Java: Immutable and mutable strings and primitive objects

Recall

What does it mean when we say an object is immutable?

Which class among the classes you have learnt so far is immutable?

Immutable Objects and their advantages

- An object is *immutable* if its state cannot be change after it is created.
- String class is *immutable* because any modification methods on string object (like replace() or substring()) does not change the original string.
- Immutable objects are thread-safe!
- Therefore they useful in applications that have multiple threads concurrently executing.

Tell me how

- How are Immutable objects thread-safe?

  - If a thread manipulates the state of an object but not under synchronized context and at the same time if another thread manipulates the state of the same object, the object is in inconsistent state. The Account object in the previous section clearly demonstrates this.
  - A immutable object does not require the synchronized context. This is because once they are created they cannot be changed. So there is no question of inconsistent state.
  - Therefore, they are thread-safe.

Tell me how

- How to create a thread-safe object?

- The thread safe class should not allow any manipulations to its member variables after creation. What will you do in your class to make this happen?

- Make member variables **`final`**
- Provide no setters
- Don’t allow subclasses to make the object mutable. So either make the class itself as **`final`** or provide a **`private`** constructor.
- Also if member variables are instances of other classes, they themselves have to be made mutable by this class– it gets more complicated!

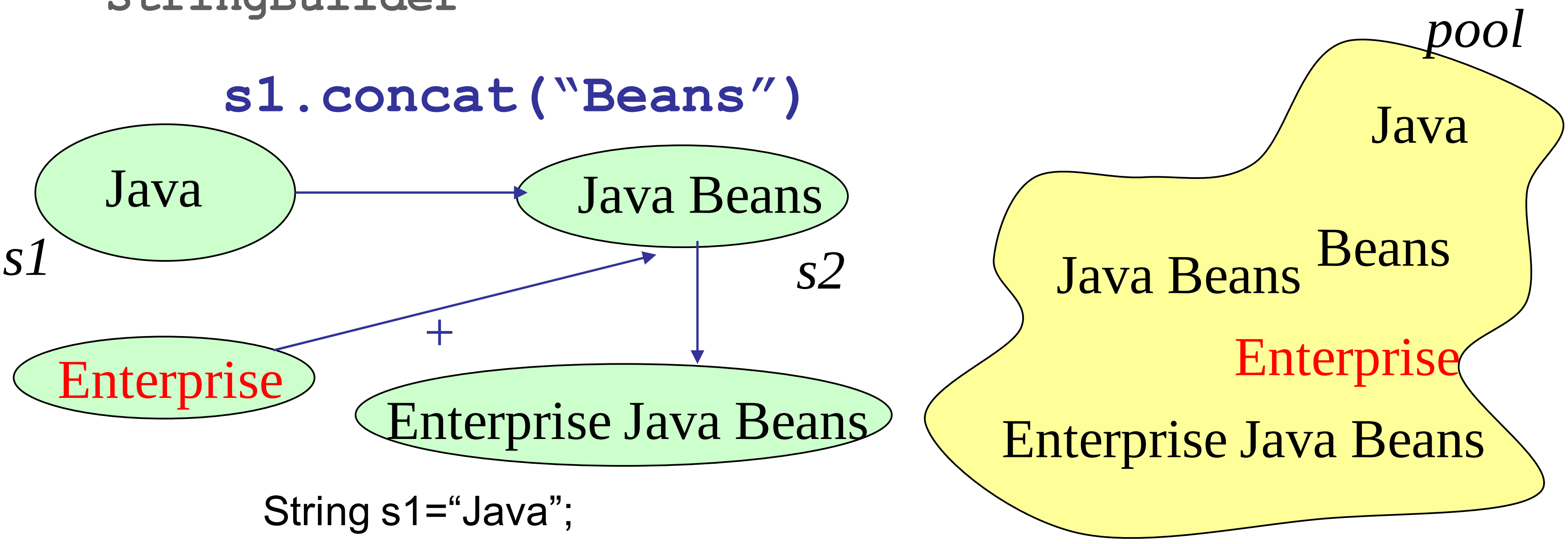
Disadvantages of Immutable objects

- Some implementations create and return a new object if immutable objects are changed (like `String` which we have seen). The cost of creating a new object is more compared to updating an object in place.
- Immutable objects are not always desirable for thread-safe code. For instance, does Account object being immutable make sense!

