Recap on Java Programming

Initialization

```
public class Foo {
    static int bar;

    public static void main (String args []) {
        bar += 1;
        System.out.println("bar = " + bar);
    }
}
```

Does the code compile? If no, why?
Does the code throw a runtime exception?
If yes, why? If no, what is the output?

The output is:
`bar = 1`

Initialization

```
public class Test {
    private int a = getB();
    private int b = 5;

    private int getB() {
        return b;
    }

    public static void main(String args[]) {
        System.out.println((new Test()).a);
    }
}
```

Does the code compile? If no, why?
Does the code throw a runtime exception?
If yes, why? If no, what is the output?

The output is:
0

Initialization

```
public class Test {
  private int b = 5;
  private int a = getB();

  private int getB() {
      return b;
  }

  public static void main(String args[]) {
      System.out.println((new Test()).a);
  }
}
```

Does the code compile? If no, why?
Does the code throw a runtime exception?
If yes, why? If no, what is the output?

The output is:
5

Exceptions

```
int i=1, j=1;
try {
   i++;
   j--;
   if (i/j > 1)
      i++;
} catch(ArithmeticException e) {
   System.out.println(1);
} catch(Exception e) {
   System.out.println(2);
} finally {
   System.out.println(3);
}
```

What is the output?

The output is:
1
3

On Parameter Passing

- All Java parameters are called by value
  - Inside the called method, act like local variables
  - Value of the parameter can be changed
- Changes do not affect value in the calling method
- Parameter value might itself be a reference to an object
  - Changes to the parameter do not affect value of the reference in the calling method
  - Values inside the object referred to can be changed
- Maximum one value returned as the function value
  - Can be a primitive or a reference to an object
- Parameter values cannot be changed
  - Values within an object referred to by a parameter can be changed

Pass by Value

```
public class PassTest1{
  public static void changeInt(int value) {
      value = 55;
  }
  public static void main(String args[]) {
      int val;
      val = 11; //Assign the int
      changeInt(val); //Try to change it
      //What is the current value?
      System.out.println(val);
  }
}
```

Does the code compile? If no, why?
Does the code throw a runtime exception?
If yes, why? If no, what is the output?

The output is:
11

Pass by Value

```
public class PassTest2 {
 public static void
 changeObjectRef(MyPoint ref){
  ref = new MyPoint(1,1); }
 public static void
  main(String args[]) {
   MyPoint point;
   //Assign the point
   point = new MyPoint(22,7);
   //Try to change it
   changeObjectRef(point);
   //What is the current value?
   System.out.println(point);
  }
}
```

```
public class MyPoint {
 int x;
 int y;
 public MyPoint(int x,int y){
  this.x = x;
  this.y = y; }
 @Override
 public String toString() {
  return "("+x+","+y+")"; }
}
```

Does the code compile? If no, why?
Does the code throw a runtime exception?
If yes, why? If no, what is the output?

The output is:
(22,7)

Pass by Value

```
public class PassTest3 {
 public static void
  changeObjectAttr(MyPoint ref){
    ref.setX(4); }
 public static void main(String
   args[]) {
    MyPoint point;
    //Assign the point
    point= new MyPoint(22,7);
    changeObjectAttr(point);
    //What is the current value?
    System.out.println(point);
    }
}
```

```
public class MyPoint {
 int x;
 int y;
 public MyPoint(int x,int y){
  this.x = x;
  this.y = y; }
 public void setX(int x) {
  this.x = x; }
  @Override
 public String toString() {
  return "("+x+","+y+")"; }
}
```

Does the code compile? If no, why?
Does the code throw a runtime exception?
If yes, why? If no, what is the output?

The output is:
(4,7)

9

Pitfall: Final Parameters

- The notion of final parameters in Java is rather weak
  - A final parameter of a method may not be assigned a new value in the body of the method. However, if the parameter is of reference type, it is allowed to modify the object or array referenced by the final parameter
- Let's consider the following method :

```
void aMethod(final IntRef i) {
      // ...
      i = new IntRef(2);    // not allowed
}
```

  it will cause a compilation error because it is not allowed to assign a final parameter a new value
- However, the following method is allowed

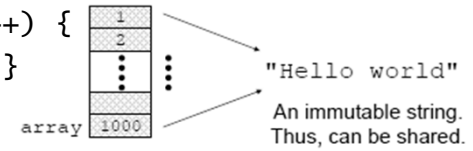
```
void aMethod(final IntRef i) {
      // ...
      i.val++;         //ok
}
```

