The Checklist

Slideshop-2014

INTRODUCTION TO PROGRAMMING && JAVA FUNDAMENTALS

CHECKLIST - HEXADECIMAL

INTRODUCTION TO PROGRAMMING

Hexadecimal refers to the base-16 number system, which consists of 16 unique symbols, in contrast to the ten unique symbols of the commonly used decimal (i.e., base 10) numbering system.

The hexadecimal system is commonly used by programmers to describe locations in memory because it can represent every byte (i.e., eight bits) as two consecutive hexadecimal digits instead of the eight digits that would be required by binary (i.e., base 2) numbers

Hexadecimal numbers are indicated by the addition of either an *0x* prefix or an *h* suffix. For example, the hexadecimal number 0x2F5B translates to the binary number 0010 1111 0101 1011.

A common use of hexadecimal numbers is to describe colors on web pages. Each of the three primary colors (i.e., red, green and blue) is represented by two hexadecimal digits to create 255 possible values, thus resulting in more than 16 million possible colors. Example: <body bgcolor="#FF0000">

CHECKLIST – LOGICAL OPERATORS

INTRODUCTION TO PROGRAMMING

p	q	p and q
true	true	true
true	false	false
false	true	false
false	false	false

p	q	p or q
true	true	true
true	false	true
false	true	true
false	false	false

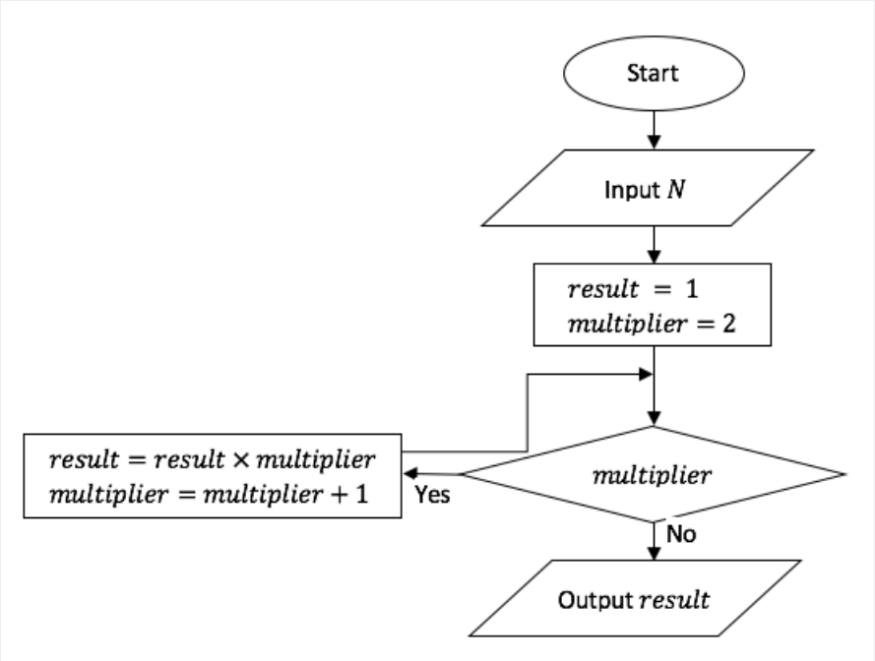
p	not p
true	false
false	true

The concept of logical operators is simple. They allow a program to make a decision based on multiple conditions. Each operand is considered a condition that can be evaluated to a true or false value.

They are used in if conditions also in while or for loops.

CHECKLIST - FLOWCHARTS

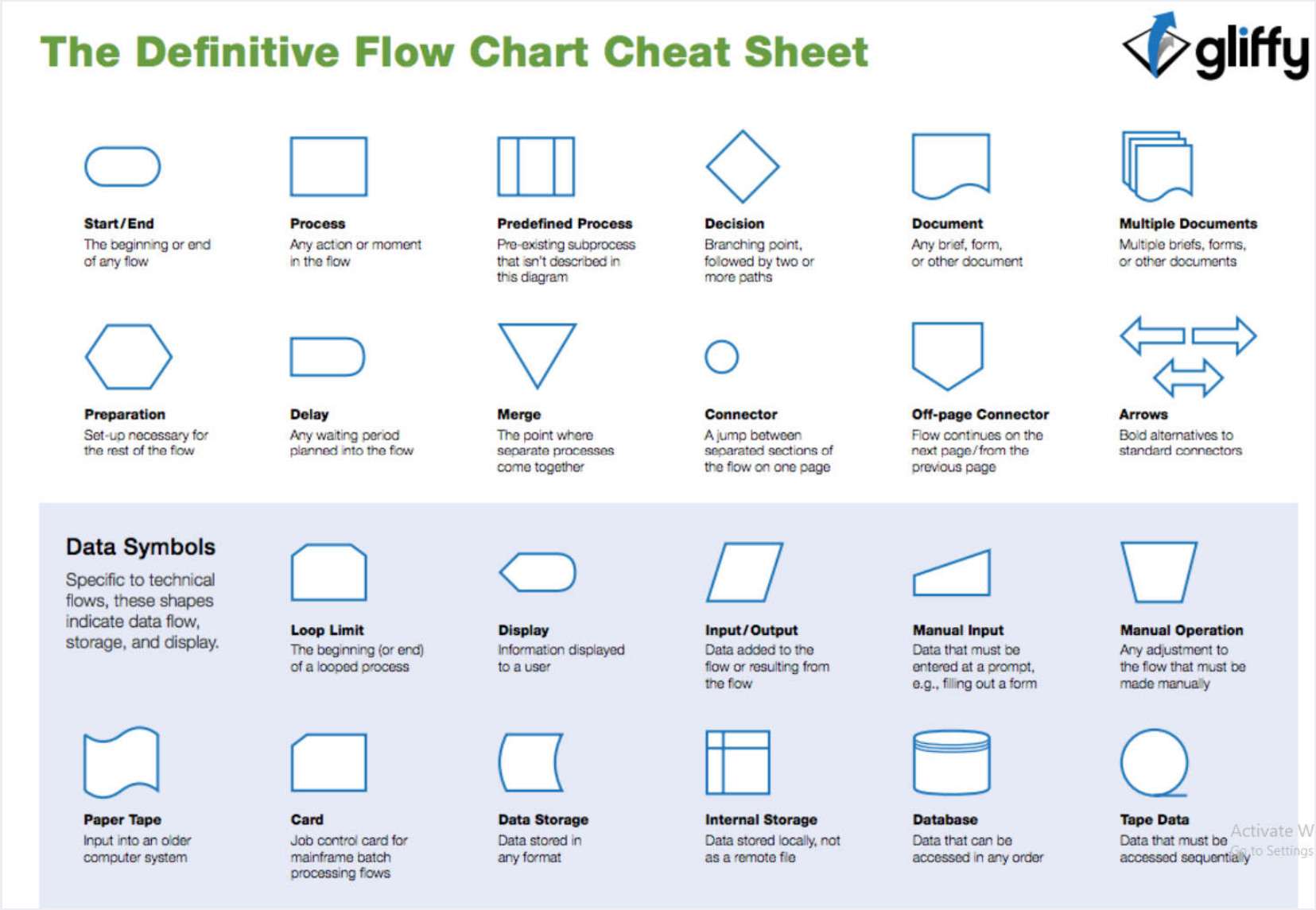
INTRODUCTION TO PROGRAMMING



One tool to consider is a flowchart, which is a pictorial or graphical representation of a process. ... The purpose of flowcharts are to communicate how a process works or should work without any confusing technical jargon.

