

JAVA BASICS

The **type system** of a programming language describes the way the language interprets data stored in memory. Type checking in a Java program is done at compileime to ensure type compatibility; information about data types is also maintained during runtime to facilitate polymorphism. Java has two varieties of types.

Primitive types: Primitive types are irreducible data types, hard coded into the Java language. They include boolean values Unicode characters, integers, and floating point numbers. Java has eight implementation-independent primitive types:

- boolean: true or false
- char: 16-bit Unicode character
- byte: 8-bit signed two's complement integer
- short: 16-bit signed two's complement integer
- int: 32-bit signed two's complement integer
- long: 64-bit signed two's complement integer
- float: IEEE 754 single-precision floating-point value double: IEEE 754 double-precision floating-point value

Note to C users: In Java, booleans and ints are distinct variable types; they cannot be used interchangeably or cast to one another.

- Casting primitive types: Since Java is a strongly typed language, data sometimes must be **cast** from one primitive type (or object) to another. Casts allow the compiler to treat an expression as having the type specified by the cast. An explicit cast, one written directly by the programmer, has the following syntax, with the variable to be cast preceded by the desired cast type, in parentheses:
 - (type) variableName
- type variable = (type) expression;
 Java will make widening casts—casts from one type to another type that is at least as large as the first—automatically when necessary, often to avoid overflow.
- The Java compiler will throw a warning when asked to make a narrowing cast. Narrowing casts must be made explicitly, as they can result in a loss of some data. They usually are used to assign an expression to a variable of a narrower type.

Reference types: Reference types are data types that hold multiple values and do not have one standard size. A reference type variable holds a reference to the data, rather than the actual data itself. Arrays and classes are reference types in Java.

- Arrays are collections of elements, all of the same type, in which each element is assigned an index through which it is accessed. The indices are numbered sequentially.
- Classes are the foundation of object-oriented programming They contain a collection of data, as well as any operations that can be performed on that data. Fields store the data and methods hold the operations. Classes exist in a hierarchy based on similar functionality and types, thus allowing reusable code to be inherited (See Inheritance). Classes are referred to by programmer-designated names such as Date, LinkedList, or MyAmazingClass

DECLARING VARIABLES

Before any variable can be used to refer to data, it must be declared—the programmer must tell Java what type of the data the variable will hold (e.g., an int, a double, or a StringTokenizer object). Java will then allocate an appropriate amount of memory for the variable.

Variables are declared with the following syntax:

- type varName; type varName1, varName2, varName3 ...;
 - **Examples:**
- boolean trueOrFalse; · float cost, revenue, profit;
- InputStream myInputStream;
- A variable can be declared final, meaning that its value can not be changed later in the program.

INITIALIZING/INSTANTIATING VARIABLES

Unlike in C, variables are set to default values when they are declared. However, it is poor practice to use a variable without explicitly setting it to its initial value.

Initializing primitive types: Primitive types are initialized using the assignment operator =. The syntax is:
• variable = value;

Examples:

- number = 3:
- letter = 'f); /* use single quotes to denote a Unicode character */
- Variables also can be initialized as they are declared. However,
- only one type of variable can be declared in a statement:
- type varName = value:

Examples:

- boolean b = false: double d = Math.sqrt(2);
- int x = 0, y = 0, z = -2; char ch = 'X', int i = 5; // syntax error!!!

Instantiating objects: Objects are instantiated with the new operator, which calls the object's appropriate constructor (See Classes and objects). The constructor's return value is assigned to the variable

Examples:

- cal = new Calendar();
- Cal = flew Catendar();
 Hashtable ht = new Hashtable();
 thread = new Thread(target);

Objects can be instantiated at the same time as their variables

Examples:

- Calendar cal = new Calendar();
 Integer i1 = new Integer(5), i2 = new Integer(-1);
 Thread t1 = new Thread(runnable1);
- Objects can also be instantiated without being assigned to a variable (See Anonymous inner classes).