10

On String Interning and Immutability

- Avoids duplicate strings

```
String[] array = new String[1000];
for (int i=0 ; i<1000 ; i++) {
array[i] = "Hello world"; }
```



- Strings are constants

```
String s = " Tea ";
s = s.trim();
s = s.replace('T', 'S');
```

- A string reference may be set:

```
String s = "Tea";
s = "Sea";
```

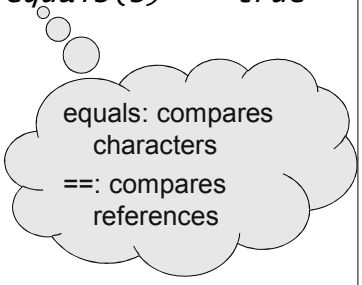
- All string literals and string-valued constant expressions are interned

11

On String Interning and Immutability

- The `String` class has a static private **pool** of internal strings
  - `myString.intern()` implementation:

```
if exists s in pool: myString.equals(s) == true
 return s;
else
 add myString to the pool
 return myString;
```



- Use implicit constructor:

```
String s = "Hello";
```

(string literals are interned)
Instead of:

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

(causes extra memory allocation)

12

On Substrings and Concatenation

- Substrings are created without copying
  - `String.substring()` is very efficient

```
String smile = "smile";
String mile = "smile".substring(1,4);
```

- String conversion and concatenation:

```
"Hello " + "World" is "Hello World"
"19" + 8 + 9 is "1989"
```

  - Conversion by `toString()`
  - Concatenation by `StringBuffer`
- Example: `String x = "19" + 8 + 9;`
  - is compiled to the equivalent of:

```
String x = new StringBuffer().
        append("19").append(8).append(9).toString();
```


String Interning

```
public static void main(String args[]) {
 if ("hello".substring(0) == "hello")
  System.out.println("statement 1 is true");

 if ("hello".substring(1) == "ello")
  System.out.println("statement 2 is true");

 if ("hello".replace('l','l') == "hello")
  System.out.println("statement 3 is true");

 if ("hello".replace('h','H') == "Hello")
  System.out.println("statement 4 is true");

 if ("hello".replace('h','H') == "hello".replace('h','H'))
  System.out.println("statement 5 is true");

}
```

What is the output?

The output is:

```
statement 1 is true
statement 3 is true
statement 4 is true
```

String Interning

```
public static void main(String args[]) {
  String s1 = "he";
  String s2 = "he" + "llo";
  String s3 = s1 + "llo";

 if (s2.equals(s3))
  System.out.println("statement 1 is true");
 if (s2 == "hello")
  System.out.println("statement 2 is true");
 if (s2 == s3)
  System.out.println("statement 3 is true");
 if (s2 == new String("hello"))
  System.out.println("statement 4 is true");
 if (s2 == s3.intern())
  System.out.println("statement 5 is true");
 if (s3 == s2.intern())
  System.out.println("statement 6 is true");
}
```

What is the output?

The output is:

```
statement 1 is true
statement 2 is true
statement 5 is true
```


`String` vs. `StringBuffer`

- Inefficient version using `String` (immutable):

```
public static String duplicate(String s, int times)
 {
 String result = s;
 for (int i=1; i<times; i++) {
  result = result + s; }
 return result;
}
```

`String.concat` creates a new `String` object

- More efficient version using `StringBuffer` (mutable):

```
public static String duplicate(String s, int times)
 {
 StringBuffer result = new
 StringBuffer(s.length() *
 for (int i=0; i<times; i+
  result.append(s); }
 return result.toString();
}
```

`StringBuffer.append` mutates the old object

Much more efficient version

```
StringBuffer result = new
  StringBuffer(s);
for (int i=1; i<times; i++) {
 result.append(s); }
return result.toString();
```

Visibility

```
public class A {
  private int bar =0;

  public boolean isEqual(A a) {
      return (bar == a.bar);
  }

  public static void main(String[] args) {
      A a1 = new A();
      A a2 = new A();
      System.out.println(a1.isEqual(a2));
  }
}
```

Does the code compile? If no, why?
Does the code throw a runtime exception?
If yes, why? If no, what is the output?

