1.Java Building Blocks

- **Writing a *main()* Method**
    - A main() method is the gateway between the startup of a Java process, which is managed by the *Java Virtual Machine* (JVM), and the beginning of the programmer's code. The JVM calls on the underlying system to allocate memory and CPU time, access files, and so on.

- ## Redundant Imports
    - some imports that don't work:

```
import java.nio.*; // NO GOOD – a wildcard only matches
//dass names, not "file.*Files"
import java.nio.*.*; // NO GOOD – you can only have one wildcard
//and it must be at the end
import java.nio.files.Paths.*; // NO GOOD – you cannot import methods
//only dass names
```

- ## Reading and Writing Object Fields
    - Reading a variable is known as getting it and Writing to a variable is known as setting it.

Ex1:

```
int numberEggs = 1; //set variable
System.out.println(numberEggs); // read variable
```

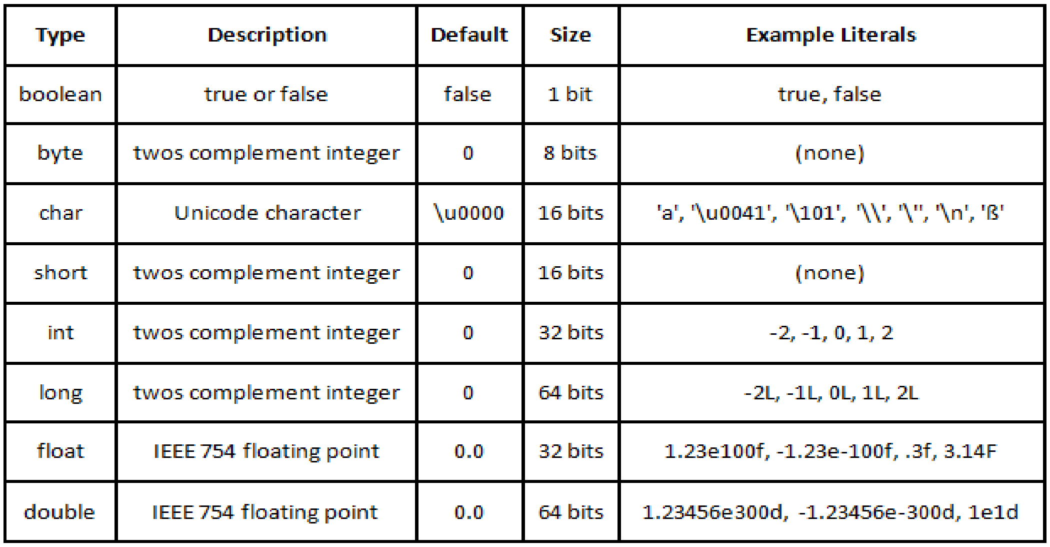
Ex2 :

```
String first = "Theodore";
String last = "Moose";
String full = first +last;//Reading and writing
```

- ## Order of Initialization
    - Fields and instance initializer blocks are run in the order in which they appear in the file.
    - The constructor runs after all fields and instance initializer blocks have run.
    - Order matters for the fields and blocks of code.

{ System.out.println(name); } // DOES NOT COMPILE
private String name = "Fluffy";

- ## Distinguishing Between Object References and Primitives
    - Java has eight built-in data types, referred to as the Java *primitive types*.



Type	Description	Default	Size	Example Literals
boolean	true or false	false	1 bit	true, false
byte	twos complement integer	0	8 bits	(none)
char	Unicode character	\u0000	16 bits	'a', '\u0041', '\101', '\\', '\'', '\n', 'ß'
short	twos complement integer	0	16 bits	(none)
int	twos complement integer	0	32 bits	-2, -1, 0, 1, 2
long	twos complement integer	0	64 bits	-2L, -1L, 0L, 1L, 2L
float	IEEE 754 floating point	0.0	32 bits	1.23e100f, -1.23e-100f, .3f, 3.14F
double	IEEE 754 floating point	0.0	64 bits	1.23456e300d, -1.23456e-300d, 1e1d

  - Examples:

    ```
    long max = 3123456789; // DOES NOT COMPILE
    long max = 3123456789L; // now Java knows it is a long
    ```

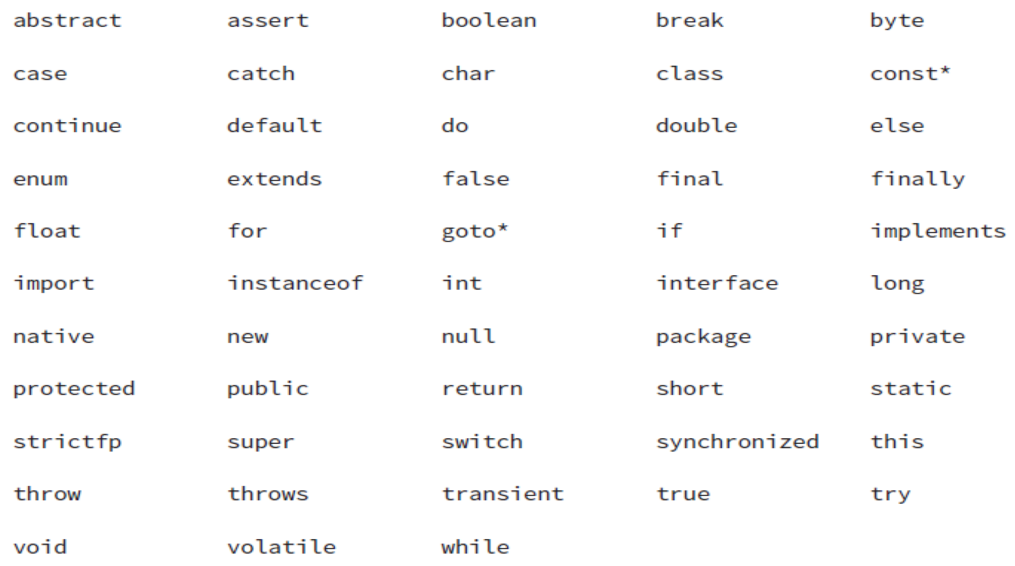
  - Java allows you to specify digits in several other formats:
    1. octal (digits 0–7), which uses the number 0 as a prefix—for example, 017
    2. hexadecimal (digits 0–9 and letters A–F), which uses the number 0 followed by x or X as a prefix—for example, 0xFF
    3. binary (digits 0–1), which uses the number 0 followed by b or B as a prefix—for example, 0b10
  - Examples:

    ```
    System.out.println(56); // 56
    System.out.println(0b11); // 3
    System.out.println(017); // 15
    System.out.println(0x1F); // 31
    ```

  - Underscore in literals. Examples;

    ```
    int million1 = 1000000;
    int million2 = 1_000_000;
    double notAtStart = _1000.00; // DOES NOT COMPILE
    double notAtEnd = 1000.00_; // DOES NOT COMPILE
    double notByDecimal = 1000_.00; // DOES NOT COMPILE
    double annoyingButLegal = 1_00_0.0_0; // this one compiles
    ```

- ## Reference Types
  - A *reference type* refers to an object (an instance of a class).
    - A reference can be assigned to another object of the same type.
    - A reference can be assigned to a new object using the new keyword.
- ## Key Differences
  - Reference types can be assigned null, Primitive types will give you a compiler error if you attempt to assign them null.
  - Reference types can be used to call methods when they do not point to null.
- ## Identifiers
  - There are only three rules to remember for legal identifiers:
    - The name must begin with a letter or the symbol $ or _.
    - Subsequent characters may also be numbers.
    - You cannot use the same name as a Java *reserved word*
  - List of Java keywords



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

- Examples:

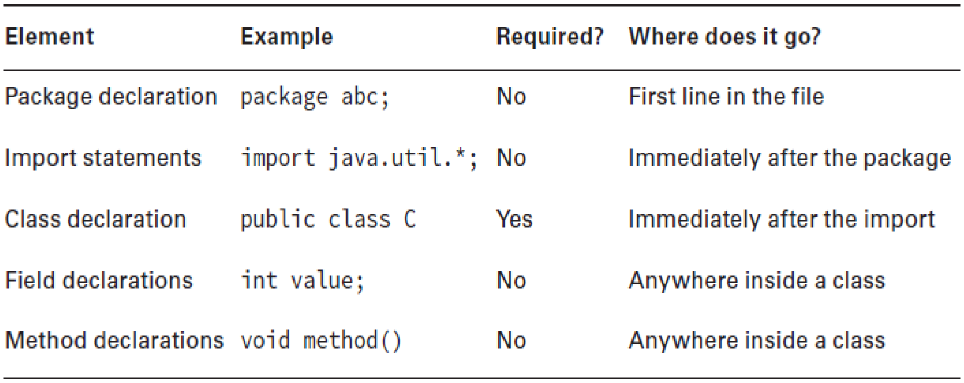
```
Okidentifier //legal
$OK2Identifier //legal
_alsoOK1d3ntifi3r //legal
_ _SStillOkbutKnotsonice$ //legal
3DPointClass // identifiers cannot begin with a number
hollywood@vine // @ is not a letter, digit, $ or _
*$coffee // * is not a letter, digit, $ or _
public // public is a reserved word
```

Understanding Default Initialization of Variables

- Local variables must be initialized before use. Compiler will not let you read an uninitialized value.
- Instance variables—in scope from declaration until object garbage collected.
- Class variables—in scope from declaration until program ends
- Tricky ex

```
public void findAnswer(boolean check) {
int answer;
int onlyOneBranch;
if (check) {
onlyOneBranch = 1;
answer = 1;
} else {
answer = 2;
}
System.out.println(answer);
System.out.println(onlyOneBranch); // DOES NOT COMPILE
}
```

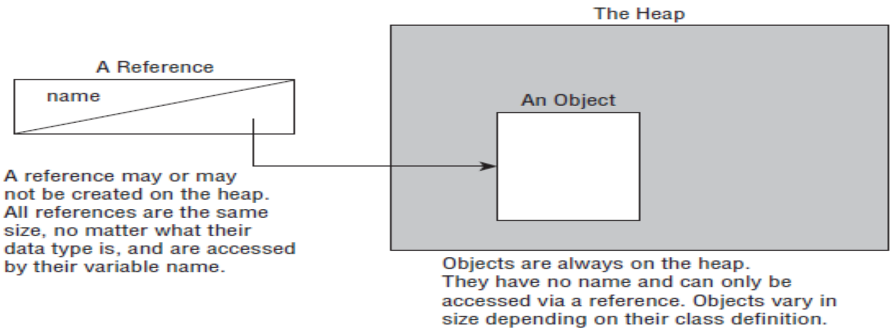
Ordering Elements in a Class



Element	Example	Required?	Where does it go?
Package declaration	`package abc;`	No	First line in the file
Import statements	`import java.util.*;`	No	Immediately after the package
Class declaration	`public class C`	Yes	Immediately after the import
Field declarations	`int value;`	No	Anywhere inside a class
Method declarations	`void method()`	No	Anywhere inside a class

- **Destroying Objects**
  - Garbage Collection
    Garbage collection refers to the process of automatically freeing memory on the heap by deleting objects that are no longer reachable in your program.
  - Java provides a method called System.gc(), System.gc() is not guaranteed to run.
  - An object is no longer reachable when one of two situations occurs:
    1. The object no longer has any references pointing to it.
    2. All references to the object have gone out of scope.

