttons.  
rbTrain.setOnAction(new EventHandler<ActionEvent>() {  
    public void handle(ActionEvent ae) {  
        response.setText("Transport selected is train.");  
    }  
});  
rbCar.setOnAction(new EventHandler<ActionEvent>() {  
    public void handle(ActionEvent ae) {  
        response.setText("Transport selected is car.");  
    }  
});
```

```
rbPlane.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent ae) {
        response.setText("Transport selected is airplane.");
    }
});
// Fire the event for the first selection. This causes
// that radio button to be selected and an action event for that button to occur.
rbTrain.fire();
// Add the label and buttons to the scene graph.
rootNode.getChildren().addAll(rbTrain, rbCar, rbPlane, response);
// Show the stage and its scene.
myStage.show();
}
}
```



# CheckBox

In JavaFX, the check box is encapsulated by the **CheckBox** class. Its immediate superclass is **ButtonBase**. Thus it is a special type of button.

**CheckBox** supports three states. The first two are **checked or unchecked**, as you would expect, and this is the default behavior. The third state is *indeterminate* (also called *undefined*). **To add 3<sup>rd</sup> state,**

```
final void setAllowIndeterminate(boolean enable)  
final boolean isIndeterminate( )
```

Here is the **CheckBox** constructor that we will use:

*CheckBox(String str)*

It creates a check box that has the text specified by *str* as a label. As with other buttons, a **CheckBox** generates an action event when it is selected.

The following program demonstrates check boxes. It displays four check boxes that represent different types of computers. They are labeled Smartphone, Tablet, Notebook, and Desktop. Each time a check-box state changes, an action event is generated. It is handled by displaying the new state (selected or cleared) and by displaying a list of all selected boxes.

```
// Demonstrate Check Boxes.  
import javafx.application.*;  
import javafx.scene.*;  
import javafx.stage.*;  
import javafx.scene.layout.*;  
import javafx.scene.control.*;  
import javafx.event.*;  
import javafx.geometry.*;  
public class CheckboxDemo extends Application {  
    CheckBox cbSmartphone;  
    CheckBox cbTablet;  
    CheckBox cbNotebook;  
    CheckBox cbDesktop;  
    Label response;  
    Label selected;  
    String computers;
```

```
public static void main(String[] args) {
    // Start the JavaFX application by calling launch().
    launch(args);
}
// Override the start() method.
public void start(Stage myStage) {
    // Give the stage a title.
    myStage.setTitle("Demonstrate Check Boxes");
    // Use a vertical FlowPane for the root node. In this case,
    // vertical and horizontal gaps of 10.
    FlowPane rootNode = new FlowPane(Orientation.VERTICAL, 10, 10);
    // Center the controls in the scene.
    rootNode.setAlignment(Pos.CENTER);
    // Create a scene.
    Scene myScene = new Scene(rootNode, 230, 200);
    // Set the scene on the stage.
    myStage.setScene(myScene);
    Label heading = new Label("What Computers Do You Own?");
```

```
// Create a label that will report the state change of a check box.  
response = new Label("");  
// Create a label that will report all selected check boxes.  
selected = new Label("");  
// Create the check boxes.  
cbSmartphone = new CheckBox("Smartphone");  
cbTablet = new CheckBox("Tablet");  
cbNotebook = new CheckBox("Notebook");  
cbDesktop = new CheckBox("Desktop");  
// Handle action events for the check boxes.  
cbSmartphone.setOnAction(new EventHandler<ActionEvent>() {  
    public void handle(ActionEvent ae) {  
        if(cbSmartphone.isSelected())  
            response.setText("Smartphone was just selected.");  
        else  
            response.setText("Smartphone was just cleared.");  
        showAll();  
    }  
});
```

```
cbTablet.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent ae) {
        if(cbTablet.isSelected())
            response.setText("Tablet was just selected.");
        else
            response.setText("Tablet was just cleared.");
        showAll();
    }
});
cbNotebook.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent ae) {
        if(cbNotebook.isSelected())
            response.setText("Notebook was just selected.");
        else
            response.setText("Notebook was just cleared.");
        showAll();
    }
});
```

