
JAVA SE
PROGRAMMING

GOOD MORNING AND WELCOME!

We will start the class soon… while waiting please visit the link for all course-related resources.

http://asmaliza.com/java-se-programming/

Thank you!

Trainer Emma

HELLO EVERYONE!

My name is Asmaliza Ahzan. You can call me Emma ☺

My email address is asmaliza@iverson.com.my

INTRODUCTION

- Name
- Job Role/Designation
- Programming Experiences (if any)
- Expectations from this course

LOGISTICS

- Sessions: 9am-5pm
- 10-15mins breaks
- 1 hour lunch break
- 5 days

AGENDAS

- Java Fundamentals
- Introducing Data Types and Operators
- Program Control Statements
- More Data Types and Operators
- A Close Look at Methods and Classes
- Inheritance
- Packages and Interfaces
- Exception Handling
- Using I/O
- Multithreaded Programming
- Enumerations, Autoboxing, Static Imports and Annotations
- Generics
- Lambda Expressions and Method References

JAVA FUNDAMENTAL

THE HISTORY AND PHILOSOPHY OF JAVA

- Java was conceived by James Gosling, Patrick Naughton, Chris Warth, Ed Frank, and Mike Sheridan at Sun Microsystems, Inc. in 1991.
- It took 18 months to develop the first working version.
- This language was initially called “Oak,” but was renamed “Java” in 1995.
- Based on C++ programming language.
- Platform independent.

OBJECT-ORIENTED PROGRAMMING

- OOP is at the core of Java.
- The 3 OOP principles
  - Encapsulation – the mechanism that binds together code and the data it manipulates and keeps both safe from outside interference and misuse.
  - Inheritance – the process by which one object acquires the properties of another object.
  - Polymorphism – a feature that allows one interface to be used for a general class of actions.

JAVA DEVELOPMENT KIT (JDK)

The Java Development Kit is an implementation of either one of the Java Platform, Standard Edition, Java Platform, Enterprise Edition, or Java Platform, Micro Edition platforms released by Oracle Corporation in the form of a binary product aimed at Java developers on Solaris, Linux, macOS or Windows.

JAVA RUNTIME ENVIRONMENT (JRE)

The Java Runtime Environment, or JRE, is a software layer that runs on top of a computer's operating system software and provides the class libraries and other resources that a specific Java program needs to run.

JAVA VIRTUAL MACHINE (JVM)

A Java virtual machine (JVM) is a virtual machine that enables a computer to run Java programs as well as programs written in other languages that are also compiled to Java bytecode. The JVM is detailed by a specification that formally describes what is required in a JVM implementation.

A FIRST SIMPLE PROGRAM

ACTIVITY

JAVA BASICS

- Java programs are enclosed in a pair of curly brackets that indicates a block of codes.
- Every statement in Java must ends with a semicolon.
- Java is case-sensitive.
- Whitespace in Java program is ignored.

IDENTIFIERS

Identifiers are used to name things, such as classes, variables, and methods.

- Must begin with a letter, underscore, or a dollar sign.
- Cannot start with a number.
- A single word.
- Case sensitive.
- Cannot use Java keywords.

COMMENTS

- // for single-line comments.
- /* */ for multiple-lines comments.
- /** */ for Javadoc comments.

THE JAVA KEYWORDS

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

THE JAVA CLASS LIBRARIES

- Full list of Java class libraries can be found at the official page https://docs.oracle.com/en/java/javase/11/docs/api/

All Modules | Java SE | JDK | Other Modules

Module	Description
java.base	Defines the foundational APIs of the Java SE Platform.
java.compiler	Defines the Language Model, Annotation Processing, and Java Compiler APIs.
java.datatransfer	Defines the API for transferring data between and within applications.
java.desktop	Defines the AWT and Swing user interface toolkits, plus APIs for accessibility, audio, imaging, printing, and JavaBeans.
java.instrument	Defines services that allow agents to instrument programs running on the JVM.
java.logging	Defines the Java Logging API.
java.management	Defines the Java Management Extensions (JMX) API.
java.management.rmi	Defines the RMI connector for the Java Management Extensions (JMX) Remote API.
java.naming	Defines the Java Naming and Directory Interface (JNDI) API.

ACTIVITY

EXERCISE 1

INTRODUCING DATA TYPES AND OPERATORS

WHY DATA TYPES ARE IMPORTANT

- Java is a strongly typed language.
- Every variable, expression has a type, and every type is strictly defined.
- All assignments, whether explicit or via parameter passing in method calls, are checked for type compatibility.
- No automatic coercions or conversions of conflicting types as in some languages.
- The Java compiler checks all expressions and parameters to ensure that the types are compatible.
- Any type mismatches are errors that must be corrected before the compiler will finish compiling the class.

JAVA'S PRIMITIVE TYPES

Java defines eight primitive types of data: byte, short, int, long, char, float, double, and boolean.

Group	Description
Integers	This group includes byte, short, int, and long, which are for whole-valued signed numbers.
Float-point	This group includes float and double, which represent numbers with fractional precision.
Characters	This group includes char, which represents symbols in a character set, like letters and numbers.
Boolean	This group includes boolean, which is a special type for representing true/false values.

INTEGERS

Java defines four integer types: byte, short, int, and long. All of these are signed, positive and negative values. Java does not support unsigned, positive-only integers.

Name	Width	Range
long	64	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
int	32	–2,147,483,648 to 2,147,483,647
short	16	–32,768 to 32,767
byte	8	–128 to 127

FLOATING POINTS

Floating-point numbers, also known as real numbers, are used when evaluating expressions that require fractional precision. For example, calculations such as square root, or transcendentals such as sine and cosine, result in a value whose precision requires a floating point type.

Name	Width	Range
double	64	4.9e–324 to 1.8e+308
float	32	1.4e–045 to 3.4e+038

CHARACTERS

In Java, the data type used to store characters is char. At the time of Java's creation, Unicode required 16 bits. Thus, in Java char is a 16-bit type. The range of a char is 0 to 65,536. There are no negative chars. The standard set of characters known as ASCII still ranges from 0 to 127 as always, and the extended 8-bit character set, ISO-Latin-1, ranges from 0 to 255.

BOOLEAN

Java has a primitive type, called boolean, for logical values. It can have only one of two possible values, true or false.

VARIABLES

The variable is the basic unit of storage in a Java program. A variable is defined by the combination of an identifier, a type, and an optional initializer. In addition, all variables have a scope, which defines their visibility, and a lifetime.

DECLARING A VARIABLE

In Java, all variables must be declared before they can be used. The basic form of a variable declaration is shown here:

type identifier [= value][, identifier [= value] …];

THE SCOPE AND LIFETIME OF VARIABLES

- A block defines a scope. Thus, each time you start a new block, you are creating a new scope.
- A scope determines what objects are visible to other parts of your program.
- It also determines the lifetime of those objects.

OPERATORS

Java provides a rich operator environment. Most of its operators can be divided into the following four groups: arithmetic, bitwise, relational, and logical.

ARITHMETIC OPERATORS

Operator	Result
+	Addition (also unary plus)
–	Subtraction (also unary minus)
*	Multiplication
/	Division
%	Modulus
++	Increment
– –	Decrement
+=	Addition assignment
– =	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Modulus assignment

BITWISE OPERATORS

Operator	Result
~	Bitwise unary NOT
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
>>	Shift right
>>>	Shift right zero fill
<<	Shift left
&=	Bitwise AND assignment
|=	Bitwise OR assignment
^=	Bitwise exclusive OR assignment
>>=	Shift right assignment
>>>=	Shift right zero fill assignment
<<=	Shift left assignment

RELATIONAL OPERATORS

Operator	Result
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

BOOLEAN LOGICAL OPERATORS

Operator	Result
&	Logical AND
|	Logical OR
^	Logical XOR (exclusive OR)
||	Short-circuit OR
&&	Short-circuit AND
!	Logical unary NOT
&=	AND assignment
|=	OR assignment
^=	XOR assignment
==	Equal to
!=	Not equal to
?:	Ternary if-then-else

BOOLEAN LOGICAL OPERATIONS

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

THE ASSIGNMENT OPERATOR

- The assignment operator is the single equal sign, =.
- The assignment operator works in Java much as it does in any other computer language.
- It has this general form:

```
var = expression;
```

OPERATOR PRECEDENCE

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

USING PARENTHESES

Parentheses raise the precedence of the operations that are inside them. This is often necessary to obtain the result you desire. For example, consider the following expression:

```
a >> b + 3
```

This expression first adds 3 to b and then shifts a right by that result. That is, this expression can be rewritten using redundant parentheses like this:

```
a >> (b + 3)
```

However, if you want to first shift a right by b positions and then add 3 to that result, you will need to parenthesize the expression like this:

```
(a >> b) + 3
```

TYPE CONVERSION IN ASSIGNMENTS

- If the two types are compatible, then Java will perform the conversion automatically. For example, it is always possible to assign an int value to a long variable.
- However, not all types are compatible, and thus, not all type conversions are implicitly allowed. For instance, there is no automatic conversion defined from double to byte.
- Fortunately, it is still possible to obtain a conversion between incompatible types.
- To do so, you must use a cast, which performs an explicit conversion between incompatible types.

JAVA'S AUTOMATIC CONVERSIONS

- When one type of data is assigned to another type of variable, an automatic type conversion will take place if the following two conditions are met:
  - The two types are compatible.
  - The destination type is larger than the source type.
- When these two conditions are met, a widening conversion takes place. For example, the int type is always large enough to hold all valid byte values, so no explicit cast statement is required.
- For widening conversions, the numeric types, including integer and floating-point types, are compatible with each other. However, there are no automatic conversions from the numeric types to char or boolean. Also, char and boolean are not compatible with each other.

CASTING INCOMPATIBLE TYPES

To create a conversion between two incompatible types, you must use a cast. A cast is simply an explicit type conversion. It has this general form:

(target-type) value

EXPRESSIONS

An expression is a construct made up of variables, operators, and method invocations, which are constructed according to the syntax of the language, that evaluates to a single value.

```
int cadence = 0;
anArray[0] = 100;
System.out.println("Element 1 at index 0: " + anArray[0]);

int result = 1 + 2; // result is now 3
if (value1 == value2)
    System.out.println("value1 == value2");
```

ACTIVITY

EXERCISE 2 AND 3

PROGRAM CONTROL STATEMENTS

DECISION CONSTRUCTS

- Java supports two selection statements: if and switch.
- These statements allow you to control the flow of your program's execution based upon conditions known only during run time.

THE IF-ELSE STATEMENT

The if statement is Java's conditional branch statement. It can be used to route program execution through two different paths. Here is the general form of the if statement:

```
if (condition) statement1;
else statement2;
```

Here, each statement may be a single statement, or a compound statement enclosed in curly braces (that is, a block). The condition is any expression that returns a boolean value. The else clause is optional.

NESTED IFS

- A nested if is an if statement that is the target of another if or else.
- Nested ifs are very common in programming.
- When you nest ifs, the main thing to remember is that an else statement always refers to the nearest if statement that is within the same block as the else and that is not already associated with an else.

THE IF-ELSE-IF LADDER

- A common programming construct that is based upon a sequence of nested ifs is the if-elseif ladder.
- It looks like this:

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

THE SWITCH STATEMENT

- The switch statement is Java's multiway branch statement.
- It provides an easy way to dispatch execution to different parts of your code based on the value of an expression.
- As such, it often provides a better alternative than a large series of if-else-if statements.
- Here is the general form of a switch statement:

```
switch (expression) {
case value1:
// statement sequence
break;
case value2:
// statement sequence
break;
.
.
.
case valueN :
// statement sequence
break;
default:
// default statement sequence
}
```

NESTED SWITCH STATEMENTS

```
switch(count) {
   case 1:
          switch(target) { // nested switch
                 case 0:
                        System.out.println("target is zero");
                        break;
                 case 1: // no conflicts with outer switch
                        System.out.println("target is one");
                        break;
          }
        break;
     case 2: // ...
```

ITERATION

- Java's iteration statements are for, while, and do-while.
- A loop repeatedly executes the same set of instructions until a termination condition is met.

WHILE

- The while loop is Java's most fundamental loop statement. It repeats a statement or block while its controlling expression is true.
- Here is its general form:

```
while(condition) {
    // body of loop
}
```

- The condition can be any Boolean expression.
- The body of the loop will be executed if the conditional expression is true. When condition becomes false, control passes to the next line of code immediately following the loop.

DO-WHILE

- The do-while loop always executes its body at least once, because its conditional expression is at the bottom of the loop. Its general form is:

```
do {
  // body of loop
} while
```

FOR

- Beginning with JDK 5, there are two forms of the for loop. The first is the traditional form that has been in use since the original version of Java. The second is the newer “for-each” form.
- Both types of for loops are discussed here, beginning with the traditional form. Here is the general form of the traditional for statement:

```
for(initialization; condition; iteration) {
    // body
}
```

FOREACH

- The for-each style of for is also referred to as the enhanced for loop.
- The general form of the for-each version of the for is shown here:

for(*type itr-var* : *collection*) *statement-block*

NESTED LOOPS

```
// Loops may be nested.
class Nested {
  public static void main(String args[]) {
        int i, j;
        for(i=0; i<10; i++) {
              for(j=i; j<10; j++)
                    System.out.print(".");
                    System.out.println();
        }
  }
}
```

BRANCHING

- Java supports three jump statements: break, continue, and return.
- These statements transfer control to another part of your program.

USE BREAK TO EXIT A LOOP

```
// Using break to exit a loop.
class BreakLoop {
  public static void main(String args[]) {
        for(int i=0; i<100; i++) {
              if(i == 10) break; // terminate loop if i is 10
              System.out.println("i: " + i);
        }
        System.out.println("Loop complete.");
  }
}
```

USE BREAK AS A FORM OF GOTO

```
// Using break as a civilized form of goto.
class Break {
    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");
                }
                System.out.println("This won't execute");
            }
            System.out.println("This is after second block.");
        }
    }
}
```

USE CONTINUE

```
// Demonstrate continue.
class Continue {
  public static void main(String args[]) {
      for(int i=0; i<10; i++) {
              System.out.print(i + " ");
              if (i%2 == 0) continue;
              System.out.println("");
        }
  }
}
```

USE RETURN

```
// Demonstrate return.
class Return {
  public static void main(String args[]) {
        boolean t = true;
        System.out.println("Before the return.");
        if(t) return; // return to caller
        System.out.println("This won't execute.");
  }
}
```

ACTIVITY

EXERCISE 4 AND 5

INTRODUCING CLASSES, OBJECTS, AND METHODS

CLASS FUNDAMENTALS

- Class defines a new data type.
- Once defined, this new type can be used to create objects of that type.
- Thus, a class is a template for an object, and an object is an instance of a class.
- A class is declared by use of the class keyword.
- The data, or variables, defined within a class are called instance variables.
- The code is contained within methods.
- Collectively, the methods and variables defined within a class are called members of the class.

