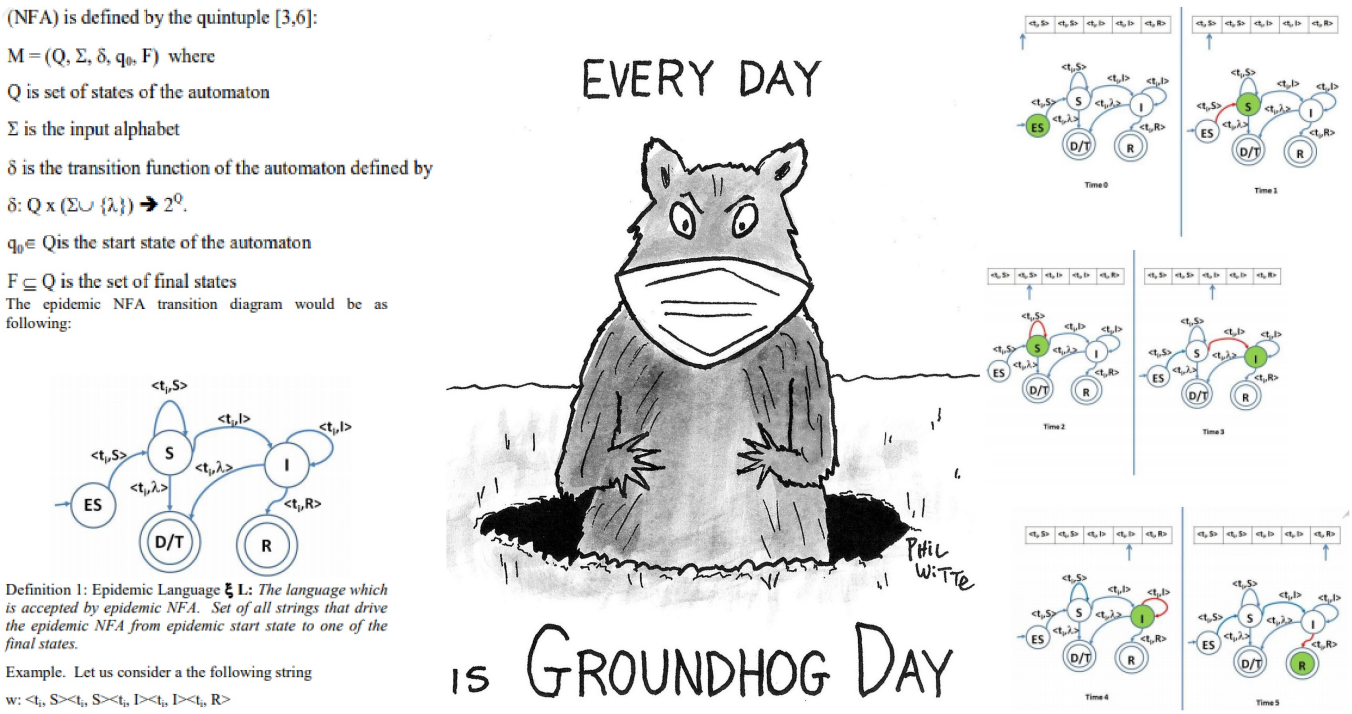
CSE250, Spring 2022 Week 1

[The first lecture will go over syllabus matters and then talk about how the evolution of the pandemic may impact course delivery---and all our psyches. That plus the first of several statements about academic integrity will be tied to the following image.]