```
cbDesktop.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent ae) {
        if(cbDesktop.isSelected())
            response.setText("Desktop was just selected.");
        else
            response.setText("Desktop was just cleared.");
        showAll();
    }
});
// Add controls to the scene graph.
rootNode.getChildren().addAll(heading, cbSmartphone, cbTablet,
                               cbNotebook, cbDesktop, response, selected);
// Show the stage and its scene.
myStage.show();
showAll();
}
```

```
// Update and show the selections.  
void showAll() {  
    computers = "";  
    if(cbSmartphone.isSelected())  
        computers = "Smartphone ";  
    if(cbTablet.isSelected())  
        computers += "Tablet ";  
    if(cbNotebook.isSelected())  
        computers += "Notebook ";  
    if(cbDesktop.isSelected())  
        computers += "Desktop";  
    selected.setText("Computers selected: " + computers);  
}  
}
```

Sample output is shown here:





## ListView

A **ListView** can display a list of entries from which you can select one or more. Scrollbars are automatically added when the number of items in the list exceeds the number that can be displayed within the control's dimensions. **ListView** is a generic class that is declared like this:

```
class ListView<T>
```

Here, **T** specifies the type of entries stored in the list view. Often, these are entries of type **String**, but other types are also allowed.

Here is the **ListView** constructor that we will use:

```
ListView(ObservableList<T> list)
```

The list of items to be displayed is specified by *list*. It is an object of type **ObservableList**. **ObservableList** supports a list of objects.

By default, a **ListView** allows only one item in the list to be selected at any one time. You can allow multiple selections by changing the selection mode, but we will use the default, single-selection mode.

To create an ObservableList for use in a ListView is to use the factory method observableArrayList( ), which is a static method defined by the FXCollections class (which is packaged in javafx.collections).

*static <E> ObservableList<E> observableArrayList(E ... elements)*

In this case, E specifies the type of elements, which are passed via elements.

to set the preferred height and/or width, size:

*final void setPrefHeight(double height)*

*final void setPrefWidth(double width)*

*void setPrefSize(double width, double height)*

you can monitor the list for changes by registering a change listener. This lets you respond each time the user changes a selection in the list. A change listener is supported by the ChangeListener interface, which is packaged in javafx.beans.value. The ChangeListener interface defines only one method, called **changed( )**. It is shown here:

*void changed(ObservableValue<? extends T> changed, T oldVal, T newVal)*

In this case, changed is the instance of ObservableValue<T> which encapsulates an object that can be watched for changes. The oldVal and newVal parameters pass the previous value and the new value, respectively. Thus, in this case, newVal holds a reference to the list item that has just been selected.

To listen for change events, you must first obtain the selection model used by the ListView. This is done by calling `getSelectionModel( )` on the list. It is shown here:

*final MultipleSelectionModel<T> getSelectionModel( )*

It returns a reference to the model.

Using the model returned by `getSelectionModel( )`, you will obtain a reference to the selected item property that defines what takes place when an element in the list is selected.

This is done by calling `selectedItemProperty( )`, shown next:

*final ReadOnlyObjectProperty<T> selectedItemProperty( )*

You will add the change listener to this property by using the `addListener( )` method on the returned property. The `addListener( )` method is shown here:

*void addListener(ChangeListener<? super T> listener)*

In this case, T specifies the type of the property.

The following example creates a list view that displays a list of computer types, allowing the user to select one. When one is chosen, the selection is displayed.