- **Objects vs. References**
  - All objects in **Java** are **stored** on the heap. The "**variables**" that hold **references** to them can be on the stack or they can be contained in other objects (then they are not really **variables**, but fields), which puts them on the heap also. The Class objects that define Classes are also heap objects.


The Heap
A Reference
name
An Object
A reference may or may not be created on the heap. All references are the same size, no matter what their data type is, and are accessed by their variable name.
Objects are always on the heap. They have no name and can only be accessed via a reference. Objects vary in size depending on their class definition.

- ***finalize()***
  - Java allows objects to implement a method called finalize() that might get called.
  - This method gets called if the garbage collector tries to collect the object. If the garbage collector doesn't run, the method doesn't get called.
  - For the exam, you need to know that this finalize() call could run zero one time.
- **Benefits of Java**
  - Object Oriented
  - Encapsulation
  - Platform Independent
  - Robust
  - Simple
  - Secure

****

2. Operators and Statements

Operators andStatements

- Three flavors of operators are available in Java: unary, binary, and ternary.
- Java operators are not necessarily evaluated from left-to-right order.
- If two operators have the same level of precedence, then Java guarantees left-to-right evaluation
- Table for Order of operator precedence

Operator	Symbols and examples
Post-unary operators	expression++, expression--
Pre-unary operators	++expression, --expression
Other unary operators	+, -, !
Multiplication/Division/Modulus	*, /, %
Addition/Subtraction	+, -
Shift operators	<<, >>, >>>
Relational operators	<, >, <=, >=, instanceof
Equal to/not equal to	==, !=
Logical operators	&, ^, |
Short-circuit logical operators	&&, ||
Ternary operators	boolean expression ? expression1 : expression2
Assignment operators	=, +=, -=, *=, /=, %=, &=, ^=, !=, <<=, >>=, >>>=

- Operators Detail
  - **>> (Signed right shift)**
  - **<< (Signed left shift)**
  - **>>> (UnSigned right shift)**→Number is stored using 32 bit 2's complement form before shift, For ex. Binary representation of -1 is all 1s (111..1).

- ### Arithmetic Operators
  - **Numeric Promotion**

  Primitive Datatype of Result = max(int, TypeOF val1, TypeOF val2,…)
  - **Unary Operators**

Unary operator	Description
+	Indicates a number is positive, although numbers are assumed to be positive in Java unless accompanied by a negative unary operator
-	Indicates a literal number is negative or negates an expression
++	Increments a value by 1
--	Decrements a value by 1
!	Inverts a Boolean's logical value

  - Example

    int x = 3;

    int y = ++x * 5 / x-- + --x;

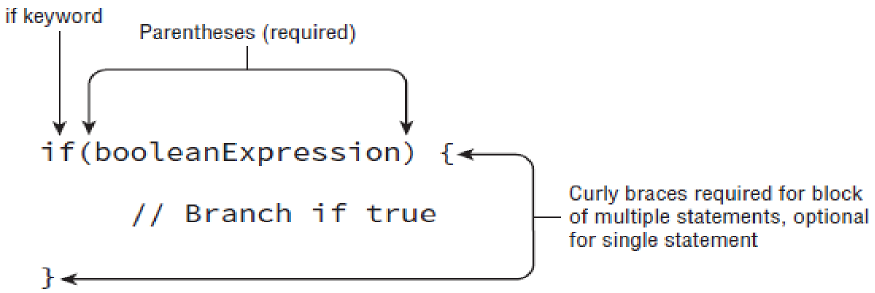
    Ans: x=2, y=7
  - Compound Assignment Operators (+=and -=)
  - Relational Operators (<, <=, >, >=)
  - Relational instanceofoperator

    a instanceof b True if the reference that a points to is an instance of a class, subclass, or class that implements a particular interface, as named in b.

- The *logical operators*, (&), (|), and (^), may be applied to both numeric and boolean data types(For Numeric they are referred to as *logical operators whereas for Boolean* referred to as *bitwise operators*.)
- conditional operators, &&and ||, which are often referred to as short-circuit operators identical to the logical operators,& and |**(except that the right-hand side of the expression may never be evaluated if the final result can be determined by the left-hand side of the expression)**
- A more common example of where short-circuit operators are used is checking for nullobjects before performing an operation,
  if(x != null &&x.getValue() < 5) {
  // Do something}
- Equality operators (== and !=) are used in one of three scenarios:
  - Comparing two numeric primitive types.
  - Comparing two boolean values.
  - Comparing two objects, including null and String values.

Understanding Java Statements

- ***if-then*** **Statement**



if keyword
Parentheses (required)
Curly braces required for block of multiple statements, optional for single statement

```
if(booleanExpression) {
      // Branch if true
}
```

- ***if-then-else*** **Statement**
  - **Verifying the *if* Statement Evaluates to a Boolean Expression**

EX1:
```
int x = 1;
if(x) { // DOES NOT COMPILE
...
}
```

Ex2:
```
int x = 1;
if(x = 5) { // DOES NOT COMPILE
...
}
```

- <u>**Ternary Operator**</u>

  Syntax: booleanExpression ? expression$_1$: expression$_2$ → **Either expression1 or expression2 is executed not both.**

  - There is no requirement that second and third expressions in ternary operations have the same data types

  Example:
  ```
  System.out.println((y > 5) ? 21 : "Zebra");
  int animal = (y < 91) ? 9 : "Horse"; // DOES NOT COMPILE
  ```

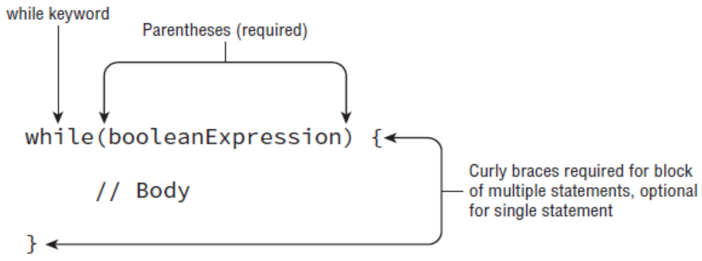
- ***switch*** **Statement**
  - Data types supported by switch statements include the following:
    - int and Integer
    - byte and Byte
    - short and Short
    - char and Character
    - String
    - enum values

- ### Compile-time Constant Values
  - The values in each case statement must be compile-time constant values of the same data type as the switch value. This means you can use only literals, enum constants, or final constant variables of the same data type.
  - *The exam creators are fond of* switch *examples that are missing* break *statements!*
  - *Example:*

    ```
    private int getSortOrder(String firstName, final String lastName) {
    String middleName = "Patricia";
    final String suffix = "JR";
    int id = 0;
    switch(firstName) {
    case "Test":
    return 52;
    case middleName: // DOES NOT COMPILE
    id = 5;
    break;
    case suffix:
    id = 0;
    break;
    case lastName: // DOES NOT COMPILE
    id = 8;
    break;
    case 5: // DOES NOT COMPILE
    id = 7;
    break;
    case 'J': // DOES NOT COMPILE
    id = 10;
    break;
    case java.time.DayOfWeek.SUNDAY: // DOES NOT COMPILE
    id=15;
    break;
    }
    return id;
    }
    ```

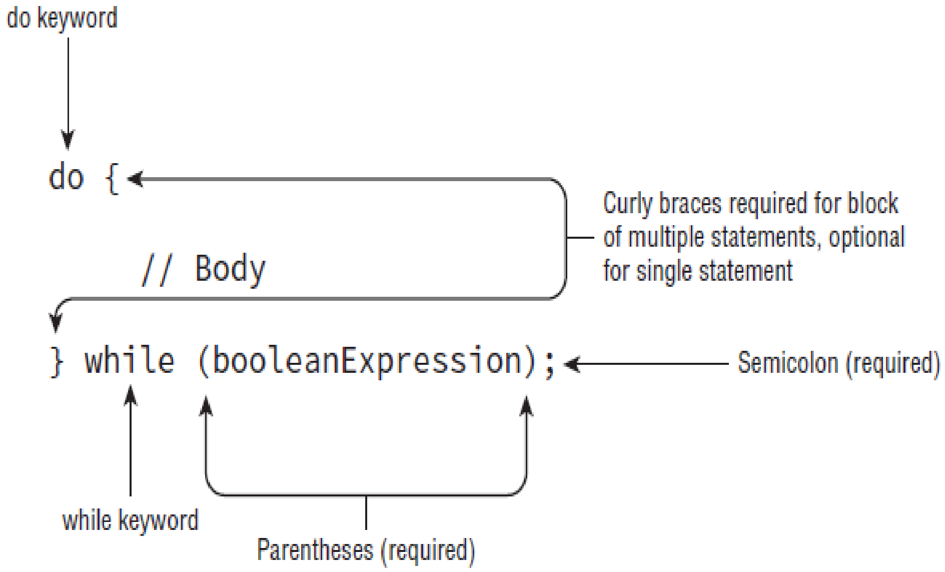
- ## <u>The *while* Statement</u>

FIGURE 2.5 The structure of a while statement


while keyword
Parentheses (required)
Curly braces required for block of multiple statements, optional for single statement

