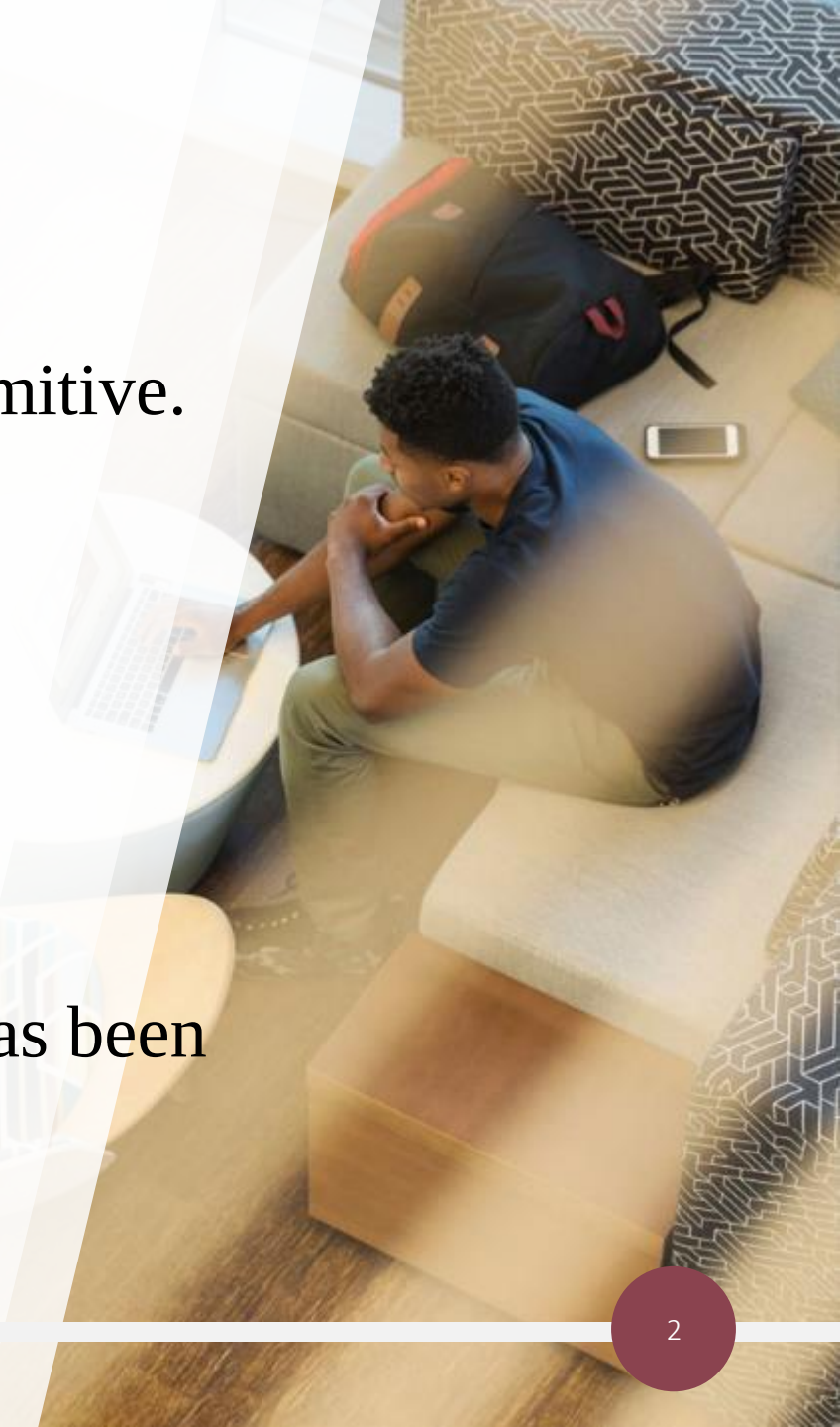
Working with Java Data Types

String

String

- A string is a sequence of characters.
- Strings are objects i.e. *String* is a class and not a primitive.
  - String s1 = new String(“Sean”);
  - String s2 = “Kennedy”; // syntactic sugar
- As it is a class, methods are available.
- Strings can be concatenated and chained together.
- A defining property of *String* objects is that they are immutable (as are wrapper types e.g. *Integer*, *Double* etc.). This means that once a *String* object has been created, it **cannot** be changed.
  - the object is immutable but the reference is mutable



String

- String literals are stored in the String Pool.
- equals() versus ==.

