Honors Computer Programming 1-2

Introduction To Chapter 3
Fundamental Data Types

Chapter Goals

- To understand integer and floating-point numbers
- To recognize the limitations of the int and double types and the overflow and round off errors that can result
- To write arithmetic expressions in Java
- To use the `String` type to define and manipulate character strings
- To learn about the `char` data type
- To learn how to read program input
- To understand the copy behavior of primitive types and object references

Number Types

In this chapter we will use a `Purse` class to demonstrate several important concepts. The general outline for the `Purse` class is shown at the right.

```
public class Purse
{
  public Purse( )
  {
    // implementation
  }

  public void addNickels(int count)
  {
    // implementation
  }

  public void addDimes(int count)
  {
    // implementation
  }

  public void addQuarters(int count)
  {
    // implementation
  }

  public double getTotal( )
  {
    // implementation
  }

  // private instance variables
}
```

Number Types

There is a constructor to make a new purse:

```
Purse myPurse = new Purse( );
```

. You can add nickels, dimes, and quarters with statements such as `myPurse.addNickels(3);`, `myPurse.addDimes(1);`, and `myPurse.addQuarters(2);`.

You can ask the `Purse` object about the total value of the coins in the purse:

```
double totalValue = myPurse.getTotal( );  // returns 0.75
```

If you look closely at the methods, you will see a variable `count` of type `int`. This `int` denotes an integer type. An integer is a number without a fractional part. For example, `3` is an integer but `0.05` is not. The number zero and negative numbers are integers. Thus, the `int` type is more restrictive than the type `double` we looked at in chapter 2.

Number Types

Why do we need both an `int` type and a `double` type? The first reason is one of _philosophy_ in which case we can't have anything other than a _whole_ number of nickels. The second reason is that integers are more _pragmatic_ than floating-point numbers since they take less _storage_ space, are processed _faster_, and don't cause _rounding_ errors.

Number Types

Now lets start implementing the `Purse` class. Any `Purse` object can be described in terms of the number of nickels, dimes, and quarters. Thus we use three instance fields to represent the state of a `Purse` object.

instance fields →

```
public class Purse
{
   ...
   private int nickels;
   private int dimes;
   private int quarters;
}
```

Number Types

We can also implement the `getTotal` method:

```
public double getTotal( )
{
  return  nickels * 0.05 + dimes * 0.1 + quarters * 0.25;
}
```

return a floating-point number

Number Types

In Java, multiplication is denoted as an asterisk `*`. Do not write commas or spaces in numbers. For example, `10,150.75` must be entered as `10150.75`. To write numbers in exponential notation in Java, use `E`.

For example, to enter the number 5.0×10^{-3} you write `5.0E-3`

The `getTotal` method computes the value of

```
nickels * 0.05 + dimes * 0.1 + quarters * 0.25
```

and returns a floating-point number of type `double`. The `return` statement returns the computed value as the method result and the method exits.

Number Types

You may be tempted to use `int n` for one of the instance fields instead of `int nickels`. Don't do it. Descriptive variable names are a better choice because they make your code easy to read without requiring comments.

Number Types

Unfortunately, `int` and `double` values do suffer one problem: they cannot represent arbitrarily large numbers. Integers have a range of `-2,147,483,648` to `2,147,483,647` (about - 2 billion to 2 billion).

If you want to represent the world population, you can't use `ints`.

`Double` numbers can go up to more than 10^{300}. However, `doubles` suffer from a lack of precision. They only store about `15` significant digits.

Number Types

Suppose your customers might find the price of 300 trillion dollars for your product a little excessive, so you want to reduce it by 5 cents. Consider the program:

```
public double AdvancedTopic
{
  public static void main( String[ ] args)
  {
    double origPrice = 3E14;
    double discountedPrice = origPrice - 0.05;
    double discount = origPrice - discountedPrice; // should be 0.05

    System.out.println(discount);          // prints 0.0625
  }
}
```

The program prints `0.0625` instead of `0.05`. It is off by more than a penny. Most of the time using `int` and `double` are acceptable. Keep in mind that overflows and loss of precision can occur.