CLASS STRUCTURE

```
class classname {
        type instance-variable1;
        type instance-variable2;
        // ...
        type instance-variableN;

        type methodname1(parameter-list) {
                // body of method
        }

        type methodname2(parameter-list) {
                // body of method
        }
        // ...

        type methodnameN(parameter-list) {
                // body of method
        }
}
```

BOX CLASS

Here is a class called Box that defines three instance variables: width, height, and depth. Currently, Box does not contain any methods.

```
class Box {
        double width;
        double height;
        double depth;
}
```

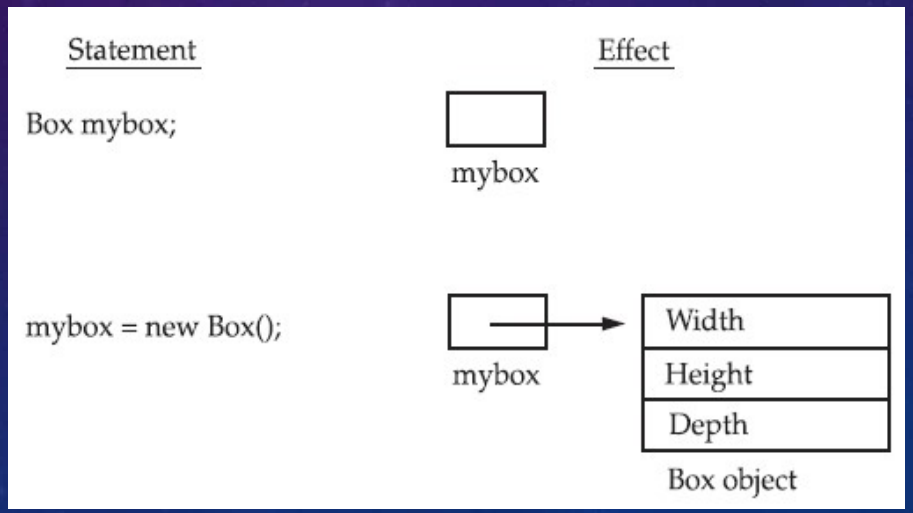
HOW OBJECTS ARE CREATED

```
// This program declares two Box objects.
class BoxDemo2 {
      public static void main(String args[]) {
            Box mybox1 = new Box();
            Box mybox2 = new Box();
            double vol;

            // assign values to mybox1's instance variables
            mybox1.width = 10;
            mybox1.height = 20;
            mybox1.depth = 15;

            /* assign different values to mybox2's instance variables */
            mybox2.width = 3;
            mybox2.height = 6;
            mybox2.depth = 9;

            // compute volume of first box
            vol = mybox1.width * mybox1.height * mybox1.depth;
            System.out.println("Volume is " + vol);

            // compute volume of second box
            vol = mybox2.width * mybox2.height * mybox2.depth;
            System.out.println("Volume is " + vol);
      }
}
```

REFERENCE VARIABLES AND ASSIGNMENT

- As just explained, the new operator dynamically allocates memory for an object. It has this general form:

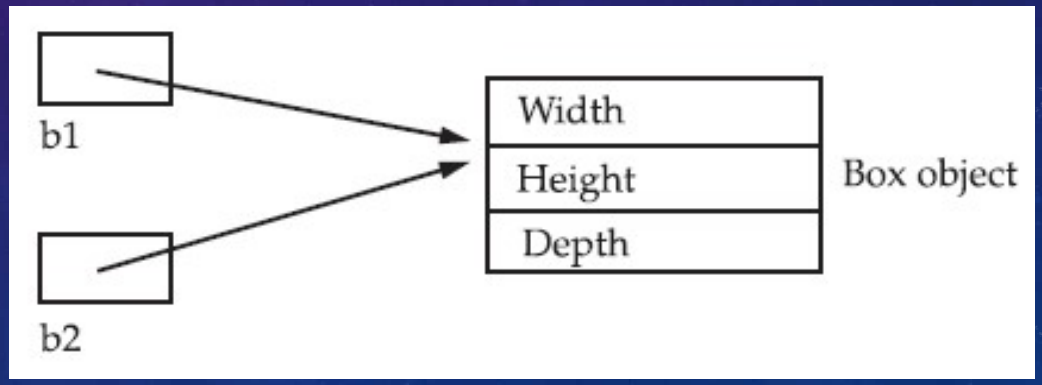
  class-var = new *classname* ( );



REFERENCE VARIABLES AND ASSIGNMENT

- Object reference variables act differently than you might expect when an assignment takes place.
- For example, what do you think the following fragment does?

```
Box b1 = new Box();
Box b2 = b1;
```



METHODS

- This is the general form of a method:

```
type name(parameter-list) {
        // body of method
}
```

- Type specifies the type of data returned by the method.
- The name of the method is specified by name.
- The parameter-list is a sequence of type and identifier pairs separated by commas.

BOX CLASS WITH METHOD

```
// This program includes a method inside the box class.
class Box {
	double width;
	double height;
	double depth;

	// display volume of a box
	void volume() {
		System.out.print("Volume is ");
		System.out.println(width * height * depth);
	}
}
```

RETURNING A VALUE

```
// Now, volume() returns the volume of a box.
class Box {
	double width;
	double height;
	double depth;

	// compute and return volume
	double volume() {
		return width * height * depth;
	}
}
```

METHOD THAT TAKES PARAMETERS

```
// This program uses a parameterized method.
class Box {
	double width;
	double height;
	double depth;

	// compute and return volume
	double volume() {
		return width * height * depth;
	}

	// sets dimensions of box
	void setDim(double w, double h, double d) {
		width = w;
		height = h;
		depth = d;
	}
}
```

CONSTRUCTORS

- A constructor initializes an object immediately upon creation.
- It has the same name as the class in which it resides and is syntactically like a method.
- The constructor is automatically called when the object is created before the new operator completes.
- Constructors look a little strange because they have no return type, not even void.
- It is the constructor's job to initialize the internal state of an object so that the code creating an instance will have a fully initialized, usable object immediately.

BOX CLASS WITH CONSTRUCTOR

```
/* Here, Box uses a constructor to initialize the dimensions of a box. */
class Box {
	double width;
	double height;
	double depth;

	// This is the constructor for Box.
	Box() {
		System.out.println("Constructing Box");
		width = 10;
		height = 10;
		depth = 10;
	}

	// compute and return volume
	double volume() {
		return width * height * depth;
	}
}
```

PARAMETERIZED CONSTRUCTORS

```
/* Here, Box uses a parameterized constructor to initialize the dimensions
of a box. */
class Box {
      double width;
      double height;
      double depth;

      // This is the constructor for Box.
      Box(double w, double h, double d) {
            width = w;
            height = h;
            depth = d;
      }

      // compute and return volume
      double volume() {
            return width * height * depth;
      }
}
```

THE THIS KEYWORD

- Sometimes a method will need to refer to the object that invoked it. To allow this, Java defines the this keyword.
- this can be used inside any method to refer to the current object.
- That is, this is always a reference to the object on which the method was invoked.

```
// Use this to resolve name-space collisions.
Box(double width, double height, double depth) {
      this.width = width;
      this.height = height;
      this.depth = depth;
}
```

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.
- It works like this: 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.

ACTIVITY

EXERCISE 6 AND 7

MORE DATA TYPES AND OPERATORS

ARRAYS

- An array is a group of like-typed variables that are referred to by a common name.
- Array of any type (primitive or class) can be created and may have one or more dimensions.
- A specific element in an array is accessed by its index.
- Arrays offer a convenient means of grouping related information.
- In Java, an array is an object.
- Array is a fixed size list of values.
- When declaring an array, the size must be supplied.

ONE-DIMENSIONAL ARRAYS

- A one-dimensional array is, essentially, a list of like-typed variables.
- Syntax to declare a one-dimensional array.

  type array-var = new *type* [*size*];

  int month_days = new int[12];

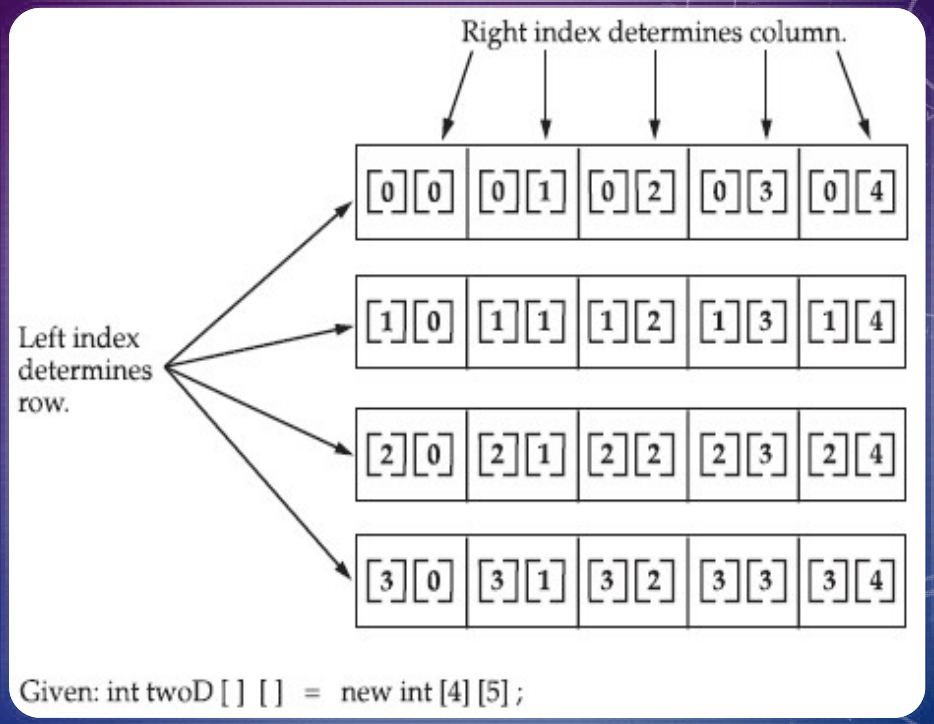
- To access and display array element, use the index.

  System.out.println(month_days[3]);

MULTIDIMENSIONAL ARRAYS

- In Java, multidimensional arrays are actually arrays of arrays.
- To declare a multidimensional array variable, specify each additional index using another set of square brackets.

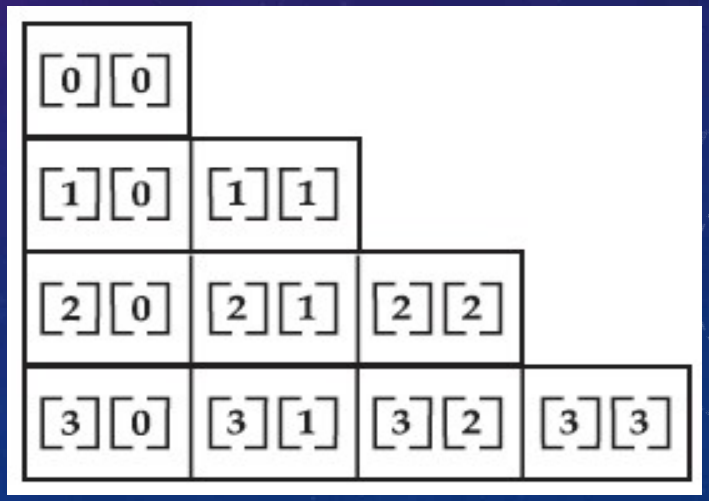
```
int twoD[][] = new int[4][5];
```



JAGGED ARRAY

- When you allocate memory for a multidimensional array, you need only specify the memory for the first (leftmost) dimension.
- You can allocate the remaining dimensions separately.

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



ALTERNATIVE ARRAY DECLARATION SYNTAX

- There is a second form that may be used to declare an array: *type*[ ] *var-name*;
- Here, the square brackets follow the type specifier, and not the name of the array variable.
- For example, the following two declarations are equivalent:

```
int al[] = new int[3];
int[] a2 = new int[3];
```

- This alternative declaration form offers convenience when declaring several arrays at the same time.

```
int[] nums, nums2, nums3; // create three arrays
int nums[], nums2[], nums3[]; // create three arrays
```

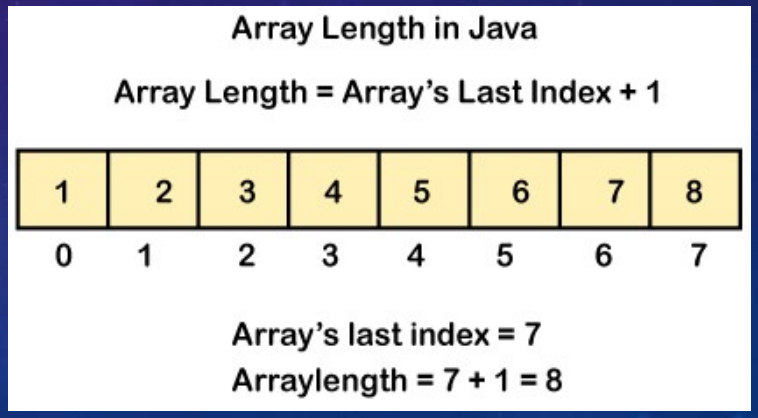
ASSIGNING ARRAY REFERENCES

- As with other objects, when you assign one array reference variable to another, you are simply changing what object that variable refers to.
- You are not causing a copy of the array to be made, nor are you causing the contents of one array to be copied to the other

USING THE LENGTH MEMBER

- All array indices in Java begin at 0.
- The number of elements in an array is stored as part of the array object in the length attribute.
- If an out-of-bounds runtime access occurs, then a runtime exception is thrown.

```
int[] arr = new int[5];
int arrayLength = arr.length;
```



THE FOR-EACH STYLE FOR LOOP

- For-each is another array traversing technique like for loop, while loop, do-while loop introduced in Java5.
- It starts with the keyword for like a normal for-loop.
- Instead of declaring and initializing a loop counter variable, you declare a variable that is the same type as the base type of the array, followed by a colon, which is then followed by the array name.
- In the loop body, you can use the loop variable you created rather than using an indexed array element.
- It’s commonly used to iterate over an array or a Collections class (eg, ArrayList)

STRINGS

- Java's string type, called String, is not a primitive type.
- It is in fact a class.
- When you declare a variable of type String, you basically creating an object.
- String class is immutable.
- When you create a String object, you are creating a string that cannot be changed. That is, once a String object has been created, you cannot change the characters that comprise that string.
- Each time you need an altered version of an existing string, a new String object is created that contains the modifications. The original string is left unchanged.

STRING DECLARATIONS

- There are two ways to declare String variables.

```
// literals
String name = "John Smith";
// constructor
String email = new String("john@example.com");
```

STRING LENGTH

- The length of a string is the number of characters that it contains. To obtain this value, call the length( ) method.

```
char chars[] = { 'a', 'b', 'c' };
String s = new String(chars);
System.out.println(s.length());
```

STRING CONCATENATION

- In general, Java does not allow operators to be applied to String objects.
- The one exception to this rule is the + operator, which concatenates two strings, producing a String object as the result.

```
String age = "9";
String s = "He is " + age + " years old.";
System.out.println(s);
```

USING COMMAND-LINE ARGUMENTS

- Sometimes you will want to pass information into a program when you run it.
- This is accomplished by passing command-line arguments to main( ).
- To access the command-line arguments inside a Java program is quite easy - they are stored as strings in a String array passed to the args parameter of main( ).
- The first command-line argument is stored at args[0], the second at args[1], and so on.

```
/*
 * access program arguments and then display to console
 */
System.out.println(args[0]);     // hello
System.out.println(args[1]);     // 1
System.out.println(args[2]);     // 2
System.out.println(args[3]);     // 3
```

USING TYPE INFERENCE WITH LOCAL VARIABLES

- Java 10 introduced a new shiny language feature called local variable type inference.
- Until Java 9, we had to mention the type of the local variable explicitly and ensure it was compatible with the initializer used to initialize it:

```
String message = "Good bye, Java 9";
```

- In Java 10, this is how we could declare a local variable:

```
var message = "Hello, Java 10";
```

- Note that this feature is available only for local variables with the initializer.
- It cannot be used for member variables, method parameters, return types, etc. – the initializer is required as without which compiler won't be able to infer the type.

USING TYPE INFERENCE WITH LOCAL VARIABLES

- This enhancement helps in reducing the boilerplate code; for example:

```
Map<Integer, String> map = new HashMap<>();
```

- This can now be rewritten as:

```
var idToNameMap = new HashMap<Integer, String>();
```

- Another thing to note is that var is not a keyword – this ensures backward compatibility for programs using var say, as a function or variable name.
- var is a reserved type name, just like int.
- Finally, note that there is no runtime overhead in using var nor does it make Java a dynamically typed language.
- The type of the variable is still inferred at compile time and cannot be changed later.