Strings: In Java, Strings are a class type, not a primitive type. However, to facilitate the manipulation of Strings, Java allows a String variable to be instantiated explicitly as an object or implicitly by the assignment of a string literal, a string of characters in quotation marks. The following are equivalent; both are valid Java syntax:

- String str = new String("Hello world!");
 String str = "Hello World!";

Strings also can be operated on either as literals or as objects with either the String.concat(String str) method or the operator. Both the concatenation operator and the concat() method concatenate their second operand or parameter to the end of the first. The following are equivalent:

- String str = "Hello" + " world!";
 String str = "Hello".concat(" world!");
- String str = Hello"; String s1 = "Hello"; String s2 = " world!"; String str = s1 + s2;

CONTROL FLOW

Java starts executing programs at the public static main (String[] args) line in the class specified on the command line, and it continues by executing each command sequentially. When the Java interpreter arrives at a method, it transfers control to that method, sequentially executing its statements. At the method's end, or when it reaches a return statement, Java transfers control back to the calling method. A program's flow of control can be modified with conditional statements and iterative statements

With all control flow statements except switch, the braces are optional if there is only one code statement.

Example: if (x < y)

- System.out.println("x is less than " + y); if (isTrue) {
- System.out.println("Added one to x."):}

CONDITIONAL STATEMENTS

if: Java's basic decision-making command is if, which is followed by a conditional statement and one or more command statements. The conditional must evaluate to a boolean. The command statements are sequentially executed if the conditional evaluates to true and skipped if it evaluates to false.

• if (condition) {

execute code statements here

if/else: The if statement can be complemented with an else statement followed by one or more command statements. The commands following if are executed if the conditional state ment is true, and the commands following else are executed if the conditional statement is false.

- if (condition) { execute code statements here }
- execute other code statements here } Example:
- if (number < 5) System.out.println("Less than five."); else if (number == 5)
- System.out.println("Equals five."); else System.out.println("Greater than five."):

switch: The switch statement is equivalent to a series of else if statements. The switch statement is best used when only a single variable needs to be tested for a series of alternatives.

The variable is matched against each case value. When or if there is a match, the accompanying code segment is executed. The optional default statement can be placed at the end of the switch code; its associated code is executed if none of the other cases are executed.

```
switch (variable) {
   case value1:
       execute code statements here
       break;
   case value2:
       execute code statements here
   default:
       execute code statements here
Example:
switch (ch) {
       System.out.println("Yes!"):
   case 'n':
       System.out.println("No!");
       break;
   default:
```

ITERATIVE STATEMENTS

Java's iterative control structures are similar to those of C. including while, do, and for loops. Loop control flow can be interrupted by the break and continue commands. The break command exits a loop; the continue command exits the current iteration of the loop and goes on to the next iteration. However, good programming practice minimizes the use of break and continue statements.

System.out.println("Invalid input!");

while: The while loop is the basic loop structure in Java. It consists of a conditional statement and a code segment. The code segment is executed and re-executed as long as the conditional is true.

```
while (condition) {
   execute code segments here }
```

Examples:

```
int x = number; // assign x the value of number
/\!/ counts down from the current value of x to 0
if (x \ll 0)
   System.out.println("Invalid value for x: " + x);
else
   System.out.println(--x);
System.out.println( "x equals 0" );
// picks random numbers r, stops when 0.6 < r < 0.7
double r = Math.random();
while (!( r > 0.6 \& r < 0.7)) {
       System.out.println( r );
        r = Math.random(); }
```

do: The do loop is similar to the while loop, except that the conditional statement of the do loop is evaluated after the loop executes once. The code segment is re-executed as long as the conditional is true.

- do { execute code statements here
- while (condition); Example:
- flips a coin, stops when the coin is heads (true) */ boolean heads:

myCoin = myCoin.flip(); heads = myCoin.getFace(); } while (!heads);

for: The for statement governs a block of code through an initialization statement, conditional statement, and increment statement. The initialization, conditional, and increment statements are all optional, although the statement for (an infinite loop. Within a for loop, Java first executes the initialization statement and then executes and re-executes the block of code and increment statement—in that order—as long as the condition is true. The condition is evaluated each time before the code block is executed.

for (initialization; condition; increment) {
 execute code statements here }

int \dot{x} = number; // assign x the value of number // counts down from the current value of x to 0 for (; x > 0 ; x--)
 System.out.println(x); System.out.println(x = 0);

SAN Copyright © 2003 by SparkNotes LLC.
All rights reserved.
SporkCharts is a registered trademark
of SparkNotes LLC.
A Barnes & Noble Publication
10 9 8 7 6 5 4 3 2 1
Printed in the USA \$4.95 | \$7.95 CAN

"THERE ARE 10 PEOPLE IN THE WORLD: THOSE WHO UNDERSTAND BINARY AND THOSE WHO DON'T." RAND RANDY CASSINGHAM

```
// sorts a char array k using bubble sort
for (int i = 0; i < k.length(); i++) {</pre>
     char temp;
for (int j = i; j < k.length() - 1; j++) {
           if (k[j] > k[j + 1]) {
  temp = k[j];
  k[j] = k[j + 1];
                 k[j + 1] = temp;
    }
```