- I paid $11 for license from CartoonStock to use the middle part in classroom environments. (Web publishing would have been $55, print publishing $50---what does that say?)
- The side panels come from an academic paper by researchers at the Mody Institute in India, https://www.ijert.org/research/finite-automata-for-sir-epidemic-model-IJERTV2IS90886.pdf, from the International Journal of Engineering Research & Technology (IJERT). Usage from an academic paper *with citation* is generally understood as free if the original access is free.
- By including it in lecture notes that are posted, am I "web publishing" it? Grey area. In this course, IMHO OK, but if I used this on the blog Gödel's Lost Letter and P=NP which I co-write with Richard J. Lipton, then it would be a violation---even though the blog is free.
- My own course materials **bear my copyright**, and reposting them to other web-accessible sites is a definite violation. Most in particular, you may not post my assignments to Chegg or CourseHero or similar sites. UB is currently formulating a targeted policy against this that will enable legal consequences.
- I used this image last year in the first lecture of CSE396, which was on Groundhog Day. The surrounding drawings exemplified course material in action ("epidemic NFA"). We won't have this *per-se* in this course, but linked data structures represented as trees and other graphs, yes.
- A cheerier thought: Happy Chinese New Year! Tiger Link

[If there is time at the end of the first lecture for to go up through the following "Hello World" example, great. Else, it will pick up on Wednesday. Survey results will tell what % of the class has had prior experience with Scala. In any event, Scala will be covered from the ground up at a level accessible to those whose only prior experience is of Python and/or Java (and/or JavaScript) while not boring those who have had Scala already. Allusions to C, C++, and other languages are FYI, big-picture chitchat.]

(Re-)Introducing Scala

Scala can be approached as an amalgam of Java and Python that fixes issues with each and grows into a modern "data-centric" language. It is bytecode-*equivalent* to Java, hence completely interoperable with it. The name means "scalable language" and is also Italian for "ladder" and "stairway" (to heaven, or at least to the *La Scala* opera house in Milan). Data-centric aspects include:

- Details of execution can be left implicit and farmed out to frameworks like *MapReduce*, *Hadoop, Spark* (these three are connected and have a Java base), and others.
- Potentially data-parallel without being explicitly so in the core language.
- Pure value types are convenient and almost automatic, unlike having to work through references for all objects in Java, and having to (remember to) use `const` in C/C++ and JavaScript, or `final` in Java. ("First-class values")
- Functions can be freely given as arguments to other functions/methods, at least when they operate on pure values. ("First-class functions")
- Easy to create container objects for data structures and do client-level operations on them in single lines of code.
- Executions branch according to logical structure of data ("First-class patterns").

All these points will be reflected in the design of code for data structures and their clients in this course. But lets begin with basic details.

1. Just like Java, Scala has a compiler `scalac` which produces `.class` files, and a run-time system `scala` which runs them. Both are built on top of Java, in fact.
2. Like Python, Scala has an interactive command line to interpret code. It is called REPL for "Read, Evaluate, Print Loop."
3. Scala uses C/C++/Java style braces rather than indentation as in Python. (Scala 3, which we are not yet using, makes more Python-like syntax optional.)
4. Like in Python, one can omit declaring basic integer, floating-point, char, and string types.
5. Those types are full-fledged objects; there are no primitive `int` and `double` (etc.) types to go wioth the `Int` and `Double` (etc.) classes as in Java.
6. Type name---if needed or desired in a declaration---comes *after* the variable or method header, with a colon : in between. (Like in the old Pascal language's family tree.)
7. What comes before is the keyword `val` or `var` (and `def` for functions/methods as in Python).

Time to see a "Hello World" program but with a few twists.

```
object Hello {                          //"object" not "class" is like "static" in C++/Java
    def main(args: Array[String]) = {   //templates are [...] not <...> as in C++/Java
        val s = "Hello"                 //semicolon is optional
        var t = "Would"
        s(0) = 'J'                      //arrays use (...) not [...]
        t(2) = 'r'
        println(s + " " + t + "!")
    }
}
```

No `public` keyword is needed (or even available); public visibility is the default. The name `main` is special. It is "static" because it is in an `object` which is standalone in Scala. There is no need here to specify a "void" return type, which when needed is `Unit` in Scala. The "`args: Array[String]`" is special syntax for command-line arguments like in Java. Scala shifts from angle brackets to square brackets for templates and from square brackets to parentheses for array indexing; the latter is treated just like a function. The `println` statement is like in Java. We could instead use a **substitutable String** prefaced by `s"` in which a `$` indicates that the next chars are a legal identifier giving a `String` value. This shortcuts many uses of output formats in C/C++ and is similar to how strings behave in Perl.