ACTIVITY

EXERCISE 8 AND 9

A CLOSE LOOK AT METHODS AND CLASSES

CONTROLLING ACCESS TO CLASS MEMBERS

- Access level modifiers determine whether other classes can use a particular field or invoke a particular method.
- A class may be declared with the modifier public, in which case that class is visible to all classes everywhere.
- If a class has no modifier (the default, also known as package-private), it is visible only within its own package.
- At the member level, you can also use the public modifier or no modifier (package-private) just as with top-level classes, and with the same meaning.
- For members, there are two additional access modifiers: private and protected.
- The private modifier specifies that the member can only be accessed in its own class.
- The protected modifier specifies that the member can only be accessed within its own package (as with package-private) and, in addition, by a subclass of its class in another package.

ACCESS MODIFIERS

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
default	Y	Y	N	N
private	Y	N	N	N

TIPS ON CHOOSING AN ACCESS LEVEL

- If other programmers use your class, you want to ensure that errors from misuse cannot happen.
- Access levels can help you do this.
  - Use the most restrictive access level that makes sense for a particular member. Use private unless you have a good reason not to.
  - Avoid public fields except for constants.

ENCAPSULATION

- Encapsulation in Java is a mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit.
- In encapsulation, the variables of a class will be hidden from other classes and can be accessed only through the methods of their current class.
- Therefore, it is also known as data hiding.

PROPERLY ENCAPSULATED

- To achieve encapsulation in Java
  - Declare the variables of a class as private.
  - Provide public setter and getter methods to modify and view the variables values.
- The public setXXX() and getXXX() methods are the access points of the instance variables of the EncapTest class.
- Normally, these methods are referred as getters and setters.

```
public class EncapTest {
   private String name;
   private String idNum;
   private int age;

   public int getAge() {
      return age;
   }

   public String getName() {
      return name;
   }

   public String getIdNum() {
      return idNum;
   }

   public void setAge( int newAge) {
      age = newAge;
   }

   public void setName(String newName) {
      name = newName;
   }

   public void setIdNum( String newId) {
      idNum = newId;
   }
}
```

PASS OBJECTS TO METHODS

- Java is strictly pass-by-value.
- Object references can be parameters.
- Call by value is used, but now the value is an object reference.
- This reference can be used to access the object and possibly change it.

```
public void moveCircle(Circle circle, int deltaX, int deltaY) {
    // code to move origin of circle to x+deltaX, y+deltaY
    circle.setX(circle.getX() + deltaX);
    circle.setY(circle.getY() + deltaY);

    // code to assign a new reference to circle
    circle = new Circle(0, 0);
}
```

```
moveCircle(myCircle, 23, 56)
```

RETURNING OBJECTS

```
// 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);
	}
}
```

METHOD OVERLOADING

- 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 this is the case, the methods are said to be overloaded, and the process is referred to as method overloading.
- Method overloading is one of the ways that Java supports polymorphism.
- 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.
- Thus, overloaded methods must differ in the type and/or number of their parameters.

METHOD OVERLOADING

```
// Demonstrate method overloading.
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;
	}
}
```

OVERLOADING CONSTRUCTORS

- In addition to overloading normal methods, you can also overload constructor methods.
- In fact, for most real-world classes that you create, overloaded constructors will be the norm, not the exception.

```
// constructor used when all dimensions specified
Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
}

// constructor used when no dimensions specified
Box() {
        width = -1; // use -1 to indicate
        height = -1; // an uninitialized
        depth = -1; // box
}

// constructor used when cube is created
Box(double len) {
        width = height = depth = len;
}
```

RECURSION

- Java supports recursion. Recursion is the process of defining something in terms of itself.
- As it relates to Java programming, recursion is the attribute that allows a method to call itself.
- A method that calls itself is said to be recursive.

FACTORIAL

```
// A simple example of recursion.
class Factorial {
// this is a recursive method
	int fact(int n) {
		int result;
		if (n == 1)
			return 1;
		result = fact(n - 1) * n;
		return result;
	}
}

class Recursion {
	public static void main(String args[]) {
		Factorial f = new Factorial();
		System.out.println("Factorial of 3 is " + f.fact(3));
		System.out.println("Factorial of 4 is " + f.fact(4));
		System.out.println("Factorial of 5 is " + f.fact(5));
	}
}
```

UNDERSTANDING STATIC

- There will be times when you will want to define a class member that will be used independently of any object of that class.
- Normally, a class member must be accessed only in conjunction with an object of its class.
- However, it is possible to create a member that can be used by itself, without reference to a specific instance.
- To create such a member, precede its declaration with the keyword 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.
- You can declare both methods and variables to be static.

INTRODUCING NESTED AND INNER CLASSES

- It is possible to define a class within another class; such classes are known as nested classes.
- The scope of a nested class is bounded by the scope of its enclosing class. Thus, if class B is defined within class A, then B does not exist independently of A.
- A nested class has access to the members, including private members, of the class in which it is nested. However, the enclosing class does not have access to the members of the nested class.
- A nested class that is declared directly within its enclosing class scope is a member of its enclosing class.
- It is also possible to declare a nested class that is local to a block.

TYPES OF INNER CLASSES

There are two types of nested classes: static and non-static.

- A static nested class is one that has the static modifier applied.
- Because it is static, it must access the non-static members of its enclosing class through an object.
- That is, it cannot refer to non-static members of its enclosing class directly.
- Because of this restriction, static nested classes are seldom used.

INNER CLASS

```
// Demonstrate an inner class.
class Outer {
	int outer_x = 100;

	void test() {
		Inner inner = new Inner();
		inner.display();
	}

// this is an inner class
	class Inner {
		void display() {
			System.out.println("display: outer_x = " + outer_x);
		}
	}
}

class InnerClassDemo {
	public static void main(String args[]) {
		Outer outer = new Outer();
		outer.test();
	}
}
```

VARARGS: VARIABLE-LENGTH ARGUMENTS

- 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.

```java
// Use an array to pass a variable number of
// arguments to a method. This is the old-style
// approach to variable-length arguments.
class PassArray {
    static void vaTest(int v[]) {
            System.out.print("Number of args: " + v.length + " Contents: ");
            for (int x : v)
                    System.out.print(x + " ");
            System.out.println();
    }

    public static void main(String args[]) {
            // Notice how an array must be created to
            // hold the arguments.
            int n1[] = { 10 };
            int n2[] = { 1, 2, 3 };
            int n3[] = {};
            vaTest(n1); // 1 arg
            vaTest(n2); // 3 args
            vaTest(n3); // no args
    }
}
```

ACTIVITY

EXERCISE 10 AND 11

INHERITANCE

INHERITANCE BASICS

- 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 terminology of Java, a class that is inherited is called a superclass.
- The class that does the inheriting is called a subclass.
- Therefore, a subclass is a specialized version of a superclass.
- It inherits all of the members defined by the superclass and adds its own, unique elements.

```java
// A simple example of inheritance.
// Create a superclass.
class A {
	int i, j;

	void showij() {
		System.out.println("i and j: " + i + " " + j);
	}
}

// Create a subclass by extending class A.
class B extends A {
	int k;

	void showk() {
		System.out.println("k: " + k);
	}

	void sum() {
		System.out.println("i+j+k: " + (i + j + k));
	}
}
```

MEMBER ACCESS AND INHERITANCE

- Although a subclass includes all of the members of its superclass, it cannot access those members of the superclass that have been declared as private.
- Superclass also have no access to its subclasses' members.

```
// Create a superclass.
class A {
	int i; // public by default
	private int j; // private to A

	void setij(int x, int y) {
		i = x;
		j = y;
	}
}
```

```
// A's j is not accessible here.
class B extends A {
	int total;

	void sum() {
		total = i + j; // ERROR, j is not accessible here
	}
}

class Access {
	public static void main(String args[]) {
		B subOb = new B();
		subOb.setij(10, 12);
		subOb.sum();
		System.out.println("Total is " + subOb.total);
	}
}
```

CONSTRUCTORS AND INHERITANCE

- Although a subclass inherits all of the methods and variables from superclass, it does not inherit constructors.
- Subclasses need to create its own constructor and add a call to the superclass's constructor.

```
public class Manager extends Employee {

	private String department;

	public Manager(String name, double salary, String department) {
		super(name, salary);
		this.department = department;
	}

	public Manager(String name, String department) {
		super(name);
		this.department = department;
	}
}
```

USING SUPER TO CALL SUPERCLASS CONSTRUCTORS

- A superclass's constructor is always called in addition to the a subclass's constructor.
- The call super() can take any number of arguments appropriate to the various constructors available in the parent class, but it must be the first statement in the constructor.
- Example as above.

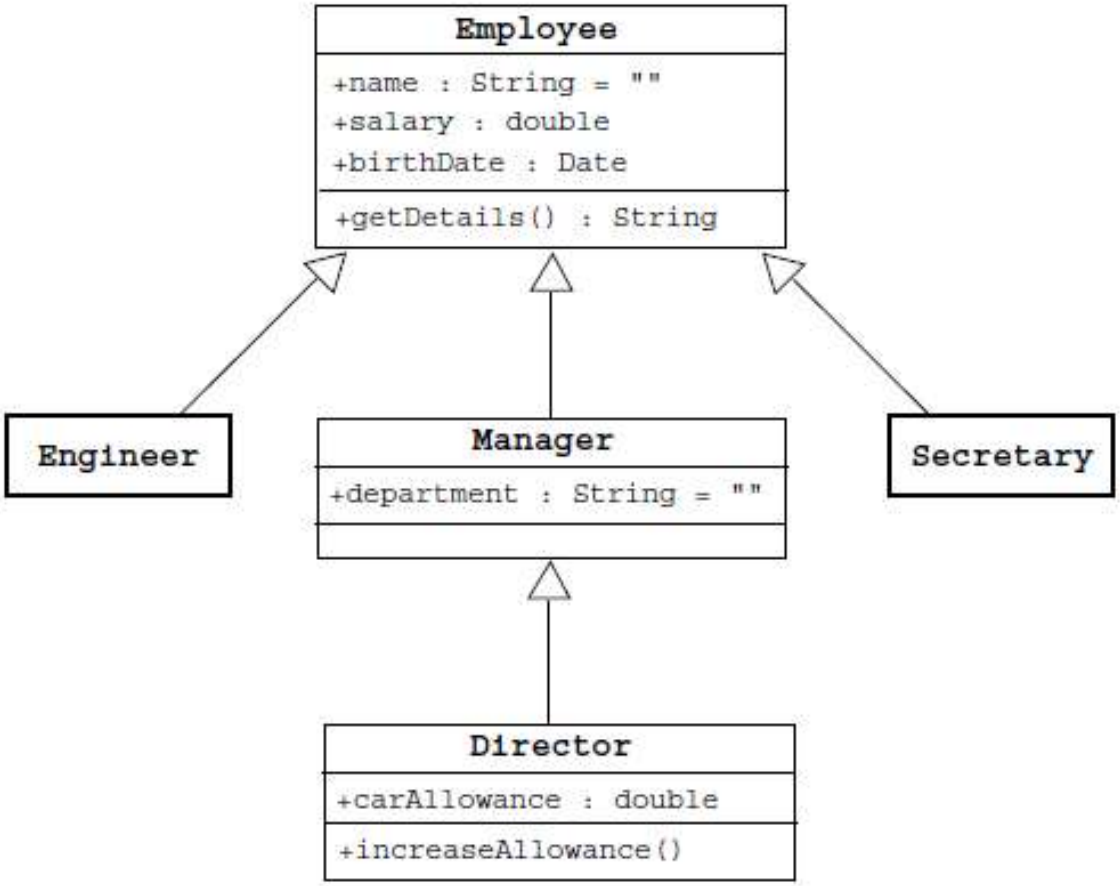
USING SUPER TO ACCESS SUPERCLASS MEMBERS

- The second form of super acts somewhat like this, except that it always 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.

```
// toString method in Employee class
@Override
public String toString() {
	return "Name: " + this.name +
			"\nSalary: " + this.salary +
			"\nBirthdate: " + this.birthDate;
}


// toString method in Manager class
@Override
public String toString() {
	return super.toString() + "\nDepartment: " + this.department;
}
```

CREATING A MULTILEVEL HIERARCHY

- Up to this point, we have been using simple class hierarchies that consist of only a superclass and a subclass.
- However, you can build hierarchies that contain as many layers of inheritance as you like.
- As mentioned, it is perfectly acceptable to use a subclass as a superclass of another.


Employee
+name : String = ""
+salary : double
+birthDate : Date
+getDetails() : String
Engineer
Manager
+department : String = ""
Secretary
Director
+carAllowance : double
+increaseAllowance()

WHEN ARE CONSTRUCTORS EXECUTED?

- When a class hierarchy is created, in what order are the constructors for the classes that make up the hierarchy executed?
- For example, given a subclass called B and a superclass called A, is A's constructor executed before B's, or vice versa?
- The answer is that in a class hierarchy, constructors complete their execution in order of derivation, from superclass to subclass.
- Further, since super( ) must be the first statement executed in a subclass' constructor, this order is the same whether or not super( ) is used.
- If super( ) is not used, then the default or parameterless constructor of each superclass will be executed.

```java
// Demonstrate when constructors are executed.
// Create a super class.
class A {
	A() {
		System.out.println("Inside A's constructor.");
	}
}// Create a subclass by extending class A.

class B extends A {
	B() {
		System.out.println("Inside B's constructor.");
	}
}

// Create another subclass by extending B.
class C extends B {
	C() {
		System.out.println("Inside C's constructor.");
	}
}

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

METHOD OVERRIDING

- In addition to producing a new class based on an old one by additional features, you can modify existing behaviour of the parent class.
- If a method is defined in a subclass so that the name, return type, and argument list match exactly those of a method in the parent class, then the new method is said to override the old one.

```
// getDetails method in Employee class
public String getDetails() {
	return "Name: " + this.name +
			"\nSalary: " + this.salary +
			"\nBirthdate: " + this.birthDate;
}


// getDetails method in Manager class
public String getDetails() {
	return super.toString() + "\nDepartment: " + this.department;
}
```

OVERRIDDEN METHODS SUPPORT POLYMORPHISM

- While the examples in the preceding section demonstrate the mechanics of method overriding, they do not show its power.
- Indeed, if there were nothing more to method overriding than a name space convention, then it would be, at best, an interesting curiosity, but of little real value. However, this is not the case.
- Method overriding forms the basis for one of Java's most powerful concepts: dynamic method dispatch.
- Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.
- Dynamic method dispatch is important because this is how Java implements run-time polymorphism.

POLYMORPHISM

- An object has only one form (the one that is given to it when constructed).
- However, a variable is polymorphic because it can refer to objects of different forms.
- Java permits you to refer to an object with a variable that is one of the parent class type.

```
Employee e = new Manager("John", "IT");// legal
e.getDetails();
```

WHY OVERRIDE METHODS?

- As stated earlier, overridden methods allow Java to support run-time polymorphism.
- Polymorphism is essential to object-oriented programming for one reason: it 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 are another way that Java implements the “one interface, multiple methods” aspect of polymorphism.

USING ABSTRACT METHODS

- Abstract methods are those defined with abstract keyword at its method signature.

  abstract *type name*(*parameter-list*);

- Abstract methods mean there is no implementation provided, that is not method body.

  abstract void callme();

There are situations in which you will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method. That is, sometimes you will want to create a superclass that only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details.

USING ABSTRACT CLASSES

- Any class that contains one or more abstract methods must also be declared abstract.
- To declare a class abstract, you simply use the abstract keyword in front of the class keyword at the beginning of the class declaration.
- There can be no objects of an abstract class. That is, an abstract class cannot be directly instantiated with the new operator.
- Any subclass of an abstract class must either implement all of the abstract methods in the superclass, or be declared abstract itself.