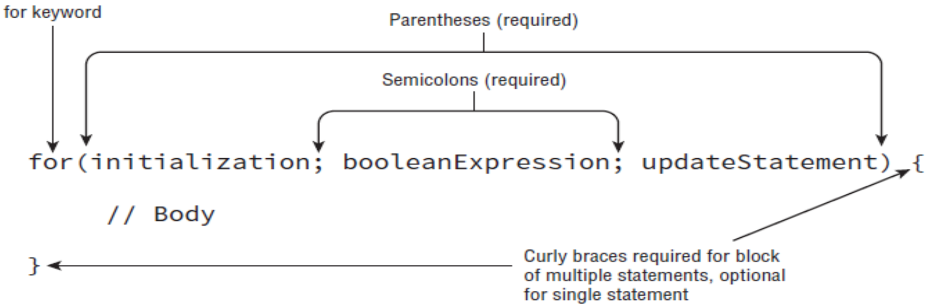
```
while(booleanExpression) {
    // Body
}
```

- **The *do-while* Statement**


do keyword
do {
// Body
} while (booleanExpression);
Curly braces required for block of multiple statements, optional for single statement
Semicolon (required)
while keyword
Parentheses (required)

- **The *for* Statement**

FIGURE 2.7 The structure of a basic for statement


for keyword
Parentheses (required)
Semicolons (required)
for(initialization; booleanExpression; updateStatement) {
// Body
}
Curly braces required for block of multiple statements, optional for single statement

1. Initialization statement executes
2. If booleanExpression is true continue, else exit loop
3. Body executes
4. Execute updateStatements
5. Return to Step 2

○ <u>Examples:</u>

1. **Creating an Infinite Loop**

```
for( ; ; ) {
System.out.println("Hello World");
}
```

2. **Adding Multiple Terms to the for Statement**

```
int x = 0;
for(long y = 0, z = 4; x < 5 && y < 10; x++, y++) {
System.out.print(y + " ");
}
System.out.print(x);
```

3. **Redeclaring a Variable in the Initialization Block**

```
int x = 0;
for(long y = 0, x = 4; x < 5 && y < 10; x++, y++) { // DOES NOT COMPILE
System.out.print(x + " ");
          }
```

4. **Using Incompatible Data Types in the Initialization Block**

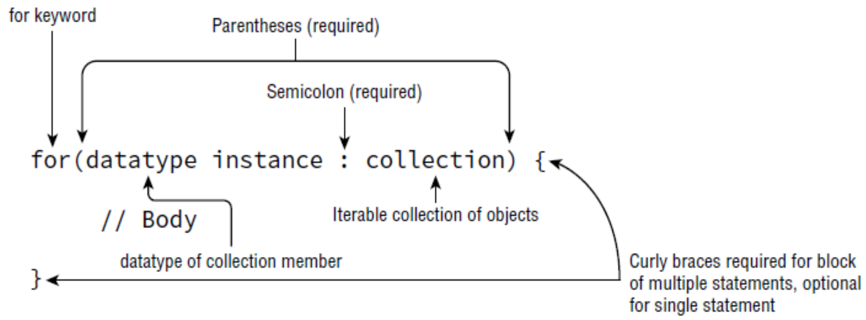
```
for(long y = 0, int x = 4; x < 5 && y<10; x++, y++) { // DOES NOT COMPILE
System.out.print(x + " ");
}
```

5. **Using Loop Variables Outside the Loop**

```
for(long y = 0, x = 4; x < 5 && y < 10; x++, y++) {
System.out.print(y + " ");
}
System.out.print(x); // DOES NOT COMPILE
```

- ***for-each*** **Statement**

FIGURE 2.8 The structure of an enhancement for statement


for keyword
Parentheses (required)
Semicolon (required)
for(datatype instance : collection) {
// Body
}
Iterable collection of objects
datatype of collection member
Curly braces required for block of multiple statements, optional for single statement

Understanding Advanced Flow Control

- **Nested Loops**

```
int[][] myComplexArray = {{5,2,1,3},{3,9,8,9},{5,7,12,7}};
for(int[] mySimpleArray : myComplexArray) {
for(int i=0; i<mySimpleArray.length; i++) {
System.out.print(mySimpleArray[i]+"\t");
}
System.out.println();
}
```

Output:

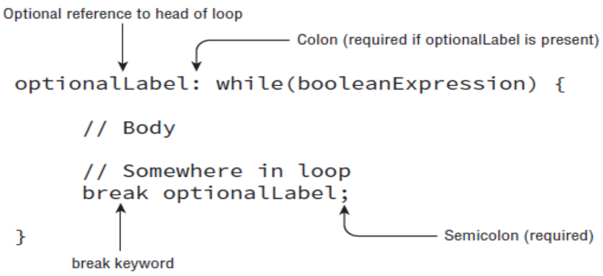
```
5        2        1        3
3        9        8        9
5        7        12       7
```

- **Adding Optional Labels**
  - if-then statements, switch statements, and loops can all have optional labels. It is a single word that is proceeded by a colon (:). For example, we can add optional labels to one of the previous examples:

```
int[][] myComplexArray = {{5,2,1,3},{3,9,8,9},{5,7,12,7}};
OUTER_LOOP: for(int[] mySimpleArray : myComplexArray) {
INNER_LOOP: for(int i=0; i<mySimpleArray.length; i++) {
System.out.print(mySimpleArray[i]+"\t");
}
System.out.println();
}
```

- ## The *break* Statement
  - A *break* statement transfers the flow of control out to the enclosing statement.

FIGURE 2.9 The structure of a break statement

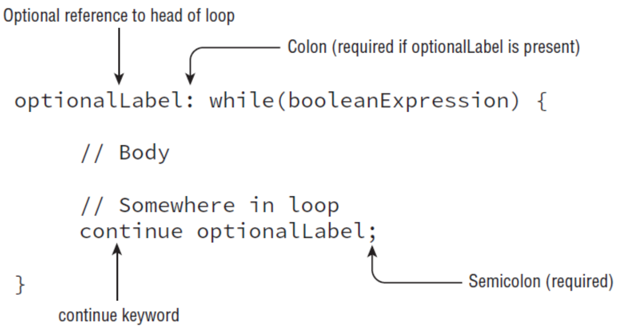


```
optionalLabel: while(booleanExpression) {

     // Body

     // Somewhere in loop
     break optionalLabel;

}
```

Optional reference to head of loop
Colon (required if optionalLabel is present)
Semicolon (required)
break keyword

- ## The *continue* Statement
  - *continue* statement, a statement that causes flow to finish the execution of the current loop.

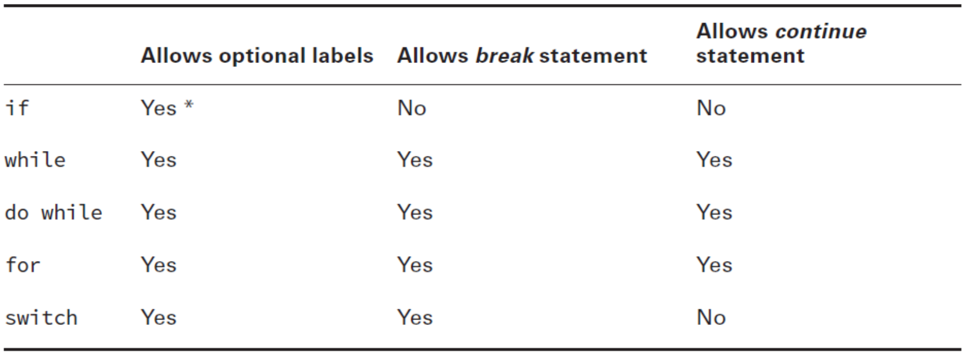
FIGURE 2.10 The structure of a continue statement



```
optionalLabel: while(booleanExpression) {

     // Body

     // Somewhere in loop
     continue optionalLabel;

}
```

Optional reference to head of loop
Colon (required if optionalLabel is present)
Semicolon (required)
continue keyword

TABLE 2.5 Advanced flow control usage



	Allows optional labels	Allows ***break*** statement	Allows ***continue*** statement
`if`	Yes *	No	No
`while`	Yes	Yes	Yes
`do while`	Yes	Yes	Yes
`for`	Yes	Yes	Yes
`switch`	Yes	Yes	No

* Labels are allowed for any block statement, including those that are preceded with an `if-then` statement.

****

3. Core Java APIs

Creating and Manipulating Strings

- Creating String
  String name = "Fluffy";
  String name = new String("Fluffy");
- **Concatenation**
  - + operatorcan be usedin two ways within thesame lineof code
    - i. If both operands are numeric, + means numericaddition.
    - ii. If either operand is a String, + means concatenation.
    - iii. The expression is evaluated left to right.

- **Immutability**
  - Example

```
final class Immutable {
private String s = "name";
public String getS() { return s; }
                    }
```

- **The String Pool**
  - The*string pool*, also known as the intern pool, is a location in the Java virtual machine (JVM)that collects all these strings.
  - Two imp scenario
    String name = "Fluffy";→Use the string pool normally
    String name = new String("Fluffy");→"No, JVM. I reallydon't want you to use the string pool. Please create a new object or me even thoughit isless efficient".