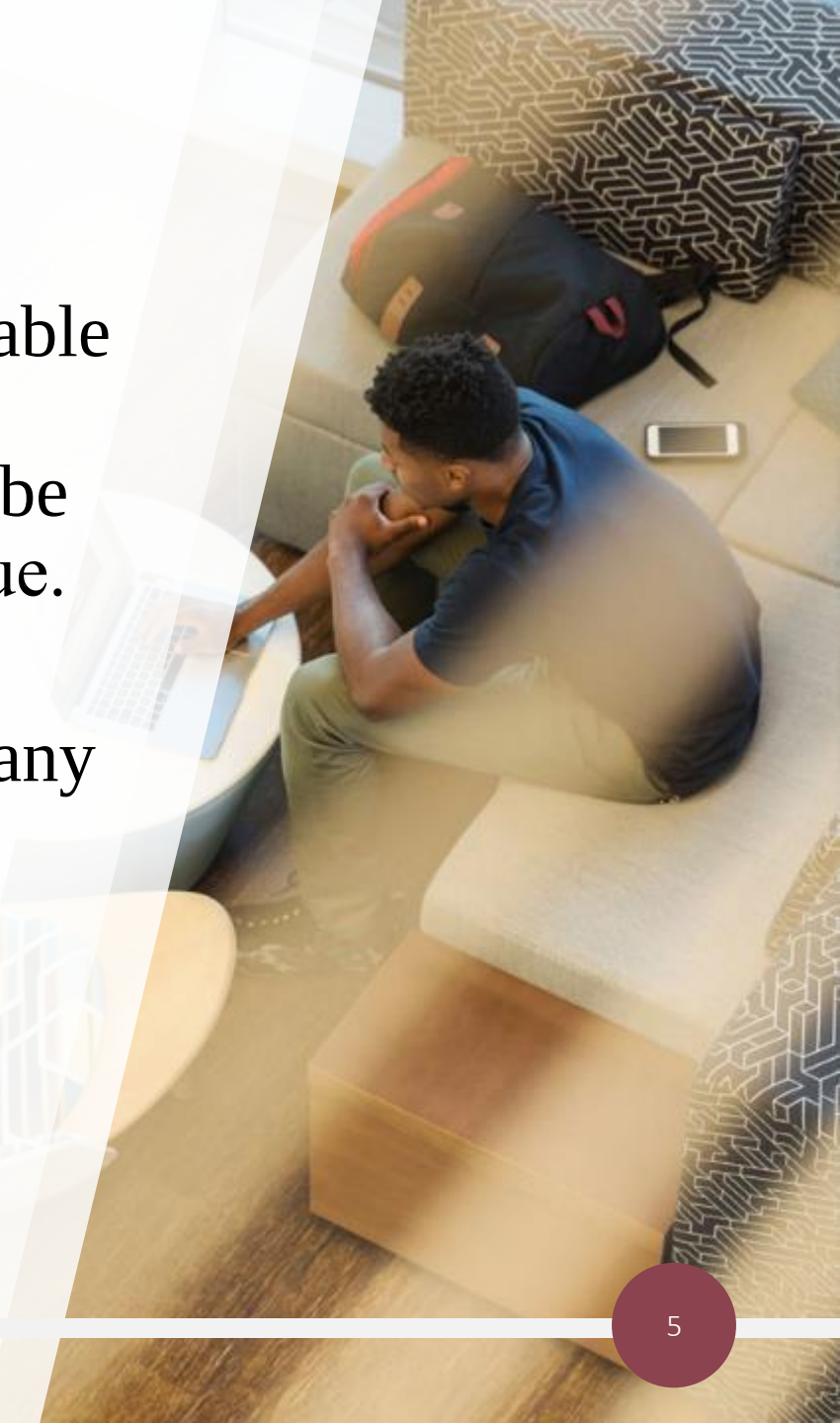
- Example of String immutability…

String Pool

- The efficient use of memory is a key goal of any programming language.
- String literals (e.g. “Jack”) often occupy large portions of memory, often leading to redundancy. To make Java more memory efficient, Java sets aside a special area of memory called the S*tring Pool*.
- When the compiler encounters a *String* literal, it checks the pool to see if an identical *String* already exists. If one exists, then the reference is directed to the *String* in the pool and no new *String* literal object is created! The existing *String* simply has an additional reference referring to it.