```java
//A Simple demonstration of abstract.
abstract class A {
	abstract void callme();

	//concrete methods are still allowed in abstract classes
	void callmetoo() {
		System.out.println("This is a concrete method.");
	}
}

class B extends A {
	void callme() {
		System.out.println("B's implementation of callme.");
	}
}
```

USING FINAL WITH INHERITANCE

The keyword final has three uses. First, it can be used to create the equivalent of a named constant. This use was described in the preceding chapter. The other two uses of final apply to inheritance.

- Using final to prevent overriding.
- Using final to prevent inheritance.

THE OBJECT CLASS

- There is one special class, Object, defined by Java.
- All other classes are subclasses of Object.
- That is, Object is a superclass of all other classes.
- This means that a reference variable of type Object can refer to an object of any other class.
- Object defines the following methods, which means that they are available in every object.

ACTIVITY

EXERCISE 12 AND 13

PACKAGES AND INTERFACES

PACKAGES

- Packages are containers for classes.
- Packages are used to keep the class name space compartmentalized.
- Packages are stored in a hierarchical manner and are explicitly imported into new class definitions.
- The package is both a naming and a visibility control mechanism.
  - Define classes inside a package that are not accessible by code outside that package.
  - Define class members that are exposed only to other members of the same package.

```
package <top_pkg_name>[.<sub_pkg_name>];
```

PACKAGES AND MEMBER ACCESS

- Packages act as containers for classes and other subordinate packages.
- Classes act as containers for data and code.
- Java addresses four categories of visibility for class members.
  - Subclasses in the same package.
  - Non-subclasses in the same package.
  - Subclasses in different packages.
  - Classes that are neither in the same package nor subclasses.

UNDERSTANDING PROTECTED MEMBERS

	Private	Default	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	yes
Different package non-subclass	No	No	No	Yes

IMPORTING PACKAGES

- Java includes the import statement to bring certain classes, or entire packages, into visibility.
- In a Java source file, import statement occur immediately after the package statement, and before any class definitions.
- The import statement syntax.

    import *pkg1* [.*pkg2*].(*classname* | *);

JAVA'S CLASS LIBRARY IS CONTAINED IN PACKAGES

- All the standard Java classes included with Java are stored in a package called java.
- The basic language functions are stored in a package inside of the java package called java.lang.
- The java.lang package is imported implicitly by the compiler for all programs.

INTERFACES

- Using the interface keyword, Java allows you to fully abstract a class's interface from its implementation.
- Interfaces are syntactically like classes, but they lack instance variables, and as a rule, their methods are declared without any body.
- Once it is defined, any number of classes can implement an interface.
- Also, one class can implement any number of interfaces.

DEFINING AN INTERFACE

```
access interface name {
    return-type method-name1(parameter-list);
    return-type method-name2(parameter-list);
    type final-varname1 = value;
    type final-varname2 = value;
    //...
    return-type method-nameN(parameter-list);
    type final-varnameN = value;
}
```

IMPLEMENTING INTERFACES

- To implement an interface, include the implements clause in a class definition, and then create the methods required by the interface.

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

- If a class implements more than one interface, the interfaces are separated with a comma.

USING INTERFACES REFERENCES

- You can declare variables as object references that use an interface rather than a class type.
- Any instance of any class that implements the declared interface can be referred to by such a 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.
- The method to be executed is looked up dynamically at run time.

VARIABLES IN INTERFACES

- Variables declared in an interface are constants that can be shared by multiple classes.

```
interface SharedConstants {
    int NO = 0;
    int YES = 1;
    int MAYBE = 2;
    int LATER = 3;
    int SOON = 4;
    int NEVER = 5;
}
```

INTERFACES CAN BE EXTENDED

- One 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 required by the interface inheritance chain.

```java
interface FirstInterface {
	public void first();
}

interface SecondInterface extends FirstInterface {
	public void second();
}

class MyClass implements SecondInterface {

	@Override
	public void first() {
		System.out.println("This is the first method");
	}

	@Override
	public void second() {
		System.out.println("This is the second method");
	}

}
```

DEFAULT INTERFACE METHODS

- Since JDK 8, Java added a new capability to interface called the default method.
- A default method lets you define a default implementation for an interface method.
- A primary motivation for the default method was to provide a means by which interfaces could be expanded without breaking existing code.
- The declaration is preceded by the keyword default.

```java
public interface MyIF {
	// This is a "normal" interface method declaration.
	// It does NOT define a default implementation.
	int getNumber();

	// This is a default method. Notice that it provides
	// a default implementation.
	default String getString() {
		return "Default String";
	}
}
```

USE STATIC METHODS IN AN INTERFACE

- JDK 8 added another new capability to interface; the ability to define one or more static methods.
- Like static methods in a class, a static method defined by an interface can be called independently of any object.
- A static method is called by prefixing the interface name.

PRIVATE INTERFACE METHODS

- Java 9 onward, you are allowed to include private methods in interfaces.
- Using private methods, now encapsulation is possible in interfaces as well.
- These private methods will improve code re-usability inside interfaces.
- For example, if two default methods needed to share code, a private interface method would allow them to do so, but without exposing that private method to its implementing classes.

PRIVATE INTERFACE METHODS

- Using private methods in interfaces have four rules:
  - Private interface method cannot be abstract.
  - Private method can be used only inside interface.
  - Private static method can be used inside other static and non-static interface methods.
  - Private non-static methods cannot be used inside private static methods.

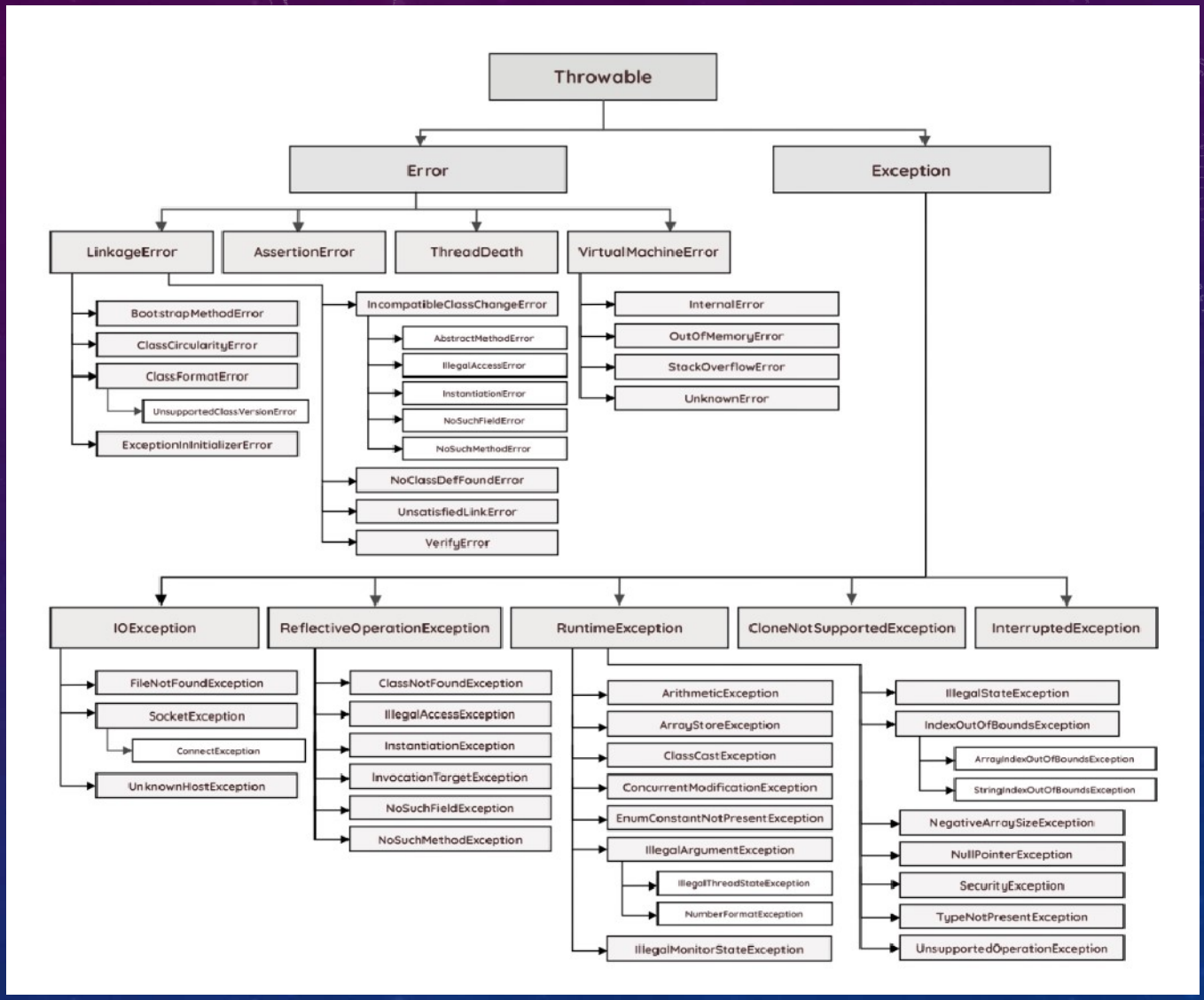
```java
interface MyInterface {
	default void first () {
		System.out.println();
		second();
	}

	private void second() {
		System.out.println();
	}
}
```

ACTIVITY

EXCEPTION HANDLING

EXCEPTION HANDLING

- Exceptions are a mechanism used by many programming languages to describe what to do when something unexpected happens.
- Typically, something unexpected is an error of some sort, for example when a method is invoked with unacceptable arguments, or a network connection fails, or the use asks to open a non-existent file.


Throwable
Error
Exception
LinkageError
AssertionError
ThreadDeath
VirtualMachineError
BootstrapMethodError
ClassCircularityError
ClassFormatError
UnsupportedClassVersionError
ExceptionInInitializerError
IncompatibleClassChangeError
AbstractMethodError
IllegalAccessError
InstantiationError
NoSuchFieldError
NoSuchMethodError
NoClassDefFoundError
UnsatisfiedLinkError
VerifyError
InternalError
OutOfMemoryError
StackOverflowError
UnknownError
IOException
ReflectiveOperationException
RuntimeException
CloneNotSupportedException
InterruptedException
FileNotFoundException
SocketException
ConnectException
UnknownHostException
ClassNotFoundException
IllegalAccessException
InstantiationException
InvocationTargetException
NoSuchFieldException
NoSuchMethodException
ArithmeticException
ArrayStoreException
ClassCastException
ConcurrentModificationException
EnumConstantNotPresentException
IllegalArgumentException
IllegalThreadStateException
NumberFormatException
IllegalMonitorStateException
IllegalStateException
IndexOutOfBoundsException
ArrayIndexOutOfBoundsException
StringIndexOutOfBoundsException
NegativeArraySizeException
NullPointerException
SecurityException
TypeNotPresentException
UnsupportedOperationException

EXCEPTION HANDLING FUNDAMENTALS

- 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 handle the exception itself or pass it on.
- Either way, at some point, the exception is caught and processed.
- Exceptions can be generated by the Java run-time system, or they can be manually generated by your code.
- Java exception handling is managed via five keywords: try, catch, throw, throws, and finally.

```
try {
        // block of code to monitor for errors
}
catch (ExceptionType1 exOb) {
        // exception handler for ExceptionType1
}
catch (ExceptionType2 exOb) {
        // exception handler for ExceptionType2
}
// ...
finally {
        // block of code to be executed after try block ends
}
```

THE CONSEQUENCES OF AN UNCAUGHT EXCEPTION

- Technically, any thrown exceptions must be handled some where within the program. Any exception that is not caught by your program will ultimately be processed by the default handler. The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates the program.

USING MULTIPLE CATCH STATEMENTS

- In some cases, more than one exception could be raised by a single piece of code.
- To handle this type of situation, you can specify two or more catch clauses, each catching a different type of exception.
- When an exception is thrown, each catch statement is inspected in order, and the first one whose type matches that of the exception is executed.

```java
//Demonstrate multiple catch statements.
class MultipleCatches {
	public static void main(String args[]) {
		try {
			int a = args.length;
			System.out.println("a = " + a);
			int b = 42 / a;
			int c[] = { 1 };
			c[42] = 99;
		} catch (ArithmeticException e) {
			System.out.println("Divide by 0: " + e);
		} catch (ArrayIndexOutOfBoundsException e) {
			System.out.println("Array index oob: " + e);
		}
		System.out.println("After try/catch blocks.");
	}
}
```

CATCHING SUBCLASS EXCEPTIONS

- A catch clause for a superclass will also match any of its subclasses.
- For example, since the superclass of all exceptions is Throwable, to catch all possible exceptions, catch Throwable.
- If you want to catch exceptions of both a superclass type and a subclass type, put the subclass first in the catch sequence. If you don’t, then the superclass catch will also catch all derived classes.
- This rule is self-enforcing because putting the superclass first causes unreachable code to be created, since the subclass catch clause can never execute.
- In Java, unreachable code is an error.

TRY BLOCKS CAN BE NESTED

- The try statement can be nested. That is, a try statement can be inside another try block.
- If an inner try statement does not have a catch handler for a particular exception, the stack in unwound and the next try statement's catch handlers are inspected for a match.
- This continues until one of the catch statements succeed, or until all the nested try statements are exhausted.
- If no catch statement matches, then the Java run-time system (default handler) will handle the exception.

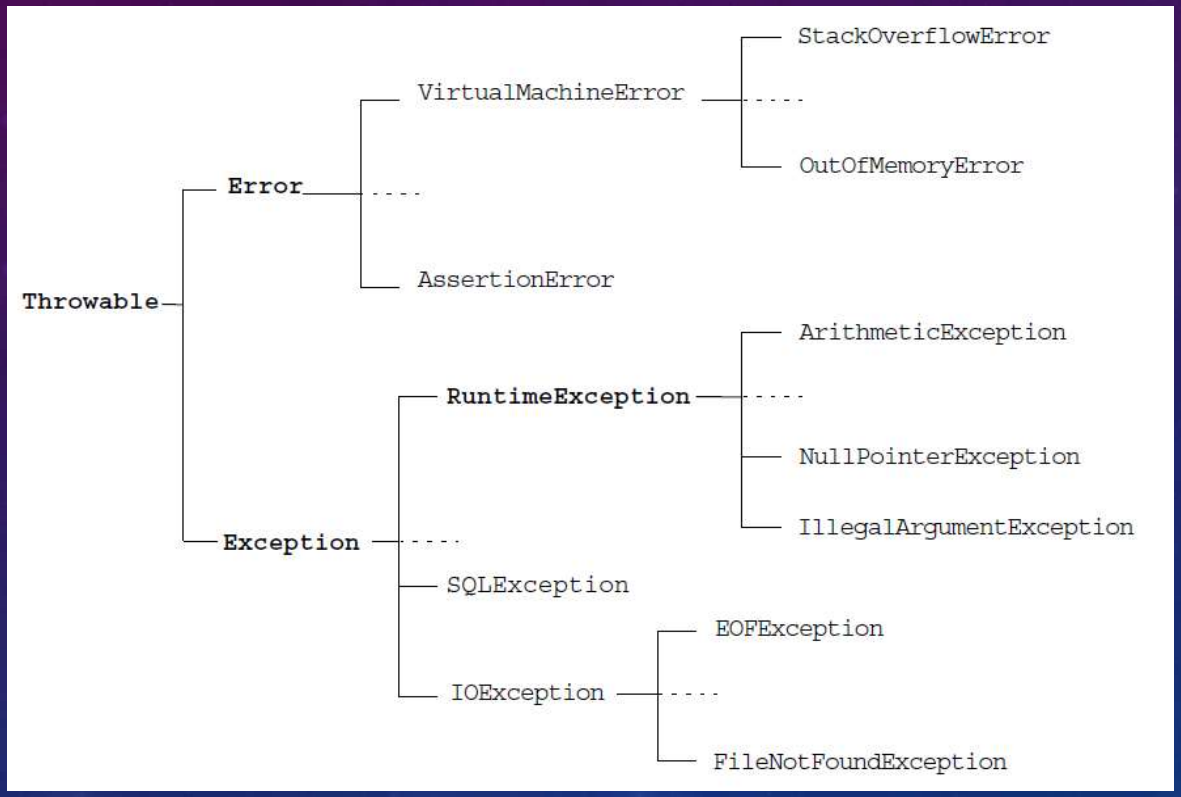
```java
//An example of nested try statements.
class NestTry {
	public static void main(String args[]) {
		try {
			int a = args.length;
			/*
			 * If no command-line args are present, the following
statement will generate a
			 * divide-by-zero exception.
			 */
			int b = 42 / a;
			System.out.println("a = " + a);
			try { // nested try block
				/*
				 * If one command-line arg is used, then a divide-by-
zero exception will be
				 * generated by the following code.
				 */
				if (a == 1)
					a = a / (a - a); // division by zero
				/*
				 * If two command-line args are used, then generate an
out-of-bounds exception.
				 */
				if (a == 2) {
					int c[] = { 1 };
					c[42] = 99; // generate an out-of-bounds
exception
				}
			} catch (ArrayIndexOutOfBoundsException e) {
				System.out.println("Array index out-of-bounds: " + e);
			}
		} catch (ArithmeticException e) {
			System.out.println("Divide by 0: " + e);
		}
	}
}
```

THROWING AN EXCEPTION

- Before you can catch an exception, some code somewhere must throw one.
- Any code can throw an exception: your code, code from a package written by someone else such as the packages that come with the Java platform, or the Java runtime environment.
- Regardless of what throws the exception, it’s always thrown with the throw statement.

A CLOSER LOOK AT THROWABLE

- The class java.lang.Throwable acts as the parent class for all objects that can be thrown and caught using the exception-handling mechanisms.
- Methods defined in the Throwable class retrieve the error message associated with the exception and print the stack trace showing where the exception occurred.
- There are three key subclasses of Throwable: Error, RuntimeException and Exception.


Throwable
Error
VirtualMachineError
StackOverflowError
.....
OutOfMemoryError
....
AssertionError
Exception
RuntimeException
ArithmeticException
.....
NullPointerException
IllegalArgumentException
.....
SQLException
IOException
EOFException
.....
FileNotFoundException

USING THROW

- To throw an exception explicitly, using the throw statement. The general form of throw is shown here:

  throw *ThrowableInstance*;

- The flow of execution stops immediately after the throw statement; any subsequent statements are not executed.
- The nearest enclosing try block is inspected to see if it has a catch statement that matches the type of exception.
- If it does find a match, control is transferred to that statement. If not, then the next enclosing try statement is inspected, and so on. If no matching catch is found, then the default exception handler halts the program and prints the stack trace.

