

The “this” reference

Every object in Java is uniquely identified by its **this** reference. The **this** reference is the “handle” (pointer) to the object that is used to invoke methods or explicitly reference instance variables.

When we create an object by calling `new`, a reference variable is initialized with the value of the object’s location in memory. When we use a reference variable to invoke an operation on an object, we are really invoking a method that is associated with some class, and we dynamically associate the state of some particular object with the method.

```
String s = new String("This is a string");  
int x = s.length(); // compute length of string
```

The compiler turns the invocation of `s.length` into the following procedure call:

```
int x = String.length(this=s);
```

So, the `length` method of the `String` class is invoked, and it will have a **this** reference that is initialized to the memory address of the object pointed to by `s`. Conceptually, what you have to keep in mind in an object oriented language like Java or C++, is that an object at run-time is just a chunk of memory somewhere in the heap that contains the data elements of the object, along with some bookkeeping information generated by the compiler for the object. At run-time, the object consists of data in the heap, and a single copy of the methods for all objects of a particular class are separate.

When you invoke a method on an object, the method and the object’s state are dynamically “tied together” for the duration of the method’s execution. The “**this**” reference is the mechanism that is used to tie them together at run-time. The **this** reference as a hidden first argument to every non-static class method. Static “class” methods do not have a **this** reference passed to them.

Using the “this” reference in a class

Most of the time, you will not explicitly refer to the **this** reference, unless you are invoking a constructor from another constructor, as shown previously. However, some people prefer a programming style that requires the use of the **this** reference to avoid variable name ambiguity:

```
public class Complex {
    private double real, imag;

    Complex(double real, double imag) { this.real = real; this.imag = imag; }
}
```

In this case, the programmer wants to convey to the user of the class as much information as possible about the arguments expected by the constructor, so the same names as the local variables are used as formal parameters. However, it is required that the **this** reference be used to distinguish the local **real** and **imag** variables from those used as formal parameters to the constructor. So, you sometimes see this explicit use of the **this** pointer when referring to local instance variables. It is not possible to refer to the **this** variable in a static method of a class.

```
class Foo {
    int x;
    static int f() { return this.x; } // ERROR
}
```

Since **f()** is static, it is called on the class **Foo**, not an object of type **Foo**. So, it is NOT the case that **f()** is passed an implicit first argument to which the “**this**” reference is initialized. So, within the body of **f()**, the “**this**” reference is uninitialized, and therefore unusable.

Static methods

The most common static method you will write is the **main** method., but static methods are used all the time in Java. For example, to create new objects initialized with some specific properties.

```
import java.net.*; // need to import InetAddress class
public class Host {
    static String host;
    static String user;
    static final int err = -1; // same as const int err = -1 in C++

    public static void main (String[] args)
    {
        try {
            InetAddress addr = InetAddress.getLocalHost();
            host = addr.getHostName();
            user = System.getProperty("user.name");
            System.out.println(user + "@" + host); // string concatenation
        }
        catch (Exception e) { System.err.println(e); System.exit(err); }
    }
}
```

The call to `InetAddress.getLocalHost()` returns a new `InetAddress` object, containing the Internet address of the local machine on which the program is running. Using that `InetAddress` object, you can then obtain the hostname by calling the non-static `getHostName` method on the `InetAddress` object. Similarly, the `System.getProperty` method is a static method defined as part of the `java.lang.System` class that returns a `String` containing the value of system property "user.name".

Static data

The `java.lang.System` class is one that you should familiarize yourself with, as it consists of mostly static methods and static data. Three static `System` class data members are very important:

```
java.lang.System.in  
java.lang.System.out  
java.lang.System.err
```

```
import java.io.*; // open Java I/O package  
public static void main(String[] args) {  
    String input = null; // local vars should be explicitly initialized  
    BufferedReader rdr = new BufferedReader(new InputStreamReader(System.in));  
    // read line buffered input from System.in until EOF  
    try {  
        do {  
            input = rdr.readLine();  
            if (input != null)  
                System.out.println(input);  
        } while (input != null);  
    } catch (IOException e) { System.err.println(e); System.exit(-1); }  
}
```

`System.in` is defined as an `InputStream` object, which is not line buffered, so you have to create an `InputStreamReader` object and use it to initialize a line `BufferedReader` object. In Java, you will find that you have several different types of Input/Output stream objects, and you need to be able to convert them for different uses.

Static data initialization

Static data is usually initialized at the point of declaration within a class:

```
class Program {
    static String errorMessage = "fatal error";
    static int errorCode = -1;
    ...
}
```

Alternatively, you can use a **static block** when the initialization is more complicated.

```
class Program {
    static String[] errMessages;
    static int errorCode;

    static {
        errorCode = -1;
        errMessages = new String[10];
        errMessages[0] = "fatal error";
        ...
    }
    ...
}
```

A static block is NOT the same thing as a constructor. Static variables are initialized when either a static method first is called on the class that contains the static data, or when the first object of the class is first instantiated.

“Wrapper” Classes

Java builtin types: `boolean`, `char`, `byte`, `short`, `int`, `long`, `float`, `double` are not objects. They are designed for efficiency. However, since all other types in Java are subtypes of the type `Object`, it is often necessary to have objects representing values of the builtin types.