Assignment

The default constructor for the Purse class is shown below.

The = operator is called the assignment operator. On the left, you need a variable name. The right-hand side can be a single value or an expression. The assignment operator sets the variable to the given value.

```
public Purse( )
{
    nickels = 0;
    dimes = 0;
    quarters = 0;
}
```

left-hand side is a variable

right-hand side is a value or expression

Assignment

Now look at the code:

```
public void addNickels(int count)
{
   nickels = nickels + count;
}
```

It means: compute the value of the expression `nickels + count` and places the result into the variable `nickels` .

The = sign doesn't mean that the right side is equal to the left side but that the right side is copied into the left-hand side variable.

The statement `nickels = nickels + 1` has the effect of incrementing `nickels` . So if nickels was 3 before the statement , then nickels is 4 after the statement. This operation is so common that there is a special shorthand for it: `nickels++;`

So the ++ is called the increment operator. There is also a decrement operator. The statement `nickels--;` subtracts 1 from nickels .

Assignment

In Java, you can combine arithmetic and assignment.

For example, the statement `nickels += count` is a shortcut for `nickels = nickels + count`. Similarly, the statement `nickels += 2` is a shortcut for `nickels = nickels + 2`.

Constants

The statement

```
nickels * 0.05 + dimes * 0.1 + quarters * 0.25
```

depends on the numeric quantities `0.05`, `0.1`, and `0.25`. The code would be easier to understand if it were written as:

```
nickels * nickelValue + dimes * dimeValue
                      + quarters * quarterValue
```

There is a difference between the `nickels` and the `nickelValue` variables. The variable `nickels` will vary in value during the lifetime of the program. But `nickelValue` is always `0.05`. That is, `nickelValue` is constant. In Java, constants are declared with the keyword `final`. As a matter of style, we will use all uppercase letters to identify constants. So the above statement might appear:

```
nickels * NICKEL_VALUE + dimes * DIME_VALUE
                       + quarters * QUARTER_VALUE
```

Constants

Frequently constants are needed in several methods of the class.

Then you need to declare them together with instance variables of the class and tag them as `static final` .

The keyword `static` will be discussed in chapter 6.

The general setup is shown at the right.

```
public class Purse( )
{
   // methods
   ...
   // constants
   private static final double NICKEL_VALUE = 0.05;
   private static final double DIME_VALUE = 0.1;
   private static final double QUARTER_VALUE = 0.25;

   // instance variables
   private int nickels;
   private int dimes;
   private int quarters;
}
```

constants are declared as `static final`

Constants

It is possible to declare constants as `public` :

```
public class Math( )
{
    ...
    public static final double E = 2.71828182845904523536;
    public static final double PI = 3.14159265358979323846;
}
```

An example of this comes from the `Math` class which is part of the standard library. You can refer to the `public` constants shown as `Math.E` and `Math.PI` . For example,

```
double circumference = Math.PI * diameter;
```

Arithmetic and Mathematical Functions

Division is indicated with a slash / not a fraction bar. For example,

$\frac{a+b}{2}$ becomes `(a + b) / 2` . Parenthesis are used to indicate the order in which subexpressions are computed . For example,

in the expression `(a + b) / 2` the sum `a + b` is computed first and then the result is divided by 2 . In contrast,

in the expression `a + b / 2` only `b` is divided by 2 and then sum of `a` and `b / 2` is formed.

Arithmetic and Mathematical Functions

Division works as you would expect as long as one of the arguments is a floating-point number. That is, `12.0 / 8` and `12 / 8.0` and `12.0 / 8.0` all yield `1.5`. However, if all arguments are integers then the result is an integer with the remainder discarded.

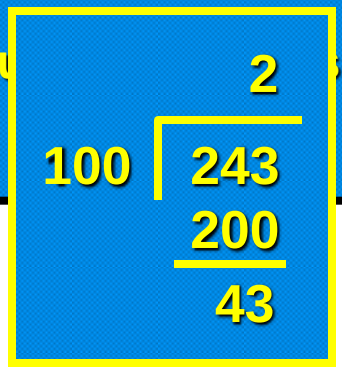
That is, `7 / 4` evaluates to `1` because 7 divided by 4 is 1 with a remainder of 3 (which is discarded).