- **Important String Methods**
  - string is a sequence of charactersand Java counts from 0 when indexed.
    - i. ***length()***→returns the number of characters in the String.
    - ii. ***charAt()***→*find* out what character is at a specificindex.
    - iii. ***indexOf()***→looks at the characters in the string and finds the first index thatmatches the desired value. indexOfcan work with an individual character or a whole String as input.
      doesn't throw an exception if it can't find a match, instead itreturns -1.
      Examples:-

```
System.out.println(string.indexOf('a')); // 0
System.out.println(string.indexOf("al")); // 4
System.out.println(string.indexOf('a', 4)); // 4→start looking at char from index 4.
System.out.println(string.indexOf("al", 5)); // -1
```

    - iv. ***substring()***→looks for characters in a string.

```
intsubstring(intbeginIndex)
intsubstring(intbeginIndex(including), intendIndex(excluding))
```

    - v. ***toLowerCase()* and *toUpperCase()***
    - vi. ***equals()* and *equalsIgnoreCase()***
    - vii. ***startsWith()* and *endsWith()***
    - viii. ***contains()***
    - ix. ***replace()***
      String replace(char oldChar, char newChar)

String replace(CharSequenceoldChar, CharSequencenewChar)

x. ***trim() : public String trim()***

- **Method Chaining**

  Example:

  ```
  String result = "AniMaL".trim().toLowerCase().replace('a', 'A');//Animal
  ```

Using the *StringBuilder*Class

- The StringBuilderclasscreates a String withoutstoring all those interim String values
- Mutability and Chaining: When we chained String method calls, theresult was a new String with the answer. Instead, the StringBuilderchanges its own state and returns a reference to itself!
- Important example:

  ```
  4: StringBuilder a = new StringBuilder("abc");
  5: StringBuilder b = a.append("de");
  6: b = b.append("f").append("g");
  7: System.out.println("a=" + a); //abcdefg
  8: System.out.println("b=" + b);//abcdefg
  ```

- **Creating a *StringBuilder***

  ```
  StringBuilder sb1 = new StringBuilder();
  StringBuilder sb2 = new StringBuilder("animal");
  StringBuilder sb3 = new StringBuilder(10); //Default capacity is 16
  ```

- **Size vs. Capacity**→ Size is the number of characters currently in the sequence, and capacity is the numberof characters the sequence can currently hold.
- Java automatically increase the capacity of StringBuilder object when it is required(Size exceeding Capacity value).
- **Important *StringBuilder*Methods**
  - ***charAt()*, *indexOf()*, *length()*, and *substring()***

    Imp Example:

    ```
    StringBuildersb = new StringBuilder("animals");
    String sub = sb.substring(sb.indexOf("a"), sb.indexOf("al"));
    intlen = sb.length();
    char ch = sb.charAt(6);
    System.out.println(sub + " " + len + " " + ch);//anim 7 s
    ```

  - ***append() : StringBuilder append(String str)***
  - ***insert():StringBuilder insert(int offset, String str)***
  - ***delete()* and *deleteCharAt() :StringBuilder delete(int start, int end)***

    StringBuilderdeleteCharAt(int index)
  - ***reverse():StringBuilder reverse()***
  - ***toString() : String toString()***

- Understanding Equality
  - Imp Example
    String x = "Hello World";
    String z = " Hello World".trim();
    System.out.println(x == z); // false

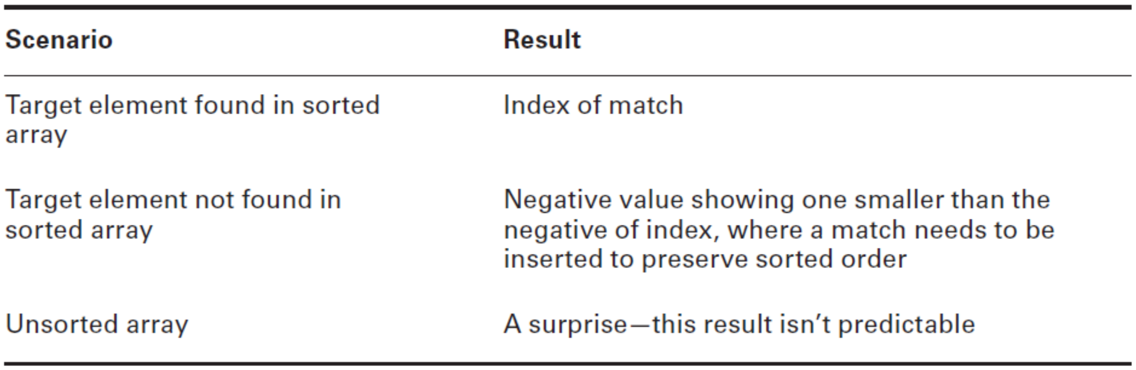
Understanding Java Arrays

- An array is an area of memory on the heap with space for a designated number of elements.
- Creating Array
  int[] numbers1 = new int[3];
  int[] numbers2 = new int[] {42, 55, 99};
  int[] numbers2 = {42, 55, 99};→This approach is called an anonymous array.
- Valid Array Declaration
  int[] numAnimals;
  int [] numAnimals2;
  int numAnimals3[];
  int numAnimals4 [];
- **Multiple "Arrays" in Declarations**
  int[] ids, types;→ Both var of type int[].
  int ids[], types;→one variable of type int[] and one variable of type int.

- **Creating an Array with Reference Variables**
  - We can call equals() because an array is an object.
  - The array does not allocate space for the String objects. Instead, it allocates space for a reference to where the objects are really stored.
  - Typecasting in Array
    3: String[] strings = { "stringValue" };
    4: Object[] objects = strings;
    5: String[] againStrings = (String[]) objects;
    6: againStrings[0] = new StringBuilder(); // DOES NOT COMPILE
    7: objects[0] = new StringBuilder(); // careful!
- **Sorting an Array**
  - Example
    int[] numbers = { 6, 9, 1 };
    Arrays.sort(numbers);
  - Imp example
    String[] strings = { "10", "9", "100" };
    Arrays.sort(strings);
    for (String string : strings)
    System.out.print(string + " ");//10 100 9.
  - String sorts in alphabetic order, and 1 sorts before 9.
  - Numbers sort before letters and uppercase sorts before lowercase, in case you were wondering.

- ## Searching an Array
    - Java provides a convenient way to search—but only if the array is already sorted.

TABLE 3.1 Binary search rules



Scenario	Result
Target element found in sorted array	Index of match
Target element not found in sorted array	Negative value showing one smaller than the negative of index, where a match needs to be inserted to preserve sorted order
Unsorted array	A surprise—this result isn't predictable

    - Example

```
3: int[] numbers = {2,4,6,8};
4: System.out.println(Arrays.binarySearch(numbers, 2)); // 0
5: System.out.println(Arrays.binarySearch(numbers, 4)); // 1
6: System.out.println(Arrays.binarySearch(numbers, 1)); // -1
7: System.out.println(Arrays.binarySearch(numbers, 3)); // -2
8: System.out.println(Arrays.binarySearch(numbers, 9)); // -5
```

- ## Varargs
    - Example

```
public static void main(String... args) // varargs
```

Multidimensional Arrays

- ### Creating a Multidimensional Array
    - **Examples**

```
Ex1: int[][] vars1; // 2D array
     int vars2 [][]; // 2D array
     int[] vars3[]; // 2D array
     int[] vars4 [], space [][]; // a 2D AND a 3D array

Ex2: String [][] rectangle = new String[3][2];

Ex3: rectangle[0][1] = "set";

Ex4: int[][] differentSize = {{1, 4}, {3}, {9,8,7}};

Ex5: int [][] args = new int[4][];
     args[0] = new int[5];
     args[1] = new int[3];
```

- ### Using a Multidimensional Array

Example:

```
int[][] twoD = new int[3][2];
for (int i = 0; i < twoD.length; i++) {
for (int j = 0; j < twoD[i].length; j++)
System.out.print(twoD[i][j] + " "); // print element
System.out.println(); // time for a new row
        }
```

OR

```
for (int[] inner : twoD) {
for (int num : inner)
System.out.print(num + " ");
System.out.println();
}
```

Understanding an *ArrayList*

- An ArrayList is an ordered sequence that allows duplicates.
- ArrayList implements toString() so you can easily see the
- contents just by printing it
- ## Creating an *ArrayList*
  - three ways to create an ArrayList:
    ```
    ArrayList list1 = new ArrayList();
    ArrayList list2 = new ArrayList(10);
    ArrayList list3 = new ArrayList(list2);
    ```
  - Java 5 introduced *generics*, which allow you to specify the type of class that the ArrayList will contain.
    ```
    ArrayList<String> list4 = new ArrayList<String>();
    ArrayList<String> list5 = new ArrayList<>();
    ```
  - < and > are diamond operator because <> looks like a diamond.
- ## Using an *ArrayList*
  - ***add() :*** boolean add(E element)
    void add(int index, E element)

    Imp Examples:
    ```
    List<String> birds = new ArrayList<>();
    birds.add("hawk"); // [hawk]
    birds.add(1, "robin"); // [hawk, robin]
    birds.add(0, "blue jay"); // [blue jay, hawk, robin]
    birds.add(1, "cardinal"); // [blue jay, cardinal, hawk, robin]
    System.out.println(birds); // [blue jay, cardinal, hawk, robin]
    ```
  - ***remove() :*** boolean remove(Object object)
    E remove(int index)
    - The E return type is the element that actually got removed.

    Imp Examples:
    ```
    List<String> birds = new ArrayList<>();
    birds.add("hawk"); // [hawk]
    birds.add("hawk"); // [hawk, hawk]
    System.out.println(birds.remove("cardinal")); // prints false
    System.out.println(birds.remove("hawk")); // prints true
    System.out.println(birds.remove(0)); // prints hawk
    System.out.println(birds); // []
    ```
  - ***set() :*** E set(int index, E newElement)
    - The set() method changes one of the elements of the ArrayList without changing the size.
    - The E return type is the element that got replaced.

