

## Some General OOP Terminology

An **object** is a run-time entity that represents an *encapsulation* of data, and operations defined on that data. An object has a unique identity (i.e., a memory address) and a persistent state (i.e., a set of values associated with the data locations that make up the object). At run-time, we invoke operations on objects, called **methods**, to query the state of the object or to effect a change of state, which persists after completion of the operation. Hence, most methods defined on an object are *side-effecting* procedures. An object is created at run-time by invoking a special method called a **constructor**. In some OO languages (e.g., Java), methods may throw **exceptions** on the occurrence of an error, which must be caught and handled by the caller.

A **class** is a compile-time entity that provides a specification of an abstract data type. The specification consists of variable declarations and method declarations defined with the *scope* of the 'class' syntactic construct. In general, variable declarations are considered to be **private** and method declarations are considered to be **public**. However, variables can be declared public and methods can be declared private.

Hence, a class is a linguistic mechanism used for defining new types from existing types: either builtin types, such as 'int', or previously defined class types, such as String. By also specifying the access policy (i.e., public, private) for variables and methods, a class allows the specification of information hiding properties for objects of that class.

Every object is an **instance** of some class, and hence every object is governed (constrained) by its type. Most OO languages are statically typed, so that type errors are detected at compile-time. The variables declared in a class that are associated with an object at run-time are called **instance variables**, and each object has its own unique storage locations associated with its set of instance variables. A class may define **class variables**, which are variables associated with run-time storage locations that are shared by all instances of that class.

## Examples: An Integer Stack Class in Java (Stack.java)

```
public class Stack { // open class scope
    // private data declarations
    private int[] theStack;
    private int size;
    private int top;

    // stack constructor
    public Stack(int s) { size = s; theStack = new int[size]; top = 0; }

    // public methods
    public boolean empty() { return top == 0; }
    public boolean full() { return top == size; }
    public void push (int x) throws StackOverflowException {
        if (!full()) {
            theStack[top] = x;
            top++;
        }
        else throw new StackOverflowException();
    }
    public int pop() throws StackUnderflowException {
        if (!empty())
            return theStack[--top];
        else throw new StackUnderflowException();
    }
}
```

## Example: Creating and using a Stack Object

```
// A "main" class that uses a stack

public class Main {

    public static void main(String[] args) {
        // allocate heap storage and call constructor to "instantiate"
        // a new Stack object, and assign its address to the reference
        // variable s of type Stack.

        Stack s = new Stack(10);
        ...
        // Attempt a push, and catch overflow exception
        try {
            s.push(99);
        }
        catch (StackOverflowException e) {
            System.err.println(e);
            System.exit(-1);
        }

        ...// similarly for s.pop()
    }
}
```

## Java Data Types

Java has three categories of data types: **primitive types**, **arrays**, and **reference types**.

Because java programs execute on a virtual machine, the primitive types are defined to be the same across all real machine architectures. Note that this is different from C/C++, where the real machine architecture has an impact on the precision. Q: What is the max n for fact(n) in C/C++?

Variables defined as a primitive **boolean** type default to **false** on declaration within a class (i.e., heap allocated), but must be explicitly initialized within methods (i.e., stack allocated):

boolean	true or false
---------	---------------

Variables defined as one of the following primitive integral types default to a value of zero on declaration with a class (i.e., heap allocated), but must be explicitly initialized within methods (i.e., stack allocated):

byte	-128..127
short	-32768..32767
int	-2147483648..2147483647
long	-922337203685477508..9223372036854775807
char	0..65535

## Java Data Types

Characters in java are **Unicode** characters (16 bits).

The Java String class implements a Unicode character string object, so strings can contain non-ASCII character for international character sets (e.g., ISO-8859-1, Kanji, etc.). Sorting of strings uses the Unicode collating sequences

This is a very important feature in software for today, since software must be written to permit internalization of character values and strings. For example, so that multi-lingual error message catalogs can be used.