CHECKLIST - FLOWCHARTS

INTRODUCTION TO PROGRAMMING



CHECKLIST - PSEUDOCODE

INTRODUCTION TO PROGRAMMING

```
Start
Input N
result = 1
multiplier = 2
While multiplier <= N
    result = result * multiplier
    multiplier = multiplier + 1
End While
ouput result
End
```

Pseudocode is an informal high-level description of the operating principle of a computer program or other algorithm. It uses the structural conventions of a normal programming language, but is intended for human reading rather than machine reading.

Flowcharts are mostly used in business life. In case of complex decision makings flowchart can easy your life by having a visual decision map

CHECKLIST - CHARACTER ENCODING

INTRODUCTION TO PROGRAMMING

- If you're used to writing programs in other languages, you may be aware of something called ASCII Character Encoding.
- Most languages use ASCII; Java uses Unicode.
- In the old ASCII representation, each character takes up only 8 bits, but in Unicode, each character takes up 8, 16, or 32 bits.
- ASCII stores the letters of the familiar Roman (English) alphabet, Unicode has room for characters from most of the world's commonly spoken languages.

CHECKLIST - CHARACTER ENCODING

INTRODUCTION TO PROGRAMMING

- UTF-8 is a method for encoding Unicode characters using 8-bit sequences.

- Main difference between UTF-8, UTF-16 and UTF-32 character encoding is how many bytes it require to represent a character in memory.
  - UTF-8 uses minimum one byte
  - UTF-16 uses minimum 2 bytes
  - UTF-32 is fixed width encoding scheme and always uses 4 bytes to encode a Unicode code point.

- Now, let's start with what is character encoding and why it's important?
  - Well, character encoding is an important concept in process of converting byte streams into characters, which can be displayed.
  - There are two things, which are important to convert bytes to characters, a **character set and an encoding**.
  - Since there are so many characters and symbols in the world, a character set is required to support all those characters. A character set is nothing but list of characters, where each symbol or character is mapped to a numeric value, also known as code points.

CHECKLIST - CHARACTER ENCODING

INTRODUCTION TO PROGRAMMING

- On the other hand UTF-16, UTF-32 and UTF-8 are encoding schemes, which describe how these values (code points) are mapped to bytes (using different bit values as a basis; e.g. 16-bit for UTF-16, 32 bits for UTF-32 and 8-bit for UTF-8).
- For example, for character A, which is Latin Capital A, Unicode code point is U+0041, UTF-8 encoded bytes are 41, UTF-16 encoding is 0041 and Java char literal is '\u0041'.
- In short, you must need a character encoding scheme to interpret stream of bytes, in the absence of character encoding, you cannot show them correctly.
- Java programming language has extensive support for different charset and character encoding, by default it use UTF-8.

CHECKLIST - CHARACTER ENCODING

INTRODUCTION TO PROGRAMMING

- UTF-8 has an advantage where ASCII are most used characters, in that case most characters only need one byte.
- UTF-8 file containing only ASCII characters has the same encoding as an ASCII file, which means English text looks exactly the same in UTF-8 as it did in ASCII.
- Given dominance of ASCII in past this was the main reason of initial acceptance of Unicode and UTF-8.

CHECKLIST - ABOUT JAVA

JAVA FUNDAMENTALS

1. Architecture neutral
2. Distributed
3. Dynamic
4. High Performance
5. Interpreted
6. Multithreaded
7. Object-Oriented
8. Platform independent
9. Portable
10. Robust
11. Secured
12. Simple

ARCHITECTURE NEUTRAL

- The compiler will generate an architecture-neutral object file meaning that compiled Java code (bytecode) can run on many processors given the presence of a Java runtime
- It's no need to recompile Java source code for 32-bit or 64-bit

DISTRIBUTED

- It is possible to create distributed applications in Java
- Programs can be design to run on computer networks
- Support for TCP, UDP, and basic Socket communication is excellent and getting better
- Also RMI and EJB are used for creating distributed applications

DYNAMIC

- Compiler doesn't understand which method to called in advance
- JVM decide which method to called at run time
- All Java objects are dynamically allocated

HIGH PERFORMANCE

- Java is faster than traditional interpretation since byte code is "close" to native code
- Java uses Just-In-Time compiler - a computer program that turns Java byte codes into instructions that can directly be sent to compilers

INTERPRETED

- Java is a compiled programming Language which compiles the Java program into Java byte codes
- This JVM is then interpreted to run the program

MULTITHREADED

- A thread in Java refers to an independent program, executing concurrently
- We can write Java programs that deal with many tasks at once by defining multiple threads
- The main advantage of multi-threading is that it doesn't occupy memory for each thread. It shares a common memory area

CHECKLIST – ABOUT JAVA

JAVA FUNDAMENTALS

1. Architecture neutral
2. Distributed
3. Dynamic
4. High Performance
5. Interpreted
6. Multithreaded
7. Object-Oriented
8. Platform independent
9. Portable
10. Robust
11. Secured
12. Simple

OBJECT-ORIENTED

- The code is organized as a combination of different types of objects which have data and behaviour
- Basic concepts of OOP: class, object, abstraction, encapsulation, inheritence, polimorphism

PORTABLE

- Output of a Java compiler is Non Executable Code i.e Bytecode
- Bytecode is executed by Java run-time system - Java Virtual Machine (JVM)

ROBUST

- Java uses strong memory management
- There are lack of pointers that avoids security problem
- There is exception handling and type checking mechanism
- There is automatic garbage collection

PLATFORM INDEPENDENT

- Java code can be run on multiple platforms (Windows, Linux, Mac/OS etc)
- When we compile Java code then .class file is generated by javac compiler
- These codes are readable by JVM and every operating system have its own JVM
- So, Java is platform independent but JVM is platform dependent

