Chapter 2: Everything is an Object

Java is more of a “pure” object-oriented language.

Both C++ and Java are hybrid languages, but in Java the designers felt that the hybridization was not as important as it was in C++.

The reason C++ is hybrid is to support backward compatibility with the C language.

The Java language assumes that you want to do only object-oriented programming.

Everything in Java is an object, even a Java program.

2.1 Manipulating objects with references

In Java, You treat everything as an object, so there is a single consistent syntax used everywhere.

The identifier you manipulate is actually a “reference” to an object.

Having a reference doesn’t mean there’s necessarily an object connected to it. ---- String s;

A safer practice is always to initialize a reference when you create it. ---- String s = “asdf”; (Strings can be initialized with quoted text.)

2.2 You must create all the objects

new says, “Make me a new one of these objects.”

```
String s = new String(“asdf”);
```

Java comes with a lot of ready-made types. What’s more important is that we can create our own types.

(1) Where storage lives

1. **Registers.** This is the fastest storage. You don’t have direct control, nor do you see any evidence.
2. **The stack.** This lives in the general RAM area, but has direct support from the processor via its *stack pointer*. Extremely fast and efficient. Java objects themselves are not placed on the stack.

3. **The heap**. This is a general-purpose pool of memory where all Java objects live. Flexible, but there's a price : it takes more time to allocate heap storage than it does to allocate stack storage.

4. **Static storage.** In a fixed location. Static storage contains data that is available for the entire time a program is running. Java objects themselves are never placed in static storage.

5. **Constant storage**. Constant values are often placed directly in the program code.

6. **Non-RAM storage**. The trick is turning the objects into something that can exist on the other medium, and yet can be resurrected into a regular RAM-based object when necessary.

(2) Special case: primitive types

For these types Java falls back on the approach taken by C and C++. Instead of creating the variable using **new**, an "automatic" variable is created that *is not a reference*. The variable holds the value, and it's placed on the stack.

Java determines the size of each primitive type. These sizes don't change from one machine architecture to another.

Primitive type	Size	Minimum	Maximum	Wrapper type
boolean	—	—	—	**Boolean**
char	16-bit	Unicode 0	Unicode $2^{16}-1$	**Character**
byte	8-bit	-128	+127	**Byte**
short	16-bit	-2^{15}	$+2^{15}-1$	**Short**
int	32-bit	-2^{31}	$+2^{31}-1$	**Integer**
long	64-bit	-2^{63}	$+2^{63}-1$	**Long**
float	32-bit	IEEE754	IEEE754	**Float**
double	64-bit	IEEE754	IEEE754	**Double**
void	—	—	—	**Void**

All numeric types are signed. *boolean* is able to take the literal values **true** or **false** only.

The primitive data types also have "wrapper" classes. If you want to make a nonprimitive object on the heap to represent that primitive type, you use the associated wrapper.

char c = 'x';

Character C = new Character(c);

Java includes two classes for performing high-precision arithmetic: BigInteger and BigDecimal. Neither one has a primitive analogue. You can do anything with a BigInteger or BigDecimal that you can with an **int** or **float**, just using method calls instead of operators. ---- exchange speed for accuracy.

(3) Arrays in Java

Using arrays in C and C++ is dangerous because those arrays are only blocks of memory.

One of the primary goals of Java is safety. A Java array is guaranteed to be initialized and cannot be accessed outside of its range. The *tradeoff* : having a small amount of memory overhead on each array as well as verifying the index at run-time.

When you create an array of objects, you are really creating an array of references, and each of those references is initialized to **null**. You must assign an object to each reference before using it.

You can also create an array of primitives. The compiler zeroes the memory for the array.

2.3 You never need to destroy an object

(1) Scoping

In C, C++ and Java, scope is determined by the placement of curly braces **{ }**.

```
{
  int x = 12;
  /* only x available */
  {
    int q = 96;
    /* both x & q available */
  }
  /* only x available */
  /* q "out of scope" */
}
```

```
{
  int x = 12;
  {
    int x = 96; /* illegal */
  }
}
```

The compiler will announce that the variable **x** has already been defined. Thus the C and C++ ability to “hide” a variable in a larger scope is not allowed in Java.