The program *does not compile* for the same reason similar Java code won't: `String` is an **immutable type** so you can't change individual characters. You need the analogue of Java `StringBuffer` which is `StringBuilder` in Scala:

```
object Hello {
    def main(args: Array[String]) = {
        val s = new StringBuilder("Hello")
        var t = new StringBuilder("Would")
        s(0) = 'J'
        t(2) = 'r'
        println(s"$s $t!")   // s" is special syntax.
    }
}
```

The difference between `val` and `var` is highlighted. Variables declared with `val` cannot be assigned to---they too are immutable and the compiler will flag attempts to assign to them. So what happens when we try to compile and run this code?

a) The code does not compile.
b) The code compiles but gives a runtime exception when we try to execute `s(0) = 'J'`.
c) The code compiles and runs with `s(0) = 'J'` treated as a no-op since `s` is a value, so that it outputs "Hello World!"
d) The code compiles and runs as if `s` were `var` not `val`, so that it outputs "Jello World!"

[this space intentionally skipped to move the answer offscreen]

Ah-ha-ha. The answer means that we cannot avoid a distinction that hits you in the face with C, C++, and Java but which Scala does even better than Python to try to hide: that between a value and a **reference**. The identifiers `s` and `t` are references to what are really data-structure objects at the level of `ArrayList[Char]`. Making `s` be `val` prevents you from changing `s` itself to refer to another object but doesn't prevent you from modifying its object "under the hood." Instead, in Scala we will have `mutable` versus `immutable` *container classes* to use in data structures.

More ways in which Scala is like and unlike Python and Java will occupy us the first two weeks.

Aspects of Scala that (In My Programmer's Humble Opinion) Improve on Java and Python

I. Intent to Declare

The presence of `val` and `var` fixes for me the scariest issue with Python. Consider this Python code:

```
sassafrasConcentration = 0.1
....
if openWound:
   sassafrassConcentration = 0.3
....
```

The typo is not flagged as an error; instead, a new variable is silently created. You might get and notice warnings about the variable never being used. The analogous Scala code

```
var sassafrasConcentration = 0.1
....
if (openWound) {
   sassafrassConcentration = 0.3
}  ....
```

gets flagged *at compile time* because the typo creates a new name without `val` or `var` to make it a declaration.

II. No `Const`-ipation or `final` agony

JavaScript and C/C++ have a modifier `const` to tell the compiler that a value cannot change. Java has `final` for that and some other uses. This helps with logical analysis and often enables a compiler to generate more-efficient code. On a method, `const` is supposed to mean that the underlying object will not change during a call, but we will see more "fake const" cases even with Scala where that gets dodged. The real moral we'll see is t*o make the underlying data immutable*, not just the variable or method name through which it is accessed. With this understood, the lack of a constant-method qualifier in Scala doesn't matter so much. The first big win is that whereas `const`-consistency across a C++ framework is often tricky (so much so that a workaround called `mutable` is resorted to ad-hoc), Scala keeps things simple.

The second and more immediate big win in Scala is that *no extra work* is needed to write `val` instead of `var`. This gives Scala compilers hope of approaching the optimization level of well-crafted C/C++ code. Hope. (I would actually like to make mutable declarations require extra work.)

III. No static with `static`

Every Scala class allows declaring an `object` of the same name that holds code shared as-is by all (other) instances of the class. Sometimes this is done implicitly. All fields and methods that would be declared `static` in C++ and Java simply go in the "companion object", which has full visibility of private members in the principal class and vice-versa.

You can also have an `object` by itself---this defines a single-instance class.