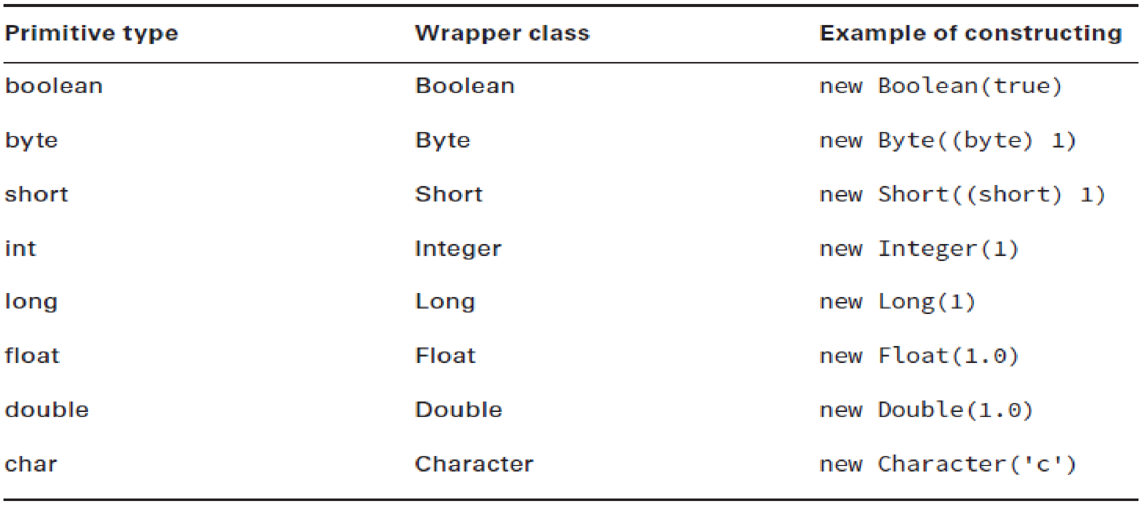
    Imp Examples:
    ```
    List<String> birds = new ArrayList<>();
    birds.add("hawk"); // [hawk]
    System.out.println(birds.size()); // 1
    birds.set(0, "robin"); // [robin]
    System.out.println(birds.size()); // 1
    birds.set(1, "robin"); // IndexOutOfBoundsException
    ```
  - ***isEmpty() and size() :*** boolean isEmpty(),int size()
  - ***clear():***void clear()
    - The clear() method provides an easy way to discard all elements of the ArrayList.

- ***contains():*** boolean contains(Object object)
    - The contains() method checks whether a certain value is in the ArrayList.
- ***equals():*** boolean equals(Object object)
    - ArrayList has a custom implementation of equals() so you can compare two lists to see if they contain the same elements in the same order.

Wrapper Classes

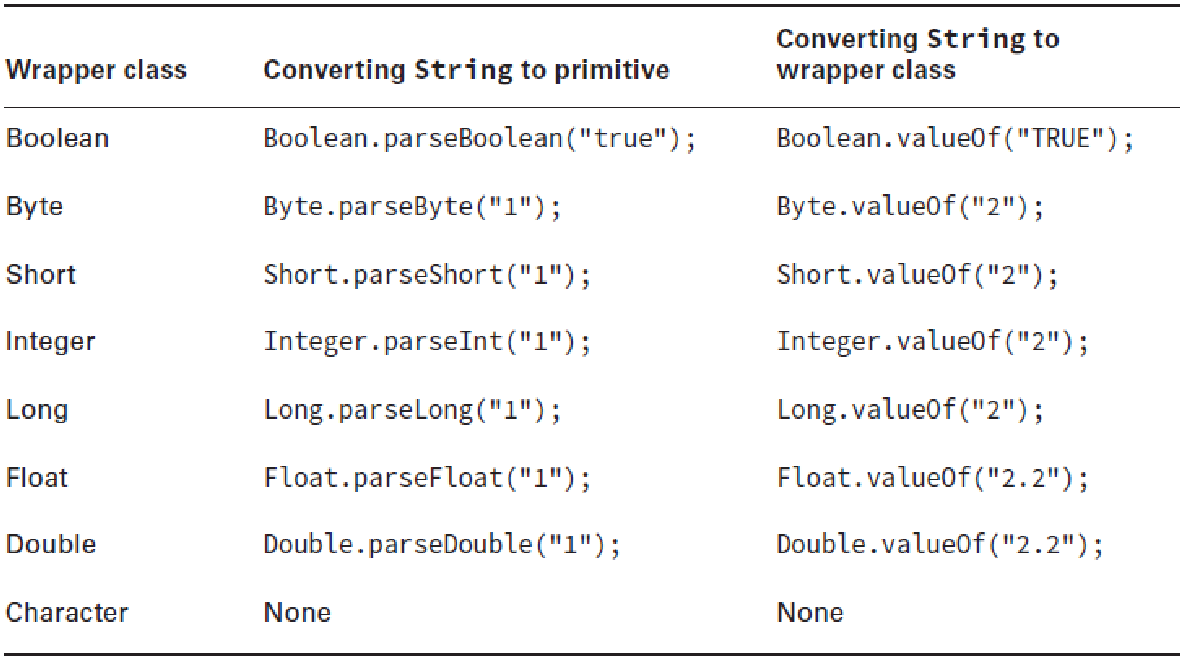
U

TABLE 3.2 Wrapper classes



Primitive type	Wrapper class	Example of constructing
boolean	Boolean	`new Boolean(true)`
byte	Byte	`new Byte((byte) 1)`
short	Short	`new Short((short) 1)`
int	Integer	`new Integer(1)`
long	Long	`new Long(1)`
float	Float	`new Float(1.0)`
double	Double	`new Double(1.0)`
char	Character	`new Character('c')`

TABLE 3.3 Converting from a String



Wrapper class	Converting String to primitive	Converting String to wrapper class
Boolean	`Boolean.parseBoolean("true");`	`Boolean.valueOf("TRUE");`
Byte	`Byte.parseByte("1");`	`Byte.valueOf("2");`
Short	`Short.parseShort("1");`	`Short.valueOf("2");`
Integer	`Integer.parseInt("1");`	`Integer.valueOf("2");`
Long	`Long.parseLong("1");`	`Long.valueOf("2");`
Float	`Float.parseFloat("1");`	`Float.valueOf("2.2");`
Double	`Double.parseDouble("1");`	`Double.valueOf("2.2");`
Character	None	None

Autoboxing

- you can just type the primitive value and Java will convert it to the relevant wrapper class for you. This is called *autoboxing*.
- Be careful when autoboxing into Integer.
  Tricky Example:
  ```
  List<Integer> numbers = new ArrayList<>();
  numbers.add(1);
  numbers.add(2);
  numbers.remove(1);
  System.out.println(numbers);// 1 → element with index 1 be removed not with value 1.
  ```

Converting Between *array* and *List*

- **ArrayList into an array:**
  ```
  List<String> list = new ArrayList<>();
  list.add("hawk");
  list.add("robin");
  Object[] objectArray = list.toArray();
  System.out.println(objectArray.length); // 2
  String[] stringArray = list.toArray(new String[0]);
  System.out.println(stringArray.length); // 2
  ```
- The only problem is that it defaults to an array of class Object.
- **Array to a List is more interesting.**
  - The original array and created array backed List are linked.
  - It is a fixed-size, backed version of a List. It updates both array and list because they point to the same data store.

    Example:
    ```
    20: String[] array = { "hawk", "robin" }; // [hawk, robin]
    21: List<String> list = Arrays.asList(array); // returns fixed size list
    22: System.out.println(list.size()); // 2
    23: list.set(1, "test"); // [hawk, test]
    24: array[0] = "new"; // [new, test]
    25: for (String b : array) System.out.print(b + " "); // new test
    26: list.remove(1); // throws UnsupportedOperation Exception
    ```
    - Line 26 throws an exception because we are not allowed to change the size of the list.

Sorting an ArrayList

- Example:
  ```
  List<Integer> numbers = new ArrayList<>();
  numbers.add(99);
  numbers.add(5);
  numbers.add(81);
  Collections.sort(numbers);
  System.out.println(numbers); [5, 81, 99]
  ```

Working with Dates and Times

- ## Creating Dates and Times

  LocalDate Contains just a date—no time and no time zone.
  LocalTime Contains just a time—no date and no time zone.
  LocalDateTime Contains both a date and time but no time zone.
  Examples:

  ```
  System.out.println(LocalDate.now()); // 2015-01-20
  System.out.println(LocalTime.now());//12:45:18.401
  System.out.println(LocalDateTime.now());//2015-01-20T12:45:18.401
  ```

  - ### Creating Date with no time

    ```
    LocalDate date1 = LocalDate.of(2015, Month.JANUARY, 20);
    LocalDate date2 = LocalDate.of(2015, 1, 20);
    Syntax:
    publicstatic LocalDate of(int year, int month, int dayOfMonth)
    publicstatic LocalDate of(int year, Month month, int dayOfMonth)
    ```

  - ### creating time with no date

    ```
    LocalTime time1 = LocalTime.of(6, 15); // hour and minute
    LocalTime time2 = LocalTime.of(6, 15, 30); // + seconds
    LocalTime time3 = LocalTime.of(6, 15, 30, 200); // + nanoseconds
    ```

  - ### Creating Data and Time

    ```
    LocalDateTime dateTime1 = LocalDateTime.of(2015, Month.JANUARY, 20, 6, 15, 30);
    LocalDateTime dateTime2 = LocalDateTime.of(date1, time1);
    ```

  - ### Different method signatures

```
publicstatic LocalDateTime of(int year, int month, int dayOfMonth, int hour, int minute)
publicstatic LocalDateTime of(int year, int month,int dayOfMonth, int hour, int minute, int second)
publicstatic LocalDateTime of(int year, int month,int dayOfMonth, int hour, int minute, int second,int nanos)
publicstatic LocalDateTime of(int year, Month month,int dayOfMonth, int hour, int minute)
publicstatic LocalDateTime of(int year, Month month,int dayOfMonth, int hour, int minute, int second)
publicstatic LocalDateTime of(int year, Month month,int dayOfMonth, int hour, int minute, int second,int nanos)
publicstatic LocalDateTime of(LocalDate date, LocalTime)
```

  - Doesn't Compile Example

    ```
    LocalDate d = new LocalDate(); // DOES NOT COMPILE
    LocalDate.of(2015, Month.JANUARY, 32) // throws DateTimeException
    ```

- ## Manipulating Dates and Times

  - Date and time classes are immutable, just like String.
  - Manupulating Dates

    ```
    12: LocalDate date = LocalDate.of(2014, Month.JANUARY, 20);
    13: System.out.println(date); // 2014-01-20
    14: date = date.plusDays(2);
    15: System.out.println(date); // 2014-01-22
    16: date = date.plusWeeks(1);
    17: System.out.println(date); // 2014-01-29
    18: date = date.plusMonths(1);
    19: System.out.println(date); // 2014-02-28
    20: date = date.plusYears(5);
    21: System.out.println(date); // 2019-02-28
    ```

  - **On line 18, we add a month. This would bring us to February 29, 2014. Java is smart enough to realize February 29, 2014 does not exist and gives us February 28, 2014 instead.**