Although array elements are referenced by sequential integers, Java does not necessarily store the elements of an array in consecutive memory locations. There are no pointers in Java because the details of data storage are encapsulated from the user. Arrays in Java are Objects, and therefore they are instantiated with the new operator:

type[] arrayName = new type[size];

Arrays are not dynamic data structures; an array initialized to a particular size will stay that size throughout its lifetime. Java will throw an ArrayIndexOutOfBoundsException at runtime

if you access an array index greater than length than zero. The java.util.Vector and java.util.ArrayList classes each simulate a dynamically growable array.

· All the elements in an array must be the same type or subclasses of the same type.

Examples:

- kompres:
 int[] digits = new int[10];
 File[] directory = new File[myDir.getSize()];
 Object[] array = new Object[SIZE]; /* This in red
 interpretation of the control of
- will hold instances of any class, but will not hold primitive types. In order to store a primitive type in this array, use a wrapper class such as java.lang.Boolean or java.lang.Double. */

REFERENCING ARRAY ELEMENTS

Array elements are referenced with the [] operator. The array name is followed by the element's index in square brackets. arrayName[index]

The first element of an array has index number zero. Any attempt to reference an element beyond the array's length will throw an ArrayIndexOutOfBoundsException

- String s = args[0] /* references the first param-
- eter on the command line */
 File myFile = directory[k] /* references the (k - 1)th element of directory */

ARRAY VARIABLES AND METHODS

The java.util.Arrays class contains several useful methods and variables for working with arrays.

- length: All arrays have an internal variable called length that stores the maximum number of elements the array can hold. The last index of array k is k.length -
- binarySearch(Object[] array, Object obj) Searches array for the object obj.
- equals(Object[] array1, Object[] array2) Returns true if array1 equals array2.
- fill(Object[] array, Object obj)
- Assigns the value obj to each element of array. sort(Object[] array)
- Sorts the elements of array in ascending order.
- All of these methods are overloaded to handle arrays of primitive types and, in some cases, Comparables. See http://iava.sun.com/i2se/1.4.1/docs/api/iava/util/Arrays.html

<u>CLASSES AND OBJECTS</u>

Object-oriented languages such as Java are designed to simulate real-life objects (both tangible and abstract by combining a group of data and operations on that data into classes. A **class** organizes attributes into data, or **fields**, and behavior into blocks of code, or **methods**. Fields and methods are called **members** of a class. A specific instance of a class can be declared as a variable in a program (unless the class is static), and this instantiation of the class is called an object.

CREATING CLASSES

Classes are created using the class keyword followed by the name of the class. The class keyword is often preceded by a visibility modifier as well as other modifiers such as static or abstract (See Modifiers). The entire class description, its fields and methods, is enclosed in braces

class myClass { ... }private class Node { ... }

CLASS HIERARCHY

All Java classes fit into a hierarchy that relates classes in terms of the fields and methods that they share Every class—except Object—is said to be a **subclass** of another class closer to the root of the hierarchy

At the top of the hierarchy is the Object class, from which all other classes descend. Since all classes are derived from Object, an instance of any Java class can call the methods that Object can call; any object is an Object.

Inheritance: Every class inherits all of the methods and fields of its superclass, the class immediately above it in the class hierarchy. This means that any method that can be called on the superclass can be called on the subclass (see Polymorphism). The default superclass for any new class is Object, and Object does not need to be specified as a superclass. Use the extends keyword after the class name to allow a new class to inherit methods from a class other than Object.

- public class Rectangle extends Polygon { .. • public class Square extends Rectangle {
- Every class (except Object) must extend exactly one class although there is no limit to the number of interfaces it car implement (see Interfaces)

Static classes: A static class, denoted with the static key word, is a class that cannot be instantiated. Its methods belong to the class, and are independent of a specific instantiation of an object. Static classes are called using the class name and the dot (method invocation) operator.

Example: The static class java.lang.Math performs many of the mathematical functions of a scientific calculato

These functions, such as the sine function and the natural logarithm function, exist outside of any particular object. The methods of the Math class are called as follows:

- double cos = Math.cos(angle);
- double rand = Math.random();

Abstract classes: An abstract class serves as a design template for subclasses to extend and fully implement or over ride some or all of its methods. Abstract classes are useful for designing a class hierarchy in which a superclass is too "abstract" to have specific implementations for all its methods. An abstract class does not require abstract methods, but a class with an abstract method must be declared abstract (See Abstract methods).