In cases where the object has to under go a lot of updations by design, then it is desirable to have mutable objects with synchronized methods.

Revisiting problem with String

- **`String`** class objects are immutable.
- In cases where we have lots of string manipulation we may end up with creating lot of strings in the string pool which are unnecessary.
- Therefore, in such cases we need to go for **`StringBuffer`** or **`StringBuilder`**



```
String s1=“Java”;
String s2=s1.concat(“Beans”);
String s3=“ Enterprise ” + s2
```

StringBuilder

- This **`final`** class can be used in the situations where we require lot of string manipulations.
- **`StringBuilder`** objects are mutable .
- Constructors
  - **`StringBuilder()`**
  - **`StringBuilder(String str)`**

Methods

Methods that are common in both String and StringBuilder classes:

- **char charAt(int index)**
- **int length()**
- **String substring(int start)**
- **String substring(int start, int end)**
- **int indexOf(String str)**
- **int indexOf(String str,int fromIndex)**
- **int lastIndexOf(String str)**
- **int lastIndexOf(String str, int fromIndex)**

- ***Concatenation***

```
StringBuilder append(String str)
StringBuilder append(StringBuffer str)
StringBuilder append(char[] c)
StringBuilder append(xx b)
```

where **`xx`** is **`boolean, char, int, long float and double`**

Example: **`s1= new StringBuilder("Now");`**

```
s1.append(" Showing"); // Now Showing
```

How do you achieve this in String class?

- ***Replacing characters***

```
StringBuilder replace(int start, int end, String s)
```

Example: **`StringBuilder s1= new StringBuilder("now");`**

```
s1.replace(0,0,"S");// Snow
s1.replace(0,1,"S");// Sow
s1.replace(0,2,"S"); //Sw
```

Compare this to the **`replace()`** method in **`String`** class

- *`Insertion and deletion of characters`*

```
    StringBuilder insert(int offset, Object str)
    StringBuilder insert(int offset, String str)
    StringBuilder insert(int offset, xx b)
```

where `xx is boolean, char, int, long float` and `double`

In case of `Object` as 2nd argument the string that is returned by `toString()` method is inserted.

Example:
```
StringBuilder s1= new
 StringBuilder("Teacher():");
 s1.insert(8, new Teacher("Tom"));
// Teacher(Tom (1)):
```

- `StringBuilder delete(int start, int end)`

Example:
```
StringBuilder deleteCharAt(int index)
StringBuilder s1= new
StringBuilder("Teacher():");
s1.delete(7, s1.length()); // Teacher
```

Example: `StringBuilder`

- ***Reverse:***

```
StringBuilder reverse()
```

Check if a string is a palindrome.. Try to do this with String class.
And then compare your code with the code here.
It turns out that the code here is far simpler!

```
public class Palindrome {
public static void main(String[] args) {
   String palindrome = "MalayalaM";

   StringBuilder sb = new StringBuilder(palindrome);
   System.out.println(sb.equals(sb.reverse()));
   System.out.println(sb);
}

}
```

Beware!

- Unlike **`String`**, **`StringBuilder`** (**`StringBuffer`** in the next slide) does not override **`equals()`** method.
- That is,

```
StringBuilder sb = new StringBuilder(palindrome);
StringBuilder sb3 = new StringBuilder(palindrome);
System.out.println(sb3.equals(sb));
```

return **`false!`**

- Then how did previous example work?
- Recall that by default **`equals()`** method of **`Object`** class functions the same as the that of the **`==`**. Having said this, since **`sb.reverse()`** changes string in that same location, **`sb`** before and after calling reverse are same!
- Also note that **`StringBuilder(StringBuffer)`** is not **`Comparable`** (unlike **`String`**)

StringBuffer

- **`StringBuffer`** has same methods as **`StringBuilder`** and it can also be used to create mutable Strings. It is also a **`final`** class
- Only difference between both the classes is
  - **`StringBuffer`** is thread-safe while **`StringBuilder`** is not thread-safe
- Thread-safe class have methods that are synchronized.
- **`synchronized`** makes only one thread at a time access an object's **`synchronized`** methods. This may impact performance.
- Thinking of this issue, JSE built **`StringBuilder`** class that does not have **`synchronized`** methods. So, now it is up to programmers to make sure of the consistency of strings from **`StringBuilder`** class by providing **`synchronized`** blocks locking the object.