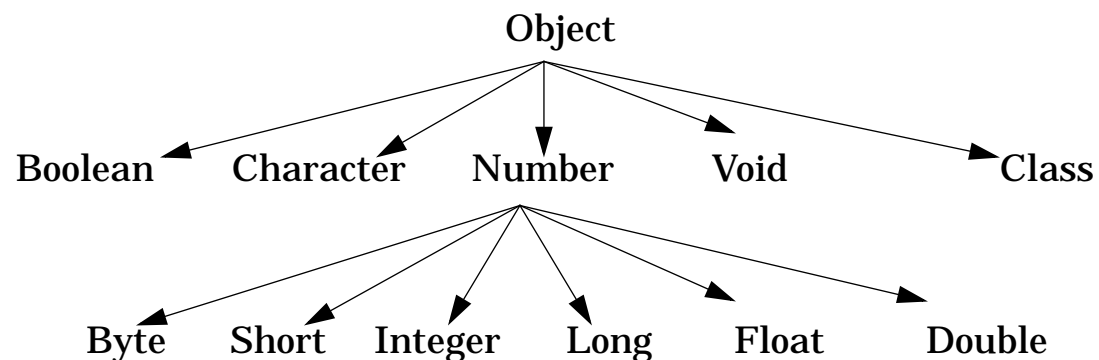
For example, a heterogeneous collection of Objects:

```
Object[] collection = new Object[10];
```

We cannot put an ‘`int`’ type into this collection, because an ‘`int`’ is not type compatible with the type `Object`. We can however put an **Integer** object (or any other object of some class type) into the collection.

```
collection[0] = new Integer(5);
```

To allow collections of objects to include values of builtin types, the core Java language package (`java.lang`) defines a set of **Wrapper Classes** for each of the builtin types, which inherit from the class `Object`. The inheritance hierarchy for the Wrapper classes is as follows:



Number “wrapper” class structure

```
package java.lang;

public final class wrapper-class-name extends Number {
    public static final builtin-typename MIN_VALUE = constant-expr;
    public static final builtin-typename MAX_VALUE = constant-expr;
    public static final Class TYPE =
        Class.getPrimitiveClass("builtin-typename");
    private builtin-typename value; // the primitive value

    public wrapper-class-name (builtin-typename value) { this.value = value; }
    public wrapper-class-name (String s) throws NumberFormatException
        { stmt-list; }
    public static wrapper-class-name valueOf(String s)
        throws NumberFormatException { stmt-list; }
    public boolean equals(Object obj) { stmt-list; }

    // builtin type conversions methods
    public String toString () { stmt-list; }
    public byte byteValue () { return (byte)value; }
    public short shortValue () { return (short)value; }
    public int intValue () { return (int)value; }
    public long longValue () { return (long)value; }
    public float floatValue () { return (float)value; }
    public double doubleValue () { return (double)value; }
}
```

Integer.java: an example wrapper class implementation

```
package java.lang;

public final class Integer extends Number {
    public static final int MIN_VALUE = 0x80000000;
    public static final int MAX_VALUE = 0x7fffffff;
    public static final Class TYPE = Class.getPrimitiveClass("int");
    private int value;

    public Integer(int value) { this.value = value; }
    public Integer(String s) throws NumberFormatException
    { this.value = parseInt(s, 10); }
    public static Integer valueOf(String s) throws NumberFormatException
    { return new Integer(parseInt(s, 10)); }

    public boolean equals(Object obj) {
        if ((obj != null) && (obj instanceof Integer)) {
            return value == ((Integer)obj).intValue();
        }
        return false;
    }

    public String toString() { return String.valueOf(value); }

    // builtin type conversion methods....
}
```

Wrapper class usage

Note however that to obtain an 'int' back from an Object, you have to perform some explicit **type conversions.**

```
// explicitly type cast "down" from an Object to an Integer
// an invalid type cast exception will be thrown if cast is to
// the wrong type
Integer i = (Integer) collection[0];

// explicitly convert to a String
String s = i.toString();

// implicitly convert to a String for printing
System.out.println( i ); // implicitly calls i.toString()

// explicitly convert back to a primitive int, or byte, etc.
// perhaps with loss of precision

int x = i.intValue();
byte b = i.byteValue();
long l = i.longValue();
float f = i.floatValue();
double d = i.doubleValue();
```

Wrapper class usage

Wrapper classes also provide some public static class methods to provide common type conversions:

```
boolean b = Boolean.toBoolean("true");
```

```
int x = Integer.parseInt("10");
```

These String to Wrapper type conversions are most useful when converting String arguments to values at run-time:

```
public static void main(String[] args) {  
    try {  
        boolean b = Boolean.valueOf(args[0]);  
        int x = Integer.parseInt(args[1]);  
    }  
    catch (ArrayIndexOutOfBoundsException e) {  
        System.err.println("Usage: pgm boolean-value integer-value");  
        System.exit(-1);  
    }  
    catch (NumberFormatException e) {  
        System.err.println("Second argument must be an integer value!");  
        System.exit(-1);  
    }  
}
```