The 7-bit ASCII character set occupies the first 128 characters of the Unicode character set, so conversions to/from ASCII are straightforward.

One important “gotcha” that arises from using Unicode strings is that character set conversions are common operations. For example, most Internet protocols (e.g., SMTP, NNTP, HTTP) use ASCII, so it is necessary to convert from/to Java String objects and byte arrays when sending/receiving data over a network socket.

Java’s floating point types implement IEEE-754 single-precision (32-bit) and double precision (64-bit) values. They default to 0.0f or 0.0d when declared in a class, but must be explicitly initialized when declared in a method:

```
float  
double
```

## Arrays

Arrays are created by calling `new`, which returns a reference to the array. Arrays are automatically garbage collected when no longer referenced in any scope.

```
byte buffer[] = new byte[1024];
```

Each byte in the array 'buffer' is initialized to its default value, which is zero for integral types.

```
Button button[] = new Button[10];
```

An array of type `Button` results in an array of `Button` references, all of which are initialized to null.

If you attempt to reference an element of an array, and invoke an operation on a button object, a `NullPointerException` will be generated.

```
button[i].push();
```

So, you must explicitly initialize each array reference with a new object.

```
for (int i = 0; i < button.length; i++)  
    button[i] = new Button();
```

Note that every array has a builtin public "constant" length field, which is a read-only variable that returns the length of the array as an integer. The length is set to the extent of the array when the array is first created by a call to `new`.

## Arrays

You also can initialize an array with a static initializer:

```
String commands[] = { "File", "Edit", "Format", "Help" };
```

Which is equivalent to:

```
String[] commands = { new String("File"), new String("Edit"),  
                     new String("Format"), new String("Help") };
```

Note that in Java, you can place the ' []' syntax, indicating an array type, next to the typename T instead of next to the variable name. This syntax is perhaps more intuitive since what you are declaring is an array of some type T. The syntax of using the [] after the variable name in an array declaration is in the language as a courtesy to long-time C/C++ programmers.

For example:

```
String[] s; // declare an uninitialized reference to an array of Strings  
s = new String[100];  
s = new String[10];
```

The second time the reference variable 's' is initialized, the previous array becomes garbage and is subject to garbage collection on the next sweep of the garbage collector.

## References and Objects

The non-primitive data types in Java are **objects** and **arrays**, which are called reference types because objects and arrays are accessed using reference variables, which contain a reference to the actual object or array. When you declare a variable for a reference type as an instance variable in a class declaration, it is initialized to the special value **null**, which is a reserved word in Java.

If you declare a reference variable in a method, then you must explicitly initialize before you use it as an r-value (value on the RHS of some expression). So, it is a good habit to initialize all variables whenever you declare them.

There are NO pointers in Java. That implies that there is also no address-of operator '&', no dereference operator '\*', and no pointer access operator '->', as there are in C/C++. However, it is still possible to get an error as the result of a null pointer reference (called the `NullPointerException`).

Example:

```
Button p = null; // should explicitly initialize reference var
Button q = null;
p = new Button();
q = p;           // p and q reference the same Button object

p.setLabel("OK");
String s = q.getLabel(); // gets the value "OK" from button

p = null; // explicitly "drop" references to p and q
q = null; // which might help the gc "know" about garbage sooner
```

## Some Simple Facts about Strings

Strings have somewhat special treatment in Java because they are so often used. You can write the following:

```
String s = "Hello, world";
```

instead of

```
String s = new String("Hello, world");
```

The Java compiler turns all string literals, like "Hello, world", into heap allocated, garbage collected, String objects automatically.

Once a String object is initialized, it is treated as a constant string value.

Unlike C++, user-defined operator overloading is not allowed in Java. However, Java does provide all the standard C operators, and includes a few builtin overloads. For example, the '+' operator is overloaded for String objects, so that you can perform concatenation.

```
System.out.println("hello," + " world");
```

Many methods are defined in Java classes that take String objects as arguments and return String objects as results. It is very common for objects to define a **toString()** method, which implements a conversion to a String object. If an object is used where a String is expected, the Java run-time automatically invokes the toString method to implicitly convert the object to a String.