Wrapper classes

- Wrapper classes in java are classes that wrap other classes.
- Therefore wrapper class’ constructor will always take the object of the class they wrap as a parameter.

Did we come across this term before?

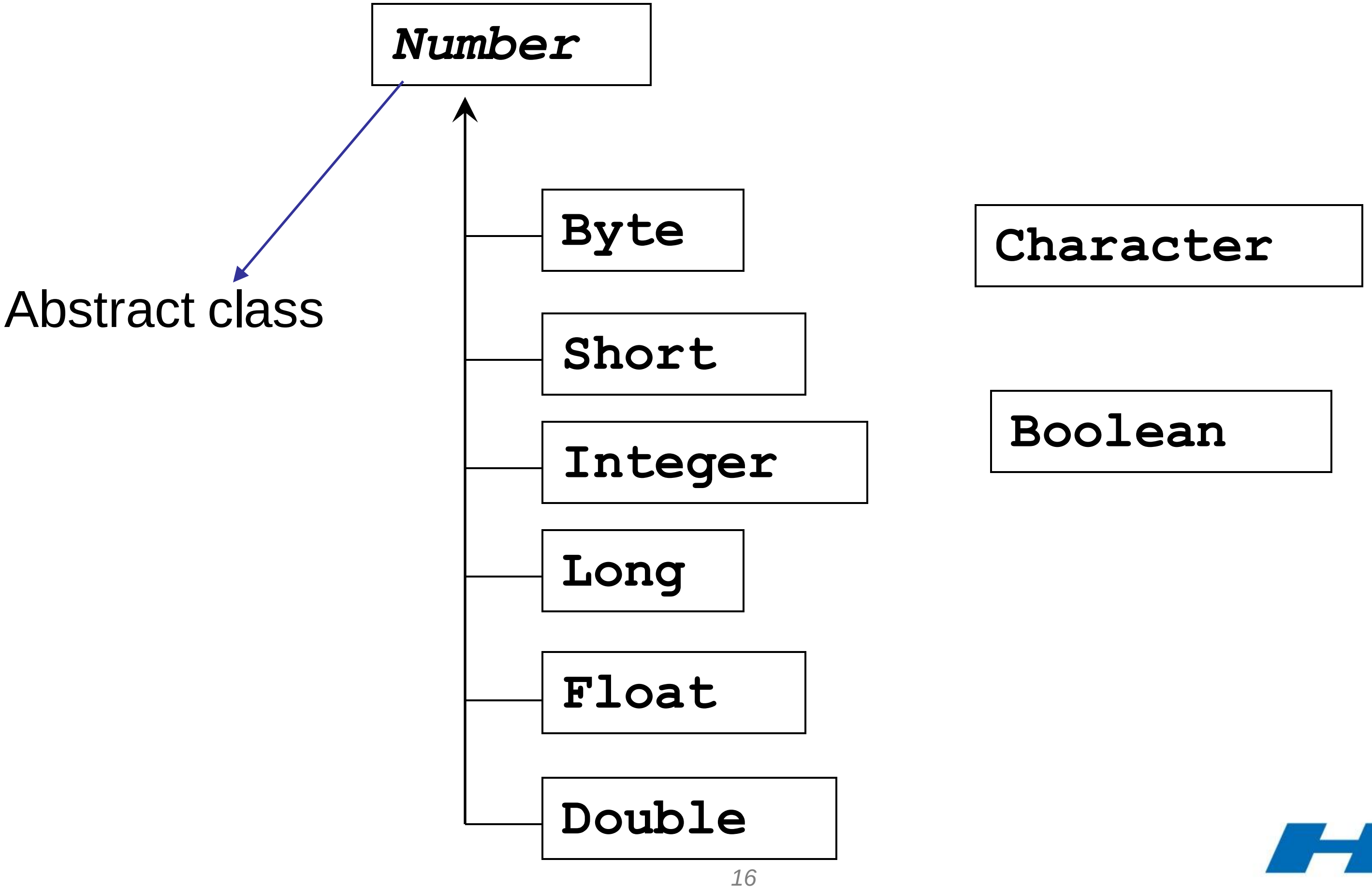
Flask

Cover is a wrapper enhancing the bottles functionality

The two important packages that have wrapper classes are in **`java.lang`** and **`java.io`**

Primitive Wrappers

- There are some special kinds of wrapper classes in **`java.lang`** package which wrap primitive types.
- All wrapper classes are **`final`** classes except **`Number`** which is **`abstract`** class.



Why should one wrap the primitive types?

1. For various conversions
   - These classes have methods that allow conversion between string and numeric types

2. Collection work with objects
   - Collection framework in java has numerous classes like **`LinkedList`**, **`ArrayList`** etc that allow storing collection of objects. But only objects can be stored in collection. So