IV. No Null Values

Scala retains the `null` reference for a variable declared via `var`, but does not allow `null` as a value. Even though a Java `NullPointerException` is recoverable in a way that a C/C++ `Segmentation fault` is not, it is still a nasty run-time error. Here is a simple example:

```
object NullTest extends App {
    var x:String = null;
    println(x.toString)
}
```

This code gives a **`java`**`.lang.NullPointerException` when trying to call `toString` through the null reference. If it were just `println(x)`, then `null` would be printed.

When you work with values, Scala can shift the detection of possible errors to compile time, but at cost

of more layering of good code. The key component is the special template class

```
Option[T] = None | Some[T]
```

which works with any type `T`. Here's an example building off one in the official online Scala reference, https://docs.scala-lang.org/overviews/scala-book/no-null-values.html

```
class Address (                     //no braces! Defines constructor at same time
    var street1: String,            //mandatory address line
    var street2: Option[String],    //since second street-address line often absent
    var city: String,
    var state: String,
    var zip: String
)

object AddressTest extends App {
    val houseadr = new Address("10 Downing Street", None, "Amherst", "NY", "14226")
    val deptadr = new Address("338 Davis Hall", Some("UB North Campus"),
                              "Amherst", "NY", "14260")
    println(s"Home address second line is ${houseadr.street2}")
    println(s"Work address second line is ${deptadr.street2}")
    var s1 = houseadr.street1    //makes Scala infer s1: String
    //s1 = houseadr.street2      //streng verboten.  Whereas, a null reference
                                 //in Java would silently propagate.
    s1 = houseadr.street2.toString   //no () needed
    println(s"toString of Option[String] prints None case as $s1")
    s1 = deptadr.street2 match {
        case None => ""
        case Some(s) => s
    }
    println(s"Work address second line cleaned up is $s1")
}
```

Scala makes the `toString` method of the Option template class implicitly called within an `s"` string substitution expression but not in an assignment to a `String` variable. The former is runtime-OK; the latter is caught *at compile time*.

The cost for avoiding runtime bombs is an extra code layer to unpack---here by a `match` expression that is needed for the `street2` field but never for `street1`. Such lack of consistency can cause clunky code. Also problematic is making the decision to print the `None` case as an empty string local and *ad-hoc*. One can make a global treatment, but it needs another code element; Lazy Programmers Will Do What They Do. (Files of games from chess tournaments I monitor have ad-hoc conventions for

the `Rating` field of an unrated player: "Unr", "??", "?", "--", "", 0, 1000, -1. A new one that recently fooled my scripts was "0.0000". My code has become spaghettified, alas.)

V. No Accidental Overrides

Java, by removing the `virtual` keyword from C++, made it harder to tell when a method in a subclass overrides a method in a superclass. Java introduced an "annotation" `@Override` for this, with the compiler option to make it mandatory, but it is not a core language requirement. Scala makes `override` required when the compiler detects an ancestor method with the same type signature.

VI. Streamlined Construction and Access

The `Address` class above exemplifies a "Plain Old Data" (pod) class. The only constructor needed is one that initializes each field.

```
class Address (
    var street1: String,
    var street2: Option[String],
    var city: String,
    var state: String,
    var zip: String
) {
    override def toString() = { s"$street1, $street2, $city $state $zip" }
}   //note no return keyword needed; last line's value is returned

//class Test extends App    //compiler compains about lack of "static"
object Test extends App {
    val pmr = new Address("10 Downing Street", None, "Westminster", "ENG", "SW1")
    println(pmr.toString)
    pmr.city = "London"
    println(pmr.toString)
}
```

Note again that "`val pmr =`" did not prevent a field of `pmr` from being changed. To make the *data* immutable, you need `val` on each of the five fields.