PROGRAMS WITH MORE THAN ONE CLASS

Most Java programs use a combination of classes in the Java API and classes created by the programme (which may also be extensions of classes in the API)

The import statement tells the Java interpreter which external classes a particular class will refer to. Without the import statement, Java expects methods and objects outside the current class to be fully qualified.

Examples:

- import myClass /* myClass is in the same directory as the current class */
 import java.util.StringTokenizer;

Each import statement is associated with only one class. To import all of the classes in a package, replace the class name

Example:

import java.io.*; /* A class headed by this statement can refer to any of the classes in the java.io package. */

Packages: Packages are groups of related classes and interfaces that can be defined as a unit to form libraries. Classes that share a package have special visibility privileges with respect to one another (See Visibility modifiers). Use the package statement to associate a class with a package. The package statement must be the first line of a class file, and each class can be associated only with a single package:

package packageName;

Although package names are delineated by dots (e.g. java.util.regex), the dots do not signify any sort of subordinating relationship between the elements of the package name. Therefore, the statement import java.util.* does not import the objects in the java.util.regex package

Nested classes: Classes (and interfaces) can be nested within any level of braces or within a method. Nested, or inner, classes allow related objects to be connected efficiently in terms of

design and access. Nested classes can have static, final, or abstract modifiers just as any class; they can also be declared anonymous (see below). If a nested class is enclosed in a block of code, it is a local inner class, and is treated as a local variable. Otherwise, a nested class is treated as a member of the outer class and can access other members of the outer class Example:

```
• class Car {
     class Transmission {
         code for Transmission here
     more code for Car here
```

Anonymous inner classes: An anonymous inner class is used to instantiate an object for immediate and onetime use without assigning it to a variable. An anonymous inner class is introduced with the new operator followed by a class name (the class's constructor) and body:

Example:

- Dice rollDice() { return new Dice() { int number = (int)(Math.random() * 6) + 1:
- · Java also supports anonymous inner interfaces, which must support all the methods that the outer interface supports.

STATIC FIELDS

Static fields, declared with the keyword static, are members that belong to the class rather than an instance of the class. Only one copy exists for all instances of a class. Static methods can reference only static fields (See Static classes)

OBJECT METHODS

The following is a partial list of the methods of java,lana,Object.

- equals(Object obj): Returns true if obj "equals" the calling object. The programmer determines the meaning of "equals" in this context by overriding the method. If this method is not overridden, it returns true if the hashcode of objequals the hashcode of the calling object. Unless either equals() or hashCode() has been overridden, this method is equivalent to using the == operator with two Objects as operands: equals() will return true if both operands refer to the same object.
- getClass(): Returns the class of the calling object
- hashCode(): Returns a hashcode for the calling object.
- clone(): Returns a copy of the calling object, provided the calling object implements the Cloneable interface.
- toString(): Returns a String representation of the calling object. If this method is not overridden, it will return the calling object's hashcode.

METHODS

Methods, reusable segments of Java code, come in two varieties: object and static. A method is called, or invoked, from any portion of a program, at which point control passes from that line to the method body A method can have values passed in as parameters from the calling segment. After the method finishes execution, it can return a value to the calling segment as well

VARIABLE SCOPE AND ENCAPSULATION

As a general rule, a variable exists only within the pair of braces or block in which it is declared

- Variables declared outside of any method in a class can be used anywhere in their class after they are declared and, depending on their visibility, can be accessed from outside the class using the dot operator (See Visibility modifiers).
- Variables declared in a method are accessible only from within the method in which they are declared. An attempt

to use a method variable outside its own method will result in a compile-time error.

Variables declared in a for or while loop are accessible only from within the block of code from which they are declared. Good programming practice dictates that the programmer try to avoid declaring variables in loops. Unless the variable is declared in the initialization of a for loop, each time the loop is executed, the variable would be redeclared and more memory allocated rather than reused.

CONTINUED ON OTHER SIDE

METHODS (CONTINUED)

METHOD HEADERS

Methods are introduced with method headers, which consist of the method's optional modifiers, return type, name, and a method signature that consists of a set of parameters enclosed in parentheses. The parentheses denote a method header or method invocation and are not optional, even if the method takes no parameters. The method header is followed by an open brace, not a semicolon, and the compiler will catch any stray semicolons following method headers.