SIMPLE

- There is no confusing rarely used features: explicit pointers, operator overloading, multiple inheritance etc
- Syntax is based on C++
- Garbage Collection - no need to remove unreferenced objects

SECURED

- Java program is executed by the JVM
- The JVM prevent java code from generating side effects outside of the system

CHECKLIST – Primitive Types & Keywords

JAVA FUNDAMENTALS

Table 4-1 Java's Primitive Types

Type Name	What a Literal Looks Like	Range of Values
Whole number types		
`byte`	`(byte)42`	–128 to 127
`short`	`(short)42`	–32768 to 32767
`int`	`42`	–2147483648 to 2147483647
`long`	`42L`	–9223372036854775808 to 9223372036854775807
Decimal number types		
`float`	`42.0F`	-3.4×10^{38} to 3.4×10^{38}
`double`	`42.0`	-1.8×10^{308} to 1.8×10^{308}

(continued)

Table 4-1 *(continued)*

Type Name	What a Literal Looks Like	Range of Values
Character type		
`char`	`'A'`	Thousands of characters, glyphs, and symbols
Logical type		
`boolean`	`true`	true, false

JAVA KEYWORDS

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

CHECKLIST – VARIABLES

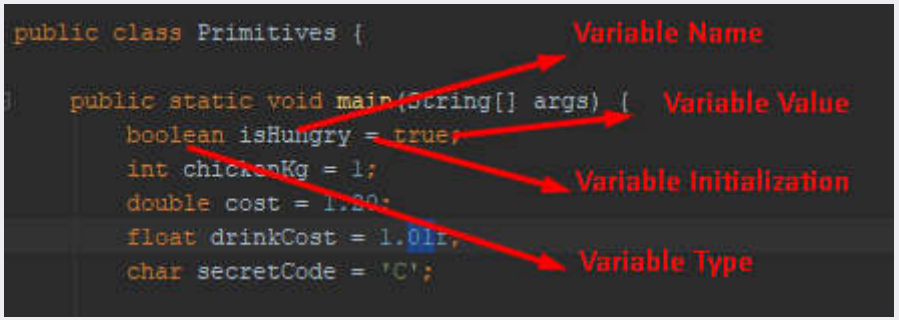
JAVA FUNDAMENTALS

- A variable is a placeholder. You can stick a number like 50.22 into a variable. After you place a number in the variable, you can change your mind and put a different number into the variable.
- Of course, when you put a new number in a variable, the old number is no longer there. If you didn't save the old number somewhere else, the old number is gone.

- Now you need some terminology. The thing stored in a variable is a value.
- A variable's value can change during the run of a program (when Jack gives you a million bucks, for instance).
- The value that's stored in a variable isn't necessarily a number. (For instance, you can create a variable that always stores a letter.)
- The kind of value that's stored in a variable is a variable's type.

CHECKLIST – VARIABLES

JAVA FUNDAMENTALS



3. **Variables :** Variable names should be short yet meaningful.
   - Should **not** start with underscore('_') or dollar sign '$' characters.
   - Should be mnemonic i.e, designed to indicate to the casual observer the intent of its use.
   - **One-character variable names should be avoided** except for temporary variables.
   - Common names for temporary variables are i, j, k, m, and n for integers; c, d, and e for characters.

* Start with lower case for variable name*

4. **Constant variables:**
   - Should be **all uppercase** with words separated by underscores ("_").
   - There are various constants used in predefined classes like Float, Long, String etc.

Examples:

```
static final int MIN_WIDTH = 4;

// Some  Constant variables used in predefined Float class
public static final float POSITIVE_INFINITY = 1.0f / 0.0f;
public static final float NEGATIVE_INFINITY = -1.0f / 0.0f;
public static final float NaN = 0.0f / 0.0f;
```

CHECKLIST – OPERATORS && Boxing Conversion

JAVA FUNDAMENTALS

Simple Assignment Operator

=	Simple assignment operator

Arithmetic Operators

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

Equality and Relational Operators

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

Type Comparison Operator

instanceof	Compares an object to a specified type

Unary Operators

++	Increment operator; increments a value by 1
--	Decrement operator; decrements a value by 1

Conditional Operators

&&	Conditional-AND
||	Conditional-OR
?:	Ternary (shorthand for if-then-else statement)

Auto Boxing is ***used*** to convert primitive data types to their wrapper class objects. Wrapper class provide a wide range of function to be performed on the primitive types.

The most common example is :

```
1 int a = 56;
2 Integer i = a; // Auto Boxing
```

Unboxing is opposite of Auto Boxing where we convert the wrapper class object back to its primitive type. This is done automatically by JVM so that we can use a the wrapper classes for certain operation and then convert them back to primitive types as primitives result int faster processing. For Example :

```
1 Integer s = 45;
2 int a = s; auto UnBoxing;
```

CHECKLIST – STRING

JAVA FUNDAMENTALS

```
import javax.swing.JFrame;

public class ShowAFrame {

    public static void main(String args[]) {
        JFrame myFrame = new JFrame();
        String myTitle = "Blank Frame";

        myFrame.setTitle(myTitle);
        myFrame.setSize(300, 200);
        myFrame.setDefaultCloseOperation
            (JFrame.EXIT_ON_CLOSE);
        myFrame.setVisible(true);
    }
}
```

A `String` is a bunch of characters. It's like having several `char` values in a row. So, with the `myTitle` variable declared to be of type `String`, assigning `"Blank Frame"` to the `myTitle` variable makes sense in Listing 4-6. The `String` class is declared in the Java API.

In a Java program, double quote marks surround the letters in a `String` literal.

A *String literal* is a Java language concept. This is a String literal:

```
"a String literal"
```

A *String object* is an individual instance of the `java.lang.String` class.

```
String s1 = "abcde";
String s2 = new String("abcde");
String s3 = "abcde";
```

All are valid, but have a slight difference. `s1` will refer to an *interned* String object. This means, that the character sequence `"abcde"` will be stored at a central place, and whenever the same literal `"abcde"` is used again, the JVM will not create a new String object but use the reference of the *cached* String.

`s2` is guranteed to be a *new String object*, so in this case we have:

```
s1 == s2 // is false
s1 == s3 // is true
s1.equals(s2) // is true
```

CHECKLIST – STRING

JAVA FUNDAMENTALS

STRINGS OPERATIONS - SUBSTRINGS

```
String text = "Hello, World!";
String s = text.substring(0, 5);
```

creates a new string consisting of the characters "Hello"

```
String t = text.substring(7);
```

creates a new string consisting of the characters "World!"