Arithmetic and Mathematical Functions

If you are interested only in the remainder use the __%__ operator.

So `7 % 4` is equal to __3__ . The % operator is referred to as the __modulus__ operator.

Here is a typical use of the / and % operators: convert a nu s into number of dollars and resulting change.



For example, if total is `243` , then dollars is set to __2__ and cents is set to ____ .

```
final int PENNIES_PER_NICKEL = 5;
final int PENNIES_PER_DIME   = 10;
final int PENNIES_PER_QUARTER = 25;
final int PENNIES_PER_DOLLAR = 100;

// compute total value in pennies
int total = nickels * PENNIES_PER_NICKEL
                    + dimes * PENNIES_PER_DIME
                      + quarters * PENNIES_PER_QUARTER;

// use integer division to convert to dollars & cents
int dollars = total / PENNIES_PER_DOLLAR;
int cents = total % PENNIES_PER_DOLLAR;
```

Arithmetic and Mathematical Functions

If you are interested only in the remainder use the __%__ operator.

So `7 % 4` is equal to __3__ . The % operator is referred to as the __modulus__ operator.

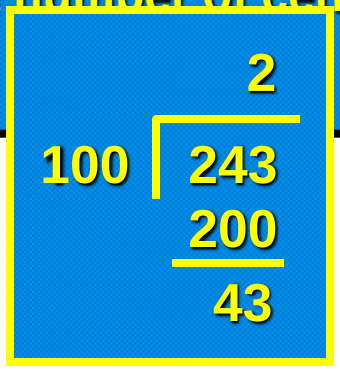
Here is a typical use of the / and % operators: convert a number of cents into number of dollars and resulting change.

For example, if total is `243` , then dollars is set to __2__ and cents is set to __43__ .

```
final int PENNIES_PER_NICKEL = 5;
final int PENNIES_PER_DIME = 10;
final int PENNIES_PER_QUARTER = 25;
final int PENNIES_PER_DOLLAR = 100;

// compute total value in pennies
int total = nickels * PENNIES_PER_NICKEL
                    + dimes * PENNIES_PER_DIME
                      + quarters * PENNIES_PER_QUARTER;

// use integer division to convert to dollars & cents
int dollars = total / PENNIES_PER_DOLLAR;
int cents = total % PENNIES_PER_DOLLAR;
```



Arithmetic and Mathematical Functions

It is unfortunate that Java uses the same / symbol for both integer and floating-point divisions. It is a common error to use integer division by accident. Consider the program:

```
int s1 = 5;   // score of test 1
int s2 = 6;   // score of test 2
int s3 = 3;   // score of test 3
double ave = (s1 + s2 + s3)/3;  // computation error

// output average test score
System.out.println("Your average is " + ave);
```

Because s1, s2, and s3 are all integers the scores add up to the integer 14 which when divided by 3 will produce a quotient of 4 with the remainder 2 being discarded.

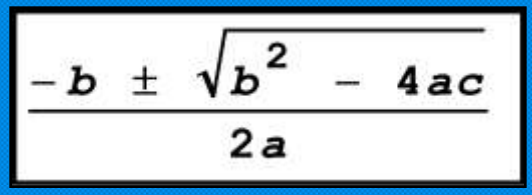
The remedy is to make either the numerator or the denominator into a floating-point number. One solution is to declare `double total = s1 + s2 + s3;` and then divide total by 3 while a second solution is to change the average calculation so that you divide by a floating-point number: `double ave = (s1 + s2 + s3) / 3.0;`

Arithmetic and Mathematical Functions

The table below (see page 95) shows some (but not all) of the methods in the `Math` class:

Function	Returns
`Math.sqrt(x)`	square root of x
`Math.pow(x, y)`	x raised to the y power
`Math.sin(x)`	sine of x (x in radians)
`Math.exp(x)`	e raised to the x power
`Math.round(x)`	closest long integer to x
`Math.abs(x)`	absolute value of x
`Math.min(x, y)`	minimum of x and y
`Math.max(x, y)`	maximum of x and y

So the subexpression of the quadratic formula



becomes:

```
(-b + Math.sqrt(b * b - 4 * a * c )) / (2 * a)
```

Arithmetic and Mathematical Functions

Consider the expression:

```
(1.5 * ((-b - Math.sqrt( b * b - 4 * a * c)) / (2 * a))
```

What's wrong with it? Count the parenthesis . The parenthesis are unbalanced since there are 5 opening parenthesis but only 4 closing parenthesis.