Modifiers returnType methodName(Parameters) { method body here }

Modifiers change the way the Java interpreter will allow a method to behave. They include visibility modifiers and keywords such as static, final, and synchronized

- Visibility modifiers: Java's visibility modifiers private, protected, and a default—encapsulate fields methods, and classes by limiting their access from other
 - · public: Public fields and methods can be accessed wherever their class is accessible. Public classes can be accessed wherever their package is accessible.
 - private: Private fields and methods can be accessed only by their containing class.
 - protected: Protected fields and methods can be accessed only by classes in the same package as their containing class, and subclasses of their containing class.
 - Default: A field or method not declared public, private. or protected is accessible only to classes in the same package as its containing class. A class not declared public will be accessible only within its package.
- Static methods function independent of any object's instantiation. As a result, they can access only static fields (which also exist independent of any object's instantiation) and other static methods. The exception is that static methods can access all of the methods available to objects instantiated by the static method itself. (See also Static classes.)
- Final methods cannot be overridden by a subclass. Final classes cannot be subclassed. Fields and variables declared final cannot have their values changed.
- Synchronized methods are used when a program is running multiple threads of execution. Only one method declared synchronized can be running at a time. See http://java.sun.com/j2se/1.4.1/docs/api for more information on threading

Return type: When a method completes its execution, it must return no more than one value to its calling method using the return statement. The return type in the method header must match the type, whether primitive or class, returned by the return statement. If a method does not return any value, its return type is void.

Method name: The method name is a programmer-specified name that should briefly describe what the method does. The method name does not have to be unique within a class (See Method overloading)

Method signature: The method signature lists a method's formal parameters separated by commas. Each formal parameter consists of its type followed by a programmer specified name. When a method is called, the variables listed as formal parameters are filled (never instantiated) with the values passed by the calling method (See Parameter passing). The combination of method name and method signature must be unique to the class (See Method overloading)

Examples:

- public static void main(String args) { ... }private Iterator getIterator(List list) { ... } • public double distance(Point p1, Point p2) { ... }
- public double 3dDistance
- (Point p1, Point p2, Point p3) { ... } protected syncronized void holdThread (int milliseconds) { ... }

INVOKING METHODS

Methods are invoked with the . (dot) operator, which separates the object invoking the method (or the class invoking a static method) from the method name and signature being invoked.

The dot operator can be used multiple times in a single statement to call multiple methods. The methods are evaluated from left to right, so the returned value of one method serves as the object on which the next method is called.

- object.method(parameters);class.method(parameters);
- object.method1(parameters).method2(parameters)

The keyword this is used to refer to the object calling a particular method, and it allows an object to refer to itself. When an object invokes a method on itself, this is optional. Therefore, the following statements are equivalent:

- this.myMethod();
- myMethod();

PARAMETER PASSING

The actual parameters sent to a method by the calling segment are listed, separated by commas, in the parentheses after the method name:

- changeColor(Color.RED); /* red is a final field of the Color class */
- double vol = myPrism.getVolume(height, width, length);

The number of parameters passed to a method must match the number of parameters called for in one of the method's signatures. The passed parameters must be of same types and in the same order as in the method signature

Java always passes parameters to methods by value

- Primitive types: When a primitive type is passed to a method, the value of the parameter is copied to the method variable. The actual value in the calling code remains unaffected by changes in the method copy.
- Reference types: When a reference type is passed to a method, then the value of the object's reference (not the actual object) is copied and used in the method. In this case, all changes to the object to which the value refers will remain after the method code terminates. Although the object reference is "passed by value," parameter passing with reference types mimics C/C++ "pass by reference," eliminating the need for pointers.

Constructors: Constructors are methods that create new instances of classes. They are invoked with the new operator. A constructor is like an ordinary method with two exceptions: (1) it has no return type because it implicitly returns an instance of its class, and (2) it must have the same name as its class.

```
• public class Count {
     int number:
     public Count(int startNumber) {
         number = startNumber; }
```

Method overloading: Two methods in the same class can have the same name as long as their method signatures are different. When the method is called, Java checks the parameters of the called method against the method signatures of all like-named methods. Java executes the method whose signature matches that of the method invocation. The following example has two methods called "Thing" with different method signatures and modifiers. When the constructor for this class is called, "public Thing()" will execute and that method will in turn call the other "Thing" method with the integer parameter.

```
public class Thing {
        int size;
       final int DEFAULT_SIZE = 20;
private Thing(int s) {
             size = s;
       public Thing() {
   new Thing( DEFAULT_SIZE );
```