```java
//Demonstrate throw.
class ThrowDemo {
	static void demoproc() {
		try {
			throw new NullPointerException("demo");
		} catch (NullPointerException e) {
			System.out.println("Caught inside demoproc.");
			throw e; // rethrow the exception
		}
	}

	public static void main(String args[]) {
		try {
			demoproc();
		} catch (NullPointerException e) {
			System.out.println("Recaught: " + e);
		}
	}
}
```

USING THROWS

- If a method can cause an exception that it does not handle, it must specify this behaviour so that callers of the method can guard themselves against that exception.
- You do this by including a throws clause in the method's declaration.
- A throws clause lists the types of exceptions that a method might throw.

```java
//This is now correct.
class ThrowsDemo {
	static void throwOne() throws IllegalAccessException {
		System.out.println("Inside throwOne.");
		throw new IllegalAccessException("demo");
	}

	public static void main(String args[]) {
		try {
			throwOne();
		} catch (IllegalAccessException e) {
			System.out.println("Caught " + e);
		}
	}
}
```

USING FINALLY

- finally creates a block of code that will be executed after a try /catch block has completed and before the code following the try/catch block.
- The finally block will execute whether or not an exception is thrown.
- If an exception is thrown, the finally block will execute even if no catch statement matches the exception.
- This can be useful for closing file handles and freeing up any other resources that might have been allocated at the beginning of a method with the intent of disposing of them before returning.
- The finally clause is optional.

```java
//Demonstrate finally.
class FinallyDemo {
	//Throw an exception out of the method.
	static void procA() {
		try {
			System.out.println("inside procA");
			throw new RuntimeException("demo");
		} finally {
			System.out.println("procA's finally");
		}
	}

	//Return from within a try block.
	static void procB() {
		try {
			System.out.println("inside procB");
			return;
		} finally {
			System.out.println("procB's finally");
		}
	}

	//Execute a try block normally.
	static void procC() {
		try {
			System.out.println("inside procC");
		} finally {
			System.out.println("procC's finally");
		}
	}

	public static void main(String args[]) {
		try {
			procA();
		} catch (Exception e) {
			System.out.println("Exception caught");
		}
		procB();
		procC();
	}
}
```

THREE ADDITIONAL EXCEPTION FEATURES

- Beginning with JDK 7, three interesting and useful features have been added to the exception system.
- The first automates the process of releasing a resource, such as a file, when it is no longer needed.
- The second feature is called multi-catch, and the third is sometimes referred to as final rethrow or more precise rethrow.

THE TRY-WITH-RESOURCES STATEMENT

- The try-with-resources statement is a try statement that declares one or more resources.
- A resource is an object that must be closed after the program is finished with it.
- The try-with-resources statement ensures that each resource is closed at the end of the statement.
- Any object that implements java.lang.AutoCloseable, which includes all objects which implement java.io.Closeable, can be used as a resource.

THE TRY-WITH-RESOURCES STATEMENT

The following example reads the first line from a file. It uses an instance of BufferedReader to read data from the file. BufferedReader is a resource that must be closed after the program is finished with it:

```
static String readFirstLineFromFile(String path) throws IOException {
    try (BufferedReader br =
                   new BufferedReader(new FileReader(path))) {
        return br.readLine();
    }
}
```

HANDLING MORE THAN ONE TYPE OF EXCEPTION

In Java SE 7 and later, a single catch block can handle more than one type of exception. This feature can reduce code duplication and lessen the temptation to catch an overly broad exception.

```
catch (IOException|SQLException ex) {
    logger.log(ex);
    throw ex;
}
```

RETHROWING EXCEPTIONS WITH MORE INCLUSIVE TYPE CHECKING

The Java SE 7 compiler performs more precise analysis of rethrown exceptions than earlier releases of Java SE. This enables you to specify more specific exception types in the throws clause of a method declaration.

```
static class FirstException extends Exception {
}

static class SecondException extends Exception {
}

public void rethrowException(String exceptionName) throws Exception {
  try {
    if (exceptionName.equals("First")) {
      throw new FirstException();
    } else {
      throw new SecondException();
    }
  } catch (Exception e) {
    throw e;
  }
}
```

RETHROWING EXCEPTIONS WITH MORE INCLUSIVE TYPE CHECKING

However, in Java SE 7, you can specify the exception types FirstException and SecondException in the throws clause in the rethrowException method declaration. The Java SE 7 compiler can determine that the exception thrown by the statement throw e must have come from the try block, and the only exceptions thrown by the try block can be FirstException and SecondException. Even though the exception parameter of the catch clause, e, is type Exception, the compiler can determine that it is an instance of either FirstException or SecondException:

```
public void rethrowException(String exceptionName)
        throws FirstException, SecondException {
    try {
        // ...
    }
    catch (Exception e) {
        throw e;
    }
}
```

JAVA'S BUILT-IN EXCEPTIONS

- Inside the standard package java.lang, Java defines several exception classes.
- There are two broad categories of exceptions, known as checked and unchecked exceptions.
- The most general of these unchecked exceptions are subclasses of the standard type RuntimeException. These exceptions need not be included in any method's throws list.
- In the language of Java, these are called unchecked exceptions because the compiler does not check to see if a method handles or throws these exceptions.
- Checked exceptions are those that the programmer is expected to handle in the program, and that arise from external conditions that can readily occur in a working program. Examples would be a requested file not being found or a network failure.

CREATING EXCEPTION SUBCLASSES

```
//This program creates a custom exception type.
class MyException extends Exception {
	private int detail;

	MyException(int a) {
		detail = a;
	}

	public String toString() {
		return "MyException[" + detail + "]";
	}
}

class ExceptionDemo {
	static void compute(int a) throws MyException {
		System.out.println("Called compute(" + a + ")");
		if (a > 10)
			throw new MyException(a);
		System.out.println("Normal exit");
	}

	public static void main(String args[]) {
		try {
			compute(1);
			compute(20);
		} catch (MyException e) {
			System.out.println("Caught " + e);
		}
	}
}
```

ACTIVITY

USING I/O

JAVA'S I/O IS BUILT UPON STREAMS

- Java programs perform I/O through streams.
- A stream is an abstraction that either produces or consumes information.
- A stream is linked to a physical device by the Java I/O system.
- This means that an input stream can abstract many kinds of input: from a disk file, a keyboard, or a network socket.
- Likewise, an output stream may refer to the console, a disk file, or a network connection.
- Java implements streams within class hierarchies defined in the java.io package.

BYTE STREAMS AND CHARACTER STREAMS

- Java defines two types of streams: byte and character.
- Byte streams provide a convenient means for handling input and output of bytes.
- Byte streams are used, for example, when reading or writing binary data.
- Character streams provide a convenient means for handling input and output of characters.
- They use Unicode and, therefore, can be internationalized.
- Also, in some cases, character streams are more efficient than byte streams.

THE BYTE STREAM CLASSES

- Byte streams are defined by using two class hierarchies.
- At the top are two abstract classes: InputStream and OutputStream.
- Each of these abstract classes has several concrete subclasses that handle the differences among various devices, such as disk files, network connections, and even memory buffers.
- The byte stream classes in java.io are shown in the next table.
- The abstract classes InputStream and OutputStream define several key methods that the other stream classes implement.
- Two of the most important are read( ) and write( ), which, respectively, read and write bytes of data.
- Each has a form that is abstract and must be overridden by derived stream classes.

Stream Class	Meaning
BufferedInputStream	Buffered input stream
BufferedOutputStream	Buffered output stream
ByteArrayInputStream	Input stream that reads from a byte array
ByteArrayOutputStream	Output stream that writes to a byte array
DataInputStream	An input stream that contains methods for reading the Java standard data types
DataOutputStream	An output stream that contains methods for writing the Java standard data types
FileInputStream	Input stream that reads from a file
FileOutputStream	Output stream that writes to a file
FilterInputStream	Implements **InputStream**
FilterOutputStream	Implements **OutputStream**
InputStream	Abstract class that describes stream input
ObjectInputStream	Input stream for objects
ObjectOutputStream	Output stream for objects
OutputStream	Abstract class that describes stream output
PipedInputStream	Input pipe
PipedOutputStream	Output pipe
PrintStream	Output stream that contains **print( )** and **println( )**
PushbackInputStream	Input stream that supports one-byte "unget," which returns a byte to the input stream
SequenceInputStream	Input stream that is a combination of two or more input streams that will be read sequentially, one after the other

THE CHARACTER STREAM CLASSES

- Character streams are defined by using two class hierarchies.
- At the top are two abstract classes: Reader and Writer.

Stream Class	Meaning
BufferedReader	Buffered input character stream
BufferedWriter	Buffered output character stream
CharArrayReader	Input stream that reads from a character array
CharArrayWriter	Output stream that writes to a character array
FileReader	Input stream that reads from a file
FileWriter	Output stream that writes to a file
FilterReader	Filtered reader
FilterWriter	Filtered writer
InputStreamReader	Input stream that translates bytes to characters
LineNumberReader	Input stream that counts lines
OutputStreamWriter	Output stream that translates characters to bytes
PipedReader	Input pipe
PipedWriter	Output pipe
PrintWriter	Output stream that contains **print( )** and **println( )**
PushbackReader	Input stream that allows characters to be returned to the input stream
Reader	Abstract class that describes character stream input
StringReader	Input stream that reads from a string
StringWriter	Output stream that writes to a string
Writer	Abstract class that describes character stream output

THE PREDEFINED STREAMS

- All Java programs automatically import the java.lang package.
- This package defines a class called System, which encapsulates several aspects of the run-time environment.
- System also contains three predefined stream variables: in, out, and err.
- These fields are declared as public, static, and final within System.
- This means that they can be used by any other part of your program and without reference to a specific System object.

USING THE BYTE STREAMS

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

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

		FileInputStream in = null;
		FileOutputStream out = null;

		try {
			in = new FileInputStream("xanadu.txt");
			out = new FileOutputStream("outagain.txt");
			int c;

			while ((c = in.read()) != -1) {
				out.write(c);
			}
		} finally {
			if (in != null) {
				in.close();
			}
			if (out != null) {
				out.close();
			}
		}
	}
}
```

READING AND WRITING FILES

```
import java.io.*;

class ShowFile {
	public static void main(String args[]) {
		int i;
		FileInputStream fin;
		// First, confirm that a filename has been specified.
		if (args.length != 1) {
			System.out.println("Usage: ShowFile filename");
			return;
		}
		// Attempt to open the file.
		try {
			fin = new FileInputStream(args[0]);
		} catch (FileNotFoundException e) {
			System.out.println("Cannot Open File");
			return;
		}
		// At this point, the file is open and can be read.
		// The following reads characters until EOF is encountered.
		try {
			do {
				i = fin.read();
				if (i != -1)
					System.out.print((char) i);
			} while (i != -1);
		} catch (IOException e) {
			System.out.println("Error Reading File");
		}
		// Close the file.
		try {
			fin.close();
		} catch (IOException e) {
			System.out.println("Error Closing File");
		}
	}
}
```

AUTOMATICALLY CLOSING A FILE

- JDK 7 added a new feature that offers another way to manage resources, such as file streams, by automating the closing process.
- This feature, sometimes referred to as automatic resource management, or ARM for short, is based on an expanded version of the try statement.
- The principal advantage of automatic resource management is that it prevents situations in which a file (or other resource) is inadvertently not released after it is no longer needed.

```
try (resource-specification) {
    // use the resource
}
```

```java
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;

public class CopyBytes {
	public static void main(String[] args) {
		// The following code uses a try-with-resources statement to open
		// a file and then automatically close it when the try block is
left.
		try (FileInputStream fin = new FileInputStream(args[0])) {
			do {
				i = fin.read();
				if (i != -1)
					System.out.print((char) i);
			} while (i != -1);
		} catch (FileNotFoundException e) {
			System.out.println("File Not Found.");
		} catch (IOException e) {
			System.out.println("An I/O Error Occurred");
		}
	}
}
```

RANDOM-ACCESS FILES

- Random access files permit nonsequential, or random, access to a file's contents. To access a file randomly, you open the file, seek a particular location, and read from or write to that file.
- This functionality is possible with the SeekableByteChannel interface. The SeekableByteChannel interface extends channel I/O with the notion of a current position. Methods enable you to set or query the position, and you can then read the data from, or write the data to, that location. The API consists of a few, easy to use, methods:
  - position – Returns the channel's current position
  - position(long) – Sets the channel's position
  - read(ByteBuffer) – Reads bytes into the buffer from the channel
  - write(ByteBuffer) – Writes bytes from the buffer to the channel
  - truncate(long) – Truncates the file (or other entity) connected to the channel

```
String s = "I was here!\n";
byte data[] = s.getBytes();
ByteBuffer out = ByteBuffer.wrap(data);

ByteBuffer copy = ByteBuffer.allocate(12);

try (FileChannel fc = (FileChannel.open(file, READ, WRITE))) {
        // Read the first 12
        // bytes of the file.
        int nread;
        do {
                nread = fc.read(copy);
        } while (nread != -1 && copy.hasRemaining());

        // Write "I was here!" at the beginning of the file.
        fc.position(0);
        while (out.hasRemaining())
                fc.write(out);
        out.rewind();

        // Move to the end of the file. Copy the first 12 bytes to
        // the end of the file. Then write "I was here!" again.
        long length = fc.size();
        fc.position(length - 1);
        copy.flip();
        while (copy.hasRemaining())
                fc.write(copy);
        while (out.hasRemaining())
                fc.write(out);
} catch (IOException x) {
        System.out.println("I/O Exception: " + x);
}
```

USING JAVA'S CHARACTER-BASED STREAMS

```
// Creating FileReader object
File file = new File("D:/myFile.txt");
FileReader reader;
try {
	reader = new FileReader(file);
	char chars[] = new char[(int) file.length()];
	// Reading data from the file
	reader.read(chars);
	// Writing data to another file
	File out = new File("D:/CopyOfmyFile.txt");
	FileWriter writer = new FileWriter(out);
	// Writing data to the file
	writer.write(chars);
	writer.flush();
} catch (FileNotFoundException e) {
	// TODO Auto-generated catch block
	e.printStackTrace();
} catch (IOException e) {
	// TODO Auto-generated catch block
	e.printStackTrace();
} finally {
	System.out.println("Data successfully written in the specified file");
}
```

USING JAVA'S TYPE WRAPPERS TO CONVERT NUMERIC TO STRINGS

- All classes inherit a method called toString from the Object class. The type-wrapper classes override this method to provide a reasonable string representation of the value held by the number object.
- The following program, ToStringDemo (in a .java source file), uses the toString method to convert a number to a string. Next, the program uses some string methods to compute the number of digits before and after the decimal point:

```
double d = 858.48;
String s = Double.toString(d);

int dot = s.indexOf('.');
System.out.println(s.substring(0, dot).length() + " digits before decimal
point.");
System.out.println(s.substring(dot + 1).length() + " digits after decimal
point.");
```

ACTIVITY

MULTITHREADED PROGRAMMING

MULTITHREADED PROGRAMMING

- Java provides built-in support for multithreaded programming.
- A multithreaded program contains two or more parts that can run concurrently.
- Each part of such a program is called a thread, and each thread defines a separate path of execution.
- Thus, multithreading is a specialized form of multitasking.