in cases where we want ints to be stored we need to wrap it in a class. Instead JSE already provides us with classes for all primitives as wrappers that can be used.

3. Serialization
   - Java stores object state in the hard disk and this is called serialization. But this technique is available only for objects. Therefore if we want to save primitives we need to wrap them inside a class. Instead of this we can use wrapper classes.

Constructors and common methods

- General form of constructor for all wrapper classes:
  a) Constructor with its corresponding primitive type
  b) Constructor with a `String` type. Excludes: `Character`

  Example:

  `Integer(String)` and `Integer(int)`

  `Double(String)` and `Double(double)`

- `Float` has an extra constructor that takes `double`.

  `Float(double)`

- Also note that

```
~~Byte  b= new Byte(1);~~ // compilation error
```

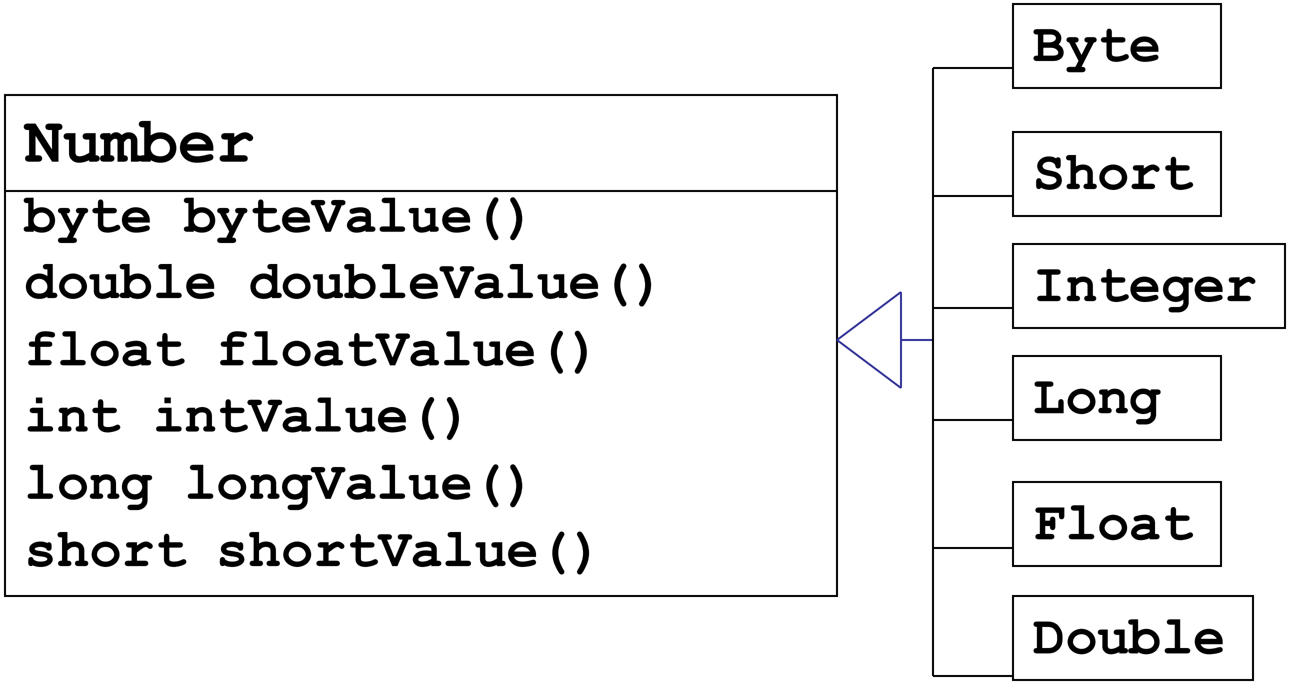
- Also all the classes have implemented `equals()`, `toString()` and `compareTo()` methods

