Primitive Data and Objects

The programmer computes on data. Data in Java is divided into primitive data and non-primitive data.

`int` is primitive, `String` is not. `double` is primitive, arrays are not.

(Wrapper classes allow primitive data to be treated like objects. The advantage of using wrapper classes is that all data can be treated uniformly. The disadvantage is some extra overhead.)
Data Abstraction

Good design enables easy handling and high volume.
Data Abstraction

Programmer not only designs the flow of control (loops, etc), but also designs the value to compute with. This may be the most important design task the programmer has, since the control flow could spring naturally from that data if the design is good.
Usual characteristics of primitive data:

- small size, all values fit in the machine’s word size
- uncomplicated (no substructure)
- operations on the data are supported in the hardware

The majority of data the programmer wants is not primitive: images, music, graphs, sequences, etc. Java supports this vast, endless variety of data by allowing programmer to define new data types and by implementing some in the libraries.

User-defined data types are created using classes. Instances of classes are allocated in a managed storage area called the heap and variables in the program refer indirectly to the instances. We call these instances *objects*, but referenced data might be a less overused term.
### Storage—Finite Collection of Machine Words

<table>
<thead>
<tr>
<th>Address</th>
<th>Binary Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>0100 1101</td>
</tr>
<tr>
<td>0x01</td>
<td>1100 0001</td>
</tr>
<tr>
<td>0x02</td>
<td>0111 1111</td>
</tr>
<tr>
<td>0x03</td>
<td>1101 1011</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td>0xFC</td>
<td>1101 1011</td>
</tr>
<tr>
<td>0xFD</td>
<td>1101 1010</td>
</tr>
<tr>
<td>0xFE</td>
<td>0000 1100</td>
</tr>
<tr>
<td>0xFF</td>
<td>0100 1010</td>
</tr>
</tbody>
</table>
## Storage—Finite Collection of Cells With Infinite Values

<table>
<thead>
<tr>
<th>Hex</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>37</td>
</tr>
<tr>
<td>0x01</td>
<td>121</td>
</tr>
<tr>
<td>0x02</td>
<td>0</td>
</tr>
<tr>
<td>0x03</td>
<td>23786341</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td>0xFD</td>
<td>3589318</td>
</tr>
<tr>
<td>0xFE</td>
<td>123743</td>
</tr>
<tr>
<td>0xFF</td>
<td>8276</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>----</td>
<td>----</td>
</tr>
<tr>
<td>a</td>
<td>37</td>
</tr>
<tr>
<td>b</td>
<td>121</td>
</tr>
<tr>
<td>c</td>
<td>0</td>
</tr>
<tr>
<td>d</td>
<td>23786341</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>w</td>
<td>1574</td>
</tr>
<tr>
<td>x</td>
<td>3589318</td>
</tr>
<tr>
<td>y</td>
<td>123743</td>
</tr>
<tr>
<td>z</td>
<td>8276</td>
</tr>
</tbody>
</table>
Storage

During execution the entire program is laid out in (virtual) storage, which we can envisioned as a gigantic array of words indexed by a (virtual) address. All the data appears somewhere in storage.
Layout of Program in Memory

0x4029 00 00 2F 87 String s

0x3D2A 00 00 00 7B int i

"Hello"
Layout of Program in Memory

```
0x403A
0x4029 00 00 2F 87
0x4028 ...
0x2F87 00 00 00 05
0x2F86 'H' 'e' 'l' 'l'
0x2F85 'o'
0x2F84
```

String s

(length)
Primitive Data and Objects

Primitive data values are stored directly ("unboxed") and objects are stored indirectly ("boxed"). For example,

```java
int i = 123; String s = "Hello";
```

![Diagram showing memory allocation for int and String objects]
The space for objects is found in the heap. Of course, the heap must be found somewhere in storage.
Other Languages