Arithmetic and Mathematical Functions

Now consider the following:

```
1.5 * (Math.sqrt(b * b - 4 * a * c)) - (( b / (2 * a))
```

If you count you will find 5 opening and 5 closing parenthesis.

But something is wrong. Here is a trick that finds the error: start counting 1 at the first opening parenthesis, add 1 whenever you see another opening parenthesis, but subtract 1 when you see a closing parenthesis.

If the count ever drops below 0 or if the count isn't 0 at the end, the parenthesis are unbalanced. In this case:

```
1.5 * (Math.sqrt(b * b - 4 * a * c) ) ) - (( b / (2 * a))
      1          2                  1 0 -1
```

error here!!

Calling Static Methods

The methods of the `Math` class, such as the `sqrt` method, are different than those of some other methods. The `getTotal` method of the `Purse` class operates on a `Purse` object. The `println` method operates on a `System.out` object.

Calling Static Methods

But the `sqrt` method does not operate on any object . That is, you do not call

```
double x = 4;
double root = x.sqrt( );  // error
```

The reason is that in Java, numbers are not objects. Actually, the number is a parameter in a method call such as `Math.sqrt(x)` .

This call makes it appear as if `Math` is an object since it looks like the call `myPurse.getTotal( )` in which case the `getTotal` method is applied to the object `myPurse` . However, `Math` is a class not an object.

A method such as `Math.round` that does not operate on an object is called a `static` method. To call a `static` method, you must specify the name of the class hence the call `Math.sqrt` or `Math.round` .

Calling Static Methods

How can you tell if `Math` is a **class** or an **object**? All classes in the Java library start with an uppercase letter (such as `System` or `Math`). Objects and methods start with a lowercase letter (such as `out` or `println`).

You can tell objects and methods apart since method calls are followed by a **parenthesis**. Therefore, `System.out.println( )` denotes a call of the `println` method on the `out` object inside the `System` class.

On the other hand, `Math.sqrt(x)` denotes a call to the `sqrt` method inside the `Math` class.

Type Conversion

When you make an assignment of an expression into a variable, the type of the variable and the expression must be compatible .

For example, it is an error to assign:

```
double total = "a lot"; // error
```

because `total` is a floating-point variable and `"a lot"` is a `String` .

However, it is legal to store an `int` expression in a `double` variable:

```
int dollars = 2;
double total = dollars;  // ok
```

makes `total` equal to `2.0`

Type Conversion

In Java, you cannot assign a floating-point expression to an `int` variable:

```
double total = 2.54;
int dollars = total;  // error
```

You must convert the floating-point value with a cast :

```
int dollars = (int)total ;
```

The cast `(int)` converts the floating-point value `total` to an `int`.

The effect of the cast is to discard the fractional part. For example, if total is `13.75` then `dollars` is set to `13` .

Type Conversion

If `total` is `13.75` then the cast `int pennies = (int)(total * 100);`

will first evaluate the expression `total * 100` to `1375.0` and then the cast `(int)` will convert the expression to `1375` .

If total is `13.75` then the cast `int pennies = (int)total * 100;`

will first convert `total` `13.75` to an `int` `13` and then multiply by `100` to `1300` .