String Pool

- Now you can see why making *String* objects immutable is such a good idea – if several references refer to the same *String* object without even knowing it, it would be very bad if any of them could change the *String*'s value.
- This is one of the main reasons the *String* class is marked *final* – nobody can override the behaviour of any of the methods i.e. the *String* objects are immutable.



String equality

```
String name1 = "Sean";
String name2 = "Sean";
System.out.println(name1 == name2);//true, are the references the same?

String name3 = new String("Sean");
System.out.println(name1 == name3);//false
System.out.println(name1.equals(name3));//true, are the object contents the same?
System.out.println(name1 == new String("Sean").intern());//true
```

- Examine what is going on in memory…

String chaining

```
String s = "qwe"
                .concat("rty")
                .toUpperCase()
                .replace('E', 'O');
System.out.println(s); // QWORTY
```

- Examine what is going on in memory…

Important String Methods

- *public char charAt(int index)*
  - returns the character located at the *String*'s specified index; remember, *String* indexes are zero-based.
    - String x = "phone";
    - System.out.println(x.charAt(2)); // output is 'o'
- *public String concat(String s)*
  - returns a *String* with the value of the String passed in to the method appended to the end of the *String* used to invoke the method.
    - String x = "light";
    - System.out.println(x.concat(" switch")); // "light switch"

Important String Methods

- *public boolean equalsIgnoreCase(String s)*
  - returns *boolean* (*true* or *false*) depending on whether the value of the *String* in the argument is the same as the value of the *String* used to invoke the method (case does not matter with this method).
    - String x = “Exit”;
    - System.out.println(x.*equalsIgnoreCase(“EXIT”*); // “true”
    - System.out.println(x.*equalsIgnoreCase(“tixe”*); // “false”

Important String Methods

- *public int length()*
  - returns the length of the *String* used to invoke the method.
    - String x = “01234567”;
    - System.out.println(x.*length()*); // 8

- *public String replace (char old, char new)*
  - returns a *String* whose value is that of the *String* used to invoke the method , updated so that **any occurrence** of the char in the first argument is replaced by the char in the second argument.
    - String x = “oxoxoxox”;
    - System.out.println(x.*replace(‘x’, ‘X’)); // “oXoXoXoX”*