Scala has no top-level distinction between accessing a field and calling a getter method to access the field. Any zero-parameter method can behave like a "dynamic field"---we could have defined toString without the parens and braces:

```
    override def toString = s"$street1, $street2, $city $state $zip"
```

The only difference from "`toString`" being a true field is that it changed when the `city` field was changed to "London". You can say the method gave a *dynamic value*, and that is just what `def` means as opposed to `val`. The immediate value is a block of executable code and it is assigned to the identifier `toString` via the usual assignment syntax with `=`. This is what makes functions passable as arguments to other functions, which is the defining point of "first-class functions." More on this next---for now the point is that Scala avoids special getter/setter syntax.

[Later: exactly how constructors work in Scala. Notes are into Friday by now.]

Functions and Types and Values (parallel to section 1.4.1 of the Lewis-Lacher text)

We can expand on the above `Address` example to illustrate some of what the text says about functions and "rocket types" in section 1.4.1. The code also illustrates a Scala **match expression** while defining a method `address` that uses parens with a Boolean argument too.

```
class Address1 (                    //no braces! Defines constructor at same time
    var street1: String,            //mandatory address line
    var street2: Option[String],    //since second street-address line often absent
    var city: String,
    var state: String,
    var zip: String
) {
    def address(give2nd: Boolean) = {
        if (give2nd) {
            street2 match {
                case None => s"$street1, $city $state $zip"
                case Some(s2) => s"$street1, $s2, $city $state $zip"
            }
        } else {
            s"$street1, $city $state $zip"
        }
    }
}

object Address1 extends App {

    /** A generic higher-order function
     */
    def dynaprint[T](obj:T, fun:T => String) = {
```

```
        println(fun(obj))
    }

    var pmr = new Address1("10 Downing Street", None, "Westminster", "ENG", "SW1")

    dynaprint(pmr, (a:Address1) => a.address(false))

    pmr.city = "London"

    dynaprint(pmr, (a:Address1) => a.address(false))
}
```

The second argument in the calls to `dynaprint` is a function given as a **lambda expression**. This one also represents a **closure** of the address method, fixing the Boolean argument to be `false` in the `address` method so that the wonky `street2` line is not printed. (Note: Uses of the "rocket" **=>** often need parentheses to disambiguate---when you don't expect it.)

Such "functional" topics are worth knowing about and sometimes make code nicer in larger examples. In the "CSE305: Programming Languages" course they receive more emphasis for themselves. Here we will defer them until some good examples of their use in data structures come along. In this example, IMHO the body of the code is more readable if we define a new function and pass that as a named argument rather than the lambda expression.

```
    /** A "closure" of the address method (not "curried")
     */
    def simpleAddress(a1:Address1) = a1.address(false)

    var pmr = new Address4("10 Downing Street", None, "Westminster", "ENG", "SW1")

    dynaprint(pmr, simpleAddress)

    pmr.city = "London"

    dynaprint(pmr, simpleAddress)
```

Later we will expand on examples like this to show how Scala uses **call by name** with function parameters. What this means is that the name of the function is treated as a `val` whose value is a textual block of code. The code is left as text when being substituted into the body of the calling function (here, `dynaprint`) rather than being evaluated ahead of time. This gives just the behavior that you would expect from mathematical substitution of formulas. For now, however, I prefer to get back to object-oriented topics that will better set up for chapters 2,3,4 and beyond.

Inheritance and Immutability

A storied conundrum in OOP is whether a `Square` "Is-A" `Rectangle`. The purely mathematical answer is *yes*, since squares are a subset of rectangles. The problem in OOP, however, is exemplified in Scala by the following code:

```
class Rectangle(var length: Double, var width: Double) {
   def dims = s"$length x $width"
}
class Square(var side: Double) extends Rectangle(side, side)

object SquareTest extends App {
   val r: Rectangle = new Square(3.0)
   r.length = 4.0    //note again: "fake const"
   println("The Square object has dimensions " + r.dims)
}
```