Other languages, like Haskell and C# do not require the programmer to distinguish between boxed and unboxed data. But both languages have boxed and unboxed values, integers, for instance. But in these languages the programmers only have one data type for integers. Unboxed is more efficient for computation and boxed is more uniform. Haskell and C# go back and forth between the two implementations automatically. Java too, goes back and forth automatically. But for each primitive data type there are two distinct types in the language: for example, `int` and `Integer`.
Comparing Integer Wrapper Class With Primitive int
Creating Objects

When you declare a variable for a value of a primitive data type, an address is assigned which *must* hold a value of the primitive type (int, etc.)
When you declare a variable for objects, a box is assigned which can hold a reference to an instance of that type. No object-instance is created. An object-instance can only be created by `new`.
All non-primitive data objects are created (directly or indirectly) by executing the `new`.
The syntax of the `new` expression:

```
new <class name> ( [ arguments ] )
```

it creates and returns an object of type `class name`. 
There is an implicit `new` in special cases: strings, arrays, wrapper classes.

```java
String s = new String("abc");  // redundant
Integer i = new Integer(123);
int[] a = new int[] {1,2,3};

String s = "abc";
Integer i = 123;  // Auto-boxing!
int[] a = {1,2,3};  // new is optional in decl
```
new java.lang.Object ()
new java.lang.StringBuilder (s)
new java.math.BigDecimal (203.99)
new java.awt.Color (r,g,b)
new java.util.Scanner (System.in)
new java.util.Locale (lang, country)
new java.io.File (dir, name)
new java.io.URL (protocol, host, file)
new java.util.ArrayList<String> ()
new javax.swing.JApplet ()

- 0, 1, or more arguments
- package name may be omitted if one uses an import
- generic instantiation
- some classes are never instantiated, i.e, Math
- other ways to get objects: “factory methods,” methods which create the object for you.
It is (regrettably) necessary to have a clear picture of boxed and unboxed values in your mind, in order to program correctly in Java.
Let me try to illustrate what I think is in my head.
Joan Miró i Ferrà (1893–1983), *Frustrated Cat*
int i = 123;
BigDecimal b = new BigDecimal (212.99);
long[] a = new long[] {1,2,3};
float g = 1000.25f;
String s = "Hello";
float f = -1.5f;

String s
BigDecimal b
float f 00 00 00 7B
long[] a 44 7A 10 00
int i 00 00 00 00
float g BF C0 00 00

"Hello"
212.99
[1,2,3]
int i;
BigDecimal b;
long[] a;
float g;
String s;
float f;

<table>
<thead>
<tr>
<th>String s</th>
<th>null</th>
</tr>
</thead>
<tbody>
<tr>
<td>BigDecimal b</td>
<td>null</td>
</tr>
<tr>
<td>float f</td>
<td>00 00 00 00</td>
</tr>
<tr>
<td>long[] a</td>
<td>null</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>int i</td>
<td>00 00 00 00</td>
</tr>
<tr>
<td>float g</td>
<td>00 00 00 00</td>
</tr>
</tbody>
</table>
i = 123;
b = new BigDecimal (212.99);
a = new long[] {1,2,3};
g = 1000.25f;
s = "Hello";
f = -1.5f;
i = 123;
b = new BigDecimal (212.99);
a = new long[] {1,2,3};
g = 1000.25f;
s = "Hello";
f = -1.5f;
i = 123;
b = new BigDecimal (212.99);
a = new long[] {1,2,3};
g = 1000.25f;
s = "Hello";
f = -1.5f;
i = 123;
b = new BigDecimal (212.99);
a = new long[] {1,2,3};
g = 1000.25f;
s = "Hello";
f = -1.5f;
i = 123;
b = new BigDecimal (212.99);
a = new long[] {1,2,3};
g = 1000.25f;
s = "Hello";
f = -1.5f;
i = 123;
b = new BigDecimal (212.99);
a = new long[] {1,2,3};
g = 1000.25f;
s = "Hello";
f = -1.5f;
i = 123;
b = new BigDecimal (212.99);
a = new long[] {1,2,3};
g = 1000.25f;
s = "Hello";
f = -1.5f;
i = 123;
b = new BigDecimal (212.99);
a = new long[] {1,2,3};
g = 1000.25f;
s = "Hello";
f = -1.5f;
i = 123;
b = new BigDecimal (212.99);
a = new long[] {1,2,3};
g = 1000.25f;
s = "Hello";
f = -1.5f;
i = 123;
b = new BigDecimal (212.99);
a = new long[] {1,2,3};
g = 1000.25f;
s = "Hello";
f = -1.5f;
i = 123;
b = new BigDecimal(212.99);
a = new long[]{1, 2, 3};
g = 1000.25f;
s = "Hello";
f = -1.5f;
i = 123;
b = new BigDecimal (212.99);
a = new long[] {1,2,3};
g = 1000.25f;
s = "Hello";
f = -1.5f;
i = 123;
b = new BigDecimal (212.99);
a = new long[] {1,2,3};
g = 1000.25f;
s = "Hello";
f = -1.5f;
Null