```
// Demonstrate a list view.  
import javafx.application.*;  
import javafx.scene.*;  
import javafx.stage.*;  
import javafx.scene.layout.*;  
import javafx.scene.control.*;  
import javafx.geometry.*;  
import javafx.beans.value.*;  
import javafx.collections.*;  
public class ListViewDemo extends Application {  
    Label response;  
    public static void main(String[] args) {  
        // Start the JavaFX application by calling launch().  
        launch(args);  
    }  
    // Override the start() method.
```

```
public void start(Stage myStage) {  
    // Give the stage a title.  
    myStage.setTitle("ListView Demo");  
    // Use a FlowPane for the root node. In this case,  
    // vertical and horizontal gaps of 10.  
    FlowPane rootNode = new FlowPane(10, 10);  
    // Center the controls in the scene.  
    rootNode.setAlignment(Pos.CENTER);  
    // Create a scene.  
    Scene myScene = new Scene(rootNode, 200, 120);  
    // Set the scene on the stage.  
    myStage.setScene(myScene);  
    // Create a label.  
    response = new Label("Select Computer Type");  
    // Create an ObservableList of entries for the list view.  
    ObservableList<String> computerTypes =  
        FXCollections.observableArrayList("Smartphone", "Tablet", "Notebook", "Desktop" );
```

```
// Create the list view.  
ListView<String> lvComputers = new ListView<String>(computerTypes);  
// Set the preferred height and width.  
lvComputers.setPrefSize(100, 70);  
// Get the list view selection model.  
MultipleSelectionModel<String> lvSelModel = lvComputers.getSelectionModel();  
// Use a change listener to respond to a change of selection within a list view.  
lvSelModel.selectedItemProperty().addListener( new ChangeListener<String>() {  
    public void changed(ObservableValue<? extends String> changed,  
                        String oldVal,  
                        String newVal) {  
        // Display the selection.  
        response.setText("Computer selected is " + newVal);  
    }  
});  
// Add the label and list view to the scene graph.  
rootNode.getChildren().addAll(lvComputers, response);
```

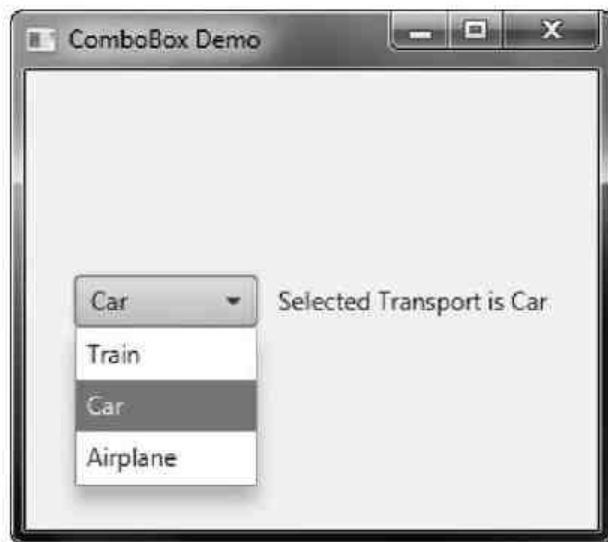
```
// Show the stage and its scene.  
myStage.show();  
}  
}  
}
```

Sample output is shown here.

Notice that a vertical scroll bar has been included so that the list can be scrolled to see all of its entries.



# JavaFX ComboBox



# ComboBox

- A combo box displays one selection, but it will also display a drop-down list that allows the user to select a different item.
- **ComboBox** inherits **ComboBoxBase** which provides much of its functionality.
- Unlike the **ListView**, which can allow multiple selections, **ComboBox** is designed for single-selection.
- **ComboBox** is a generic class that is declared like this:  
`class ComboBox<T>`
- Here, **T** specifies the type of entries. Often, these are entries of type **String**, but other types are also allowed.

# ComboBox constructors

- The default constructor creates an empty **ComboBox**.
- **ComboBox (ObservableList<T>/list)**  
*list* specifies a list of the items that will be displayed
- It is an object of type **ObservableList**, which defines a list of observable objects.
- **ObservableList** inherits **java.util.List**.
- An easy way to create an **ObservableList** is to use the factory method
- **observableArrayList( )**, which is a static method defined by the **FXCollections** class.

# Events Generated by Combobox

- A **ComboBox** generates an action event when its selection changes
- It will also generate a change event
- Alternatively, it is also possible to ignore events and simply obtain the current selection when needed
- To obtain the current selection by calling **getValue( )**, shown here:  
    final T getValue( )
- If the value of a combo box has not yet been set (by the user or under program control), then **getValue( )** will return **null**.
- To set the value of a **ComboBox** under program control, call **setValue( )**:  
    final void setValue(T *newVal*)