- Manupulating LocalDateTime

  ```
  LocalDate date = LocalDate.of(2020, Month.JANUARY, 20);
  LocalTime time = LocalTime.of(5, 15);
  LocalDateTime dateTime = LocalDateTime.of(date, time);
  System.out.println(dateTime); // 2020-01-20T05:15
  dateTime = dateTime.minusDays(1);
  System.out.println(dateTime); // 2020-01-19T05:15
  dateTime = dateTime.minusHours(10);
  System.out.println(dateTime); // 2020-01-18T19:15
  dateTime = dateTime.minusSeconds(30);
  System.out.println(dateTime); // 2020-01-18T19:14:30
  ```
- Chaining Date and Time methods

  ```
  LocalDate date = LocalDate.of(2020, Month.JANUARY, 20);
  LocalTime time = LocalTime.of(5, 15);
  LocalDateTime dateTime = LocalDateTime.of(date2, time).minusDays(1).minusHours(10).minusSeconds(30);
  ```
- ***Tricky one***

  ```
  LocalDate date = LocalDate.of(2020, Month.JANUARY, 20);
  date.plusDays(10);
  System.out.println(date);// January 20, 2020.
  ```

Working with Periods

- LocalDate has toEpochDay(), which is the number of days since January 1, 1970.
- LocalDateTime has toEpochTime(), which is the number of seconds since January 1, 1970.
- LocalTime does not have an epoch method.
- There are five ways to create a Period class:

  ```
  Period annually = Period.ofYears(1); // every 1 year
  Period quarterly = Period.ofMonths(3); // every 3 months
  Period everyThreeWeeks = Period.ofWeeks(3); // every 3 weeks
  Period everyOtherDay = Period.ofDays(2); // every 2 days
  Period everyYearAndAWeek = Period.of(1, 0, 7); // every year and 7 days
  ```
- You cannot chain methods when creating a Period. Only last method is called. Becoz it makes other parameter zero. Refer ofMonths() body

  ```
  public static Period ofMonths(int months) {
      return create(0, months, 0);
    }
  ```

  Example:

  ```
  Period wrong = Period.ofYears(1).ofWeeks(1); // every week
  ```

  This tricky code is really like writing the following:

  ```
  Period wrong = Period.ofYears(1);
  wrong = Period.ofWeeks(7);
  ```
- Period is what objects it can be used with.

  ```
  LocalDate date = LocalDate.of(2015, 1, 20);
  LocalTime time = LocalTime.of(6, 15);
  LocalDateTime dateTime = LocalDateTime.of(date, time);
  Period period = Period.ofMonths(1);
  System.out.println(date.plus(period)); // 2015-02-20
  System.out.println(dateTime.plus(period)); // 2015-02-20T06:15
  System.out.println(time.plus(period)); // UnsupportedTemporalTypeException
  ```

Formatting Dates and Times(convert a date or time to a formatted String)

- The date and time classes support many methods to get data out of them:

  ```
  LocalDate date = LocalDate.of(2020, Month.JANUARY, 20);
  System.out.println(date.getDayOfWeek()); // MONDAY
  System.out.println(date.getMonth()); // JANUARY
  System.out.println(date.getYear()); // 2020
  System.out.println(date.getDayOfYear()); // 20
  ```

- Java provides a class called java.time.format.DateTimeFormatter to format Dates and Times.
- **<u>Examples:</u>**

```
LocalDate date = LocalDate.of(2020, Month.JANUARY, 20);
LocalTime time = LocalTime.of(11, 12, 34);
LocalDateTime dateTime = LocalDateTime.of(date, time);
System.out.println(date.format(DateTimeFormatter.ISO_LOCAL_DATE));// 2020-01-20
System.out.println(time.format(DateTimeFormatter.ISO_LOCAL_TIME));// 11:12:34
System.out.println(dateTime.format(DateTimeFormatter.ISO_LOCAL_DATE_TIME));// 2020-01-20T11:12:34
```

- Predefined formats

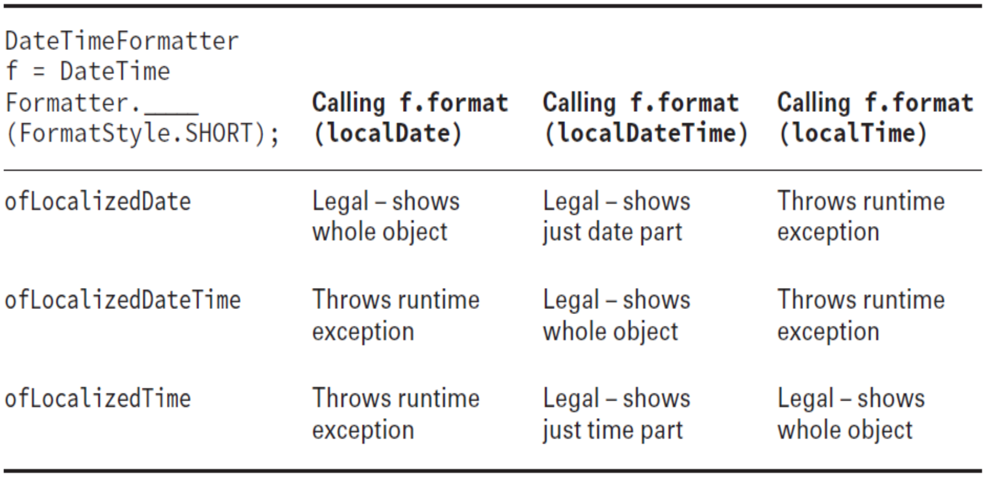
```
DateTimeFormatter shortDateTime =DateTimeFormatter.ofLocalizedDate(FormatStyle.SHORT);
System.out.println(shortDateTime.format(dateTime)); // 1/20/20
System.out.println(shortDateTime.format(date)); // 1/20/20
System.out.println(shortDateTime.format(time)); // UnsupportedTemporalTypeException
```

- The following statements print exactly the same thing as the previous code:

```
DateTimeFormatter shortDateTime =DateTimeFormatter.ofLocalizedDate(FormatStyle.SHORT);
System.out.println(dateTime.format(shortDateTime));
System.out.println(date.format(shortDateTime));
System.out.println(time.format(shortDateTime));
```

- Table 3.5 shows the legal and illegal localized formatting methods.

TABLE 3.5 ofLocalized methods



DateTimeFormatter f = DateTime Formatter.____ (FormatStyle.SHORT);	Calling f.format (localDate)	Calling f.format (localDateTime)	Calling f.format (localTime)
ofLocalizedDate	Legal – shows whole object	Legal – shows just date part	Throws runtime exception
ofLocalizedDateTime	Throws runtime exception	Legal – shows whole object	Throws runtime exception
ofLocalizedTime	Throws runtime exception	Legal – shows just time part	Legal – shows whole object

- There are two predefined formats that can show up on the exam: SHORT and MEDIUM.

```
LocalDate date = LocalDate.of(2020, Month.JANUARY, 20);
LocalTime time = LocalTime.of(11, 12, 34);
LocalDateTime dateTime = LocalDateTime.of(date, time);

DateTimeFormatter shortF = DateTimeFormatter.ofLocalizedDateTime(FormatStyle.SHORT);
DateTimeFormatter mediumF = DateTimeFormatter.ofLocalizedDateTime(FormatStyle.MEDIUM);
System.out.println(shortF.format(dateTime)); // 1/20/20 11:12 AM
System.out.println(mediumF.format(dateTime)); // Jan 20, 2020 11:12:34 AM
```

- If you don't want to use one of the predefined formats, you can create your own.

```
DateTimeFormatter f = DateTimeFormatter.ofPattern("MMMM dd, yyyy, hh:mm");
System.out.println(dateTime.format(f)); // January 20, 2020, 11:12
```

- **Tricky one**

4: DateTimeFormatter f = DateTimeFormatter.ofPattern("hh:mm");
5: f.format(dateTime);
6: f.format(date);//**throws exception**
7: f.format(time);

- **Parsing Dates and Times(String to a date or time)**
  - Just like the format() method, the parse() method takes a formatter as well.
  - Examples:

```
DateTimeFormatter f = DateTimeFormatter.ofPattern("MM dd yyyy");
LocalDate date = LocalDate.parse("01 02 2015", f);
LocalTime time = LocalTime.parse("11:22");
System.out.println(date); // 2015-01-02
System.out.println(time); // 11:22
```

****

4.Methods and Encapsulation

Access Modifiers

- Java offers four choices of access modifier:
    - ***public*** The method can be called from any class.
    - ***private*** The method can only be called from within the same class.
    - ***protected*** The method can only be called from classes in the same package or subclasses.
    - ***Default (Package Private) Access*** The method can only be called from classes in the same package. This one is tricky because there is no keyword for default access. You simply omit the access modifier.

Working with Varargs

- A vararg parameter must be the last element in a method's parameter list.
- you are only allowed to have one vararg parameter per method.
- Java will create an empty array if no parameters are passed for a vararg.
- It is still possible to pass null explicitly to varargs, Java treats it as an array reference that happens to be null.

Designing Static Methods and Fields

- main() can be called just like any other static method.
- Regular imports are for importing classes. Static imports are for importing static members of classes, you can use a wildcard or import a specific member.

Overloading Methods

- *Method overloading* occurs whenthere are different methodsignatures with the same name but different type parameters and different no of parameters.
- public void fly(int[] lengths) { }and public void fly(int... lengths) { } are treated same by compiler so can not be compile for method overloading.
- We can call either method by passing an array: fly(new int[] { 1, 2, 3 });
- But can only call the varargs version withstand-alone parameters:fly(1, 2, 3);

Creating Constructors

- A constructor is typically used to initialize instance variables.

Order of Initialization

1. If there is a superclass, initialize it first .
2. Static variable declarations and static initializers in the order they appear in the file.
3. Instance variable declarations and instance initializers in the order they appear in the file.
4. The constructor.

Encapsulating Data

- Encapsulation means we set up the class so only methods in the class with the variables can refer to the instance variables.