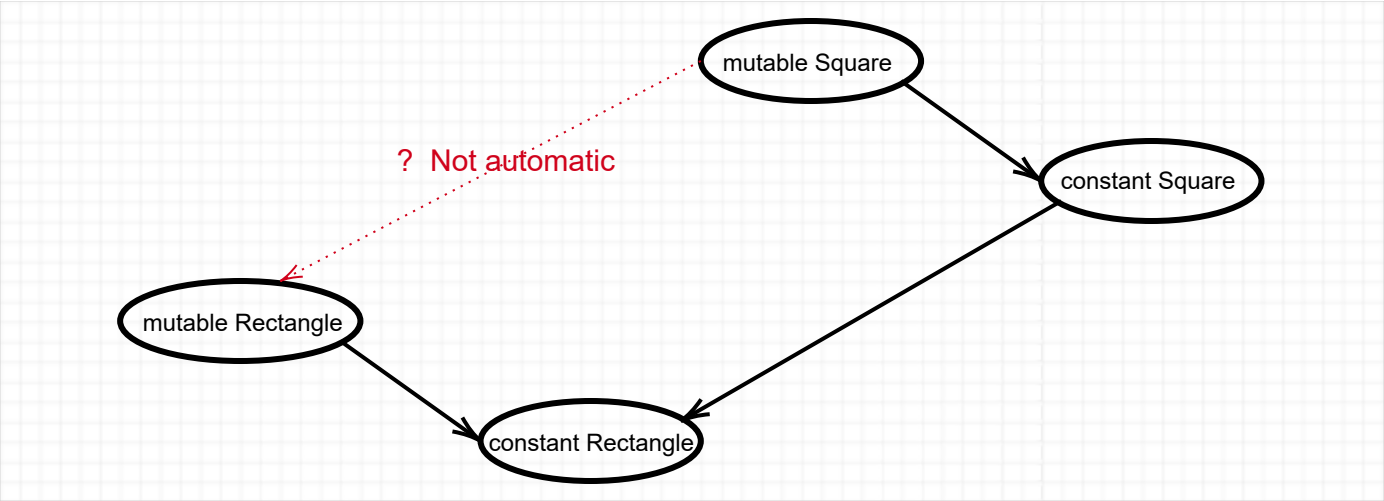
The `Square` object is supposed to be holdable by a parent-class variable like `r`, but doing so is not **safe** because `r` can invoke a setter to make the sides unequal.

But if we make the underlying fields immutable then this issue does not arise:

```
class Rectangle(val length: Double, val width: Double) {
   def dims = s"$length x $width"
}
class Square(val side: Double) extends Rectangle(side, side)

object SquareTest2 extends App {
   val r: Rectangle = new Square(3.0)
   //r.length = 4.0    //caught by compiler since *data* is immutable
   println("The Square object has dimensions " + r.dims)
}
```

That is to say, a *constant* `Square` really "Is-A" *constant* `Rectangle`. I once had the idea that base classes `Foo` in OOP should be immutable by default, with an inner subclass `Foo.Mutable` allowing mutation. By the rules of inner subclasses (in Java, say), we could have `Square` inherit from `Rectangle` without `Square.Mutable` inheriting from `Rectangle.Mutable`:



It is OOP-safe for a constant-Rectangle variable to hold a mutable Rectangle, even a mutable Square, so long as the const can't be cast away. However, it doesn't sound sensible to say "a mutable rectangle is a constant rectangle." For numerous other reasons I shelved the idea.