Test your knowledge

- Can you write the declaration for **`equals()`**, **`toString()`** and **`compareTo()`** for **`Byte?`**
  - **`String toString()`**
  - **`boolean equals(Object o)`**
  - **`int compareTo(Byte a)`**
  - The first 2 methods remain the same for all classes but the last method parameter changes based on the wrapper class.

- Where did you come across these methods?
  - **`equals()`** and **`toString()`** are **`Object`** class methods that wrapper classes have overridden and **`compareTo`** is in **`Comparable`** which obviously means that wrapper classes have implemented **`Comparable!`**

Methods in Number Subclass



Number

byte byteValue()
double doubleValue()
float floatValue()
int intValue()
long longValue()
short shortValue()

Byte

Short

Integer

Long

Float

Double

Conversion of String to numeric primitive type

- General form of method:

  `static xxx parseXxx(String s)`

  where xxx is all numeric primitives

- **`parseXxx`** method are there in all **`Number`** subclasses
- Each class has Xxx replaced by the type it encapsulates.
- For example

  `Byte` class has

  `static byte parseByte(String s)`

  `Float` class has

  `static float parseFloat(String s)`

- This method throws a **`NumberFormatException`** if the string is not convertible to number.

Examples

- int x=Integer.parseInt(“-3”);//OK
- float x=Float.parseFloat("3.14"); //OK
- byte x=Byte.parseByte("10”); //OK
- Integer.parseInt(“abc”);

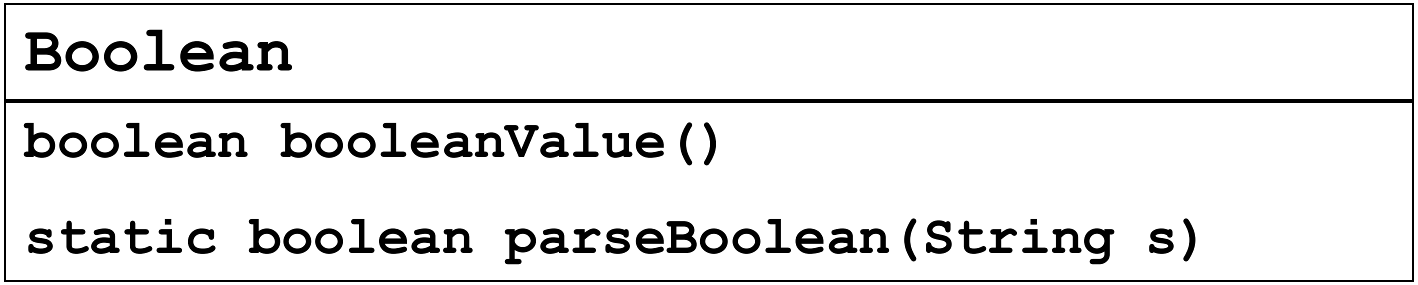
  NumberFormatException thrown at runtime
- Integer.parseInt("12.23");