POLYMORPHISM

Polymorphism is the end effect of the Java class hierarchy. A specific object can also be thought of as a member of a class of more general objects which, in turn, can be thought of as a member of a class of even more general objects, all the way up to the Object class. Not only does this allow for a more logical object implementation, it also allows the programmer to design a general class once and then use it as a blueprint to create more specific classes. There are two ways in which Java classes exhibit polymorphism: by extending other classes (See Class hierarchy) and through interfaces

EXTENDING OTHER CLASSES

A (sub)class that extends another (super)class already has all of the attributes, the methods and fields, of the superclass. The subclass can (and should) also include methods that aren't found in the superclass, override methods from the superclass to fit the specific needs of the subclass, or initialize fields that cannot be set to specific values in the superclass. The subclass should also have a constructor of its own, which may call the superclass constructor (Java implicitly calls the superclass constructor if it is not explicitly called by the program to ensure proper initialization).

Casting objects: When an object is instantiated, its type is set to whatever type the constructor returns. Like primitive types, however, objects may sometimes need to be cast. The syntax for casting objects is the same as that for casting primitive types:

- (className) variable; className variable = (className) expression;
- When necessary, Java will automatically make widening casts where the destination class is further up the class hierarchy (i.e., closer to the Object class) than the original class.
- Narrowing casts, where the destination class is deeper in the class hierarcy than the original class, must be made explicitly by the programmer. The compiler will throw a Type Mismatch exception if the cast is necessary but not made.
- It is illegal to try to cast an object to a destination class deeper in the hierarchy than the class in which the object was instantiated

Encapsulation: Although a subclass's constructor can have no obvious relation to its superclass's constructor (classes that extend java.lang.Object are often like this), the real power of object-oriented programming comes from re-using as much of the superclass as possible and then tweaking the superclass for the specificities of the subclass. The subclass can call the methods of the superclass without being aware of its implementation details. This encapsulates the su providing separate modules and a layer of protection by hiding code and private data from the user

The super keyword is used to refer to the superclass. Therefore, super() refers to the constructor of the superclass, super.name refers to the field name in the superclass, and super.mvMethod() refers to the method mvMethod() in the superclass.

Example:

```
    public class Polygon throws Exception {

     int sides:
     public Polygon(int n) {
      if (n < 3) throw TooFewSidesException}
     public int angleMeasure() {
   return 180 * (sides - 2); }
 public class Triangle extends Polygon {
     final int SIDES = 3;
     public Triangle() {
          super(SIDES);
```

Method overriding: To reflect the specific behavior of a subclass in reference to its superclass, Java supports method overriding. "Overriding" means defining a method in the subclass with the same name and signature as a method in the superclass. When a method is called on a subclassed object, Java executes either the method in the subclass or the superclass, depending on the current class of the object. If the subclass has no method with the same name and signature as the called method. Java executes the method in the superclass.

Abstract methods are methods with names and signatures, but without bodies. Any class that contains abstract methods must be declared abstract and cannot be instantiated. Abstract methods must be overridden by subclasses or supported by implemented interfaces.

INTERFACES

To simulate the multiple inheritance features of languages such as C++, Java supports interfaces, which are a contract between the programmer and the Java interpreter. Interfaces tend to have names (such as Runnable, Throwable, and Observable) that describe a function a particular class can perform, without making any claims about the class's fields and structure.

Implementing interfaces: An interface is a specification that any class implementing the interface is guaranteed to support. The keyword implements declares that a class supports a particular interface, and although a class can extend only one other class, it can implement an unlimited number of interfaces:

Example:

- class Suburban extends Car implements FourWheelDrive. SixCylinder
- · Attempting to let a class implement an interface without writing methods (and method bodies) for all the methods declared in the interface will cause a compile-time error

Writing interfaces: Interfaces are written just like classes, except with the class keyword replaced by interface.

public interface Runnable { ... • public interface MouseMotionListener

The body of an interface can contain only abstract methods and/or constant fields (declared as final and static). Any class implementing the interface must either supply the implementation for the abstract methods or be labeled abstract itself.

```
Examples:
```

```
public interface Runnable {
    public void run() {}
public interface MouseMotionListener {
   public void mouseDragged(MouseEvent e) {}
    public void mouseMoved(MouseEvent e) {}
```

Creating exceptions: Since all exceptions are descendants of java.lang.Exception, it is possible to create program-specific exceptions by creating a new class that extends Exception or one of its subclasses.

