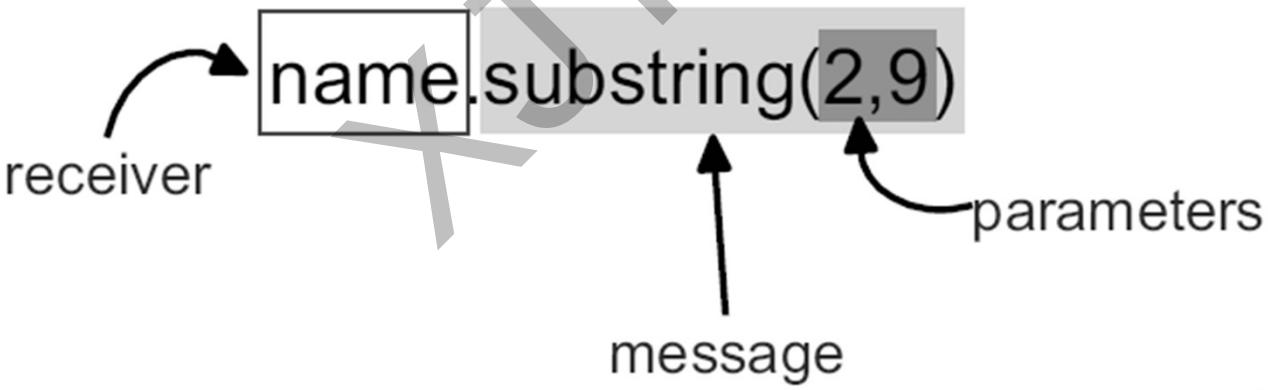
Unit objectives

- After completing this unit, you should be able to:
  - Create and initialize objects
  - Use the identity (==) operator
  - Identify and use primitive wrapper classes
  - Outline Java's implementation of Strings, and work with Strings
  - Explain the difference between the String and StringBuffrer classes
  - Use conditional statements
  - Use looping and branching structures
  - Explain variable scope

Objects and messages

- Objects provide more complex behavior than primitive data types
- Objects respond to messages
  - The dot "." operator is used to send a message to an object



Declaring and initializing objects

- Just like primitives and arrays, objects must be declared before they can be used
  - The declaration requires the *type* of the object
    - The type is the class of the object
- Use = for assignment (including initialization)
- Initialization of an object often uses the new operator
  - The new operator is used if you want to create a new object
- An object can be initialized to null
- Arrays of objects are declared just like arrays of primitives
  - Arrays of objects default to initialization with null
- Examples:

```
Employee emp1 = new Employee(123456);
Employee emp2;
emp2 = emp1;
Department dept[] = new Department[100];
Test[] t = {new Test(1), new Test(2)};
```

Identity

- The == relational operator
  - When this operator is used on objects, it tests for exact object identity
  - Checks whether two variables reference the same object
  - When this operator is used on primitive types, it checks for equal values

```
Employee a = new Employee(1);
Employee b = new Employee(1);
if (a==b)... // false
```

```
Employee a = new Employee(1);
Employee b = a;
if (a==b)... // true
```

```
int a = 1;
int b = 1;
if (a==b)... // true
```

Wrapper classes

- Primitives have no associated methods; there is no behavior associated with primitive data types
- Each primitive data type has a corresponding class, called a *wrapper*
  - Each wrapper object simply stores a single primitive variable and offers methods with which to process it
- Wrapper classes are included as part of the base Java API

Primitive Type	Wrapper Class
boolean	Boolean
byte	Byte
char	Character
double	Double
float	Float
int	Integer
long	Long
short	Short

Using wrapper classes

```
double number = Double.parseDouble("42.76");
```

```
String hex = Integer.toHexString(42);
```

```
double value = new Integer("1234").doubleValue();
```

```
String input = "test 1-2-3";
int output = 0;
for (int index = 0; index < input.length(); index++) {
  char c = input.charAt(index);
  if (Character.isDigit(c))
    output = output * 10 + Character.digit(c, 10);
}
System.out.println(output) // 123
```