Important String Methods

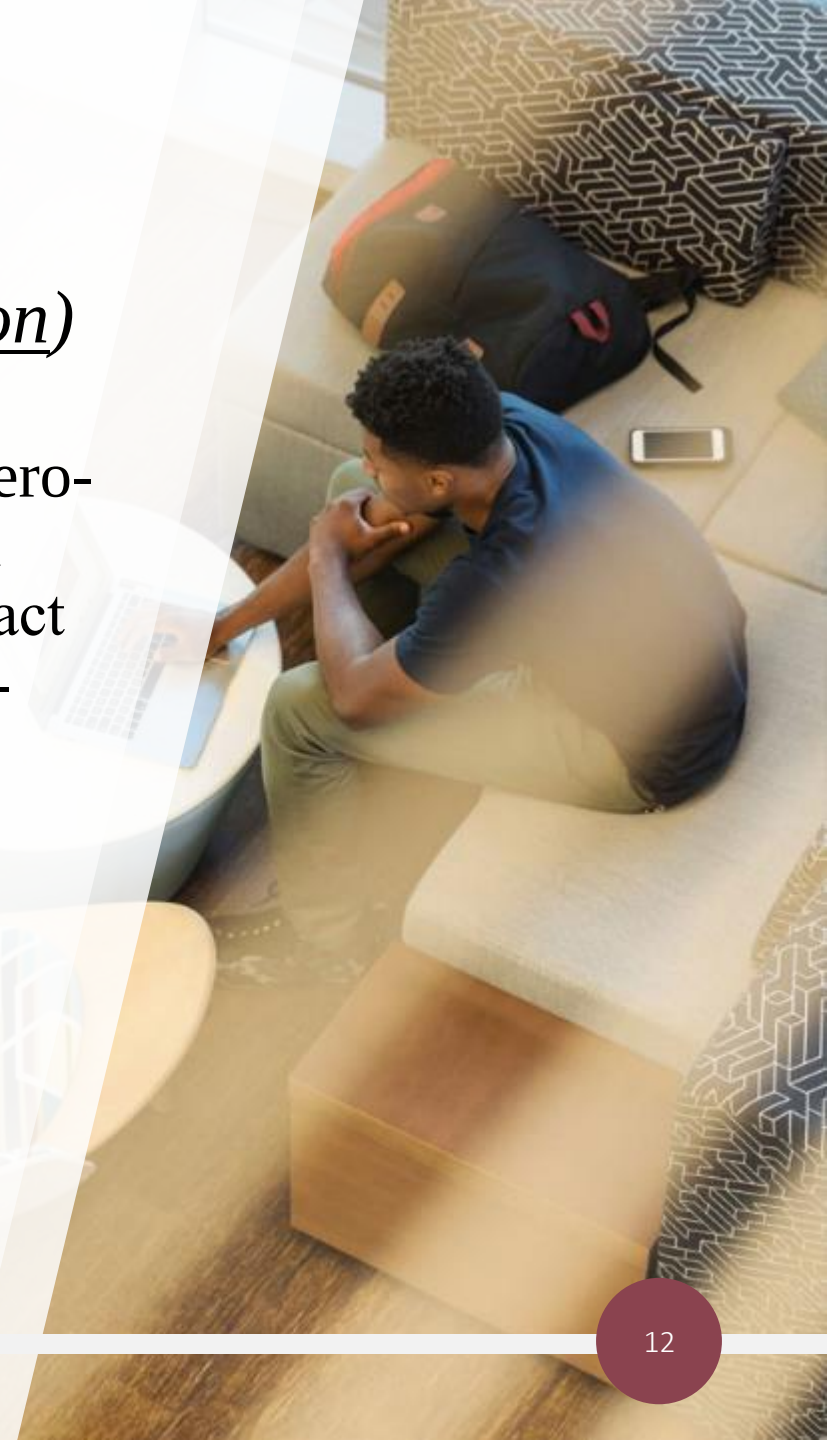
- *public String substring(int beginIndex)*
  - returns a part (substring) of the *String* used to invoke the method. The index is zero-based and goes from the *beginIndex* to the end of the string.
    - String x = “0123456789”;
    - System.out.println(*x.substring(5)*); // “56789”

```
"unhappy".substring(2) returns "happy"
"Harbison".substring(3) returns "bison"
"emptiness".substring(9) returns "" (an empty string)
```

Important String Methods

- *public String substring(int beginIndex, int endPosition)*
  - returns a part (substring) of the *String* used to invoke the method. The first argument represents the starting index (zero-based). Unfortunately, the 2nd argument is NOT zero-based (hence the term "position"). For example, position 7 is in fact index 6. A useful rule-of-thumb is to calculate *endPosition-beginIndex* characters starting at *beginIndex*.
    - String x = "0123456789";
    - System.out.println(*x.substring(5, 8)*); // "567"

```
"hamburger".substring(4, 8) returns "urge"
"smiles".substring(1, 5) returns "mile"
```



Important String Methods

- *public String toLowerCase()*
  - converts all characters in the *String* to lowercase
    - String x = “All Has CHanged”;
    - System.out.println(*x.toLowerCase()*); // “all has changed”

- *public String toUpperCase()*
  - converts all characters in the *String* to uppercase
    - String x = “All Has CHanged”;
    - System.out.println(*x.toUpperCase()*); // “ALL HAS CHANGED”

Important String Methods

- *public String trim()*
  - returns a *String* whose value is the *String* used to invoke the method, but with any leading or trailing whitespace removed:
    - String x = “ hi ”;
    - System.out.println(x + “t”); // “ hi t”
    - System.out.println(*x.trim()* + “t”); // “hit”