```
// Demonstrate a combo box.

import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.geometry.*;
import javafx.collections.*;
import javafx.event.*;

public class ComboBoxDemo extends Application {
    ComboBox<String> cbTransport;
    Label response;

    public static void main(String[] args) {
        // Start the JavaFX application by calling launch().
        launch(args);
    }
}
```

```
// Override the start() method.
public void start(Stage myStage) {

    // Give the stage a title.
    myStage.setTitle("ComboBox Demo");

    // Use a FlowPane for the root node. In this case,
    // vertical and horizontal gaps of 10.
    FlowPane rootNode = new FlowPane(10, 10);

    // Center the controls in the scene.
    rootNode.setAlignment(Pos.CENTER);

    // Create a scene.
    Scene myScene = new Scene(rootNode, 280, 120);

    // Set the scene on the stage.
    myStage.setScene(myScene);

    // Create a label.
    response = new Label();
```

```
// Create an ObservableList of entries for the combo box.  
ObservableList<String> transportTypes =  
    FXCollections.observableArrayList( "Train", "Car", "Airplane" );  
  
// Create a combo box.  
cbTransport = new ComboBox<String>(transportTypes);  
  
// Set the default value.  
cbTransport.setValue("Train");  
  
// Set the response label to indicate the default selection.  
response.setText("Selected Transport is " + cbTransport.getValue());  
  
// Listen for action events on the combo box.  
cbTransport.setOnAction(new EventHandler<ActionEvent>() {  
    public void handle(ActionEvent ae) {  
        response.setText("Selected Transport is " + cbTransport.getValue());  
    }  
});
```

```
// Add the label and combo box to the scene graph.  
rootNode.getChildren().addAll(cbTransport, response);  
  
// Show the stage and its scene.  
myStage.show();  
}  
}
```

# CERT JAVA CODING STANDARD

# Introduction

- Java programming language is a well-documented and enforceable coding standard.
- The CERT Oracle Secure Coding Standard for Java provides rules for secure coding in the Java programming language
- *The goal of these rules is to eliminate insecure coding practices that can lead to exploitable vulnerabilities*

# Introduction

- The application of the secure coding standard leads to higher quality systems that are safe, secure, reliable, dependable, robust, resilient, available, and maintainable and can be used as a metric to evaluate source code for these properties.
- Languages such as C and C++ allow undefined, unspecified, or implementation-defined behaviors, which can lead to *vulnerabilities when a programmer makes incorrect assumptions about the underlying behavior of an API or language construct.*
- The Java Language Specification goes further to standardize language requirements because Java is designed to be a “write once, run anywhere” language.

# Rules and Recommendations

- Rules are the requirements that we should or must follow otherwise it leads to some vulnerabilities
- Rules are the requirements for conformance with the standard.
- ***Recommendation*** describe good practices or useful advice. And *do not establish conformance requirements*.

# Recommendations

## 1. Do not declare more than one variable per declaration

- Declaring multiple variables in a single declaration could cause confusion about the types of variables and their initial values. In particular, do not declare any of the following in a single declaration:
  - n Variables of different types n
  - A mixture of initialized and uninitialized variablesIn general, *you should declare each variable on its own line with an explanatory comment regarding its role.*

### **Noncompliant Code Example (Initialization) :**

**This noncompliant code example:** might lead a programmer or reviewer to mistakenly believe that both i and j are initialized to 1. In fact, only j is initialized, while i remains uninitialized:

```
int i, j = 1;  
int i=10;double d=4.5;
```

**Compliant Solution (Initialization)** In this compliant solution, it is readily apparent that both i and j are initialized to 1:

```
int i = 1; // Purpose of i...  
int j = 1; // Purpose of j...
```

# Recommendations

## 2. Use meaningful symbolic constant to represent literal value in program logic

**Noncompliant Code Example** This noncompliant code example calculates approximate dimensions of a sphere, given its radius:

```
double area(double radius) { return 3.14 * radius * radius; }
double volume(double radius) { return 4.19 * radius * radius * radius; }
double greatCircleCircumference(double radius) { return 6.28 * radius; }
```

Although it removes some of the ambiguity from the literals, it complicates code maintenance. **If the programmer were to decide that a more precise value of  $\pi$  is desired, all occurrences of 3.14 in the code would have to be found and replaced.**

### Compliant Solution

In this compliant solution, **a constant PI is declared and initialized to 3.14**. Thereafter, it is referenced in the code whenever the value of  $\pi$  is needed.