STRINGS OPERATIONS - CONCATENATION

```
String h = "Hello";
String w = "World!";
String text = h + ", " + w;
```

creates a new string consisting of the characters "Hello, World!"

public class String

	String(String s)	*create a string with the same value as* `s`
	String(char[] a)	*create a string that represents the same sequence of characters as in* `a[]`
int	length()	*number of characters*
char	charAt(int i)	*the character at index* `i`
String	substring(int i, int j)	*characters at indices* `i` *through* `(j-1)`
boolean	contains(String substring)	*does this string contain* `substring`*?*
boolean	startsWith(String prefix)	*does this string start with* `prefix`*?*
boolean	endsWith(String postfix)	*does this string end with* `postfix`*?*
int	indexOf(String pattern)	*index of first occurrence of* `pattern`
int	indexOf(String pattern, int i)	*index of first occurrence of* `pattern` *after* `i`
String	concat(String t)	*this string, with* `t` *appended*
int	compareTo(String t)	*string comparison*
String	toLowerCase()	*this string, with lowercase letters*
String	toUpperCase()	*this string, with uppercase letters*
String	replace(String a, String b)	*this string, with* `a` *replaced by* `b`*s*
String	trim()	*this string, with leading and trailing whitespace removed*
boolean	matches(String regexp)	*is this string matched by the regular expression?*
String[]	split(String delimiter)	*strings between occurrences of* `delimiter`
boolean	equals(Object t)	*is this string's value the same as* `t`*'s?*
int	hashCode()	*an integer hash code*

CHECKLIST – LOOPS

JAVA FUNDAMENTALS

```
int x = 3;
while (x > 1) {
    System.out.println(x);
    x--;
}
```

```
do {
  // statement or block of code
} while (expression);
```

```
for (initialization; condition; iteration) {
    // statement or block of code
}
```

```
for (int x = 0; x < 10; x++) {
    System.out.println("x is " + x);
}
```

```
for (declaration : expression) {
    // statement or block of code
}
```

CHECKLIST – LOOPS && IF && SWITCH

JAVA FUNDAMENTALS

LOOP CONTROL FLOW

code in loop	behaviour
`break`	execution jumps immediately to the first statement after the loop
`continue`	stops just the current iteration
`return`	execution jumps immediately back to the calling method
`System.exit()`	all program execution stops; the VM shuts down

IF STATEMENT

```
if (booleanExpression) {
    // statement or block of code
}
```

```
if (booleanExpression) {
    // statement or block of code
} else {
    // statement or block of code
}
```

```
if (booleanExpression) {
    // statement or block of code
} else if (booleanExpression) {
    // statement or block of code
  ...
} else {
    // statement or block of code
}
```

SWITCH STATEMENT - EXAMPLE

```
switch (direction) {
    case 'n':
        System.out.println("You are going North");
        break;
    case 's':
        System.out.println("You are going South");
        break;
    case 'e':
        System.out.println("You are going East");
        break;
    case 'w':
        System.out.println("You are going West");
        break;
    default:
        System.out.println("Bad direction!");
```

CHECKLIST – ARRAYS

JAVA FUNDAMENTALS

The Java Motel, with its ten comfortable rooms, sits in a quiet place off the main highway. Aside from a small, separate office, the motel is just one long row of ground floor rooms. Each room is easily accessible from the spacious front parking lot.

Oddly enough, the motel's rooms are numbered 0 through 9. I could say that the numbering is a fluke — something to do with the builder's original design plan. But the truth is that starting with 0 makes the examples in this chapter easier to write.

Anyway, you're trying to keep track of the number of guests in each room. Because you have ten rooms, you may think about declaring ten variables:

```
int guestsInRoomNum0, guestsInRoomNum1, guestsInRoomNum2,
    guestsInRoomNum3, guestsInRoomNum4, guestsInRoomNum5,
    guestsInRoomNum6, guestsInRoomNum7, guestsInRoomNum8,
    guestsInRoomNum9;
```

CHECKLIST – ARRAYS

JAVA FUNDAMENTALS

Doing it this way may seem a bit inefficient — but inefficiency isn't the only thing wrong with this code. Even more problematic is the fact that you can't loop through these variables. To read a value for each variable, you have to copy the `nextInt` method ten times.

```
guestsInRoomNum0 = diskScanner.nextInt();
guestsInRoomNum1 = diskScanner.nextInt();
guestsInRoomNum2 = diskScanner.nextInt();
... and so on.
```

Surely a better way exists.

That better way involves an array. An *array* is a row of values, like the row of rooms in a one-floor motel. To picture the array, just picture the Java Motel:

- First, picture the rooms, lined up next to one another.
- Next, picture the same rooms with their front walls missing. Inside each room you can see a certain number of guests.
- If you can, forget that the two guests in Room 9 are putting piles of bills into a big briefcase. Ignore the fact that the guests in Room 6 haven't moved away from the TV set in a day and a half. Instead of all these details, just see numbers. In each room, see a number representing the count of guests in that room. (If free-form visualization isn't your strong point, look at Figure 11-1.)

CHECKLIST – ARRAYS

JAVA FUNDAMENTALS

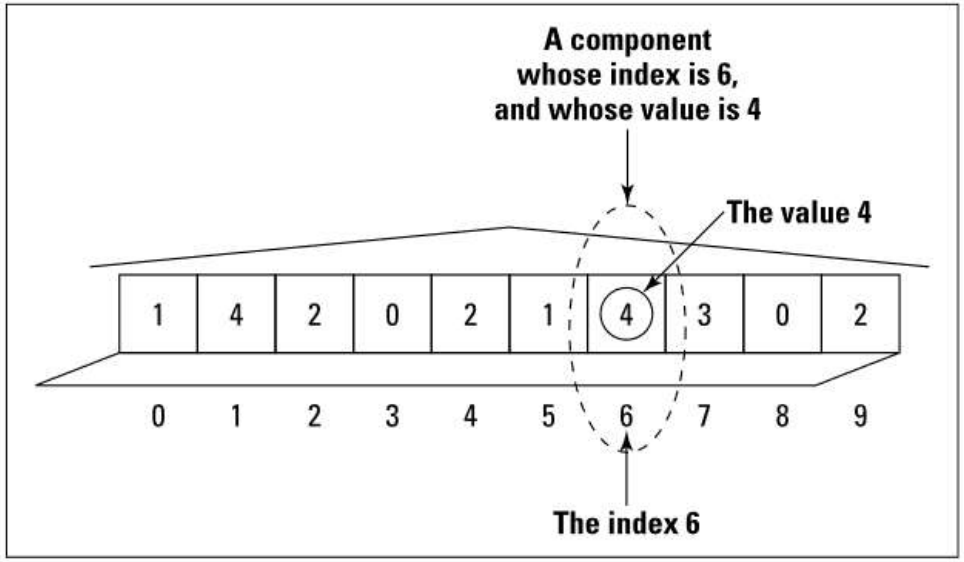


Figure 11-1: An abstract snapshot of rooms in the Java Motel.