(2) Scope of objects

When you create a Java object using **new**, it hangs around past the end of the scope.

Objects created with **new** stay around for as long as you want them.

Java has a *garbage collector*, which looks at all the objects that were created with **new** and figures out which ones are not being referenced anymore. Then it releases the memory for those objects.

You never need to worry about reclaiming memory yourself. ---- Eliminating “memory leak”.

2.4 Creating new data types: class

(1) Data members

If it is a reference to an object, you must initialize that reference to connect it to an actual object in the *constructor*. If it is a primitive type you can initialize it directly at the point of definition in the class.

Each object keeps its own storage for its data members; the data members are not shared among objects.

Primitive type	Default
boolean	false
char	‘\uoooo’ (null)
byte	(byte)o
short	(short)o
int	o
long	oL
float	o.of
double	o.od

Note carefully that the default values are what Java guarantees when the variable is used *as a member of a class*. This guarantee doesn’t apply to “local” variables. If you forget to initialize them, you get a compile-time error.

(2) Methods

Methods in Java determine the messages an object can receive.

The method name and argument list together uniquely identify the method.

int x = a.f( ); ---- This act of calling a method is commonly referred to as *sending a message to an object*.

2.5 Building a Java program

(1) Name visibility

To produce an unambiguous name for a library, the specifier used is like an Internet domain name. Now the entire package name is lowercase.

This mechanism means that all of your files live in their own namespaces, and each class within a file must have a unique identifier.

(2) Using other components

Java eliminates the "*forward referencing*" problem.

You can tell the Java compiler exactly what classes you want using the ***import*** keyword. **import** tells the compiler to bring in a *package*, which is a library of classes.

Using components from the standard Java libraries that come with your compiler, you don't need to worry about long domain names.

import java.util.ArrayList; import java.util.*;

(3) The static keyword

One situation is if you want to have only one piece of storage for a particular piece of data. The other is if you need a method that isn't associated with any particular object of this class.

Even if you've never created an object of that class you can call a **static** method or access a piece of **static** data.

```
class StaticTest{
        static int i = 47;
}
StaticTest st1 = new StaticTest( );
StaticTest st2 = new StaticTest( );
```

There are two ways to refer to a **static** variable : you can name it via an object, or you can refer to it directly through its class name (preferred).

An important use of **static** for methods is to allow you to call that method without creating an object. **----** Defining the **main( )** method

2.6 The first Java program (HelloDate.java)

You can put in the following bit of code at the end of **main( )** to pause the output:

```
try{
        System.in.read( );
}  catch (Exception e) { }
```

This will pause the output until you press “Enter” (or any other key).

You must place the **import** statement at the beginning of each program file. There's a certain library of classes that are automatically brought into every Java file: **java.lang**.

If you don't know the library where a particular class is, or if you want to see all of the classes, you can select "Tree" in the Java documentation. Now you can find every single class that comes with Java. Then you can use the browser's "find" function to find a specific class.

"out" is a **static PrintStream** object.

The name of the class is the same as the name of the file. That class must contain a method called **main( )** with the signature shown:

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

2.7 Comments and embedded documentation

Link the code to the documentation. The easiest way to do this is to put everything in the same file.

The tool to extract the comments is called *javadoc*. The output of javadoc is an HTML file. This tool allows you to create and maintain a single source file and automatically generate useful documentation.

All of the javadoc comments occur only within /** comments. The comments end with */ as usual.

There are two primary ways to use javadoc: embed HTML, or use "doc tags." Doc tags are commands that start with a '@' and are placed at the beginning of a comment line.

Note : javadoc will process comment documentation for only **public** and **protected** members. This makes sense, since only **public** and **protected** members are available outside the file.

```
/**
* <pre>
* System.out.println(new Date());
* </pre>
*/
```

@see Class : @version @author

Method : @param @return @throws @deprecated

2.8 Documentation example

The unofficial standard in Java is to capitalize the first letter of a class name. If the class name consists of several words, they are run together and the first letter of each embedded word is capitalized.

For almost everything else: methods, fields and object reference names, the accepted style is just as it is for classes *except* that the first letter of the identifier is lowercase.