```
private static final double PI = 3.14;
double area(double radius) { return PI * radius * radius; }
double volume(double radius) { return 4.0/3.0 * PI * radius * radius * radius; }
double greatCircleCircumference(double radius) { return 2 * PI * radius; }
```

# Rules

- Rules have a consistent structure. Each rule has a unique identifier, which is included in the title. The title of the rules and the introductory paragraphs define the conformance requirements. This is typically followed by one or more sets of noncompliant code examples and corresponding compliant solutions

# Rules

**Identifiers** : Each rule has a unique identifier, consisting of three parts:

- A three-letter mnemonic, representing the section of the standard, is used to group similar rules and make them easier to find.
  - A two-digit numeric value in the range of 00 to 99, which ensures each rule has a unique identifier.
  - The letter J, which indicates that this is a Java language rule and is included to prevent ambiguity with similar rules in CERT secure coding standards for other languages
- 
- Example 1. EXP00-J.
  - 2. NUM01-J

# EXPRESSIONS

EXP00-J. Do not ignore values returned by methods

EXP01-J. Never dereference null pointers

EXP02-J. Use the two-argument Arrays.equals()

method to compare the contents of arrays

# EXP00-J. Do not ignore values returned by methods

## Noncompliant Code

- Example (String Replacement) This **noncompliant code example ignores the return value of the String.replace() method**, failing to update the original string. The String.replace() method cannot modify the state of the String (because String objects are immutable); rather, it returns a reference to a new String object containing the modified string:

```
public class Replace
{
    public static void main(String[] args) {
        String original =
            "insecure";
        original.replace('i', '9');
        System.out.println(original);
    }
}
```

## **EXP00-J. Do not ignore values returned by methods**

- It is especially important to process the return values of immutable object methods. While many methods of mutable objects operate by changing some internal state of the object, methods of immutable objects cannot change the object and often return a mutated new object, leaving the original object unchanged.

### **Compliant Solution :**

- This compliant solution correctly updates the String reference `original` with the return value from the `String.replace()` method.

```
public class Replace
{
    public static void main(String[] args)
    {
        String original = "insecure";
        original = original.replace('i', '9');
        System.out.println(original);
    }
}
```

## EXP01-J. Never dereference null pointers

### Noncompliant Code Example

- The cardinality method was designed to return the number of occurrences of object obj in collection col. One valid use of the cardinality method is to determine how many objects in the collection are null. However, because membership in the collection is checked using the expression obj.equals(elt), a null pointer dereference is guaranteed whenever obj is null and elt is not null.

```
public static int cardinality(Object obj, final Collection col)
{ int count = 0; Iterator it = col.iterator();
  while (it.hasNext())
  { Object elt = it.next(); // null pointer dereference
    if ((null == obj && null == elt) || obj.equals(elt))
    { count++; }
  }
  return count;
}
```

## EXP01-J. Never dereference null pointers

### Compliant Code Example

- This compliant solution eliminates the null pointer dereference.

```
public static int cardinality(Object obj, final Collection col)
```

```
{ int count = 0;  
    Iterator it = col.iterator();  
    while (it.hasNext()) {  
        Object elt = it.next();  
        if ((null == obj && null == elt) || (null != obj && obj.equals(elt)))  
            { count++; }  
    }  
    return count;  
}
```

- Explicit null checks as shown here are an acceptable approach to eliminating null pointer dereferences.

# **EXP03-J. Use the two-argument Arrays.equals() method to compare the contents of arrays**

## **Noncompliant Code Example**

- This noncompliant code example incorrectly uses the Object.equals() method to compare two arrays.