In the lingo of this chapter, the entire row of rooms is called an *array*. Each room in the array is called a *component* of the array (also known as an array *element*). Each component has two numbers associated with it:

CHECKLIST – ARRAYS

JAVA FUNDAMENTALS

- The room number (a number from 0 to 9), which is called an *index* of the array
- A number of guests, which is a *value* stored in a component of the array

Using an array saves you from all the repetitive nonsense in the sample code shown at the beginning of this section. For instance, to declare an array with ten values in it, you can write one fairly short statement:

```
int guests[] = new int[10];
```

If you're especially verbose, you can expand this statement so that it becomes two separate statements:

```
int guests[];
guests = new int[10];
```

In either of these code snippets, notice the use of the number 10. This number tells the computer to make the `guests` array have ten components. Each component of the array has a name of its own. The starting component is named *guests[0]*, the next is named *guests[1]*, and so on. The last of the ten components is named *guests[9]*.

CHECKLIST – ARRAYS

JAVA FUNDAMENTALS

Each line serves its own distinct purpose:

- `int guests[]`: This first line is a declaration. The declaration reserves the array name (a name like *guests*) for use in the rest of the program. In the Java Motel metaphor, this line says, "I plan to build a motel here and put a certain number of guests in each room." (See Figure 11-2.)

  Never mind what the declaration `int guests[]` actually does. It's more important to notice what the declaration `int guests[]` *doesn't* do. The declaration doesn't reserve ten memory locations. Indeed, a declaration like `int guests[]` doesn't really create an array. All the declaration does is set up the `guests` variable. At that point in the code, the `guests` variable still doesn't refer to a real array. (In other words, the motel has a name, but the motel hasn't been built yet.)

- `guests = new int[10]`: This second line is an assignment statement. The assignment statement reserves space in the computer's memory for ten `int` values. In terms of real estate, this line says, "I've finally built the motel. Go ahead and put guests in each room." (Again, see Figure 11-2.)

CHECKLIST – ARRAYS

JAVA FUNDAMENTALS

accessing

```
int[] arrayOfInts = {10, 15, 20, 25, 30};
System.out.println(arrayOfInts[0]);    // print 10
System.out.println(arrayOfInts[2]);    // print 20
System.out.println(arrayOfInts[4]);    // print 30

for (int i = 0; i < arrayOfInts.length; i++) {
    System.out.print(arrayOfInts[i] + " ");
}
// print 10 15 20 25 30

for (int i : arrayOfInts) {
    System.out.print(i + " ");
}
// print 10 15 20 25 30
```

CHECKLIST – ARRAYS

JAVA FUNDAMENTALS

1.Printing array

We always use for loop or some other iteration on the array then print all the elements , but we can do something different like

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

System.out.println(array); //[I@1234be4e

String arrStr = Arrays.toString(array);

System.out.println(array); //[1,2,3,4,5];
```

2.Creating an ArrayList from an Array

```
String[] array = { "a", "b", "c", "d", "e" };

ArrayList<String> arrayList =
                        new ArrayList<String>(Arrays.asList(array));

System.out.println(arrayList);

// [a, b, c, d, e]
```

3.Check the value persent in an array

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

boolean isContain= Arrays.asList(array).contains(5);

System.out.println(isContain);
// true
```

4.Concatenate two arrays

```
int[] array1 = { 1, 2, 3, 4, 5 };

int[] array2 = { 6, 7, 8, 9, 10 };

// Apache Commons Lang library
int[] combinedIntArray = ArrayUtils.addAll(array1, array2);
```

5. Declare an array inline

```
method(new String[]{"a", "b", "c", "d", "e"});
```

CHECKLIST – ARRAYS

JAVA FUNDAMENTALS

6.Reverse an array

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

// Apache Commons Lang library
ArrayUtils.reverse(intArray);

System.out.println(Arrays.toString(intArray));
//[5, 4, 3, 2, 1]
```

7. Remove element of an array

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

int[] removed = ArrayUtils.removeElement(intArray, 3);//create a new
array

System.out.println(Arrays.toString(removed));
```

8. Convert an array to a set

```
Set<String> set = new HashSet<String>(Arrays.asList(new
String[]{"a", "b", "c", "d", "e"}));

System.out.println(set);

//[d, e, b, c, a]
```

9. Convert an ArrayList to an array

```
String[] stringArray = { "a", "b", "c", "d", "e" };

ArrayList<String> arrayList = new ArrayList<String>
(Arrays.asList(stringArray));

String[] stringArr = new String[arrayList.size()];

arrayList.toArray(stringArr);
```

10. Joins the elements of the provided array into a single String

```
// Apache common lang
String j = StringUtils.join(new String[] { "a", "b", "c" }, ", ");

System.out.println(j); //a, b, c
```

CHECKLIST – OBJECT-ORIENTED PROGRAMMING

JAVA FUNDAMENTALS

Because Java is an object-oriented programming language, your primary goal is to describe classes and objects. A class is the idea behind a certain kind of thing. An object is a concrete instance of a class. The programmer defines a class, and from the class definition, the computer makes individual objects.

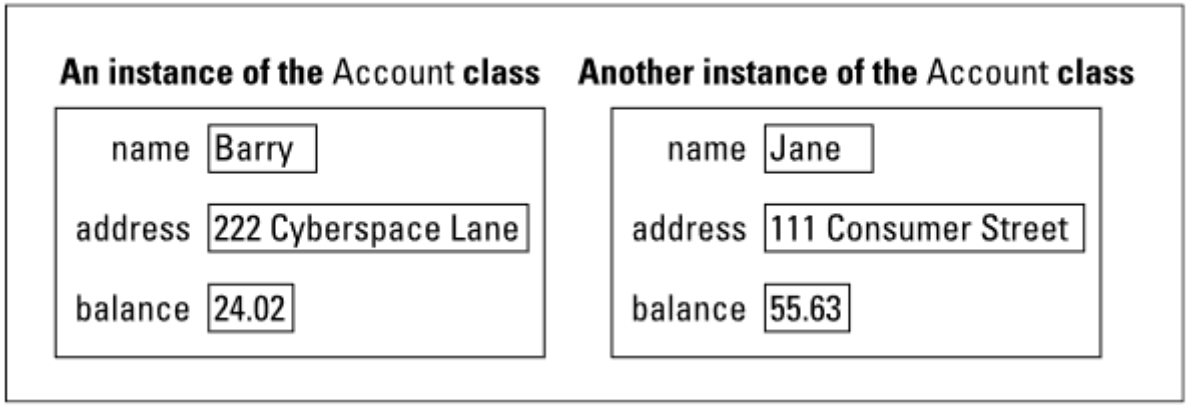
What distinguishes one bank account from another? If you ask a banker this question, you hear a long sales pitch. The banker describes interest rates, fees, penalties — the whole routine. Fortunately for you, I'm not interested in all that. Instead, I want to know how my account is different from your account. After all, my account is named *Barry Burd, trading as Burd Brain Consulting*, and your account is named *Jane Q. Reader, trading as Budding Java Expert*. My account has $24.02 in it. How about yours?