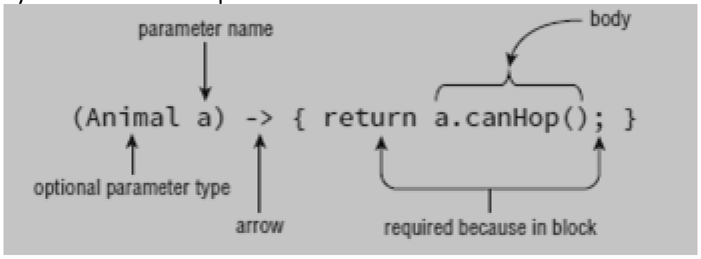
Creating Immutable Classes

-
  ```
  public class ImmutableSwan
   {
              private int numberEggs;
              public ImmutableSwan(int numberEggs) {
              this.numberEggs = numberEggs;
              }
              public int getNumberEggs() {
              return numberEggs;
              }
  }
  ```

- Defensive copy of mutable class
  ```
  public Mutable(StringBuilder b) {
  builder = new StringBuilder(b);
  }
  public StringBuildergetBuilder() {
  return new StringBuilder(builder);
              }
  ```
- Another approach for the getter is to return an immutable object:
  ```
  public String getValue() {
  return builder.toString();
  }
  ```

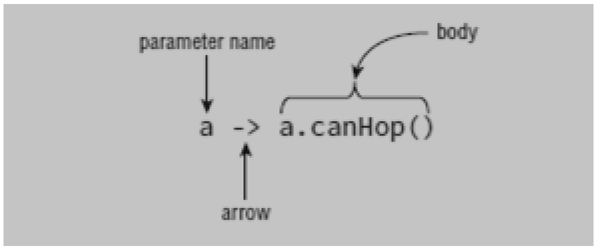
Writing Simple Lambdas

- A *lambda expression* is a block of code that gets passed around. You can think of a lambda expression as an*continued* anonymous method.
- In other words, a lambda expression is like a method that you can pass as if it were a variable.
- Syntax of Lambda expression



```
(Animal a) -> { return a.canHop(); }
```

Or



```
a -> a.canHop()
```

- The parentheses can only be omitted if there is a single parameter and its type is notexplicitly stated.

- Examples

  ```
  3: print(() -> true); // 0 parameters
  4: print(a ->a.startsWith("test")); // 1 parameter
  5: print((String a) ->a.startsWith("test")); // 1 parameter
  6: print((a, b) ->a.startsWith("test")); // 2 parameters
  7: print((String a, String b) ->a.startsWith("test")); // 2 parameters
  ```
- Java 8 even integrated the Predicate interface into some existing classes. There is only one you need to know for the exam. ArrayListdeclares a removeIf() method that takes a Predicate.
- Example

  ```
  List<String> bunnies = new ArrayList<>();
  4: bunnies.add("long ear");
  5: bunnies.add("floppy");
  6: bunnies.add("hoppy");
  7: System.out.println(bunnies); // [long ear, floppy, hoppy]
  8: bunnies.removeIf(s ->s.charAt(0) != 'h');
  9: System.out.println(bunnies); // [hoppy]
  ```

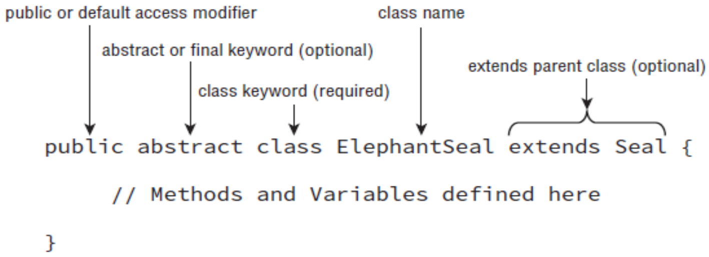
****

5. Class Design

Introducing Class Inheritance

- *Inheritance* is the process by which the new child subclass automatically includes any public or protected primitives, objects, or methods defined in the parent class.
- **Extending a Class**

FIGURE 5.2 Defining and extending a class


public or default access modifier
abstract or final keyword (optional)
class keyword (required)
class name
extends parent class (optional)

```
public abstract class ElephantSeal extends Seal {

    // Methods and Variables defined here

}
```

- **Applying Class Access Modifiers**
  - public and default are the only ones that can be applied to top-level classes within a Java file.
  - The protected and private modifiers can only be applied to inner classes, which are classes that are defined within other classes.
  - The default package private modifier indicates the class can be accessed only by a subclass or class within the same package.
  - There can be at most one public class or interface in a Java file.
- **Defining Constructors**
  - In Java, the first statement of every constructor is either a call to another constructor within the class, using this (), or a call to a constructor in the direct parent class, using super ().
  - User of both super() and super(age) in the following example:

```
public class Animal {
private int age;
public Animal(int age) {
super();
this.age = age;
}
}

public class Zebra extends Animal {
public Zebra(int age) {
super(age);
}
public Zebra() {
this(4);
}
}
```

- **Understanding Compiler Enhancements**
  - compiler automatically inserts a call to the no-argument constructor super() if the first statement is not a call to the parent constructor.
  - if the parent class doesn't have a no-argument constructor then you must create at least one constructor in your child class that explicitly calls a parent constructor via the super() command.
  - In Java, the parent constructor is always executed before the child constructor.

- Example:

  ```
  public class Mammal {
  public Mammal(int age) {
  }
  }
  public class Elephant extends Mammal { // DOES NOT COMPILE
  }
  ```

- If the parent class and child class are part of the same package, the child class may also use any default members defined in the parent class.
- ***super()* vs. *super***
  1. super(), is a statement that explicitly calls a parent constructor and may only be used in the first line of a constructor of a child class.
  2. super, is a keyword used to ref member defined in a parent class and may be used throughout the child class.

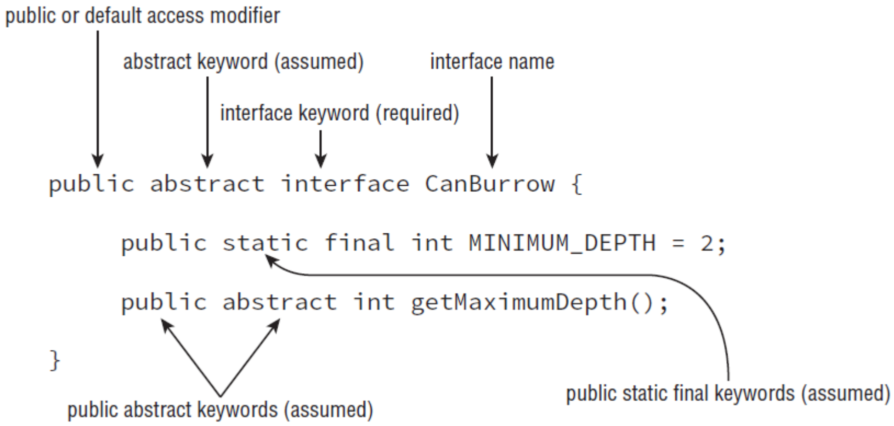
- **Inheriting Methods**
  - **Overriding a Method**
    - When you override a method, you may reference the parent version of the method using the super keyword.
    - The compiler performs the following checks when you override a non private method:
      1. The method in the child class must have the same signature as the method in the parent class.
      2. The method in the child class must be at least as accessible or more accessible than the method in the parent class.
      3. The method in the child class may not throw a checked exception that is new or broader than the class of any exception thrown in the parent class method.
      4. If the method returns a value, it must be the same or a subclass of the method in the parent class, known as *covariant return types*.
  - **Redeclaring private Methods**
    - Java permits you to redeclare a new method in the child class with the same or modified signature as the method in the parent class.
  - **Hiding Static Methods**
    - A *hidden method* occurs when a child class defines a static method with the same name and signature as a static method defined in a parent class.
    - Along with **four previous rules** new rule is added for hiding a method, namely that the usage of the static keyword for parent and child classes.
  - **Overriding vs. Hiding Methods**
    - Overridden is replaced at runtime in the parent class with the call to the child class's method.
  - **Creating *final* methods**
    - final methods cannot be overridden.
    - You cannot hide a static method in a parent class if it is marked as final.
  - **Inheriting Variables**
    1. Java doesn't allow variables to be overridden but instead hidden.
    2. This creates two copies of the variable within an instance of the child class: one instance defined for the parent reference and another defined for the child reference.

- **Default Method Implementations in Abstract Classes**
  - We note that an abstract class cannot be marked as final.
  - A method may not be marked as both abstract and private.

- Implementing an abstract method in a subclass follows the same rules for overriding a method. For example, the name and signature must be the same, and the visibility of the method in the subclass must be at least as accessible as the method in the parent class.

Implementing Interfaces

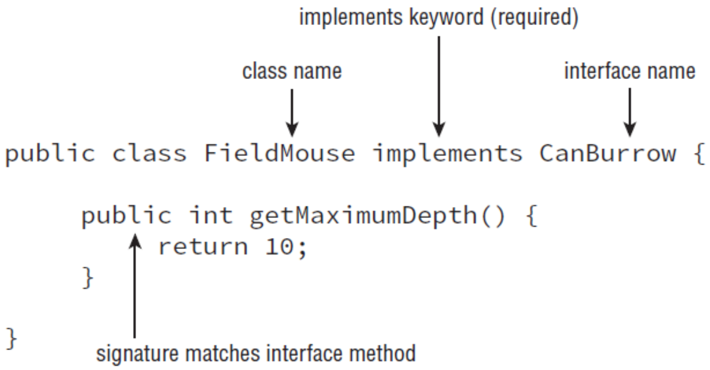
- Defining an interface



```
public abstract interface CanBurrow {

    public static final int MINIMUM_DEPTH = 2;

    public abstract int getMaximumDepth();

}
```

public or default access modifier
abstract keyword (assumed)
interface name
interface keyword (required)
public abstract keywords (assumed)
public static final keywords (assumed)

- Implementing an interface



```
public class FieldMouse implements CanBurrow {

    public int getMaximumDepth() {
        return 10;
    }

}
```

implements keyword (required)
class name
interface name
signature matches interface method

- **Defining an interface**
  1. All top-level interfaces are assumed to have public or default access, and they must include the abstract modifier in their definition.
  2. Marking an interface as private, protected, or final will trigger a compiler error, since this is incompatible with these assumptions.

- Here are two interface variables rules:
  1. Interface variables are assumed to be public, static, and final.
  2. The value of an interface variable must be set when it is declared since it is marked as final.