```
public void arrayEqualsExample()
{
    int[] arr1 = new int[20]; // initialized to 0
    int[] arr2 = new int[20]; // initialized to 0
    arr1.equals(arr2); // false
}
```

# **EXP03-J. Use the two-argument Arrays.equals() method to compare the contents of arrays**

## **Compliant Code Example**

**Compliant Solution** This compliant solution compares the two arrays using the two-argument Arrays.equals() method.

```
public static boolean equals(int[] a1, int[] a2);
public void arrayEqualsExample()
{ int[] arr1 = new int[20]; // initialized to 0
  int[] arr2 = new int[20]; // initialized to 0
  Arrays.equals(arr1, arr2); // true
}
```

# Methods

- MET00-J. Validate method arguments
- MET03-J. Methods that perform security checks must be declared private
- MET04-J. Do not increase the accessibility of overridden or hidden methods

# MET00-J. Validate method arguments

- Validate method arguments to ensure that they fall within the bounds of the method's intended design. This practice ensures that operations on the method's parameters yield valid results.  
**Failure to validate method arguments can result in incorrect calculations, runtime exceptions, violation of class invariants, and inconsistent object state.**
- Caller validation of arguments can result in faster code because the caller may be aware of invariants that prevent invalid values from being passed

# Validate method arguments

## Noncompliant Code

- In this noncompliant code example, `setState()` and `useState()` fail to validate their arguments. A malicious caller could pass an invalid state to the library, consequently corrupting the library and exposing a vulnerability.

```
private Object myState = null;  
// Sets some internal state in the library  
void setState(Object state)  
{ myState = state; }  
// Performs some action using the file passed earlier  
void useState()  
{ // Perform some action here }
```

- Such vulnerabilities are particularly severe when the internal state contains or refers to sensitive or system-critical data.

# Validate method arguments

## Compliant Solution

- This compliant solution both validates the method arguments and verifies the internal state before use. This promotes consistency in program execution and reduces the potential for vulnerabilities.

```
private Object myState = null;  
// Sets some internal state in the library  
  
void setState(Object state)  
{ if (state == null)  
    { // Handle null state  
    }  
    myState=state;  
}  
// Performs some action using the state passed earlier  
void useState()  
{ if (myState == null)  
    {  
    // Handle no state (e.g. null) condition  
    }  
    //...  
}
```

# MET003-J. Methods that perform security checks must be declared private

- Member methods of non-final classes that perform security checks can be compromised when a malicious subclass overrides the methods and omits the checks. Consequently, such methods must be declared private or final to prevent overriding.
- class Base
- { // some sensitive data
- void readData()
- { // do some security check and access the data ( file)
- }
- }
- Class Sub extend Base
- {void readData()
- { // No security check and access the data ( file)
- }
- }

# Methods that perform security checks must be declared private

## Noncompliant Code Example

- This noncompliant code example allows a subclass to override the `readSensitiveFile()` method and omit the required security check.

```
public void readSensitiveFile() {  
    try {  
        SecurityManager sm = System.getSecurityManager();  
        if (sm != null) { // Check for permission to read file  
            sm.checkRead("/temp/tempFile");  
        } // Access the file  
    } catch (SecurityException se) { // Log exception } }
```

# Methods that perform security checks must be declared private

## Compliant Solution

- This compliant solution prevents overriding of the `readSensitiveFile()` method by declaring it final.

```
public final void readSensitiveFile()  
{  
try {  
    SecurityManager sm = System.getSecurityManager();  
    if (sm != null) { // Check for permission to read file  
        sm.checkRead("/temp/tempFile");  
    }  
    // Access the file  
} catch (SecurityException se) { // Log exception } }
```

## **MET004-J. Do not increase the accessibility of overridden or hidden methods**

- Increasing the accessibility of overridden or hidden methods permits a malicious subclass to offer wider access to the restricted method than was originally intended.
- Consequently, programs must override methods only when necessary and must declare methods final whenever possible to prevent malicious subclassing.

# **Do not increase the accessibility of overridden or hidden methods**

## **Noncompliant Code Example**

This noncompliant code example demonstrates how a malicious subclass Sub can both override the doLogic() method of its superclass and increase the accessibility of the overriding method. Any user of Sub can invoke the doLogic() method because the base class Super defines it to be protected, consequently allowing class Sub to increase the accessibility of doLogic() by declaring its own version of the method to be public.