When you come right down to it, the differences between one account and another can be summarized as values of variables. Maybe there's a variable named *balance*. For me, the value of `balance` is `24.02`. For you, the value of `balance` is `55.63`. The question is, when writing a computer program to deal with accounts, how do I separate my `balance` variable from your `balance` variable?

The answer is to create two separate objects. Let one `balance` variable live inside one of the objects and let the other `balance` variable live inside the other object. While you're at it, put a `name` variable and an `address` variable in each of the objects. And there you have it — two objects, and each object represents an account. More precisely, each object is an instance of the `Account` class. (See Figure 7-1.)

CHECKLIST – OBJECT-ORIENTED PROGRAMMING

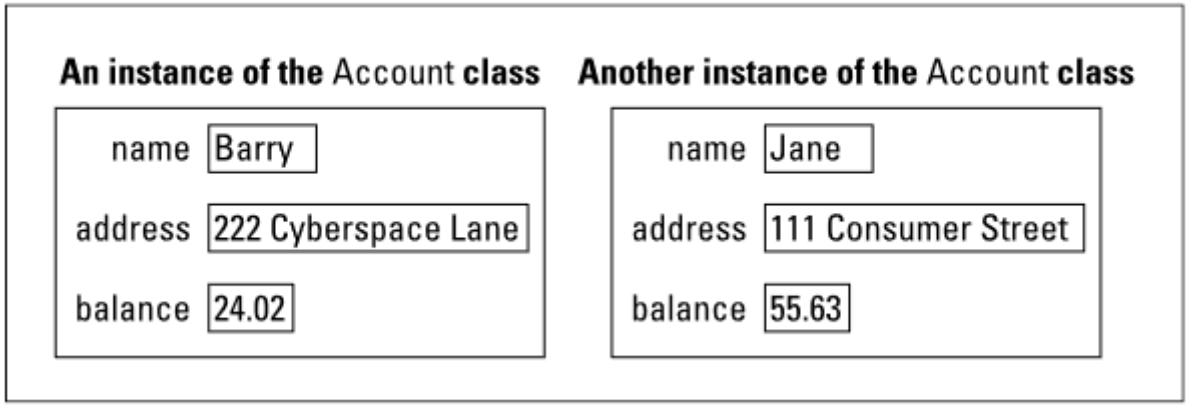
JAVA FUNDAMENTALS



So far, so good. However, you still haven't solved the original problem. In your computer program, how do you refer to my `balance` variable, as opposed to your `balance` variable? Well, you have two objects sitting around, so maybe you have variables to refer to these two objects. Create one variable named *myAccount* and another variable named *yourAccount*. The `myAccount` variable refers to my object (my instance of the `Account` class) with all the stuff that's inside it. To refer to my balance, write

CHECKLIST – OBJECT-ORIENTED PROGRAMMING

JAVA FUNDAMENTALS




```
public class Account {
    String name;
    String address;
    double balance;
}
```

The `Account` class in Listing 7-1 defines what it means to be an `Account`. In particular, Listing 7-1 tells you that each of the `Account` class's instances has three variables — `name`, `address`, and `balance`. This is consistent with the information in Figure 7-1. Java programmers have a special name for variables of this kind (variables that belong to instances of classes). Each of these variables — `name`, `address`, and `balance` — is called a *field*.

A variable declared inside a class but not inside any particular method is a *field*. In Listing 7-1, the variables `name`, `address`, and `balance` are fields. Another name for a field is an *instance variable*.

CHECKLIST – OBJECT-ORIENTED PROGRAMMING

JAVA FUNDAMENTALS

CLASS DECLARATION

Class declarations can include these components, in order:

1. Modifiers such as *public*, *private* (if any), and a number of others that you will encounter later.
2. The class name, with the initial letter capitalized by convention.
3. The class body, surrounded by braces, {}.

ACCESS MODIFIERS

There are four *access controls* (levels of access) but only three *access modifiers*.

- `public` - visible to the world
- `protected` - visible to the package and all subclasses
- default - visible to the package
- `private` - visible to the class only

CHECKLIST – Variable Scopes

JAVA FUNDAMENTALS

```
class VariableDemo
{
   static int count=0;
   public void increment()
   {
       count++;
   }
   public static void main(String args[])
   {
       VariableDemo obj1=new VariableDemo();
       VariableDemo obj2=new VariableDemo();
       obj1.increment();
       obj2.increment();
       System.out.println("Obj1: count is="+obj1.count);
       System.out.println("Obj2: count is="+obj2.count);
   }
}
```

Output:

```
Obj1: count is=2
Obj2: count is=2
```

static - they are created when the class is loaded, and they survive as long as the class stays loaded in the JVM

CHECKLIST – Variable Scopes

JAVA FUNDAMENTALS

Instance variable in java is used by Objects to store their states. Variables which are defined without the STATIC keyword and are *Outside any method declaration* are Object specific and are known as instance variables. They are called so because their values are instance specific and are ***not*** shared among instances.

```
1. class Page {
2. public String pageName;
3. // instance variable with public access
4. private int pageNumber;
5. // instance variable with private access
6. }
```

instance - they are created when a new instance is created, and they live until the instance is removed

CHECKLIST – Variable Scopes

JAVA FUNDAMENTALS

A *local variable* in Java is a variable that's declared within the body of a method. Then you can use the variable only within that method. Other methods in the class aren't even aware that the variable exists.