Type Conversion

A common task: round to the nearest `int`. One way is to add `0.5` and then cast as an `int`. This is illustrated in the code:

```
double price = 44.95;
int dollars = (int)(price + 0.5); // ok for positive values
System.out.print("The price is approximately $");
System.out.println(dollars);          // prints 45
```

Actually, there is a better way. Use the `Math.round` method in the standard Java library. However, it returns a `long integer`. You need to `cast` it as an `int`:

```
int dollars = (int)Math.round(price);
```

Type Conversion

Sometimes rounding errors occur due to the fact that numbers are stored in the CPU as binary numbers. Here is an example:

```
double f = 4.35;
int n = (int)(100 * f);
System.out.println(n);    // prints 434
```

The example should print 435 instead of 434. The reason for this error is that there is no exact binary representation for 4.35 just as there is no exact decimal representation for 1/3. The remedy is to use Math.round:

```
int n = (int)Math.round(100 * f);
```

Strings

A string is a sequence of characters such as `"Hello, World!"` enclosed in quotes " which are not themselves part of the `String`. In Java, unlike numbers, strings are objects. You can tell that `String` is a class name since it begins with an uppercase letter whereas the basic types `int` and `double` begin with a lowercase letter.

The number of characters in the string is called the length of the string. For example, the length of the string `"Hello, World!"` is `13`. You can compute the length of a string with the `length( )` method:

```
int n = message.length( );
```

Strings

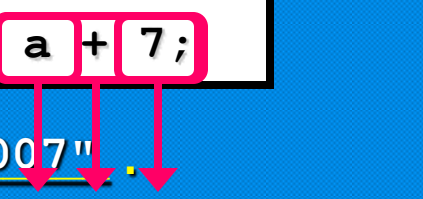
A string of length zero, containing no characters, is called the empty string and is written as "". You are reminded that you can use concatenation to put two or more strings together to form a longer string:

```
String name = "Dave";
String message = "Hello, " + name;
```

The + operator concatenates two strings. If one of the expressions either to the left or to the right of the + is a string then the other is automatically forced to be a string as well.

For example,

```
String a = "Agent 00";
int n = 7;
String bond = a + 7;
```



makes bond equal to "Agent 007".

Strings

This concatenation is very powerful and can be used to make statements such as:

```
System.out.println("The total is " + total);
```

Sometimes you have a string that contains a number, usually from user input. For example, suppose the string variable input has the value "19". To get the value 19, you use the static parseInt method of the Integer class:

```
int count = Integer.parseInt(input);  // count is the integer 19
```

Strings

The `toUpperCase` and `toLowerCase` methods make strings with only upper- or lower- case letters. For example, the code:

```
String greeting = "Hello";
System.out.println(greeting.toUpperCase( ));
System.out.println(greeting.toLowerCase( ));
```

prints `HELLO` and `hello`.

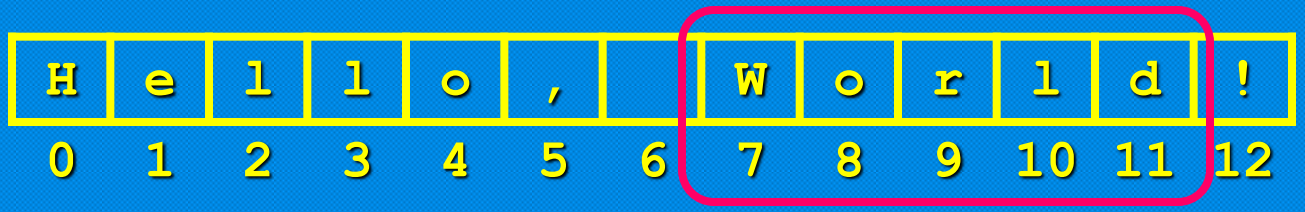
Note that the `toUpperCase` and `toLowerCase` methods don't change the original string `greeting`. They return new `objects` that contain either the uppercase or lowercase versions of the original string. In fact, no string methods modify the `String` object on which they operate. For that reason, strings are called immutable objects.

Strings

The `substring` method computes substrings of a string.