Non-primitive variable are initialized to the special value null. Null is a legal value for all non-primitive types. Unintentionally accessing a null object is a very common problem. NullPointerException.

String s;
long [] a;
char c = s.charAt(17);
long l = a[39];

Java catches some (but not all) initialization errors.
```java
public class Main {
    public static void main (String[] args) {
        String s;
        long [] a;
        char c = s.charAt(17);
        long l = a[39];
    }
}

> javac Main.java
Main.java:5: variable s might not have been initialized
    char c = s.charAt(17);
           ^
Main.java:6: variable a might not have been initialized
    long l = a[39];
           ^
2 errors
```
public class Fields {
    static String s;
    static long [] a;
    public static void main (String[] args) {
        char c = s.charAt(17);
        long l = a[39];
    }
}

> java Fields
Exception in thread "main" java.lang.NullPointerException
    at Fields.main(Fields.java:5)
I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn’t resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.

Sir Charles Athony Hoare
Garbage

```
i = 0;
b = null;
a = null;
g = 0.0f;
s = null;
f = 0.0f;
```
Garbage

```java
int i = 0;
BigDecimal b = null;
String a = null;
float g = 0.0f;
String s = null;
float f = 0.0f;
```

```
String s
BigDecimal b
float f 00 00 00 00
long[] a
int i 00 00 00 00
float g 44 7a 10 00
```

"Hello"

212.99

[1,2,3]
Garbage

```java
int i = 0;
b = null;
a = null;
g = 0.0f;
s = null;
f = 0.0f;
```

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>&quot;Hello&quot;</td>
</tr>
<tr>
<td>BigDecimal</td>
<td>212.99</td>
</tr>
<tr>
<td>float</td>
<td>BF C0 00 00</td>
</tr>
<tr>
<td>long[]</td>
<td>[1,2,3]</td>
</tr>
<tr>
<td>int</td>
<td>00 00 00 00</td>
</tr>
<tr>
<td>float</td>
<td>44 7a 10 00</td>
</tr>
</tbody>
</table>
Garbage

i = 0;
b = null;
a = null;
g = 0.0f;
s = null;
f = 0.0f;

String s
BigDecimal b
float f
long[] a
int i
float g
Garbage

```java
i = 0;
b = null;
a = null;
g = 0.0f;
s = null;
f = 0.0f;
```

```
String s
BigDecimal b  null
float f  BF C0 00 00
long[] a  null
 int i  00 00 00 00
float g  00 00 00 00
```

"Hello"

212.99

[1,2,3]
Garbage

```java
i = 0;
b = null;
a = null;
g = 0.0f;
s = null;
f = 0.0f;
```