Example:

- public class myException extends Exception { \dots }

The standard constructor for a new exception consists of only one line invoking the superclass constructor. One of the superclass constructors takes a user-defined String as a parameter to facilitate error messages. An additional field can be added in the subclass to store data obtained from the String. Example:

my data,
public myException (String someData) {
 super("Exception involving " + someData);
 data = someData;
}

The default toString() method of Exception (inherited from Object) prints the String from the constructor (which may be null) concatenated with a String including the method and line number that threw the exception.

EXCEPTIONS

Technically, an exception is any class that extends the java.lang.Exception class. More generally, an exception is **thrown** whenever anything unexpected happens during the execution of a program. It is the programmer's responsibility to make sure that a program recovers gracefully from an exception.

Exceptions and Errors: The java.lang.Throwable class has two standard subclasses: Exception and Error. Errors usually represent unexpected behavior that the program cannot posibly recover from, so errors are usually ignored by the programmer. Exceptions are problems from which the program can (and should) recover from, and they are generally caught and handled within a program.

Checked and unchecked exceptions: A RuntimeException (java.lang.RuntimeException) or a descendant of RuntimeException is considered an unchecked exception, meaning that, as it occurs at runtime, the compiler cannot determine whether the exception might be thrown. Java will allow a program to throw unchecked exceptions without specifically declaring that it might do so. All other exceptions are checked exceptions, meaning that the compiler will check for them. Any method that throws a checked exception in its method header. A class whose methods throw checked exceptions must also declare the exceptions that it can throw.

EXCEPTION HANDLING

When an exception occurs, all execution in the program more precisely, in the thread) halts and control is transferred to an exception handling mechanism. At this point, one of two things will happen: the exception can be caught, or the exception can be allowed to propagale to another part of the program.

try/catch/finally: Exception handling is done through the try and catch statements. Code that might generate an exception is enclosed in a try block, a pair of braces preceded by the keyword try. Each try block is immediately followed by one or more catch blocks, code in between a pair of braces preceded by the keyword catch and the type and name of the Exception in parentheses. Whenever code in the try block throws an exception of the above type, control is transferred to the corresponding catch block.

exception generating code here
} catch (Exception1 e1) { exception handling code
here

} catch (Exception2 e2) { exception handling code here

After the catch block executes, control is transferred back to the calling method.

A finally block can follow any catch block(s). Before transferring control to the calling method, the code in the finally block executes regardless of whether exceptions were thrown.

exception generating code here }
catch (Exception e) { exception handling code here }
finally {
clean-up code here }

Propagating exceptions: Any exception that is not caught in the method that throws it must be declared in the method header with the throws keyword:

public void findCar(Car c) throws NoCarException, MyException

THE JAVA API

Java 1.4 consists of over 2700 pre-written classes organized in the Application Programmer Interface (API). The following is a brief list of some of the most commonly used Java classes, organized by package. The complete API can be found at http://java.sun.com/izse/1.41/docs/api.

JAVA.LANG

Consists of the Java classes that form the basis of the language; classes representing the elements Java needs to function such as Object, Method, Class, System; and wrappers for the primitive types. The functionality of most of these methods is generally encapsulated by the Java VM.

- Object is at the root of the class hierarchy. Its methods serve to manipulate objects as they are stored in memory.
- Math and StrictMathcontain static methods for performing basic mathematical functions. The two classes are very similar, but StrictMath is used primarily in cryptographic applications needing platform-independent results.
- Boolean, Character, Double, Float and Integer are wrapper classes, used to treat primitive types as objects. They contain methods for conversions between Strings and primitive types or wrappers.
- System maintains the standard input, output, and error streams; the clock; and the system properties. System also has a method that copies part of an array.
- String and StringBuffer represent character strings. A String is immutable; a StringBuffer is not.
- Thread represents a thread of execution. Java is a multithreaded language allowing simultaneous execution of code segments.

JAVA.IO

Contains classes that allow Java to receive input and output from the user and to communicate with the filesystem.