## Example Java Class: java.lang.String

```
public final class String {
    private char[] value; // the value is used for character storage.
    private int offset; // the first index of the storage that is used.
    private int count; // number of characters in the String.

    // Different String "Constructors"
    public String() { value = new char[0]; }

    public String(String value) {
        count = value.length();
        this.value = new char[count];
        value.getChars(0, count, this.value, 0);
    }

    public String(char[] value) {
        this.count = value.length;
        this.value = new char[count];
        System.arraycopy(value, 0, this.value, 0, count);
    }
    ... // lots of other String related methods
};
```

A **final** class means that the class cannot be extended by a subclass. Several of the `java.lang` package classes are declared as **final**. Note the `System.arraycopy` method. It copies from `value` beginning at the first position, to `this.value` beginning at the first position, for `count` elements.

## Objects

All objects by default *inherit* from a base object class called **java.lang.Object**. So, in Java all objects are **subtypes** of the type **Object**. That is, anywhere a type **Object** is called for (e.g., in a type signature of some procedure), ANY object can be used since all objects are of type **Object**.

This means that objects of any type can all be treated generically as instance of class **Object**, which is useful for defining *heterogeneous collections* of objects, such as a vector, stack, etc. In fact, the `java.util.vector` and `java.util.stack` classes are both defined to operate on objects of type **Object**.

Java objects are allocated from the heap and a garbage collector takes care of reaping objects from the heap when there are no more references to the object. When you create a new object, the result of the **new** operator is a reference to the heap allocated object that is assigned to an object reference variable. You use this object reference variable to access the public fields and methods of an object.

The **Object** class provides some high-level methods that can be called on an object of any type. For example, the **Object** class defines a `clone` method that makes a copy of an object:

You can copy objects using a special method called `java.lang.Object.clone()`, which is defined for all objects since all objects inherit from the `java.lang.Object`.

```
Vector v = new Vector();  
Vector z = v;  
Vector c = z.clone();
```

The `java.lang.Object.clone()` method does a field by field copy of the object, or what is called a “shallow” copy. It does not do a “deep” copy.

## Object Equality

Equality between objects can be tricky. You have to distinguished between equivalent references and equivalent objects. Two references are equal is they refer to the same object.

```
v == z is true  
c == z is false
```

To check to see if two distinct objects are equivalent, you call the `equals()` method for those objects. The `equals` method can be overridden by a class to implement a type specific comparison. Every object inherits the `java.lang.Object.equals()` method, which just checks to see if two references point at the same object.

For example, the statement:

```
x.equals(null)
```

for some initialized reference type variable `x` should always yield false. Why?

## The Object Class

```
package java.lang;
/**
 * Class Object is the root of the class hierarchy.
 * Every class has Object as a superclass. All objects,
 * including arrays, implement the methods of this class.
 * @author unascribed
 * @version 1.39, 01/20/97
 * @see java.lang.Class
 * @since JDK1.0
 */
public class Object {
    public final native Class getClass();
    public native int hashCode();
    public String toString();
    public boolean equals(Object obj) { return (this == obj); }
    protected native Object clone() throws CloneNotSupportedException;
    public final native void notify();
    public final native void notifyAll();
    public final native void wait(long timeout) throws InterruptedException;
    public final void wait(long timeout, int nanos) throws InterupptedExcep-
tion { ... }
    public final void wait() throws InterruptedException { wait(0); }
    protected void finalize() throws Throwable { }
}
```

## The Object.toString Method

```
/**
 * Returns a string representation of the object. In general, the
 * toString method returns a string that
 * "textually represents" this object. The result should
 * be a concise but informative representation that is easy for a
 * person to read.
 * It is recommended that all subclasses override this method.
 * <p>
 * The toString method for class Object
 * returns a string consisting of the name of the class of which the
 * object is an instance, the at-sign character @, and
 * the unsigned hexadecimal representation of the hash code of the
 * object.
 *
 * @return a string representation of the object.
 * @since JDK1.0
 */
public String toString() {
    return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```

Notice that the default `toString` method generates a string like "Foo@0x12e8FB" for a class `Foo`, which is not very informative in practice. So, it is very common (and good programming practice) for a class to reimplement the `toString` method to do something useful. This is called "overriding" the implementation of an inherited method.

## The Object.equals Method