Scala goes halfway by defining a kind of constant class, but with restrictions on inheritance. It is called a **case class**. The larger intent is to make them cases of a wider base abstraction---like inheritance in reverse---but they also work by themselves:

```
case class Rectangle(length: Double, width: Double) {
   def dims = s"$length x $width"
}
```

The length and width are `val` by default. Then we do not need `new` when constructing a `Rectangle` object. (This is so with C++ objects as well when they are treated as values.) We can inherit from a `case class`, but only with a `class`:

```
/*case*/ class Square(val side: Double) extends Rectangle(side, side)

object SquareTest3 extends App {
   var r = Rectangle(2.0,3.0)
   r = new Square(3.0)
   //r.length = 4.0    //caught by compiler since *data* is immutable
   println("The Square object has dimensions " + r.dims)
}
```

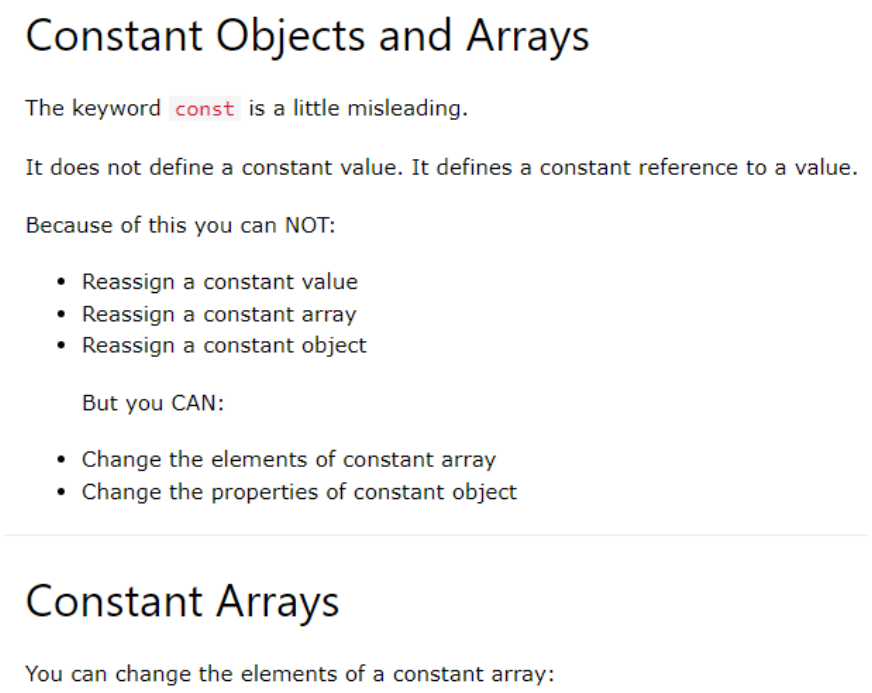
Commenting in the second word "case" gives the error

```
error: case class Square has case ancestor Rectangle, but case-to-case
inheritance is prohibited. To overcome this limitation, use extractors to
pattern match on non-leaf nodes.
```

So we can't fully solve the "Square" versus "Rectangle" issue with case classes. Instead, they are targeted toward design patterns like growing a hierarchy of `Shape`s and co-ordinating code that was written locally for each shape in the hierarchy. This can be done by **pattern matching** as a compile-time safer alternative to "dynamic casting". We'll see this next week in the course of exploring ways to manage a basic list data structure.

Footnote 1: I see that CSE116 has gotten as far as covering case classes in Scala, with reference to the page https://docs.scala-lang.org/tour/case-classes.html But I'm a believer that the second pass from a different angle of attack is what sticks...

Footnote 2: The W3Schools JavaScript website is up-front about the limitations of const here:



Constant Objects and Arrays

The keyword `const` is a little misleading.

It does not define a constant value. It defines a constant reference to a value.

Because of this you can NOT:

- Reassign a constant value
- Reassign a constant array
- Reassign a constant object

But you CAN:

- Change the elements of constant array
- Change the properties of constant object

Constant Arrays

You can change the elements of a constant array:

Footnote 3: In JavaScript, `var` defines a **dynamically scoped** variable. Great, IMPHO, for stream-of-consciousness scripting. Maybe not so great for maintaining code.

Footnote 4: The text covers the "Square Is-A Rectangle?" conundrum more broadly in section 4.2.2 on inheritance.