No compilation error:
Objects of the same class can access each other's private fields

The output is:
`true`

Interfaces

```
public interface Foo {
  public void bar()
      throws Exception;
}

public class FooImpl implements Foo {
  public void bar() {
    System.out.println("An exception is not thrown");
  }

  public static void main(String args[]) {
      Foo foo = new FooImpl();
      foo.bar();
  }
}
```

Does the code compile? If no, why?
Does the code throw a runtime exception?
If yes, why? If no, what is the output?

Compilation error:
"Unhandled exception type Exception"

Interfaces

```
public interface Foo {
  public void bar()
      throws Exception;
  }

public class FooImpl implements Foo {
  public void bar() {
    System.out.println("No exception is thrown");
  }

  public static void main(String args[]) {
      FooImpl foo = new FooImpl();
      foo.bar();
  }
}
```

Does the code compile? If no, why?
Does the code throw a runtime exception?
If yes, why? If no, what is the output?

Output:
`No exception is thrown`

Interfaces and Inheritance

Consider the following class hierarchy:

```
Interface Animal {…}
class Dog implements Animal{…}
class Poodle extends Dog {…}
class Labrador extends Dog {…}
```

Which of the following lines (if any) will not compile?

```
Poodle poodle = new Poodle();
Animal animal = (Animal) poodle;
Dog dog = new Labrador();

poodle = dog;
```

```
poodle = (Poodle)dog;
```
-No compilation error
-Runtime Exception

```
poodle = (Poodle)animal;
```
-No compilation error
-No Runtime Exception

Interfaces and Inheritance

```
class A {
  public void print() {
      System.out.println("A");
  }
}

class B extends A implements C {
}

interface C {
  void print();
}
```

Is there any error?

No compilation errors

Public by default

Interfaces and Inheritance

```
class A {
  void print() {
      System.out.println("A");
  }
}

class B extends A implements C {
}

interface C {
  void print();
}
```

Is there any error?

Compilation error:
The inherited package method `A.print()` cannot hide the public abstract method in C

Method Overloading

Does the code compile? If no, why?
Does the code throw a runtime exception?
If yes, why? If no, what is the output?

```
public class A {
  public void foo(Object o) {
      System.out.println("Object"); }

  public void foo(String s) {
      System.out.println("String"); }

  public static void main(String args[]) {
      A a = new A();
      a.foo(null); }
}
```

The code compiles and runs, printing “String”

Method Overloading

Does the code compile? If no, why?
Does the code throw a runtime exception?
If yes, why? If no, what is the output?

```
public class A {
  public void foo(StringBuffer sb) {
      System.out.println("StringBuffer");}

  public void foo(String s) {
      System.out.println("String"); }

  public static void main(String args[]) {
      A a = new A();
      a.foo(null); }
}
```

The code does not compile (an ambiguous method)

Method Overriding

```
public class A {
  public void print() {
      System.out.println("A");
  }
}

public class B extends A {
  public void print(){
      System.out.println("B");
  }
}
```

```
public class C {
  public static void
  main(String args[]){
      B b = new B();
      A a = b;
      b.print();
      a.print();
  }
}
```

Casting is unneeded

Does the code compile? If no, why?
Does the code throw a runtime exception?
If yes, why? If no, what is the output?

The output is:
B
B

Method Overriding & Visibility

```
public class A {
  public void print() {
   System.out.println("A");
  }
}

public class B extends A {
protected void print() {
  System.out.println("B");
  }
}
```

```
public class C {
  public static void
  main(String[] args) {
      B b = new B();
      b.print();
  }
}
```

Does the code compile? If no, why?
Does the code throw a runtime exception?
If yes, why? If no, what is the output?

Compilation error:
"Cannot reduce the visibility of the inherited method"

Method Overriding & Visibility

```
public class A {
  protected void print() {
      System.out.println("A");
  }
}

public class B extends A {
  public void print() {
      System.out.println("B");
  }
}
```

```
public class C {
  public static void
  main(String[] args) {
      B b = new B();
      b.print();
  }
}
```

Does the code compile? If no, why?
Does the code throw a runtime exception?
If yes, why? If no, what is the output?

The output is:
B

27

Inheritance

```
public class A {
  private void foo() {
   System.out.println("A.foo()");
  }
  public void bar() {
   System.out.println("A.bar()");
   foo();
  }
}
public class B extends A {
  public void foo() {
   System.out.println("B.foo()");
  }
}
```

```
public class D {
  public static void
  main(String[] args) {
      A a = new B();
      a.bar();
  }
}
```

Does the code compile? If no, why?
Does the code throw a runtime exception?
If yes, why? If no, what is the output?