```
class Super
{
    protected void doLogic()
    {
        System.out.println("Super invoked");
    }
}

public class Sub extends Super
{
    public void doLogic() {
        System.out.println("Sub invoked"); // Do sensitive operations } }
```

# Do not increase the accessibility of overridden or hidden methods

## Compliant Solution

- This compliant solution prevents overriding of the `readSensitiveFile()` method by declaring it `final`.

```
class Super
{ protected final void doLogic()
{ // declare as final
System.out.println("Super invoked");      // Do sensitive
operations } }
```

# Numeric Types

- NUM00-J. Detect or prevent integer overflow
- NUM01-J. Do not perform bitwise and arithmetic operations on the same data
- NUM02-J. Ensure that division and modulo operations do not result in divide-by-zero errors
- NUM03-J. Use integer types that can fully represent the possible range of unsigned data
- NUM04-J. Do not use floating-point numbers if precise computation is required
- NUM05-J. Do not use denormalized numbers

## **NUM00-J. Detect or prevent integer overflow**

- Programs must not allow mathematical operations to exceed the integer ranges provided by their primitive integer data types

**Noncompliant Code Example** Either operation in this noncompliant code example could result in an overflow. When overflow occurs, the result will be incorrect.

```
public static int multAccum(int oldAcc, int newVal, int scale)
{ // May result in overflow
    return oldAcc + (newVal * scale);
}
```

**Compliant Solution (Precondition Testing)** This compliant solution uses the safeAdd() and safeMultiply() methods defined in the Precondition Testing section to perform secure integral operations or throw ArithmeticException on overflow.

```
public static int multAccum(int oldAcc, int newVal, int scale) throws ArithmeticException
{
    return safeAdd(oldAcc, safeMultiply(newVal, scale));
}
```

## Compliant Solution (Upcasting)

This compliant solution shows the implementation of a method for checking whether value of type long falls within the representable range of an int using the upcasting technique. The implementations of range checks for the smaller primitive integer types are similar.

```
public static long intRangeCheck(long value)    throws ArithmeticException
{ if ((value < Integer.MIN_VALUE) || (value > Integer.MAX_VALUE))
    throw new ArithmeticException("Integer overflow");
    return value;
}

public static int multAccum(int oldAcc, int newVal, int scale) throws ArithmeticException
{
    final long res = intRangeCheck( ((long) oldAcc) + intRangeCheck((long) newVal *
  (long) scale) );
    return (int) res; // safe down-cast
}
```

## **NUM02-J. Ensure that division and modulo operations do not result in divide-by-zero errors**

**Noncompliant Code Example (Division)** The result of the / operator is the quotient from the division of the first arithmetic operand by the second arithmetic operand. Division operations are susceptible to divide-by-zero errors. This noncompliant code example can result in a divide-by-zero error during the division of the signed operands num1 and num2.

```
long num1, num2, result;  
/* Initialize num1 and num2 */  
result = num1 / num2;
```

### **Compliant Solution (Division)**

This compliant solution tests the divisor to guarantee there is no possibility of divide-by zero errors.

```
long num1, num2, result; /* Initialize num1 and num2 */  
if (num2 == 0)  
{ // handle error  
}  
else  
{ result = num1 / num2; }
```

## **Noncompliant Code Example (Modulo)**

- The % operator provides the remainder when two operands of integer type are divided. This noncompliant code example can result in a divide-by-zero error during the remainder operation on the signed operands num1 and num2.

```
long num1, num2, result;  
/* Initialize num1 and num2 */  
result = num1 % num2;
```

**Compliant Solution (Modulo)** This compliant solution tests the divisor to guarantee there is no possibility of a divide-by zero error.