PROCESS VS THREAD

- A process is, in essence, a program that is executing.
- Thus, process-based multitasking is the feature that allows your computer to run two or more programs concurrently.
- 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 simultaneously.

THREAD STATES

- A thread can be *running*.
- It can be *ready to run* as soon as it gets CPU time.
- A running thread can be *suspended*, which temporarily halts its activity.
- A suspended thread can then be *resumed*, allowing it to pick up where it left off.
- A thread can be *blocked* when waiting for a resource.
- At any time, a thread can be *terminated*, which halts its execution immediately.
- Once terminated, a thread cannot be resumed.

THE THREAD CLASS AND RUNNABLE INTERFACE

- Java's multithreading system is built upon the Thread class, its methods, and its companion interface, Runnable. Thread encapsulates a thread of execution. Since you can't directly refer to the ethereal state of a running thread, you will deal with it through its proxy, the Thread instance that spawned it.
- 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.

THREAD CLASS

Method	Meaning
getName	Obtain a thread’s name.
getPriority	Obtain a thread’s priority.
isAlive	Determine if a thread is still running.
join	Wait for a thread to terminate.
run	Entry point for the thread.
sleep	Suspend a thread for a period of time.
start	Start a thread by calling its run method.

THE MAIN THREAD

- When a Java program starts up, one thread begins running immediately. This is usually called the main thread of your program because it is the one that is executed when your program begins. The main thread is important for two reasons:
  - It is the thread from which other “child” threads will be spawned.
  - Often, it must be the last thread to finish execution because it performs various shutdown actions.

```java
//Controlling the main Thread.
class CurrentThreadDemo {
	public static void main(String args[]) {
		Thread t = Thread.currentThread();
		System.out.println("Current thread: " + t);
		//change the name of the thread
		t.setName("My Thread");
		System.out.println("After name change: " + t);
		try {
			for (int n = 5; n > 0; n--) {
				System.out.println(n);
				Thread.sleep(1000);
			}
		} catch (InterruptedException e) {
			System.out.println("Main thread interrupted");
		}
	}
}
```

CREATING A THREAD

```
//Create a second thread.
class NewThread implements Runnable {
        Thread t;

        NewThread() {
                //Create a new, second thread
                t = new Thread(this, "Demo Thread");
                System.out.println("Child thread: " + t);
                t.start(); // Start the thread
        }

        //This is the entry point for the second thread.
        public void run() {
                try {
                        for (int i = 5; i > 0; i--) {
                                System.out.println("Child Thread: " + i);
                                Thread.sleep(500);
                        }
                } catch (InterruptedException e) {
                        System.out.println("Child interrupted.");
                }
                System.out.println("Exiting child thread.");
        }
}
```

CREATING A THREAD

```
public class Test {
	public static void main(String args[]) {
		new NewThread(); // create a new thread
		try {
			for (int i = 5; i > 0; i--) {
				System.out.println("Main Thread: " + i);
				Thread.sleep(1000);
			}
		} catch (InterruptedException e) {
			System.out.println("Main thread interrupted.");
		}
		System.out.println("Main thread exiting.");
	}
}
```

EXTENDING THREAD

```
//Create a second thread by extending Thread
class NewThread extends Thread {
	NewThread() {
		// Create a new, second thread
		super("Demo Thread");
		System.out.println("Child thread: " + this);
		start(); // Start the thread
	}

	// This is the entry point for the second thread.
	public void run() {
		try {
			for (int i = 5; i > 0; i--) {
				System.out.println("Child Thread: " + i);
				Thread.sleep(500);
			}
		} catch (InterruptedException e) {
			System.out.println("Child interrupted.");
		}
		System.out.println("Exiting child thread.");
	}
}
```

EXTENDING THREAD

```
public class Test {
	public static void main(String args[]) {
		new NewThread(); // create a new thread
		try {
			for (int i = 5; i > 0; i--) {
				System.out.println("Main Thread: " + i);
				Thread.sleep(1000);
			}
		} catch (InterruptedException e) {
			System.out.println("Main thread interrupted.");
		}
		System.out.println("Main thread exiting.");
	}
}
```

CREATING MULTIPLE THREADS

```
//Create multiple threads.
class NewThread implements Runnable {
	String name; // name of thread
	Thread t;

	NewThread(String threadname) {
		name = threadname;
		t = new Thread(this, name);
		System.out.println("New thread: " + t);
		t.start(); // Start the thread
	}

	// This is the entry point for thread.
	public void run() {
		try {
			for (int i = 5; i > 0; i--) {
				System.out.println(name + ": " + i);
				Thread.sleep(1000);
			}
		} catch (InterruptedException e) {
			System.out.println(name + "Interrupted");
		}
		System.out.println(name + " exiting.");
	}
}
```

CREATING MULTIPLE THREADS

```
public class Test {
	public static void main(String args[]) {
		new NewThread("One"); // start threads
		new NewThread("Two");
		new NewThread("Three");
		try {
			// wait for other threads to end
			Thread.sleep(10000);
		} catch (InterruptedException e) {
			System.out.println("Main thread Interrupted");
		}
		System.out.println("Main thread exiting.");
	}
}
```

DETERMINING WHEN A THREAD ENDS

- Two ways exist to determine whether a thread has finished: isAlive() and join() methods.
  - The isAlive( ) method returns true if the thread upon which it is called is still running. It returns false otherwise.
  - The join() 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.

ISALIVE() AND JOIN()

```
//Using join() to wait for threads to finish.
class NewThread implements Runnable {
	String name; // name of thread
	Thread t;

	NewThread(String threadname) {
		name = threadname;
		t = new Thread(this, name);
		System.out.println("New thread: " + t);
		t.start(); // Start the thread
	}

	//This is the entry point for thread.
	public void run() {
		try {
			for (int i = 5; i > 0; i--) {
				System.out.println(name + ": " + i);
				Thread.sleep(1000);
			}
		} catch (InterruptedException e) {
			System.out.println(name + " interrupted.");
		}
		System.out.println(name + " exiting.");
	}
}
```

ISALIVE() AND JOIN()

```
public class Test {
	public static void main(String args[]) {
		NewThread ob1 = new NewThread("One");
		NewThread ob2 = new NewThread("Two");
		NewThread ob3 = new NewThread("Three");

		System.out.println("Thread One is alive: " + ob1.t.isAlive());
		System.out.println("Thread Two is alive: " + ob2.t.isAlive());
		System.out.println("Thread Three is alive: " + ob3.t.isAlive());
		// wait for threads to finish
		try {
			System.out.println("Waiting for threads to finish.");
			ob1.t.join();
			ob2.t.join();
			ob3.t.join();
		} catch (InterruptedException e) {
			System.out.println("Main thread Interrupted");
		}
		System.out.println("Thread One is alive: " + ob1.t.isAlive());
		System.out.println("Thread Two is alive: " + ob2.t.isAlive());
		System.out.println("Thread Three is alive: " + ob3.t.isAlive());
		System.out.println("Main thread exiting.");
	}
}
```

THREAD PRIORITIES

- Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run. In theory, over a given period of time, higher-priority threads get more CPU time than lower-priority threads.
- To set a thread's priority, use the setPriority( ) method, which is a member of Thread.

  final void setPriority(int *level*)
- 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.

SYNCHRONIZATION

- When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called *synchronization*.

SYNCHRONIZATION

- Key to synchronization is the concept of the monitor.
- A monitor is an object that is used as a mutually exclusive lock.
- Only one thread can own a monitor at a given time.
- When a thread acquires a lock, it is said to have entered the monitor.
- All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor.
- These other threads are said to be waiting for the monitor.
- A thread that owns a monitor can reenter the same monitor if it so desires.

USING SYNCHRONIZED STATEMENT

```
//This program uses a synchronized block.
class Callme {
	void call(String msg) {
		System.out.print("[" + msg);
		try {
			Thread.sleep(1000);
		} catch (InterruptedException e) {
			System.out.println("Interrupted");
		}
		System.out.println("]");
	}
}

class Caller implements Runnable {
	String msg;
	Callme target;
	Thread t;

	public Caller(Callme targ, String s) {
		target = targ;
		msg = s;
		t = new Thread(this);
		t.start();
	}

	// synchronize calls to call()
	public void run() {
		synchronized (target) { // synchronized block
			target.call(msg);
		}
	}
}
```

```
public class Test {
	public static void main(String args[]) {
		Callme target = new Callme();
		Caller ob1 = new Caller(target, "Hello");
		Caller ob2 = new Caller(target, "Synchronized");
		Caller ob3 = new Caller(target, "World");
		// wait for threads to end
		try {
			ob1.t.join();
			ob2.t.join();
			ob3.t.join();
		} catch (InterruptedException e) {
			System.out.println("Interrupted");
		}
	}
}
```

THREAD COMMUNICATION USING NOTIFY(), WAIT(), AND NOTIFYALL()

- To avoid polling, Java includes an elegant interprocess communication mechanism via the wait( ), notify( ), and notifyAll( ) methods. These methods are implemented as final methods in Object, so all classes have them. All three methods can be called only from within a synchronized context.
  - wait( ) tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls notify( ) or notifyAll( ).
  - notify( ) wakes up a thread that called wait( ) on the same object.
  - notifyAll( ) wakes up all the threads that called wait( ) on the same object. One of the threads will be granted access.

```
//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();
		}
	}
}
```

```
public class Test {
	public static void main(String args[]) {
		Q q = new Q();
		new Producer(q);
		new Consumer(q);
		System.out.println("Press Control-C to stop.");
	}
}
```

SUSPENDING, RESUMING, AND STOPPING THREADS

- 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. However, all these methods are **deprecated** by Java 2, since these methods can sometimes cause serious system failures.
- Instead, a thread must be designed so that the run( ) method periodically checks to determine whether that thread should suspend, resume, or stop its own execution. Typically, this is accomplished by establishing a flag variable that indicates the execution state of the thread. As long as this 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. If it is set to “stop,” the thread must terminate.

```java
//Suspending and resuming a thread the modern way.
class NewThread implements Runnable {
	String name; // name of thread
	Thread t;
	boolean suspendFlag;

	NewThread(String threadname) {
		name = threadname;
		t = new Thread(this, name);
		System.out.println("New thread: " + t);
		suspendFlag = false;
		t.start(); // Start the thread
	}

	//This is the entry point for thread.
	public void run() {
		try {
			for (int i = 15; i > 0; i--) {
				System.out.println(name + ": " + i);
				Thread.sleep(200);
				synchronized (this) {
					while (suspendFlag) {
						wait();
					}
				}
			}
		} catch (InterruptedException e) {
			System.out.println(name + " interrupted.");
		}
		System.out.println(name + " exiting.");
	}

	synchronized void mysuspend() {
		suspendFlag = true;
	}

	synchronized void myresume() {
		suspendFlag = false;
		notify();
	}
}
```

```
public class Test {
	public static void main(String args[]) {
		NewThread ob1 = new NewThread("One");
		NewThread ob2 = new NewThread("Two");
		try {
			Thread.sleep(1000);
			ob1.mysuspend();
			System.out.println("Suspending thread One");
			Thread.sleep(1000);
			ob1.myresume();
			System.out.println("Resuming thread One");
			ob2.mysuspend();
			System.out.println("Suspending thread Two");
			Thread.sleep(1000);
			ob2.myresume();
			System.out.println("Resuming thread Two");
		} catch (InterruptedException e) {
			System.out.println("Main thread Interrupted");
		}
		// wait for threads to finish
		try {
			System.out.println("Waiting for threads to finish.");
			ob1.t.join();
			ob2.t.join();
		} catch (InterruptedException e) {
			System.out.println("Main thread Interrupted");
		}
		System.out.println("Main thread exiting.");
	}
}
```

ACTIVITY

ENUMERATIONS, AUTOBOXING AND STATIC IMPORT

ENUMERATIONS

- Enumerations was added since JDK 5.
- An enumeration is a list of named constants.
- Java enumeration defines a class type.
- Java enumeration can have constructors, methods and instance variables.
- An enumeration is created using the enum keyword.

```
//An enumeration of apple varieties.
enum Apple {
        Jonathan, GoldenDel, RedDel, Winesap, Cortland
}
```

```
class EnumDemo {
	public static void main(String args[]) {
		Apple ap;
		ap = Apple.RedDel;
		// Output an enum value.
		System.out.println("Value of ap: " + ap);
		System.out.println();
		ap = Apple.GoldenDel;
		// Compare two enum values.
		if (ap == Apple.GoldenDel)
			System.out.println("ap contains GoldenDel.\n");
		// Use an enum to control a switch statement.
		switch (ap) {
		case Jonathan:
			System.out.println("Jonathan is red.");
			break;
		case GoldenDel:
			System.out.println("Golden Delicious is yellow.");
			break;
		case RedDel:
			System.out.println("Red Delicious is red.");
			break;
		case Winesap:
			System.out.println("Winesap is red.");
			break;
		case Cortland:
			System.out.println("Cortland is red.");
			break;
		}
	}
}
```

THE VALUES() AND VALUEOF() METHODS

- The values( ) method returns an array that contains a list of the enumeration constants.
- The valueOf( ) method returns the enumeration constant whose value corresponds to the string passed in str.
- In both cases, enum-type is the type of the enumeration.

```
class EnumDemo2 {
	public static void main(String args[]) {
		Apple ap;
		System.out.println("Here are all Apple constants:");
		// use values()
		Apple allapples[] = Apple.values();
		for (Apple a : allapples)
			System.out.println(a);
		System.out.println();
		// use valueOf()
		ap = Apple.valueOf("Winesap");
		System.out.println("ap contains " + ap);
	}
}
```