The output is:
A.bar()
A.foo()

28

Inheritance

```
public class A {
  public void foo() {…}
}

public class B extends A {
  public void foo() {…}
}
```

How can you invoke the foo method of A within B?
Answer:
Use super.foo()

Inheritance

```
public class A {
  public void foo() {…}
}

public class B extends A {
  public void foo() {…}
}

public class C extends B {
  public void foo() {…}
}
```

How can you invoke the foo method of A within C?
Answer:
No possible
(super.super.foo() is illegal)

Inheritance & Constructors

```
public class A {
  String bar = "A.bar";
  A() { foo(); }
  public void foo() {
      System.out.println("A.foo():
            bar = " + bar);
  }
}
public class B extends A {
  String bar = "B.bar";
  B() { foo(); }
  public void foo() {
      System.out.println("B.foo():
            bar = " + bar);
  }
}
```

What is the output?

```
public class D {
  public static void
  main(String[] args) {
      A a = new B();
      System.out.println(
     "a.bar = " + a.bar);
      a.foo();
  }
}
```

The output is:

```
A.foo(): bar = null
B.foo(): bar = B.bar
a.bar = A.bar
B.foo(): bar = B.bar
```


Inheritance & Constructors

What is the output?

```
public class A {
  protected B b = new B();
  public A() { System.out.println("in A: no args."); }
  public A(String s) { System.out.println("in A: s = " + s); }
}
public class B {
  public B() { System.out.println("in B: no args."); }
}
class C extends A {
  protected B b;
  public C() { System.out.println("in C: no args."); }
  public C(String s) { System.out.println("in C:
}
public Class D {
  public static void main(String args[]) {
      C c = new C();
      A a = new C(); }
}
```

The output is:

```
in B: no args.
in A: no args.
in C: no args.
in B: no args.
in A: no args.
in C: no args.
```

Inheritance & Constructors

```
public class A {
  protected B b = new B();
  public A() { System.out.println("in A: no args."); }
  public A(String s) { System.out.println("in A: s = " + s); }
}
public class B {
  public B() { System.out.println("in B: no args."); }
}
public class C extends A {
  protected B b;
  public C() { System.out.println("in C: no args."
  public C(String s) { System.out.println("in C: s
}
public class D {
  public static void main(String args[]) {
      C c = new C("c");
      A a = new C("a"); }
}
```

What is the output?

The output is:

```
in B: no args.
in A: no args.
in C: s = c
in B: no args.
in A: no args.
in C: s = a
```


Inheritance & Constructors

```
public class A {
  String bar = "A.bar";
}

public class B extends A {
  String bar = "B.bar";

  B() { foo(); }

  public void foo() {
      System.out.println("B.foo():
  bar = " + bar);
  }
}
```

```
public class D {
  public static void
  main(String[] args) {
   A a = new B();
   a.foo();
   System.out.println(a.bar);

  }
}
```

What is the result?

Compilation Error:
"The method foo is undefined for the type A"

Local Class

```
public class Test {
  public int a = 0;
  private int b = 1;

  public void foo(final int c) {
      int d = 2;

      class InnerTest {
            private void bar(int e) {


            }
      }
  }
}
```

Which variables (a, b, c, d, e) are accessible at the highlighted line?

Only a,b,c and e are accessible at the highlighted line.

35

Method Overloading

Does the code compile? If no, why?
Does the code throw a runtime exception?
If yes, why? If no, what is the output?

```
public class A {
  private static class B {}
  private static class C extends B {}

  public void foo(B b) {
      System.out.println("B"); }
  public void foo(C c) {
      System.out.println("C"); }
  public static void main(String args[]) {
      A a = new A();
      a.foo(null); }
}
```

The code compiles and runs, printing “C”

36

Pass By-Value

```
public class Test {
   private static class Value { int v = 1; }

   public static void main(String[] args) {
      Test test = new Test();
      int v = 2;
      Value value = new Value();
      value.v = 3;
      foo(value, v);
      System.out.println(value.v + " " + v);
   }
   private static void foo(Value value, int v) {
      v= 4;
      value.v = 5;
      value = new Value();
      System.out.println(value.v + " " + v);
   }
}
```

What is the output?