Hello World in Scala

- With “main” function
  ```scala
  object HelloWorld {
    def main(args:Array[String]) {
      println("Hello World!");
    }
  }
  ```

- By extending “App”
  ```scala
  object HelloWorld extends App {
    println("Hello World!");
  }
  ```

Data Types and Operators
- Data Types
  - Byte, Char, Short, Int, Long, Float, Double, Boolean
- Types are classes
  - “101”.toInt()
- String
- Array
- Operators
  - +, -, *, /, %, ==, <=, |, &&, ||, etc.
- Operators are functions
  - (1)+(2) == 3
- What is the result of “1.+(2)”?

Variables
- Definition
  - var a:Int; // Mutable Variable
  - val a:String; // Immutable Variable
  - var a:Int = 42; // Initialization
  - var a = 42; // Type inference

- Manipulation
  - a = 37;
Control Flow
- Every statement is an expression
  - if-else
    ```scala
    var y = if (x < 0) -x else x;
    ```
- while loop, do ... while loop
  ```scala
  while (a > 0) {
    ...
  }
  ```
- for loop
  ```scala
  for ( j <- 1 to 10) {
    sum += j
  }
  ```

Functions
- Definition
  ```scala
  def fn(x: Int) = {
    var y = 2*x;
    y + 1;
  }
  ```
- Invocation
  ```scala
  fn(x);
  ```

More about Functions
- Function as Objects
  ```scala
  def println(x: Int) = println("x=" + x);
  def repeat(fn: Int => Unit) = while(true) { fn(0);
    Thread.sleep 1000 } }
  ```
- Anonymous Functions
  ```scala
  repeat(x => {
    var y = 2+x;
    println("xx=" + y);
  });
  ```

Recursion
- Function calls itself in its body
- Must specify return type
  ```scala
  def factorial(n: Int): Int = if (n <= 0) 1 else factorial(n-1)*n;
  ```
- Tail Recursion
  ```scala
  import scala.annotation.tailrec
  @tailrec def factorialAcc(acc: Int, n: Int): Int = {
    if (n <= 1) acc;
    else factorialAcc(n * acc, n - 1);
  }
  ```
- Warning if the compiler cannot optimize
Classes

- Example
  ```scala
  class Complex(real: Double, imaginary: Double) {
    def re = real;
    def im = imaginary;
    def mod = sqrt(re * re + im * im);
  }
  ```

- Fields
- Methods
- Inheritance and Overriding
  - To be discussed in next lecture

Constructors

- Parameters
  - Parameters are class fields

- Mutable vs. Immutable
  - Default: Immutable
    ```scala
    class Complex(var real: Double, ...)
    ```

- Instantiation
  ```scala
  val a = new Complex(2, 3);
  ```

Functions / Methods

- No behavioral difference between Field and Method

- Field behaves like a method with no argument
  - Omit the parenthesis

- Private
  ```scala
  private[this] var a;
  ```
  - Restrict the access to only the current class

- Package-Private

Objects

- “Singleton Class”

- Allows only one instance of the class

- No parameters

- Instantiated lazily

- Companion class

- Object instance and class share the same name
Traits

- "Interface Class"
  - Defines a set of methods
  - Can be used for inheritance
  - Can also implement some of the methods

trait Ord {
  def < (that: Any): Boolean;
  def <=(that: Any): Boolean = (this < that) || (this == that);
  ...
}

Case Classes

- Example
  abstract class Tree;
  case class Sum(l: Tree, r: Tree) extends Tree;
  case class Var(n: String) extends Tree;
  case class Const(v: Int) extends Tree;

- Case Classes vs. Regular Classes
  - Constructor Parameters are directly accessible
  - No need to use "new"
  - Provides "equals" and "hashCode"
  - Provides "toString"

Pattern Matching

- Match
  def eval(tree:Tree):Int = tree match {
    case Var(n) => ...
    case Const(v) => ...
    case Sum(l, r) => ...
  }

- Pattern Matching vs. Inheritance Method
  - Pattern Matching: Easy to add a new operation
  - Inheritance Method: Easy to add new class

Generics

- Template Class
  - Since JavaSE 5.0
  - Allows a class to contain multiple different types

class ListNode[T] {
  private var content:T;
  private var nextNode:ListNode[T];
  ...
}

val startNode:ListNode[Int];

Array is a generic class
Type System

- Everything is Object
  - Scala.Any – Supertype for any object
  - Scala.AnyVal – Supertype of values
  - Scala.AnyRef – Supertype of references

- Scala.Nothing – Subtype of any object
- Scala.Null – Subtype of references
- Scala.Unit – Unit Type

Exception Handling

- Try-catch-throw
  ```scala
def fn(x:Int) throws SomeException {
  try {
    throw AnotherException
  } catch (YetAnotherException) {
    ...
  } finally {
    ...
  }
}
```

Implicit Conversion

- Used in type inference
  ```scala
  object ComplexImplicits {
    implicit def Double2Complex(value : Double) : Complex = new Complex(value,0.0)
  }

  Import ComplexImplicits._
  // Use it
  ```

Domain-Specific Languages

- Definition
  - A programming languages used only in a specific application domain
  - In contrast to Generic Programming Languages (GPLs)

- Examples
  - HTML
  - Verilog/VHDL
  - Matlab
  - SQL
Domain-Specific Languages

- **External DSL**
  - DSL outside of the scope of the corresponding GPL
  - Requires re-inventing the mechanisms

- **Internal DSL**
  - DSL using the primitives & semantics of the corresponding GPL
  - Often restricted

DSLs in Scala

- **What in Scala allows you to blend extensions with the native language?**
  - Parentheses are optional for invocation of 0-ary functions
    - `o.call is the same as o.call()`
  - Dot is optional
    - `o.call is the same as o.call and o.call()`
  - Makes the syntax more fluent

  - No instantiation required for singleton objects
    - `Object PRINT {
        ...
        }`
    - `PRINT`
    - Makes “PRINT” look like an instruction

Domain-Specific Languages

- **Examples of a DSL in Scala**
  - http://www.scala-lang.org/old/node/1403
  - https://github.com/fogus/baysick
    - Basic in Scala
    - https://github.com/gjacobrobertson/baysick
    - Improved cleanroom implementation by my former TA

  - Example of an External DSL

  - Example of an Internal DSL
    - http://debasishg.blogspot.com/2008/05/designing-internal-dsls-in-scala.html

DSLs in Scala

- **Apply**
  - Syntactic sugar method that is considered when no function call is given
    - `object PRINT {
        def apply(s : String) {
            println(s)
        }
    }`
    - `PRINT("Hello World")`

- **Implicits**
  - Automagically convert objects to match expected type
  - Can be used to synthesize auxiliary objects (see LineBuilder from Integer in Baysick through implicit int2LineBuilder)