JAVA ENUMERATIONS ARE CLASS TYPES

- Java enumeration is a class type.
- Java enum can have constructors, add instance variables and methods, and even implement interfaces.
- However, no object is created as an instance of an enum.
- Instead, the constructor of an enum is called when each enumeration constant is created.

```
//Use an enum constructor, instance variable, and method.
enum Apple {
	Jonathan(10), GoldenDel(9), RedDel(12), Winesap(15), Cortland(8);

	private int price; // price of each apple

	// Constructor
	Apple(int p) {
		price = p;
	}

	int getPrice() {
		return price;
	}
}

public class Test {
	public static void main(String args[]) {
		Apple ap;
		// Display price of Winesap.
		System.out.println("Winesap costs " + Apple.Winesap.getPrice() + "
cents.\n");
		// Display all apples and prices.
		System.out.println("All apple prices:");
		for (Apple a : Apple.values())
			System.out.println(a + " costs " + a.getPrice() + " cents.");
	}
}
```

ENUMERATIONS INHERIT ENUM

- All enumerations automatically inherit one: java.lang.Enum
- This class defines several methods that are available for use by all enumerations.
- Some methods that might be useful:
  - ordinal( ) : returns a value that indicates an enumeration constant's position in the list of constants
  - compareTo( ) : compare the ordinal value of two constants of the same enumeration
  - equals( ) : compare for equality an enumeration constant with any other object

```java
//Demonstrate ordinal(), compareTo(), and equals().
//An enumeration of apple varieties.
enum Apple {
	Jonathan, GoldenDel, RedDel, Winesap, Cortland
}

public class Test {
	public static void main(String args[]) {
		Apple ap, ap2, ap3;
		// Obtain all ordinal values using ordinal().
		System.out.println("Here are all apple constants" + " and their
ordinal values: ");
		for (Apple a : Apple.values())
			System.out.println(a + " " + a.ordinal());
		ap = Apple.RedDel;
		ap2 = Apple.GoldenDel;
		ap3 = Apple.RedDel;
		System.out.println();
		// Demonstrate compareTo() and equals()
		if (ap.compareTo(ap2) < 0)
			System.out.println(ap + " comes before " + ap2);
		if (ap.compareTo(ap2) > 0)
			System.out.println(ap2 + " comes before " + ap);
		if (ap.compareTo(ap3) == 0)
			System.out.println(ap + " equals " + ap3);
		System.out.println();
		if (ap.equals(ap2))
			System.out.println("Error!");
		if (ap.equals(ap3))
			System.out.println(ap + " equals " + ap3);
		if (ap == ap3)
			System.out.println(ap + " == " + ap3);
	}
}
```

TYPE WRAPPERS

- The type wrappers are Double, Float, Long, Integer, Short, Byte, Character, and Boolean.
- These classes offer a wide array of methods that allow you to fully integrate the primitive types into Java's object hierarchy.

CHARACTER

- Character is a wrapper around a char. The constructor for Character is

  Character(char *ch*)

- Here, ch specifies the character that will be wrapped by the Character object being created.
- To obtain the char value contained in a Character object, call charValue( ), shown here:

  char charValue( )

- It returns the encapsulated character.

BOOLEAN

- Boolean is a wrapper around boolean values. It defines these constructors:

  Boolean(boolean *boolValue*)

  Boolean(String *boolString*)

- In the first version, boolValue must be either true or false. In the second version, if boolString contains the string "true" (in uppercase or lowercase), then the new Boolean object will be true. Otherwise, it will be false.
- To obtain a boolean value from a Boolean object, use booleanValue( ), shown here:

  boolean booleanValue( )

- It returns the boolean equivalent of the invoking object.

NUMERIC TYPE WRAPPERS

- These are Byte, Short, Integer, Long, Float, and Double. All of the numeric type wrappers inherit the abstract class Number. Number declares methods that return the value of an object in each of the different number formats. These methods are shown here:

```
byte byteValue( )
double doubleValue( )
float floatValue( )
int intValue( )
long longValue( )
short shortValue( )
```

AUTOBOXING

- Beginning with JDK 5, Java added two important features: autoboxing and auto-unboxing.
- Autoboxing is the process by which a primitive type is automatically encapsulated (boxed) into its equivalent type wrapper whenever an object of that type is needed. There is no need to explicitly construct an object.
- Auto-unboxing is the process by which the value of a boxed object is automatically extracted (unboxed) from a type wrapper when its value is needed. There is no need to call a method such as intValue( ) or doubleValue( ).

```
Integer iOb = 100; // autobox an int
int i = iOb; // auto-unbox
```

AUTOBOXING AND METHODS

- In addition to the simple case of assignments, autoboxing automatically occurs whenever a primitive type must be converted into an object; auto-unboxing takes place whenever an object must be converted into a primitive type.
- Thus, autoboxing/unboxing might occur when an argument is passed to a method, or when a value is returned by a method.

```java
//Autoboxing/unboxing takes place with
//method parameters and return values.
public class Test {

	// Take an Integer parameter and return
	// an int value;
	static int m(Integer v) {
		return v; // auto-unbox to int
	}

	public static void main(String args[]) {
		// Pass an int to m() and assign the return value
		// to an Integer. Here, the argument 100 is autoboxed
		// into an Integer. The return value is also autoboxed
		// into an Integer.
		Integer iOb = m(100);
		System.out.println(iOb);
	}
}

//This program displays the following result:
//100
```

AUTOBOXING/UNBOXING OCCURS IN EXPRESSIONS

- In general, autoboxing and unboxing take place whenever a conversion into an object or from an object is required.
- This applies to expressions.
- Within an expression, a numeric object is automatically unboxed.
- The outcome of the expression is reboxed, if necessary.

```java
//Autoboxing/unboxing occurs inside expressions.
public class Test {

	public static void main(String args[]) {
		Integer iOb, iOb2;
		int i;
		iOb = 100;
		System.out.println("Original value of iOb: " + iOb);
		// The following automatically unboxes iOb,
		// performs the increment, and then reboxes
		// the result back into iOb.
		++iOb;
		System.out.println("After ++iOb: " + iOb);
		// Here, iOb is unboxed, the expression is
		// evaluated, and the result is reboxed and
		// assigned to iOb2.
		iOb2 = iOb + (iOb / 3);
		System.out.println("iOb2 after expression: " + iOb2);
		// The same expression is evaluated, but the
		// result is not reboxed.
		i = iOb + (iOb / 3);
		System.out.println("i after expression: " + i);
	}
}

/*
 * The output is shown here:
	Original value of iOb: 100
	After ++iOb: 101
	iOb2 after expression: 134
	i after expression: 134
 */
```

STATIC IMPORT

- If you have to access the static members of a class, then it is necessary to qualify the references with the class from which they come.
- Java 5 provides the static import feature that enables unqualified access to static members without having to qualify them with the class name.
- **Use static imports sparingly**. If you overuse the static import feature, it can make your program unreadable and unmaintainable, polluting its namespace with all of the static members that you import. Readers of your code (including you, a few months after you wrote it) will not know from which class a static member comes. Importing all of the static members from a class can be very harmful to readability; if you need one or two members only, import them individually. Used appropriately, static import can make your program more readable, by removing the boilerplate of repetition of class names.

```java
package com.example.test;

import com.example.domain.PlayingCard;
import static com.example.domain.Suit.*;

public class TestPlayingCardStatic {

	public static void main(String[] args) {
		PlayingCard card = new PlayingCard(CLUBS, 2); // no need to prefix
the class
		System.out.println("card is the " + card.getRank() + " of the " +
card.getSuitName());
		System.out.println("card is the " + card.getRank() + " of the " +
card.getSuit().getName());
	}

}
```

ACTIVITY

GENERICS

GENERIC FUNDAMENTALS

- The term generics means parameterized types.
- Using generics, it is possible to create a single class, for example, that automatically works with different types of data.
- A class, interface, or method that operates on a parameterized type is called generic, as in generic class or generic method.
- Generics added the type safety.
- With generics, all casts are automatic and implicit.

A SIMPLE GENERIC 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

	// 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());
	}
}
```

```java
//Demonstrate the generic class.
public class Test {
	public static void main(String args[]) {
		// Create a Gen reference for Integers.
		Gen<Integer> iOb;
		// 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);
		// 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");
		// 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: 88
	Type of T is java.lang.String
	value: Generics Test
 */
```

BOUNDED TYPES

- Sometimes it is useful to limit the types that can be passed to a type parameter.
- For example, assume that you want to create a generic class that contains a method that returns the average of an array of numbers.
- Furthermore, you want to use the class to obtain the average of an array of any type of number, including integers, floats, and doubles.
- Thus, you want to specify the type of the numbers generically, using a type parameter.

```java
//In this version of Stats, the type argument for
//T must be either Number, or a class derived
//from Number.
class Stats<T extends Number> {
	T[] nums; // array of Number or subclass

	// Pass the constructor a reference to
	// an array of type Number or subclass.
	Stats(T[] o) {
		nums = o;
	}

	// Return type double in all cases.
	double average() {
		double sum = 0.0;
		for (int i = 0; i < nums.length; i++)
			sum += nums[i].doubleValue();
		return sum / nums.length;
	}
}
```

```java
//Demonstrate Stats.
public class Test {
	public static void main(String args[]) {
		Integer inums[] = { 1, 2, 3, 4, 5 };
		Stats<Integer> iob = new Stats<Integer>(inums);
		double v = iob.average();
		System.out.println("iob average is " + v);
		Double dnums[] = { 1.1, 2.2, 3.3, 4.4, 5.5 };
		Stats<Double> dob = new Stats<Double>(dnums);
		double w = dob.average();
		System.out.println("dob average is " + w);
	}
}

/*
 * The output is shown here:
	Average is 3.0
	Average is 3.3
 */
```

```java
//Use a wildcard.
class Stats<T extends Number> {
	T[] nums; // array of Number or subclass

	// Pass the constructor a reference to
	// an array of type Number or subclass.
	Stats(T[] o) {
		nums = o;
	}

	// Return type double in all cases.
	double average() {
		double sum = 0.0;
		for (int i = 0; i < nums.length; i++)
			sum += nums[i].doubleValue();
		return sum / nums.length;
	}

	// Determine if two averages are the same.
	// Notice the use of the wildcard.
	boolean sameAvg(Stats<?> ob) {
		if (average() == ob.average())
			return true;
		return false;
	}
}
```

USING WILDCARD ARGUMENTS

```
// Demonstrate wildcard.
public class Test {
	public static void main(String args[]) {
		Integer inums[] = { 1, 2, 3, 4, 5 };
		Stats<Integer> iob = new Stats<Integer>(inums);
		double v = iob.average();
		System.out.println("iob average is " + v);
		Double dnums[] = { 1.1, 2.2, 3.3, 4.4, 5.5 };
		Stats<Double> dob = new Stats<Double>(dnums);
		double w = dob.average();
		System.out.println("dob average is " + w);
		Float fnums[] = { 1.0F, 2.0F, 3.0F, 4.0F, 5.0F };
		Stats<Float> fob = new Stats<Float>(fnums);
		double x = fob.average();
		System.out.println("fob average is " + x);
		// See which arrays have same average.
		System.out.print("Averages of iob and dob ");
		if (iob.sameAvg(dob))
			System.out.println("are the same.");
		else
			System.out.println("differ.");
		System.out.print("Averages of iob and fob ");
		if (iob.sameAvg(fob))
			System.out.println("are the same.");
		else
			System.out.println("differ.");
	}
}
```

```java
//Two-dimensional coordinates.
class TwoD {
	int x, y;

	TwoD(int a, int b) {
		x = a;
		y = b;
	}
}

//Three-dimensional coordinates.
class ThreeD extends TwoD {
	int z;

	ThreeD(int a, int b, int c) {
		super(a, b);
		z = c;
	}
}

//Four-dimensional coordinates.
class FourD extends ThreeD {
	int t;

	FourD(int a, int b, int c, int d) {
		super(a, b, c);
		t = d;
	}
}
```

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 generic type that will operate on a class hierarchy.

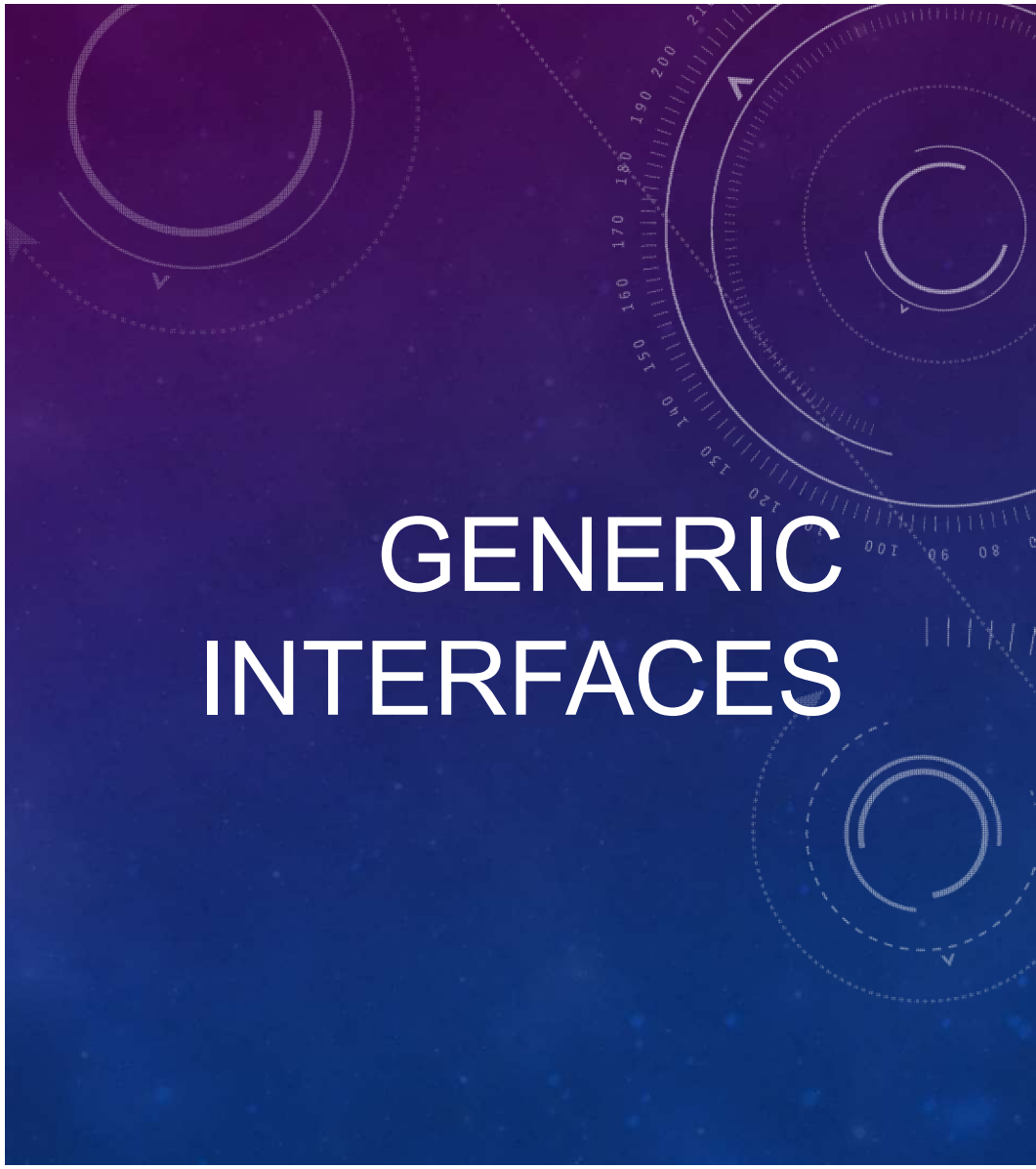
GENERIC METHODS