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>String s</td>
<td>null</td>
</tr>
<tr>
<td>BigDecimal b</td>
<td>null</td>
</tr>
<tr>
<td>float f</td>
<td>BF C0 00 00</td>
</tr>
<tr>
<td>long[] a</td>
<td>null</td>
</tr>
<tr>
<td>int i</td>
<td>00 00 00 00</td>
</tr>
<tr>
<td>float g</td>
<td>00 00 00 00</td>
</tr>
</tbody>
</table>

"Hello"

212.99

[1,2,3]
Garbage

```java
i = 0;
b = null;
a = null;
g = 0.0f;
s = null;
f = 0.0f;
```

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>String s</td>
<td>null</td>
</tr>
<tr>
<td>BigDecimal b</td>
<td>null</td>
</tr>
<tr>
<td>Float f</td>
<td>0.0f</td>
</tr>
<tr>
<td>Long[] a</td>
<td>null</td>
</tr>
<tr>
<td>Integer i</td>
<td>0</td>
</tr>
<tr>
<td>Float g</td>
<td>0.0f</td>
</tr>
</tbody>
</table>

"Hello"

212.99

[1, 2, 3]
i = 0;
b = null;
a = null;
g = 0.0f;
s = null;
f = 0.0f;
Garbage

HEAP

garbage
Java and Garbage Collection

Java does garbage collection.

One of Java’s most significant features is its ability to automatically manage memory. The idea is to free the programmers from the responsibility of managing memory by keeping track of orphaned objects and returning the memory they use to a pool of free memory.

Sedgwick & Wayne, CS: An Interdisciplinary Approach, page 357
Immutability

In *Effective Java*, Joshua Bloch makes this important recommendation:

*Classes should be immutable unless there's a very good reason to make them mutable. If a class cannot be made immutable, limit its mutability as much as possible.*
Immutability
Mutability Leads To Disaster

Mutability

• There is no direct support of immutability in the Java programming language.
• Therefore, one must learn to design immutable data structures using the features of the Java programming language.
Mutable Objects

Objects in Java, like all data, can be divided into two types: mutable and immutable. An *immutable* object is one that no operation can change. For example, an object of type `String`. A good example of a mutable object is an array (of any type). Changing an element of an array, changes the state of the array. Thus an array is a *mutable* object, an object that “has a state which may be modified by certain operations without changing the identity of the object.”

Do not confuse immutable with the term *constant*. An identifier is said to be *constant* if it always refers to the same object (immutable or not). An object is said to be *immutable* if no operations can change it. All primitive types are immutable. Not all objects are mutable.
```java
int i = 123;
StringBuilder b = new StringBuilder("world");
StringBuilder f = b
int j = i;
String s = "Hello";
String t = s;
f.deleteCharAt(3);
```
int i = 123;
StringBuilder b = new StringBuilder ("world");
StringBuilder f = b
int j = i;
String s = "Hello";
String t = s;
f.deleteCharAt (3);
```java
int i = 123;
StringBuilder b = new StringBuilder ("world");
StringBuilder f = b
int j = i;
String s = "Hello";
String t = s;
f.deleteCharAt (3);
```

```
int i 00 00 00 7B
```
int i = 123;
StringBuilder b = new StringBuilder ("world");
StringBuilder f = b
int j = i;
String s = "Hello";
String t = s;
f.deleteCharAt (3);
int i = 123;
StringBuilder b = new StringBuilder("world");
StringBuilder f = b
int j = i;
String s = "Hello";
String t = s;
f.deleteCharAt(3);
int i = 123;
StringBuilder b = new StringBuilder("world");
StringBuilder f = b
int j = i;
String s = "Hello";
String t = s;
f.deleteCharAt(3);
```java
int i = 123;
StringBuilder b = new StringBuilder("world");
StringBuilder f = b
int j = i;
String s = "Hello";
String t = s;
f.deleteCharAt(3);
```
int i = 123;
StringBuilder b = new StringBuilder ("world");
StringBuilder f = b
int j = i;
String s = "Hello";
String t = s;
f.deleteCharAt (3);
int i = 123;
StringBuilder b = new StringBuilder("world");
StringBuilder f = b
int j = i;
String s = "Hello";
String t = s;
f.deleteCharAt(3);
```java
int i = 123;
StringBuilder b = new StringBuilder("world");
StringBuilder f = b
int j = i;
String s = "Hello";
String t = s;
f.deleteCharAt(3);
```
int i = 123;
StringBuilder b = new StringBuilder ("world");
StringBuilder f = b
int j = i;
String s = "Hello";
String t = s;
f.deleteCharAt (3);
```java
int i = 123;
StringBuilder b = new StringBuilder("world");
StringBuilder f = b
int j = i;
String s = "Hello";
String t = s;
f.deleteCharAt(3);
```
int i = 123;
StringBuilder b = new StringBuilder ("world");
StringBuilder f = b
int j = i;
String s = "Hello";
String t = s;
f.deleteCharAt (3);
int i = 123;
StringBuilder b = new StringBuilder ("world");
StringBuilder f = b
int j = i;
String s = "Hello";
String t = s;
f.deleteCharAt (3);
Where does sharing come from? Obviously it comes from assignment. But what causes some of the biggest problems, is we often overlook that that sharing comes from method calls. Parameter passing in Java (call by value) is like an assignment to a local variable and it causes sharing for all non-primitive types.