  NumberFormatException thrown at runtime
- Float.parseFloat("12e10"); //OK

Methods for `Character`

```
Character

char charValue()

boolean isLetter(char ch)
boolean isDigit(char ch)

boolean isUpperCase(char ch)
boolean isLowerCase(char ch)

char toUpperCase(char ch)
char toLowerCase(char ch)

boolean isWhitespace(char ch)
```

Methods for `Boolean`



Boolean

`boolean booleanValue()`

`static boolean parseBoolean(String s)`

Autoboxing

- Autoboxing refers to the automatic conversion of
  - wrapper class type to its primitive type
    - Called Boxing
  - and vice versa
    - Called Unboxing
- This was included in JSE 5.0.
- Older Java code needed explicit calls to constructors or **`xxxValue()`** methods for these conversions.
- Now compiler does these conversions automatically.
- Boxing and unboxing may hit performance. So be careful where you use it.

Autoboxing Examples

- Boxing Examples
  1. `Integer ii=10; ii++;`
  2. `Boolean bb=true; if(bb){}`
  3. `Long ll=34L;` but `Long ll=34` leads to error
  4. `Byte bt=34;` but `Byte bt= 1000;` leads to error
  5. `Float fl= 3.14f;` but `Float fl= 3.14;` leads to error
  6. `Double dl= 3.14;`
  7. `Character c=‘a’;`

  You will find that literal conversions here is very similar to what we had for primitives.

- Unboxing is the reverse. That is a wrapper objects automatically convertible to its primitive.
  - `int i=ii; boolean b=bb; long l=ll;`

Pros and Cons

- When compiler allows **`Integer ii=10,`** what it does is it converts the code as

  `Integer ii = new Integer(10);`
- Advantage of boxing are
  - the code I neat and clutter free.
  - Less coding for developers
- Disadvantages of boxing is
  - Experiments and experience yields the fact that when boxing conversions are used within a loop, it affects the performance of a program. Therefore while allowing both wrapper and primitives gives greater flexibility in a collection (like array or List (coming up)), care must be taken when to use them. For instance, if only primitive **`int`** is required, then restricted **`int`** array can be created instead of an **`Integer`** array. Or **`Integer List.`**

Overloading Resolution including autoboxing

- Compiler resolves overloaded methods using the following sequence
  - Finds if there are any exact match possible
  - If not, finds if there are any automatic conversion possible
  - If not, finds if any specific conversion apply
  - If not, finds if there are any auto-boxing conversion possible
  - If not, finds if there are any var-args conversion possible
- No boxing does not work in case of arrays

```
Integer[] i= new int[10]; // error
```

Let us work out some code to understand this better.

Test your understanding

-
  ```
  static void change(int i){
  System.out.println("int");}
  static void change(Integer i){
  System.out.println("Integer");}
  ```
  Call: `change(123);`

Prints : int

-
  ```
  static void change(int i){
  System.out.println("int");}
  static void change(Byte i){
  System.out.println(“Byte");}
  ```
  Call: `byte b1=45; change(b1);`

Prints : int

-
  ```
  static void change(char i){
  System.out.println(“char");}
  static void change(Byte i){
  System.out.println("Byte");}
  ```
  Call: `byte b1=45; change(b1);`

Prints : Byte

-
  ```
  static void change(Integer i){
  System.out.println("integer");}
  static void change(Number i){
  System.out.println("number");}
  ```
  Call: `change(45);`

  Prints : integer

-
  ```
  static void change(Number i){
  System.out.println(“number");}
  static void change(int i){
  System.out.println(“int");}
  ```
  Call: `Byte b1=45; change(b1);`

  Prints : number

-
  ```
  static void change(Integer i){
  System.out.println("integer");}
  static void change(Number i){
  System.out.println("number");}
  ```
  Call: `byte b1=25;change(b1);`

  Prints : number

-
  ```
  static void change(Object i){
  System.out.println("Object");}
  static void change(Long i){
  System.out.println("Long");}
  ```
  Call: `change(10);`

Prints : Object

-
  ```
  static void change(Integer i){
  System.out.println("integer");}
  static void change(int... i){
  System.out.println("int");}
  ```
  Call: `change(10);`