Strings

- The String type is a class, and not a primitive data type
- A String literal is made up of any number of characters between double quotes:

```
String a = "A String";
String b = "";
```

- String objects can be initialized in other ways:

```
String c = new String();
String d = new String("Another String");
String e = String.valueOf(1.23);
String f = null;
```

Concatenating strings

- The + operator concatenates Strings:
  - String a = "This" + " is a " + "String";
  - There are more efficient ways to concatenate Strings (this will be discussed later)
- Primitive types used in a call to println are automatically converted to Strings
  - System.out.println("answer = " + 1 + 2 + 3);
  - System.out.println("answer = " + (1+2+3));
  - Do you get the same output from the above examples?

String messages

- Strings are objects; objects respond to messages
  - Use the dot (.) operator to send a message
  - String is a class, with methods (more later)

```
String name = "Joe Smith";
name.toLowerCase();  // "joe smith"
name.toUpperCase();  // "JOE SMITH"
" Joe Smith ".trim();// "Joe Smith"
"Joe Smith".indexOf('e');   // 2
"Joe Smith".length();       // 9
"Joe Smith".charAt(5);      // 'm'
"Joe Smith".substring(5);   // "mith"
"Joe Smith".substring(2,5); // "e S"
```

Comparing strings

- Several messages can be sent to a String to test for equivalence with another String
- oneString.equals(anotherString)
  - Tests for equivalence
  - Returns true or false
- oneString.equalsIgnoreCase(anotherString)
  - Case insensitive test for equivalence
  - Returns true or false
- oneString == anotherString is problematic

```
String name = "Joe";
if ("Joe".equals(name))
   name += " Smith";
```

```
boolean same = "Joe".equalsIgnoreCase("joe");
```

StringBuffer

- The StringBuffer class provides a more efficient mechanism for building strings
  - String concatenation can get very expensive
  - String concatenation is converted by most compilers -including IBM Rational Application Developer - into a StringBuffer implementation
- If building a simple String, just concatenate; if building a String through a loop, use a StringBuffer

```
StringBuffer buffer = new StringBuffer(15);
buffer.append("This is ");
buffer.append("String");
buffer.insert(7," a");
buffer.append('.');
System.out.println(buffer.length());    // 17
System.out.println(buffer.capacity());  // 32
String output = buffer.toString();
System.out.println(output);  // "This is a String."
```

Conditional statement: if-else

- Conditional expression must evaluate to a boolean
- The else clause is optional
- Braces are not needed for single statements but are highly recommended for clarity

```
if (x > 10) {
    if (x != 20) {
        System.out.println("x is not 20");
    }
    else {
        System.out.println("x = " + x);
    }
}
else {
    System.out.println("x is less than 11");
}
```

Shortcut for if-else: the ternary operator

- Shortcut for if-else statement:
  - (<boolean-expr> ? <true-choice> : <false-choice>)
- Can result in shorter code
  - Make sure code is still readable

Code using `if-else`

```
if (x>LIMIT) {
   warning = "Too Big";
} else {
   warning = null;
}
```

Code using ternary operator

```
warning = (x>LIMIT) ? "Too Big" : null ;
```

Conditional statement: switch

- Tests a single variable for several alternative values and executes the corresponding case
- Any case without break will “fall through”
  - Next case will also be executed
- default clause handles values not explicitly handled by a case

```
switch (day) {
   case 0:
   case 1:
      rule = “weekend”;
      break;
   case 2:
      …
   case 6:
      rule = “weekday”;
      break;
   default:
      rule = “error”;
}
```

```
if (day == 0 || day == 1) {
   rule = “weekend”;
} else if (day > 1 && day <7) {
   rule = “weekday”;
} else {
   rule = error;
}
```