```java
//Demonstrate a simple generic method.
public class Test {

	// Determine if an object is in an array.
	static <T extends Comparable<T>, V extends T> boolean isIn(T x, V[] y) {
		for (int i = 0; i < y.length; i++)
			if (x.equals(y[i]))
				return true;
		return false;
	}

	public static void main(String args[]) {
		// Use isIn() on Integers.
		Integer nums[] = { 1, 2, 3, 4, 5 };
		if (isIn(2, nums))
			System.out.println("2 is in nums");
		if (!isIn(7, nums))
			System.out.println("7 is not in nums");
		System.out.println();
		// Use isIn() on Strings.
		String strs[] = { "one", "two", "three", "four", "five" };
		if (isIn("two", strs))
			System.out.println("two is in strs");
		if (!isIn("seven", strs))
			System.out.println("seven is not in strs");

		// Oops! Won't compile! Types must be compatible.
		// if(isIn("two", nums))
		// System.out.println("two is in strs");
	}
}

/*
 * The output from the program is shown here:
	2 is in nums
	7 is not in nums
 */
```

GENERIC CONSTRUCTORS

```
//Use a generic constructor.
class GenCons {
	private double val;

	<T extends Number> GenCons(T arg) {
		val = arg.doubleValue();
	}

	void showval() {
		System.out.println("val: " + val);
	}
}

public class Test {

	public static void main(String args[]) {
		GenCons test = new GenCons(100);
		GenCons test2 = new GenCons(123.5F);
		test.showval();
		test2.showval();
	}
}

/*
 * The output is shown here:
	val: 100.0
	val: 123.5
 */
```

```java
//A generic interface example.
//A Min/Max interface.
interface MinMax<T extends Comparable<T>> {
	T min();

	T max();
}

//Now, implement MinMax
class MyClass<T extends Comparable<T>> implements MinMax<T> {
	T[] vals;

	MyClass(T[] o) {
		vals = o;
	}

	// Return the minimum value in vals.
	public T min() {
		T v = vals[0];
		for (int i = 1; i < vals.length; i++)
			if (vals[i].compareTo(v) < 0)
				v = vals[i];
		return v;
	}

	// Return the maximum value in vals.
	public T max() {
		T v = vals[0];
		for (int i = 1; i < vals.length; i++)
			if (vals[i].compareTo(v) > 0)
				v = vals[i];
		return v;
	}
}
```



```java
public class Test {

	public static void main(String args[]) {
		Integer inums[] = { 3, 6, 2, 8, 6 };
		Character chs[] = { 'b', 'r', 'p', 'w' };
		MyClass<Integer> iob = new MyClass<Integer>(inums);
		MyClass<Character> cob = new MyClass<Character>(chs);
		System.out.println("Max value in inums: " + iob.max());
		System.out.println("Min value in inums: " + iob.min());
		System.out.println("Max value in chs: " + cob.max());
		System.out.println("Min value in chs: " + cob.min());
	}
}
```

RAW TYPES AND LEGACY CODE

```
//Demonstrate a raw type.
class Gen<T> {
	T ob; // declare an object of type T

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

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

```java
//Demonstrate raw type.
public class Test {

	public static void main(String args[]) {
		// Create a Gen object for Integers.
		Gen<Integer> iOb = new Gen<Integer>(88);
		// Create a Gen object for Strings.
		Gen<String> strOb = new Gen<String>("Generics Test");
		// Create a raw-type Gen object and give it
		// a Double value.
		Gen raw = new Gen(new Double(98.6));
		// Cast here is necessary because type is unknown.
		double d = (Double) raw.getob();
		System.out.println("value: " + d);

		// The use of a raw type can lead to run-time
		// exceptions. Here are some examples.

		// The following cast causes a run-time error!
		// int i = (Integer) raw.getob(); // run-time error

		// This assignment overrides type safety.
		strOb = raw; // OK, but potentially wrong
		// String str = strOb.getob(); // run-time error

		// This assignment also overrides type safety.
		raw = iOb; // OK, but potentially wrong
		// d = (Double) raw.getob(); // run-time error
	}
}
```

TYPE INFERENCE WITH THE DIAMOND OPERATOR

- Prior to JDK 7, to create an instance of MyClass, you would have needed to use a statement similar to the following:

  MyClass<Integer, String> mcOb = new MyClass<Integer, String>(98, "A String");

- Today the preceding declaration can be rewritten as shown here:

  MyClass<Integer, String> mcOb = new MyClass<>(98, "A String");

SOME GENERIC RESTRICTIONS

- Type Parameters Can’t Be Instantiated
- Restrictions on Static Members
- Generic Array Restrictions
- Generic Exception Restriction

ACTIVITY

LAMBDA EXPRESSIONS

INTRODUCING LAMBDA EXPRESSIONS

- Key to understanding Java's implementation of lambda expressions are two constructs. The first is the **lambda expression**, itself. The second is the **functional interface**.
- A lambda expression is, essentially, an anonymous (that is, unnamed) method. However, this method is not executed on its own. Instead, it is used to implement a method defined by a functional interface. Thus, a lambda expression results in a form of anonymous class. Lambda expressions are also commonly referred to as closures.
- A functional interface is an interface that contains one and only one abstract method.

LAMBDA EXPRESSION FUNDAMENTALS

- The new operator, sometimes referred to as the lambda operator or the arrow operator, is –>.
  - It divides a lambda expression into two parts.
  - The left side specifies any parameters required by the lambda expression.
  - On the right side is the lambda body, which specifies the actions of the lambda expression.

LAMBDA EXPRESSION FUNDAMENTALS

- Java defines two types of lambda bodies.
  - Single expressions.
  - Block of codes.

```
() -> 123.45
() -> Math.random() * 100
(n) -> (n % 2) ==  0
```

FUNCTIONAL INTERFACES

- A functional interface is an interface that specifies only one abstract method.

```
interface MyNumber {
        double getValue();
}
```

BLOCK LAMBDA EXPRESSIONS

```
//A block lambda that computes the factorial of an int value.
interface NumericFunc {
	int func(int n);
}

public class Test {
	public static void main(String args[]) {
		// This block lambda computes the factorial of an int value.
		NumericFunc factorial = (n) -> {
			int result = 1;
			for (int i = 1; i <= n; i++)
				result = i * result;
			return result;
		};

		System.out.println("The factoral of 3 is " + factorial.func(3));
		System.out.println("The factoral of 5 is " + factorial.func(5));
	}
}

/*
 * The output is shown here:
	The factorial of 3 is 6
	The factorial of 5 is 120
*/
```

Lambdas that have block bodies are sometimes referred to as block lambdas.

GENERIC FUNCTIONAL INTERFACES

- A lambda expression, itself, cannot specify type parameters.
- Thus, a lambda expression cannot be generic. However, the functional interface associated with a lambda expression can be generic.

```java
//Use a generic functional interface with lambda expressions.
//A generic functional interface.
interface SomeFunc<T> {
	T func(T t);
}

public class Test {
	public static void main(String args[]) {
		// Use a String-based version of SomeFunc.
		SomeFunc<String> reverse = (str) -> {
			String result = "";
			int i;
			for (i = str.length() - 1; i >= 0; i--)
				result += str.charAt(i);
			return result;
		};

		System.out.println("Lambda reversed is " + reverse.func("Lambda"));
		System.out.println("Expression reversed is " +
reverse.func("Expression"));

		// Now, use an Integer-based version of SomeFunc.
		SomeFunc<Integer> factorial = (n) -> {
			int result = 1;
			for (int i = 1; i <= n; i++)
				result = i * result;
			return result;
		};

		System.out.println("The factoral of 3 is " + factorial.func(3));
		System.out.println("The factoral of 5 is " + factorial.func(5));
	}
}

/*
 * The output is shown here:
	Lambda reversed is adbmaL
	Expression reversed is noisserpxE
	The factoral of 3 is 6
	The factoral of 5 is 120
 */
```

LAMBDA EXPRESSIONS AND VARIABLE CAPTURE

- Variables defined by the enclosing scope of a lambda expression are accessible within the lambda expression.
  - A lambda expression can use an instance or static variable defined by its enclosing class.
  - A lambda expression also has access to this keyword.
- However, when a lambda expression uses a local variable from its enclosing scope, a special situation is created that is referred to as a variable capture. In this case, a lambda expression may only use local variables that are effectively final.
  - An effectively final variable is one whose value does not change after it is first assigned.
  - There is no need to explicitly declare such a variable as final, although doing so would not be an error.

```java
//An example of capturing a local variable from the enclosing scope.
interface MyFunc {
	int func(int n);
}

public class Test {
	public static void main(String args[]) {
		// A local variable that can be captured.
		int num = 10;
		MyFunc myLambda = (n) -> {
			// This use of num is OK. It does not modify num.
			int v = num + n;
			// However, the following is illegal because it attempts
			// to modify the value of num.
			// num++;
			return v;
		};
		// The following line would also cause an error, because
		// it would remove the effectively final status from num.
		// num = 9;
	}
}
```

THROW AN EXCEPTION FROM WITHIN A LAMBDA EXPRESSION

- A lambda expression can throw an exception.
- However, it if throws a checked exception, then that exception must be compatible with the exception(s) listed in the throws clause of the abstract method in the functional interface.

```java
//Throw an exception from a lambda expression.
interface DoubleNumericArrayFunc {
	double func(double[] n) throws EmptyArrayException;
}

class EmptyArrayException extends Exception {
	EmptyArrayException() {
		super("Array Empty");
	}
}

public class Test {
	public static void main(String args[]) {
		double[] values = { 1.0, 2.0, 3.0, 4.0 };

		// This block lambda computes the average of an array of doubles.
		DoubleNumericArrayFunc average = (n) -> {
			double sum = 0;
			if (n.length == 0)
				throw new EmptyArrayException();
			for (int i = 0; i < n.length; i++)
				sum += n[i];
			return sum / n.length;
		};

		System.out.println("The average is " + average.func(values));

		// This causes an exception to be thrown.
		System.out.println("The average is " + average.func(new double[0]));
	}
}
```

METHOD REFERENCES TO STATIC METHODS

```
//Demonstrate a method reference for a static method.
//A functional interface for string operations.
interface StringFunc {
	String func(String n);
}

//This class defines a static method called strReverse().
class MyStringOps {
//A static method that reverses a string.
	static String strReverse(String str) {
		String result = "";
		int i;
		for (i = str.length() - 1; i >= 0; i--)
			result += str.charAt(i);
		return result;
	}
}
```

```java
//This class defines a static method called strReverse().
public class Test {

	// This method has a functional interface as the type of
	// its first parameter. Thus, it can be passed any instance
	// of that interface, including a method reference.
	static String stringOp(StringFunc sf, String s) {
		return sf.func(s);
	}

	public static void main(String args[]) {
		String inStr = "Lambdas add power to Java";
		String outStr;

		// Here, a method reference to strReverse is passed to stringOp().
		outStr = stringOp(MyStringOps::strReverse, inStr);

		System.out.println("Original string: " + inStr);
		System.out.println("String reversed: " + outStr);
	}
}

/*
 * The output is shown here:
	Original string: Lambdas add power to Java
	String reversed: avaJ ot rewop dda sadbmaL
 */
```

METHOD REFERENCES TO INSTANCE METHODS

```
//Demonstrate a method reference to an instance method
//A functional interface for string operations.
interface StringFunc {
	String func(String n);
}

//Now, this class defines an instance method called strReverse().
class MyStringOps {
	String strReverse(String str) {
		String result = "";
		int i;
		for (i = str.length() - 1; i >= 0; i--)
			result += str.charAt(i);
		return result;
	}
}
```

```
public class Test {

	// This method has a functional interface as the type of
	// its first parameter. Thus, it can be passed any instance
	// of that interface, including method references.
	static String stringOp(StringFunc sf, String s) {
		return sf.func(s);
	}

	public static void main(String args[]) {
		String inStr = "Lambdas add power to Java";
		String outStr;

		// Create a MyStringOps object.
		MyStringOps strOps = new MyStringOps();

		// Now, a method reference to the instance method strReverse
		// is passed to stringOp().
		outStr = stringOp(strOps::strReverse, inStr);

		System.out.println("Original string: " + inStr);
		System.out.println("String reversed: " + outStr);
	}
}

/*
 * The output is shown here: Original string:
      Lambdas add power to Java String
      reversed: avaJ ot rewop dda sadbmaL
 */
```

METHOD REFERENCES WITH GENERICS

```
//Demonstrate a method reference to a generic method
//declared inside a non-generic class.
//A functional interface that operates on an array
//and a value, and returns an int result.
interface MyFunc<T> {
      int func(T[] vals, T v);
}

//This class defines a method called countMatching() that
//returns the number of items in an array that are equal
//to a specified value. Notice that countMatching()
//is generic, but MyArrayOps is not.
class MyArrayOps {
      static <T> int countMatching(T[] vals, T v) {
            int count = 0;
            for (int i = 0; i < vals.length; i++)
                  if (vals[i] == v)
                        count++;
            return count;
      }
}
```

```
public class Test {

	// This method has the MyFunc functional interface as the
	// type of its first parameter. The other two parameters
	// receive an array and a value, both of type T.
	static <T> int myOp(MyFunc<T> f, T[] vals, T v) {
		return f.func(vals, v);
	}

	public static void main(String args[]) {
		Integer[] vals = { 1, 2, 3, 4, 2, 3, 4, 4, 5 };
		String[] strs = { "One", "Two", "Three", "Two" };
		int count;

		count = myOp(MyArrayOps::<Integer>countMatching, vals, 4);
		System.out.println("vals contains " + count + " 4s");

		count = myOp(MyArrayOps::<String>countMatching, strs, "Two");
		System.out.println("strs contains " + count + " Twos");
	}
}

/*
 * The output is shown here:
	vals contains 3 4s
	strs contains 2 Twos
 */
```

CONSTRUCTOR REFERENCES

```
//Demonstrate a Constructor reference.
//MyFunc is a functional interface whose method returns
//a MyClass reference.
interface MyFunc {
        MyClass func(int n);
}

class MyClass {
        private int val;

        // This constructor takes an argument.
        MyClass(int v) {
                val = v;
        }

        // This is the default constructor.
        MyClass() {
                val = 0;
        }

        // ...
        int getVal() {
                return val;
        };
}
```

```java
public class Test {

	public static void main(String args[]) {
		// Create a reference to the MyClass constructor.
		// Because func() in MyFunc takes an argument, new
		// refers to the parameterized constructor in MyClass,
		// not the default constructor.
		MyFunc myClassCons = MyClass::new;

		// Create an instance of MyClass via that constructor reference.
		MyClass mc = myClassCons.func(100);

		// Use the instance of MyClass just created.
		System.out.println("val in mc is " + mc.getVal());
	}
}

/*
 * The output is shown here:
	val in mc is 100
 */
```

PREDEFINED FUNCTIONAL INTERFACES

Interface	Purpose
UnaryOperator<T>	Apply a unary operation to an object of type **T** and return the result, which is also of type **T**. Its method is called **apply( )**.
BinaryOperator<T>	Apply an operation to two objects of type **T** and return the result, which is also of type **T**. Its method is called **apply( )**.
Consumer<T>	Apply an operation on an object of type **T**. Its method is called **accept( )**.
Supplier<T>	Return an object of type **T**. Its method is called **get( )**.
Function<T, R>	Apply an operation to an object of type **T** and return the result as an object of type **R**. Its method is called **apply( )**.
Predicate<T>	Determine if an object of type **T** fulfills some constraint. Return a **boolean** value that indicates the outcome. Its method is called **test( )**.

```java
//Use the Function built-in functional interface.
//Import the Function interface.
import java.util.function.Function;

public class Test {

	public static void main(String args[]) {
		// This block lambda computes the factorial of an int value.
		// This time, Function is the functional interface.
		Function<Integer, Integer> factorial = (n) -> {
			int result = 1;
			for (int i = 1; i <= n; i++)
				result = i * result;
			return result;
		};

		System.out.println("The factoral of 3 is " + factorial.apply(3));
		System.out.println("The factoral of 5 is " + factorial.apply(5));
	}
}
```

ACTIVITY

THE END AND THANK YOU!