```
public void run() {
 * Create tidisplay ( finterpreter getHelloPrompt() ); //declared here only to ensure visibility in finally claus
 * this method should be invoked from the beginning the event-display in the null ntroduction to Java
            //fInterpreter's result
private statingutStreamReader inputStreamReader = new InputStreamReader ( System4.iJava Flow Control
            BufferedReader stdin = new BufferedReader ( inputStreamReader ) putStream ( buffer );
            boolean hasRequestedQuit = false
            String line snull, lang. Object
    //Creat List resulte as new ArrayList (); name> //display its data
    frame = try {JFrame("FocusConceptsDemo");
frame.setiwhile (!hasRequestedOnit)" [ava.lang.String">Iterator quarksItr = recoveredQuarks.iterator();
frame.setiwhile (!hasRequestedOnit)" [ava.lang.String">while ( quarksItr.hasNext() ) {
frame.setiwhile (!hasRequestedOnit)" [ava.lang.String">while ( quarksItr.hasNext() ) {
                 line 'stdin readline() dva lang
    //Create and/note that "result" is passed as an "out" parameter
    JComponent hasRequestedQuit = fInterpreter parseInput( line, result );
    newContentPdisplay(Sresult: ) | Sol Maring | les must be | light of lass Console |
    frame.setCorresult clear() on tentPane
    //Display the window.
    frame p catch ( IOException ex ) {
    frame.set System, erraprintln(ex) (lass name>java.util.Gregor
               The inheritance tree
                                                                                                  List [aText ] ... (ent.s.
                                                                                          ispl
            finally {class java.util.GregorianCalendar
                                                                                                   = aText.iterator()
                                                                                          extIt
                                                                                                hasNext() a)ur{uments
public static display(fBYE) ring[ args]
                                                                                               ter. next() preter) the
    //Scheduleshutdown (stdin event dispatching threa
    //creating and showing this application's GU
    javax. swing. SwingUtilities.invokeLater(new_
                                                                                                              aStdin )
          /// PRIVATE ////
          private static final String fBYE =
          private Interpreter fInterpreter
          /**
 void pri*htDisplay some ttextesto stdouten
     final String[] mvStrings = new String[2]
```

Introduction

- The focus of this module is the imperative style of programming we do in methods
 - This is generally called flow control because it is the code that controls the flow of logic
 - Flow control also generally includes the basic operators of the language
 - Java is very similar to all other C-style languages at this level of code
- However, this module assumes you already know
 - How Boolean, arithmetic and other operators work
 - What the basic flow control structures are and how they work
- What this module will focus on are the following:
 - Things that Java does with operators and flow control that are unique to Java
 - Java gotchas
 - Best practices for operators and flow control



Operators

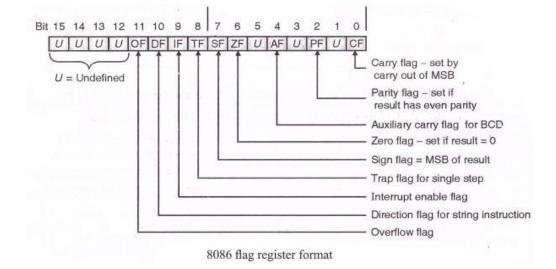
- The standard list of operators usually looks something like the chat shown
- There is also an order of precedence defined for Java operators
- Clean code tip
 - Always explicitly state the order of operations using () to group terms
 - This ensures that Java, you and anyone else reading your code agree on what is happening
- Some of the operators are holdover from C and really aren't used
 - |, &, ^, <<, >>, >>>
 - These were primarily used for bitlevel operations

Precedence	Operator	Operand type	Description
1	++,	Arithmetic	Increment and decrement
1	+, -	Arithmetic	Unary plus and minus
1	~	Integral	Bitwise complement
1	· !	Boolean	Logical complement
1	(type)	Any	Cast
2	*, /, %	Arithmetic	Multiplication, division, remainder
3	+, -	Arithmetic	Addition and subtraction
3	+	String	String concatenation
4	<<	Integral	Left shift
4	>>	Integral	Right shift with sign extension
4	>>>	Integral	Right shift with no extension
5	<, <=, >, >=	Arithmetic	Numeric comparison
5	instanceof	Object	Type comparison
6	==,!=	Primitive	Equality and inequality of value
6	==,!=	Object	Equality and inequality of reference
7	&	Integral	Bitwise AND
7	&	Boolean	Boolean AND
8	٨	Integral	Bitwise XOR
8	٨	Boolean	Boolean XOR
9	1	Integral	Bitwise OR
9	li	Boolean	Boolean OR
10	&&	Boolean	Conditional AND
11	II	Boolean	Conditional OR
12	?:	N/A	Conditional ternary operator
13	=	Any	Assignment



Bitwise Operators

- These operators were very common in C and C++ when an integer might represent a set of status flags to encode a lot of information in a small amount of data
 - This was done to save bandwidth in transmission
 - Data encoded and extracted from the integer by XORing
- Notice that we still use sub-net masks when working with IP addresses – same idea
- Java is not usually used for this sort of work so it's rare to see these operations in Java code
- There are examples in the demos





Mixed Mode Arithmetic

- In the module on data types, you saw that Java will cast data from one type to another during assignment operations
 - But if the assignment violates Java's type safety rules, Java will generate a compiler error
- Mixed mode arithmetic is when the two operands of an arithmetic operator are of different data types
 - Java can only apply the operator if the two operands are of the same type
 - Java will start casting the operands until the operands are the same type
 - The same casting rules as before apply
 - If Java cannot cast an operand, a compile error is generated
 - For example:
 - If we have int + long, then the int is cast to a long
 - If we have float + int, then the int is cast to a float
 - If we have string + int, then the int is cast to a string
- Best practices for mixed mode arithmetic
 - Avoid it like the plague because it is the source of subtle bugs
 - Instead, ensure you cast all the operands yourself so you can control the data type conversions



The Increment and Accumulate Operators

- The ++ increment and decrement operator -- are simple in concept
 - Confusion occurs as to the difference between i++ and ++i
 - For i++ the value of i is used then incremented
 - For ++i the value of i is incremented then used
 - In most cases, it doesn't matter which form is used
 - But it can be the source of subtle bugs
 - Demonstrated in the demo
- The accumulate operator is of the form x (operator)= value
 - This is the same as x = x (operator) value
 - For addition, it would be x+=3 would be the same as x = x + 3
- Generally, accumulate expressions are
 - Anything other than the simplest accumulate expression can be hard to understand
 - They can also be tricky to use and make the code more difficult to debug