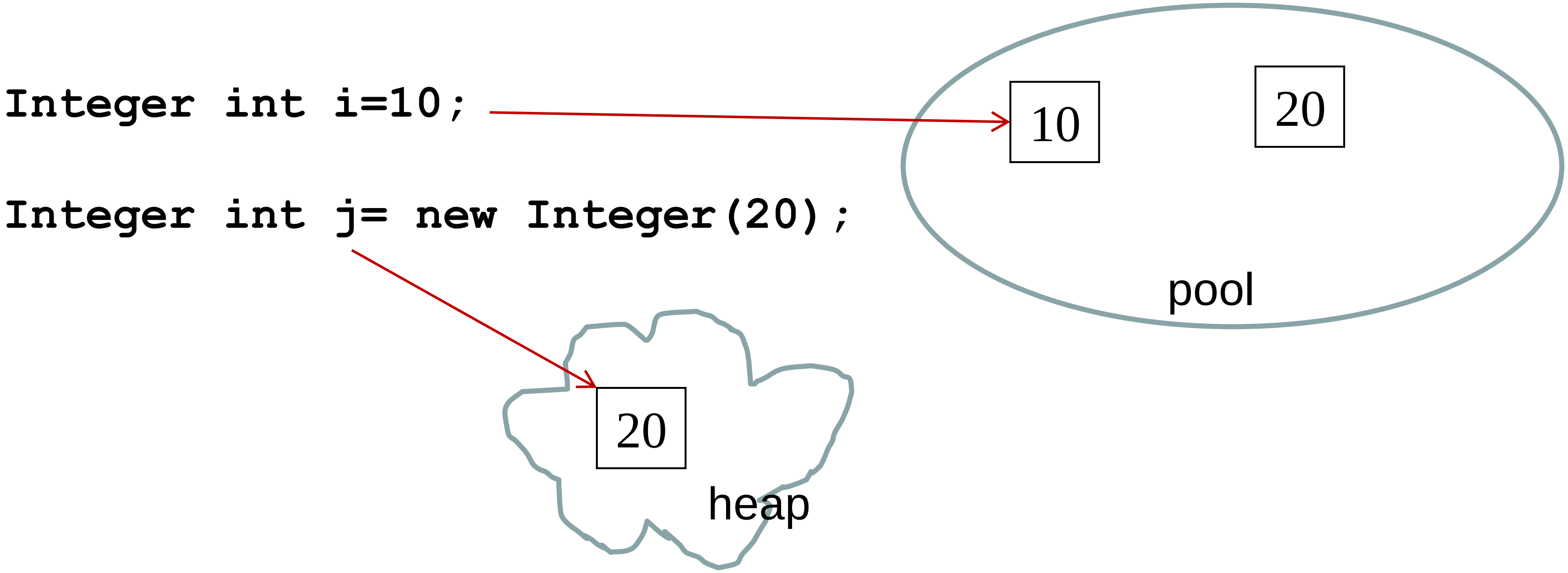
Prints : integer

-
  ```
  static void change(Integer... i){
  System.out.println("Integer");   }
  static void change(int... i ){
  System.out.println("int");       }
  ```
  Call: `change(10); // error`

Compilation error

Immutability

- Like **`String`**, Wrapper objects are also immutable.
- So using new creates primitives in the pool(if not there already) as well as in the heap and directly assigning a literal will create primitives in the heap alone.
- That is why there are no setters in wrapper classes.



```
Integer int i=10;

Integer int j= new Integer(20);
```

Test your understanding?

What will the code print?

```
class ImmutablilityTest{
public static void main(String... args){
Integer x=10;
 change(x);
 System.out.println(x);
  }
static void change(Integer i){
   i=20;
}
}
```

Prints 10!

Beware!

- How can you exchange 2 primitives?
- Will the code below work?

```
 public static void main(String[] args) {
 Integer x=10;
        Integer y=30;
        exchange(x,y);
}
 static void exchange(Integer i,Integer j){
          Integer t;
          t=i;
          i=j;
          j=t;
  }
```