```
long num1, num2, result;  
/* Initialize num1 and num2 */  
if (num2 == 0)  
{  
    // handle error  
}  
else  
{ result = num1 % num2; }
```

**NUM04-J. Do not use floating point numbers if precise computation is required**

## **Noncompliant Code Example (Modulo)**

- Noncompliant Code Example This noncompliant code example performs some basic currency calculations.

```
double dollar = 1.00;
```

```
double dime = 0.10;
```

```
int number = 7;
```

```
System.out.println("A dollar less " + number + " dimes is $" +  
(dollar - number * dime) );
```

- Because the value 0.10 lacks an exact representation in either Java floating-point type (or any floating-point format that uses a binary mantissa), on most platforms, this program prints:
- A dollar less 7 dimes is \$0.2999999999999993

## **NUM04-J. Do not use floating point numbers if precise computation is required**

- **Compliant Solution** this compliant solution uses an integer type (such as long) and works with cents rather than dollars.
- `long dollar = 100; long dime = 10;`  
`int number = 7;`
- `System.out.println("A dollar less " + number + "`  
`dimes is " + (dollar - number * dime) + " cents"`  
`');`
- This code correctly outputs:
- `A dollar less 7 dimes is 30 cents`
- `// 1 dollar=100 cents, 1dime=10cents`

**Ten dimes make a dollar**

**A dollar is 100 cents**

# Strings

- STR04-J. Do not encode non character data as String
- STR02J. Specify appropriate locale while comparing  
locale dependent data

# STR00-J. Do not encode non character data as String

- Increasingly, programmers view strings as a portable means of storing and communicating arbitrary data, such as numeric values.
- For example, a real-world system stores the binary values of encrypted passwords as strings in a database. Non-character data may not be representable as a string, because not all bit patterns represent valid characters in most character sets.

# Do not encode non character data as String

## Noncompliant Code

- This noncompliant code example attempts to convert a BigInteger value to a String and then restore it to a BigInteger value.
- The `toByteArray()` method used returns a byte array containing the two's-complement representation of this BigInteger. The byte array is in big-endian byte order: the most significant byte is in the zeroth element. The program uses the `String(byte[] bytes)` constructor to create the string from the byte array. The behavior of this constructor when the given bytes are not valid in the default character set is unspecified, which is likely to be the case.

```
BigInteger x = new BigInteger("530500452766");
byte[] byteArray = x.toByteArray();
String s = new String(byteArray);
```

# **Do not encode non character data as String**

## **Compliant Code**

This compliant solution first produces a String representation of the BigInteger object and then converts the String object to a byte array. This process is then reversed. Because the textual representation in the String object is generated by the BigInteger class, it contains valid character data in the default character set.

```
BigInteger x = new BigInteger("530500452766");
String s = x.toString(); // Valid character data
byte[] byteArray = s.getBytes();
```

- STR00-J. Specify appropriate locale while comparing locale dependent data
- Using locale-dependent methods on locale-dependent data can produce unexpected results when the locale is unspecified

# **Specify appropriate locale while comparing locale dependent data**

## **Noncompliant Code**

Using locale-dependent methods on locale-dependent data can produce unexpected results when the locale is unspecified. Programming language identifiers, protocol keys, and HTML tags are often specified in a particular locale, usually Locale.ENGLISH. Running a program in a different locale may result in unexpected program behavior

```
public static void processTag(String tag) {  
    if (tag.toUpperCase().equals("SCRIPT")) {  
        return;  
    }  
    // Process tag  
}
```

## **Specify appropriate locale while comparing locale dependent data**

### **Compliant Code**

- This compliant solution explicitly sets the locale to English to avoid unexpected results:
- Specifying Locale.ROOT is a suitable alternative when an English-specific locale would not be appropriate.

```
public static void processTag(String tag) {  
  
    if (tag.toUpperCase(Locale.ENGLISH).equals("SCRIPT")) {  
  
        return;  
  
    }  
  
    // Process tag  
  
}
```