The programmer has no choice in parameter passing and so must always be on the defensive when using mutable objects.

What if sort invoked the clear method on your data?!

```java
ArrayList<Integer> list = // ...
Collections.veryBadSort(list);
```

A means of protection:

```java
return Collections.unmodifiableList(list);
```
Sharing is cheap, but buggy; copying (large objects) is expensive, but safe. Immutable objects can be shared without problems, Always design for immutability; optimize later.

*premature optimization is the root of all evil*

Knuth, 1974 Turing Award lecture
Sharing Mutable

mutável

sharing
Sharing Immutable

sharing

immutable
The Stansifer sayings:

It is easier to make a correct program more efficient than to make a buggy program more correct.

A program that does what you think it does is much better than a program that might do what you want it do.
Reference Counting

stack

heap
class name { members }

Members may be static or instance.
Members may be methods (subprocedures) or fields (data values).

<table>
<thead>
<tr>
<th>static</th>
<th>instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>methods</td>
<td></td>
</tr>
<tr>
<td>fields</td>
<td></td>
</tr>
</tbody>
</table>
Static member are accessed: `ClassName.member`. Instance members are accessed: `expressionDenotingAnInstance.member`. Very important to observe the capitalization convention and never to access static members like this:

`expressionDenotingInstanceOfClass.member`

(which is legal but bewilders the reader).

Instance members can access both static and instance members. (But don’t take advantage of this.) Static members can only access static members.
public class Main {
    static int field = 123; // static member
    public static void main (String[] a) {
        System.out.println (new Main().field);
    }
}

public class Main {
    int field = 123;  // instance field
    public static void main (String[] a) {
        System.out.println (Main.field);
    }
}

Not caught by any checkstyle check, but is a translation warning:

javac -Xlint:static Main.java
Main.java:4: warning: [static] static variable should be by type name, Main, instead of by an expression
Parameter Passing

Call by value is about protection. Information does not flow back it is not about objects nor 'final'.

- **PassByValue.java** – Java uses "call by value"
- **PassByWrapper.java** – Using wrapper class does not provide means of creating "out" parameters.
- If you want to return something, there are no “out” parameters; use a function
Returning information

Use function return values. Don’t use secret dead drops.
Module java.base
Package java.util

Class Optional<T>

java.lang.Object
  java.util.Optional<T>

Type Parameters:
T - the type of value

public final class Optional<T>
extends Object

A container object which may or may not contain a non-null value. If a value is present, then
additional methods that depend on the presence or absence of a contained value
are available.

This is a value-based class; programmers should treat instances that are equal
as the same.

API Note:
Optional is primarily intended for use as a method return type where there
is no Optional instance.

Since:
1.8