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

You may also declare local variables within blocks of code marked by braces. For example:

```
if (taxRate > 0)
{
    double taxAmount;
    taxAmount = subTotal * taxRate;
    total = subTotal + total;
}
```

local - they live as long as their method remains on the stack

CHECKLIST – Variable Scopes

JAVA FUNDAMENTALS

These variables are defined inside blocks of code (between { and }) and can be used while the block is executed; typical blocks of code are for, while, initialization block.

```
public static void main(String args[]) {
    {
       int a = 2; //a is known only here
    }             //a will be freed
    {
       int a = 3; //you can declare it again here
    }
}
```

block - live only as long as the code block is executing

CHECKLIST – Member Variables

JAVA FUNDAMENTALS

Member variables in a class — called *fields*.

```
public class Bicycle {
    private int cadence;
    private int gear;
    private int speed;
}
```

Variables in a method or block of code — called *local variables*.

Variables in method declarations – called *parameters*.

```
public static int methodName(int a, int b) {
   // body
}
```

CHECKLIST – Constructors

JAVA FUNDAMENTALS

Here's a statement that creates an object:

```
Account myAccount = new Account();
```

I know this works — I got it from one of my own examples in Chapter 7. Anyway, in Chapter 7, I say, "when the computer executes `new Account()`, you're creating an object by calling the `Account` class's constructor." What does this mean?

Well, when you ask the computer to create a new object, the computer responds by performing certain actions. For starters, the computer finds a place in its memory to store information about the new object. If the object has fields, the fields should eventually have meaningful values.

```
public class Bicycle {
    private int cadence;
    private int gear;
    private int speed;

    public Bicycle(int cadence, int gear, int speed) {
        this.cadence = cadence;
        this.gear = gear;
        this.speed = speed;
    }
}
```

```
Bicycle bike = new Bicycle(75, 2, 20);
```

CHECKLIST – Methods

JAVA FUNDAMENTALS

Methods are fundamental building blocks of Java programs. Each Java method is a collection of statements that are grouped together to perform an operation.

Method declarations have six components, in order:
1. Modifier - it defines the access type of the method and it is optional to use.
2. Return type - method may return a value.
3. Method name.
4. Parameter list in parenthesis - it is the type, order, and number of parameters of a method.
5. Method body - defines what the method does with the statements.

```
public int sum(int a, int b) {
    // return a + b;
}
```

```
void draw(String s) {
    // perform some draw functions
}
```

```
private boolean isNew() {
    // return true or false according to some rules
}
```

CHECKLIST – Packages && Imports

JAVA FUNDAMENTALS

Packages are used in Java in order

- to prevent naming conflicts
- to control access
- to make searching/locating and usage of classes, interfaces, enumerations and annotations easier, etc

```
package vehicle;

public class Bicycle {
    // class body
}
```

If you want to use a class from a package, you can refer to it by its full name (package name plus class name).
For example, java.util.Scanner refers to the Scanner class in the java.util package:

```
java.util.Scanner in = new java.util.Scanner(System.in);
```

Instead, you can import a name with an `import` statement:

```
import java.util.Scanner;
```

or import all classes from the `java.util` package

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

and then you can write:

```
Scanner in = new Scanner(System.in);
```

CHECKLIST – Static Fields, Methods and Imports

JAVA FUNDAMENTALS

Static keyword can be used with class, variable, method and block. Static members belong to the class instead of a specific instance, this means if you make a member static, you can access it without object. Let's take an example to understand this:

Here we have a static method `myMethod()`, we can call this method without any object because when we make a member static it becomes class level. If we remove the static keyword and make it non-static then we must need to create an object of the class in order to call it.

> Static members are common for all the instances(objects) of the class but non-static members are separate for each instance of class.

Static variable initialization

1. Static variables are initialized when class is loaded.
2. Static variables are initialized before any object of that class is created.
3. Static variables are initialized before any static method of the class executes.

CHECKLIST – Static Fields, Methods and Imports

JAVA FUNDAMENTALS

Static Block

Static block is used for initializing the static variables.This block gets executed when the class is loaded in the memory. A class can have multiple Static blocks, which will execute in the same sequence in which they have been written into the program.

Example 1: Single static block

As you can see that both the static variables were intialized before we accessed them in the main method.

```
class JavaExample{
   static int num;
   static String mystr;
   static{
      num = 97;
      mystr = "Static keyword in Java";
   }
   public static void main(String args[])
   {
      System.out.println("Value of num: "+num);
      System.out.println("Value of mystr: "+mystr);
   }
}
```

Output:

```
Value of num: 97
Value of mystr: Static keyword in Java
```

CHECKLIST – Static Fields, Methods and Imports

JAVA FUNDAMENTALS

Java Static Variables

A static variable is common to all the instances (or objects) of the class because it is a class level variable. In other words you can say that only a single copy of static variable is created and shared among all the instances of the class. Memory allocation for such variables only happens once when the class is loaded in the memory.
Few Important Points:

- Static variables are also known as Class Variables.
- Unlike **non-static variables**, such variables can be accessed directly in static and non-static methods.

Example 1: Static variables can be accessed directly in Static method

Here we have a static method `disp()` and two static variables `var1` and `var2`. Both the variables are accessed directly in the static method.

```
class JavaExample3{
  static int var1;
  static String var2;
  //This is a Static Method
  static void disp(){
      System.out.println("Var1 is: "+var1);
      System.out.println("Var2 is: "+var2);
  }
  public static void main(String args[])
  {
      disp();
  }
}
```

Output:

```
Var1 is: 0
Var2 is: null
```

CHECKLIST – Static Fields, Methods and Imports

JAVA FUNDAMENTALS

Example 2: Static variables are shared among all the instances of class

In this example, String variable is non-static and integer variable is Static. As you can see in the output that the non-static variable is different for both the objects but the static variable is shared among them, thats the reason the changes made to the static variable by object `ob2` reflects in both the objects.

```
class JavaExample{
   //Static integer variable
   static int var1=77;
   //non-static string variable
   String var2;

   public static void main(String args[])
   {
        JavaExample ob1 = new JavaExample();
        JavaExample ob2 = new JavaExample();
        /* static variables can be accessed directly without
         * any instances. Just to demonstrate that static variables
         * are shared, I am accessing them using objects so that
         * we can check that the changes made to static variables
         * by one object, reflects when we access them using other
         * objects
         */
        //Assigning the value to static variable using object ob1
        ob1.var1=88;
        ob1.var2="I'm Object1";
        /* This will overwrite the value of var1 because var1 has a single
         * copy shared among both the objects.
         */
        ob2.var1=99;
        ob2.var2="I'm Object2";
        System.out.println("ob1 integer:"+ob1.var1);
        System.out.println("ob1 String:"+ob1.var2);
        System.out.println("ob2 integer:"+ob2.var1);
        System.out.println("ob2 STring:"+ob2.var2);
   }
}
```

```
ob1 integer:99
ob1 String:I'm Object1
ob2 integer:99
ob2 STring:I'm Object2
```

CHECKLIST – Static Fields, Methods and Imports

JAVA FUNDAMENTALS

Java Static Methods

Static Methods can access class variables(static variables) without using object(instance) of the class, however non-static methods and non-static variables can only be accessed using objects. Static methods can be accessed directly in static and non-static methods.