```
/**
 * Compares two Objects for equality.
 * <p>
 * The equals method for class <code>Object</code> implements the most
 * discriminating possible equivalence relation on objects; that is,
 * for any reference values <code>x</code> and <code>y</code>, this
 * method returns <code>>true</code> if and only if <code>x</code> and
 * <code>y</code> refer to the same object (<code>x==y</code> has the
 * value <code>>true</code>).
 *
 * @param   obj    the reference object with which to compare.
 * @return  <code>>true</code> if this object is the same as the obj
 *         argument; <code>>false</code> otherwise.
 * @see     java.lang.Boolean#hashCode()
 * @see     java.util.Hashtable
 * @since   JDK1.0
 */
public boolean equals(Object obj) {
    return (this == obj);
}
```

**Notice that the default equals method implements a reference equality relationship, not a semantic equality relationship. A class must implement semantic equality explicitly by overriding the `Object.equals` method.**

## For Example: String Equality

The `java.lang.Object.equals()` method is overridden by the `String` class to perform a string equality check. The type signature of the `equals` method takes a single `Object` reference, which must be type cast “down” from `Object` to a `String`, but only if at run-time the object is an “instance of” the `String` class. The builtin **instanceof** operator returns true if the object on the LHS is an instance of the class on the RHS, or implements the interface of the class on the RHS. It returns false if the object on the LHS is not an instance of the class, or if the object is null. Note that the argument of type `Object` must be explicitly “downcast” to a string after the run-time “instance of” check is performed. If you downcast from `Object` to a incorrect type, then a run-time “invalid type cast” exception will be thrown.

```
public boolean equals(Object obj) {
    if ((obj != null) && (obj instanceof String)) {
        String otherString = (String)obj; // safe downcast!
        int n = this.count;
        if (n == otherString.count) {
            char v1[] = this.value;
            char v2[] = otherString.value;;
            int i = this.offset;
            int j = otherString.offset;
            while (n-- != 0)
                if (v1[i++] != v2[j++]) return false;
            return true;
        }
    }
    return false;}
}
```

## The Object.clone Method

```
/**
 * Creates a new object of the same class as this object. It then
 * initializes each of the new object's fields by assigning it the
 * same value as the corresponding field in this object. No
 * constructor is called.
 * <p>
 * The <code>clone</code> method of class <code>Object</code> will
 * only clone an object whose class indicates that it is willing for
 * its instances to be cloned. A class indicates that its instances
 * can be cloned by declaring that it implements the
 * <code>Cloneable</code> interface.
 *
 * @return      a clone of this instance.
 * @exception   CloneNotSupportedException if the object's class does not
 *             support the <code>Cloneable</code> interface. Subclasses
 *             that override the <code>clone</code> method can also
 *             throw this exception to indicate that an instance cannot
 *             be cloned.
 * @exception   OutOfMemoryError         if there is not enough memory.
 * @see         java.lang.Cloneable
 * @since       JDK1.0
 */
protected native Object clone() throws CloneNotSupportedException;
```

## The `Object.finalize` Method

```
/**
 * Called by the garbage collector on an object when garbage collection
 * determines that there are no more references to the object.
 * A subclass overrides the finalize method to dispose of
 * system resources or to perform other cleanup.
 * <p>
 * Any exception thrown by the finalize method causes
 * the finalization of this object to be halted, but is otherwise
 * ignored.
 * <p>
 * The finalize method in Object does
 * nothing.
 *
 * @exception java.lang.Throwable [Need description!]
 * @since      JDK1.0
 */
protected void finalize() throws Throwable { }
```