Conditional statement: switch

```
import java.util.*;
public class TestEnum {
 private enum Seasons {
  winter, spring, summer, autumn
 }

 public static void main(String args[]) {
  Scanner in = new Scanner(System.in);
  System.out.println
("please press which season
 is you like most:");
  String name = in.next();

  doIt(Seasons.valueOf(name));
 }
```

```
private static void doIt(Seasons c) {
 switch (c) {
 case autumn:
  System.out.println("value is" + Seasons.autumn);
  break;
 case winter:
  System.out.println("value is" + Seasons.winter);
  break;
 case spring:
  System.out.println("value is" + Seasons.spring);
  break;
 case summer:
  System.out.println("value is" + Seasons.summer);
  break;
 default:
  System.out.println("default");

 }
}
```

Looping statements: while and do…while

- Executes a statement or block as long as the condition remains true
- while() executes zero or more times
- do...while() executes at least once

```
int x = 2;
while (x < 2) {
   x++;
   System.out.println(x);
}
```

```
int x = 2;
do {
   x++;
   System.out.println(x);
} while (x < 2);
```

Looping statement: for

- A for loop executes the statement or block { } that follows it
  - Evaluates "start expression" once
  - Continues as long as the "test expression" is true
  - Evaluates "increment expression" after each iteration
- A variable can be declared in the for statement
  - Typically used to declare a "counter" variable
  - Typically declared in the “start” expression
  - Its scope is restricted to the loop

```
for (start expr; test expr; increment expr) {
  // code to execute repeatedly
}
```

```
for (int index = 0; index < 10; index++) {
  System.out.println(index);
}
```

for versus while

- These statements provide equivalent functionality
  - Each can be implemented in terms of the other
- These looping structures are typically used in different situations
  - while tends to be used for open-ended looping
  - for tends to be used for looping over a fixed number of iterations

```
int sum = 0;
for (int index = 1; index <= 10; index++) {
   sum += index;
}
```

```
int sum = 0;
int index = 1;
while (index <= 10) {
        sum += index;
        index++;
}
```

Branching statements

- **Break**
  - Can be used outside a switch statement
  - Terminates a for, while or do...while loop
  - Two forms:
    - Labeled: execution continues at next statement after labeled loop
    - Unlabeled: execution continues at next statement outside loop
- **Continue**
  - Like break, but merely skips the remainder of this iteration of the loop, then continues by evaluating the boolean expression of the innermost loop
  - Labeled and unlabeled forms
- **Return**
  - Exits the current method
  - May include an expression to be returned
    - Type must match method's return type
    - A void return type means no value can be returned

Sample branching statements

```
public int myMethod(int x) {

    int sum = 0;
    outer: for (int i=0; i<x; i++) {
        inner: for (int j=i; j<x; j++) {
                sum++;
                if (j==1) continue;
                if (j==2) continue outer;
                if (j==3) break;
                if (j==4) break outer;
        }
    }
    return sum;
}
```

Scope

- A variable's scope is the region of a program within which the variable can be referenced
  - Variables declared in a method can only be accessed in that method
  - Variables declared in a loop or a block can only be accessed in that loop or block

```
int a = 1;
for (int b = 0; b < 3; b++){
   int c = 1;
   for (int d = 0; d <3; d++){
      if (c < 3) c++;
   }
   System.out.print(c);
   System.out.println(b);
}
a = c; // ERROR! c is out of scope
```

abcd

abc

a

Checkpoint

- 1. Provide two ways to create an array of five integers,initialized to zero
- 2. Explain the difference between the code (x==y) and x.equals(y)
- 3. What are the differences between a String and a StringBuffer object, and why would a developer choose to use one over the other?
- 4. What are Java's three branching statements?

Unit summary

- In this unit, you should have learned to:
  - Create and initialize objects
  - Use the identity (==) operator
  - Identify and use primitive wrapper classes
  - Outline Java's implementation of Strings, and work with Strings
  - Explain the difference between the String and StringBuffer classes
  - Use conditional statements
  - Use looping and branching structures
  - Explain variable scope