- File represents a file or a directory pathname.
- InputStream represents a stream of incoming byte data. It is an abstract class, overridden by such classes as FileInputStream, FilterInputStream, ObjectInputStream, and PipedInputStream. FilterInputStream is used with an instantiation of a subclass such as BufferedInputStream.
- OutputStream represents a stream of outgoing byte data. It is an abstract class, overridden by such classes as FileOutputStream, FilterOutputStream, ObjectOutputStream, and PipedOutputStream. FilterOutputStream is used with an instantiation of a subclass such as BufferedOutputStream.
- Reader is an object that reads character streams.
 It is an abstract class, overridden by such classes as BufferedReader, FileReader, PipedReader, and StringReader.
- Writer is an object that writes character streams. It is an abstract class, overridden by such classes as BufferedWriter, FileWriter, PipedWriter, and StringWriter.

JAVA.UTIL

Example:

Contains utilities such as basic data structures; classes representing times, dates and locations; fundamental event classes; and miscellaneous utilities.

- Vector and ArrayList are implementations of growable arrays. Unlike ArrayList, Vector methods are synchronized, making Vector inefficient but thread-safe. ArrayList methods are not synchronized.
- Hashtable and Hashmap are data structures that map keys to values.
- Stack represents a stack of objects.
- LinkedList and Iterator are used to implement a linked list data structure. The structure is stored in the LinkedList and returned by the Iterator.
- Date represents a specific time, precise to the nearest millisecond.
- GregorianCalendar represents a standard calendar.
 EventObject is the root class for all events (See java.awt.event).
- Timer allows applications to schedule execution of commands.
- StringTokenizer breaks a string into tokens, pieces marked by certain delimiters (generally white space or punctuation marks, but the specific delimiters can be given as parameters to the constructor).
- Random represents a pseudorandom number generator. Depending on the method called, it returns either a "random" byte array, integer, decimal, or Gaussian distributed value between zero and one and with standard deviation of one.

JAVA.AW1

Contains the classes of the **Abstract Windowing Toolkit**, which is used to create graphics and images
Many of the classes in java.awt have been updated
in Swing.

- Graphics is the base class for all graphical components.
 It contains methods for drawing shapes and Strings on a Canvas object. The Graphics2D class extends the Graphics class with several enhancements.
- Component represents a graphic object that the user can interact with, such as a Button, Label, or Scrollbar, all of which extend Component.
- Container is a type of Component that can hold other components. It contains methods for adding and placing new components into a container.
- Window is a top-level window with no borders or menubar.
- Point is a location in the coordinate plane, specified to the nearest integer.
- Polygon represents a polygon, a closed region in the coordinate plane.
- Color has a list of pre-defined fields that represent various colors, as well as methods for creating a new color from an existing Color or from three ints between 0 and 255 representing red, blue, and green components.

JAVA.AWT.EVENT

Contains classes representing **events**, and interfaces that model **listeners**. Events are generated whenever something happens in the program or with the computer's peripherals; listeners have a programmer-defined response to events. Events themselves have fields that define such attributes as the location and time of the events and the name of the component that generated the event.

- MouseEvent is an object generated whenever the mouse moves or whenever the mouse button is pressed. The MouseListener and MouseMotionListener interfaces respond to MouseEvents.
- ActionEvent is an object generated by components such as Buttons or CheckBoxes. The ActionListener interface responds to ActionEvents.
- KeyEvent is an object generated whenever a keystroke occurs. The KeyListener responds to KeyEvents.
- WindowEvent is an object generated whenever the status of a window changes (e.g., when a window is closed or maximized). The WindowListener interface responds to WindowEvents.

JAVA.UTIL.REGEX

Contains the Pattern and Matcher classes and is used for parsing regular expressions.

- Pattern represents a regular expression.
- Matcher is an object that can find and replace a Pattern in a given character sequence.

JAVAX.SWING

Describes the Java Foundation Class "Swing," which is used to make GUIs and is similar to the java.awt package but more comprehensive. The primary difference between Swing and AWT is that Swing is implemented without native code, meaning that it is more portable than AWT. For more information on Swing, see https://java.sun.com/docs/books/tutorial/uiswing.

- JComponent is analagous to an AWT Component. It is the base class for any of Swing's graphical objects, such as JButton, JFrame, or JLabel.
- JPanel and JFrame are containers, which are JComponents that can hold other JComponents.
- ImageIcon turns an Image object into an Icon. Icon is an interface of the javax.swing package.
- BoxLayout and OverlayLayout are layout managers, which determine the placement of components in a Graphical User Interface (GUI). Choose a layout manager with the component's setLayout(LayoutManager) method.

Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. SparkNotes is independent of Sun Microsystems. Inc. \$7.95 CAN