Syntax:

Static keyword followed by return type, followed by method name.

```
static return_type method_name();
```

Example 1: static method main is accessing static variables without object

```
class JavaExample{
   static int i = 10;
   static String s = "Beginnersbook";
   //This is a static method
   public static void main(String args[])
   {
       System.out.println("i:"+i);
       System.out.println("s:"+s);
   }
}
```

```
i:10
s:Beginnersbook
```

CHECKLIST – Static Fields, Methods and Imports

JAVA FUNDAMENTALS

Static Class

A class can be made **static** only if it is a nested class.

1. Nested static class doesn't need reference of Outer class
2. A static class cannot access non-static members of the Outer class

We will see these two points with the help of an example:

Static class Example

```
class JavaExample{
   private static String str = "BeginnersBook";

   //Static class
   static class MyNestedClass{
        //non-static method
        public void disp() {

           /* If you make the str variable of outer class
            * non-static then you will get compilation error
            * because: a nested static class cannot access non-
            * static members of the outer class.
            */
           System.out.println(str);
        }

   }
   public static void main(String args[])
   {
       /* To create instance of nested class we didn't need the outer
        * class instance but for a regular nested class you would need
        * to create an instance of outer class first
        */
        JavaExample.MyNestedClass obj = new JavaExample.MyNestedClass();
        obj.disp();
   }
}
```

CHECKLIST –

JAVA FUNDAMENTALS

SPECIAL METHOD

Used to start a Java application

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

or

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

ARGUMENTS VS PARAMETERS

- **arguments** - things you specify between the parentheses when you're invoking a method

```
go("abc", 12);  // "abc" and 12 are arguments
```

- **parameters** - things in the method's signature that indicate what the method must receive when it's invoked

```
void go(String s, int n) {}  // s and n are parameters
```

VARARGS

- Varargs allows the method to accept zero or muliple arguments
- There can be only one variable argument in a method
- Variable argument (varargs) must be the last argument

VARARGS

- We use three dots (...) in the method signature to make it accept variable arguments
- We don't have to provide overloaded methods so less code
- In fact varargs parameter behaves like an array of the specified type

```
int sum(int... elements) {
    int result = 0;
    for (int i : e) {
        result += i;
    }
    return result;
}

System.out.println(sum(1, 2, 3, 4);  // 10
System.out.println(sum(1);           // 1
System.out.println(sum();            // 0
```

CHECKLIST – Regular Expression (Regex)

JAVA FUNDAMENTALS

- A regular expression defines a search pattern for strings
- The search pattern can be anything from a simple character, a fixed string or a complex expression containing special characters describing the pattern
- The pattern defined by the regex may match one or several times or not at all for a given string
- Regular expressions can be used to search, edit and manipulate text

```
/**
 * \d Any digits, short of [0-9]
 * \D Any non-digit, short for [^0-9]
 * \s Any whitespace character, short for [\t\n\x0B\f\r]
 * \S Any non-whitespace character, short for [^\s]
 * \w Any word character, short for [a-zA-Z_0-9]
 * \W Any non-word character, short for [^\w]
 */
String s1="My name is Bond. James Bond.";
String replaceString=s1.replaceAll( regex: "\\s", replacement: "");
System.out.println(replaceString);

String s2="My name is Bond James Bond.?,!;:--([])([])'\"' '. . .***";
String replaceString1=s2.replaceAll( regex: "\\p{Punct}", replacement: "");
System.out.println(replaceString1);

String s3="My name is Bond James Bond.?,!;:--([])([])'\"' '. . .***1.23";
String replaceString2=s3.replaceAll( regex: "[^\\d.]", replacement: "");
System.out.println(replaceString2);

String s4="-lasdasd.23asdasd";
String digits = s4.replaceAll( regex: "[^0-9.-]",  replacement: "");
System.out.println(digits);
```

CHECKLIST – Regular Expression (Regex)

JAVA FUNDAMENTALS

Character classes

`[abc]`	matches **a** or **b**, or **c**.
`[^abc]`	negation, matches everything except **a**, **b**, or **c**.
`[a-c]`	range, matches **a** or **b**, or **c**.
`[a-c[f-h]]`	union, matches **a**, **b**, **c**, **f**, **g**, **h**.
`[a-c&&[b-c]]`	intersection, matches **b** or **c**.
`[a-c&&[^b-c]]`	subtraction, matches **a**.

Predefined character classes

`.`	Any character.
`\d`	A digit: `[0-9]`
`\D`	A non-digit: `[^0-9]`
`\s`	A whitespace character: `[ \t\n\x0B\f\r]`
`\S`	A non-whitespace character: `[^\s]`
`\w`	A word character: `[a-zA-Z_0-9]`
`\W`	A non-word character: `[^\w]`

Boundary matches

`^`	The beginning of a line.
`$`	The end of a line.
`\b`	A word boundary.
`\B`	A non-word boundary.
`\A`	The beginning of the input.
`\G`	The end of the previous match.
`\Z`	The end of the input but for the final terminator, if any.
`\z`	The end of the input.

Pattern flags

`Pattern.CASE_INSENSITIVE` - enables case-insensitive matching.

`Pattern.COMMENTS` - whitespace and comments starting with **#** are ignored until the end of a line.

`Pattern.MULTILINE` - one expression can match multiple lines.

`Pattern.UNIX_LINES` - only the '**\n**' line terminator is recognized in the behavior of **.**, **^**, and **$**.

Useful Java classes & methods

PATTERN

A pattern is a compiler representation of a regular expression.

`Pattern compile(String regex)`
Compiles the given regular expression into a pattern.

`Pattern compile(String regex, int flags)`
Compiles the given regular expression into a pattern with the given flags.

`boolean matches(String regex)`
Tells whether or not this string matches the given regular expression.

`String[] split(CharSequence input)`
Splits the given input sequence around matches of this pattern.

`String quote(String s)`
Returns a literal pattern String for the specified String.

`Predicate<String> asPredicate()`
Creates a predicate which can be used to match a string.

MATCHER

An engine that performs match operations on a character sequence by interpreting a Pattern.

`boolean matches()`
Attempts to match the entire region against the pattern.

`boolean find()`
Attempts to find the next subsequence of the input sequence that matches the pattern.

`int start()`
Returns the start index of the previous match.

`int end()`
Returns the offset after the last character matched.