Default Interface Methods

- A *default method* is a method defined within an interface with the default keyword in which a method body is provided.
- The following are the default interface method rules:
  - A default method may only be declared within an interface and not within a class or abstract class.
  - A default method must be marked with the default keyword. If a method is marked as default, it must provide a method body.
  - A default method is not assumed to be static, final, or abstract, as it may be used or overridden by a class that implements the interface.

        - Like all methods in an interface, a default method is assumed to be public and will not compile if marked as private or protected.

This rule holds true even for abstract classes that implement multiple interfaces, because
the default method could be called in a concrete method within the abstract class.

- ## Static Interface Methods
  - A static method defined in an interface is not inherited in any classes that implement the interface.
  - Rules for static method:
    - Like all methods in an interface, a static method is assumed to be public and will not compile if marked as private or protected.
    - To reference the static method, a reference to the name of the interface must be used.
- ## Object vs. Reference
  - ***The type of the object determines which properties exist within the object in memory.***
  - ***The type of the reference to the object determines which methods and variables are accessible to the Java program.***
- ## Casting Objects
  - Casting an object from a subclass to a superclass doesn't require an explicit cast.
  - Casting an object from a superclass to a subclass requires an explicit cast.
  - The compiler will not allow casts to unrelated types
  - Even when the code compiles without issue, an exception may be thrown at runtime if the object being cast is not actually an instance of that class.
  - Casting is not without its limitations. Even though two classes share a related hierarchy, that doesn't mean an instance of one can automatically be cast to another. For ex:

```
public class Rodent {
}
public class Capybara extends Rodent {
public static void main(String[] args) {
Rodent rodent = new Rodent();
Capybara capybara = (Capybara)rodent; // Throws ClassCastException at runtime
}
}
```

- ## Virtual Methods
  - A *virtual method* is a method in which the specific implementation is not determined until runtime.
  - All non-final, non static and non-private Java methods are considered virtual methods, since any of them can be overridden at runtime.
  - What makes a virtual method special in Java is that if you call a method on an object that overrides a method, you get the overridden method, even if the call to the method is on a parent reference or within the parent class. For Ex:

```
public class Bird {
public String getName() {
return "Unknown";
}
public void displayInformation() {
System.out.println("The bird name is: "+getName());
}
}
public class Peacock extends Bird {
public String getName() {
return "Peacock";
}
public static void main(String[] args) {
Bird bird = new Peacock();
bird.displayInformation();
}
}
```

This code compiles and executes without issue and outputs the following:
The bird name is: Peacock

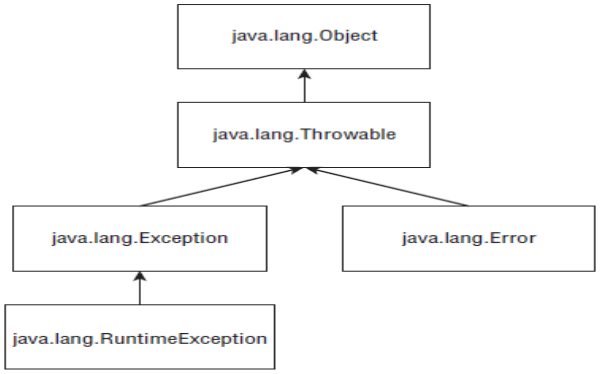
- ## Polymorphic Parameters
  - One of the most useful applications of polymorphism is the ability to pass instances of a subclass or interface to a method.

****

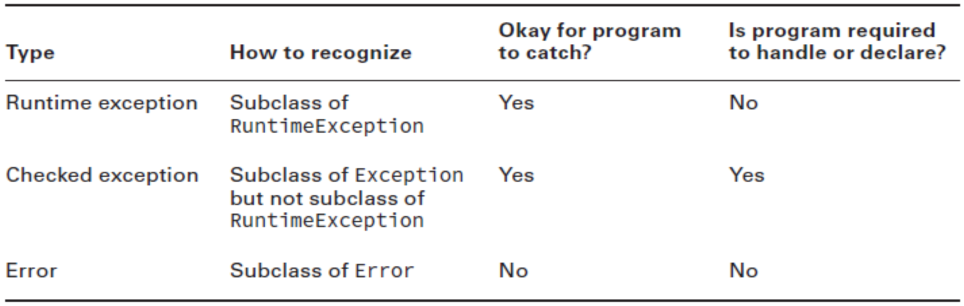
6. Exceptions

The Role of Exceptions

- An *exception* is Java's way of saying, "I give up. I don't know what to do right now. You deal with it."
- ### Understanding Exception Types

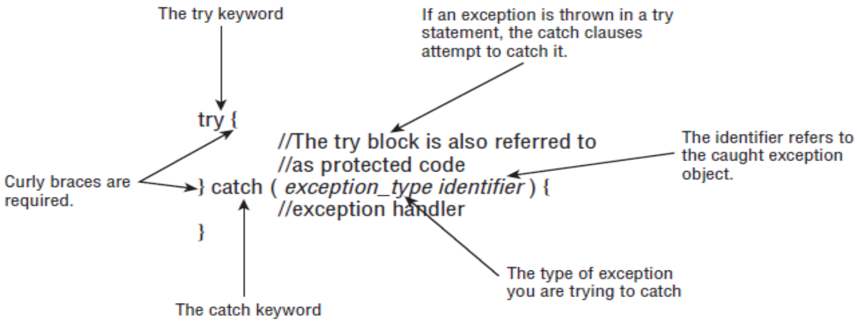


  - Error means something went so horribly wrong that your program should not attempt to**recover** from it.
  - A *runtime exception* is defined as the RuntimeExceptionclass and its subclasses.Runtime exceptions are also known as *unchecked exceptions*.
  - A *checked exception* includes Exception and all subclasses that do not extendRuntimeException. Checked Exception must be either ***handled or declared***.
- Types of exceptions



Type	How to recognize	Okay for program to catch?	Is program required to handle or declare?
Runtime exception	Subclass of RuntimeException	Yes	No
Checked exception	Subclass of Exception but not subclass of RuntimeException	Yes	Yes
Error	Subclass of Error	No	No

- Using a *try* Statement

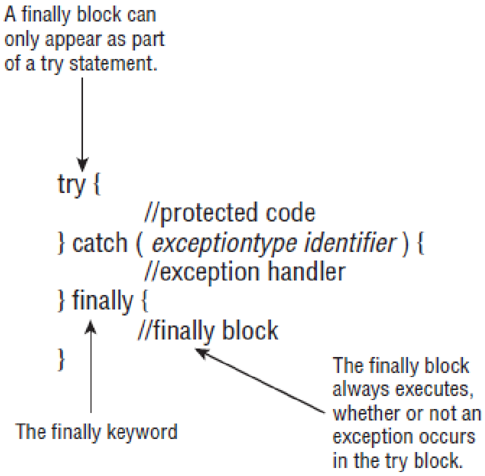


The try keyword
If an exception is thrown in a try statement, the catch clauses attempt to catch it.
The identifier refers to the caught exception object.
Curly braces are required.
The type of exception you are trying to catch
The catch keyword

```
try {
            //The try block is also referred to
            //as protected code
} catch ( exception_type identifier ) {
            //exception handler
}
```

- try statements are like methods in that the curly braces are required even if there is onlyone statement inside the code blocks.

- ## Adding a *finally* Block



A finally block can only appear as part of a try statement.
The finally block always executes, whether or not an exception occurs in the try block.
The finally keyword

```
try {
            //protected code
} catch ( exceptiontype identifier ) {
            //exception handler
} finally {
            //finally block
}
```

- There is one exception, When System.exitis called in the try or catchblock, finally does not run.
- **Tricky Example:**

```
public String exceptions() {
String result = "";
String v = null;
try {
try {
result += "before";
v.length();
result += "after";
} catch (NullPointerException e) {
result += "catch";
throw new RuntimeException();
} finally {
result += "finally";
throw new Exception();
}
} catch (Exception e) {
result += "done";
}
return result;
}
```

The correct answer is ***before catch finally done***.

- **Recognizing Common Exception Types**
    - ### Runtime Exceptions
        1. **ArithmeticException**→ code attempts to divide by zero
        2. **ArrayIndexOutOfBoundsException**→ code attempts to divide by zeroindex toaccess an array.
        3. **ClassCastException**→ when an attempt is made to cast an exceptionto a subclass of which it is not an instance.
        4. **IllegalArgumentException**→ indicate that a method hasbeen passed an illegal or inappropriate argument.
        5. **NullPointerException**→ when there is a null reference where anobject is required.
        6. **NumberFormatException**→ when an attempt is made to converta string to a numeric type but the string doesn't have an appropriate format.
    - ### Checked Exceptions
        1. **FileNotFoundException**→ Thrown programmatically when code tries to reference a filethat does not exist.
        2. **IOException**→ Thrown programmatically when there's a problem reading or writing a file.
        3. ***FileNotFoundExceptionis a subclass of IOException***.
    - ### Errors
        1. **ExceptionInInitializerError**→ Thrown by the JVM when a static initializer throwsan exception and doesn't handle it.
        2. **StackOverflowError**→ Thrown by the JVM when a method calls itself too many times(this is called *infinite recursion* because the method typically calls itself without end).
        3. **NoClassDefFoundError**→ Thrown by the JVM when a class that the code uses is availableat compile time but not runtime.
- ## Subclasses- Overriding method with exceptions
    - When a class overrides a method from asuperclass or implements a method from an interface, it's not allowed to add new **checkedexceptions** to the method signature.This rule applies only to **checked exceptions.**
- ### Printing an Exception
    - There are three ways to print an exception.
    - **Example:**

```
5: publicstatic void main(String[] args) {
: try {
7: hop();
: } catch (Exception e) {
9: System.out.println(e);
10: System.out.println(e.getMessage());
11: e.printStackTrace();
12: }
13: }
14: private static void hop() {
15: throw new RuntimeException("cannot hop");
16: }
```

```
java.lang.RuntimeException: cannot hop
cannot hop
java.lang.RuntimeException: cannot hop at trycatch.Handling.hop(Handling.java:15)
at trycatch.Handling.main(Handling.java:7)
```

****