The call `str.substring(start, pastEnd)` returns a string that is made up from the characters in the string `str` starting at character `start` and containing all characters up to, but not including, the character `pastEnd`. Here is an example:

```
String greeting = "Hello, World!";
String sub = greeting.substring(7, 12);
   // sub is "World"
```



Strings

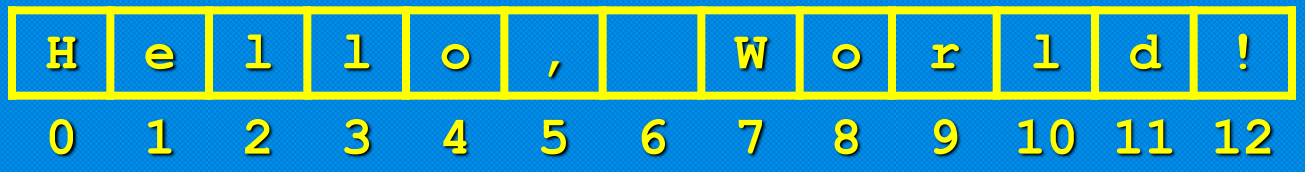
```
String greeting = "Hello, World!";
String sub = greeting.substring(7, 12);
   // sub is "World"
```

Starting position 0 means start at the beginning of the string.

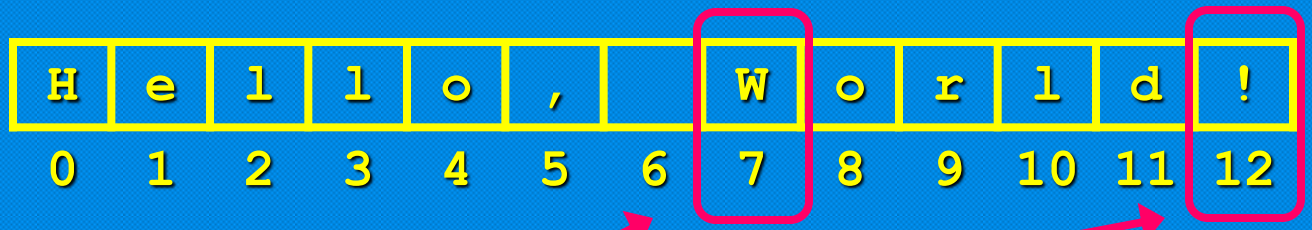
The first string position is numbered 0 and is the character H, the second one 1 and is the character e, and so on.

The position of the last character ! is 12 which is 1 less than the length of the string.

String positions:



Strings



Let us figure out how to extract the substring `"World"`. You find that W is character number 7 and the first character you don't want is ! at 12.

Therefore, the command is:

```
String w = greeting.substring(7, 12);
```

Formatting Numbers

The default format for printing numbers is not always what you would like. For example, consider the following code:

```
double total = 3.50;
final double TAX_RATE = 8.5;              // Tax rate in percent
double tax = total * TAX_RATE / 100;      // tax is 0.2975
System.out.println("Total: " + total);
System.out.println("Tax:   " + tax);
```

The output is:

```
Total: 3.5
Tax:   0.2975
```

You may prefer the numbers to be printed with two digits after the decimal point, like this:

```
Total: 3.50
Tax:   0.30
```

Formatting Numbers

You can achieve this with the `printf` method of the `PrintStream` class. The first parameter of the `printf` method is a format string that shows how output should be formatted. The format string contains:

- characters that are simply printed
- format specifiers. These are codes that start with a % character and end with a letter that indicates the format type. There are quite a few formats. The table shows the most important ones.

Format Types

code	type	example
`d`	integer	`123`
`f`	floating-point	`12.30`
`s`	string	`Tax:`
`n`	newline character	

Formatting Numbers

Format Types		
code	type	example
d	integer	123
f	floating-point	12.30
s	string	Tax:
n	newline character	

The remaining parameters of `printf` are the values to be formatted.

For example:

```
System.out.printf("Total:%5.2f", total);
```

The above prints the string `"Total:"` followed by a floating-point number with width of 5 and a precision of 2. The precision is the number of digits after the decimal point. If the value of `total` is `3.5` and if a space character is shown by the character `x`, then the output of the above statement would be `Total:x3.50`. Notice that the number is right - justified.

Formatting Numbers

If a newline character is desired, then the above example could be modified:

```
System.out.printf("Total:%5.2f%n", total);
```

The table below indicates some of the more important format flags :

Format Flags		
flag	meaning	example
-	left-justification	123 followed by spaces
0	show leading zeros	001.23
+	show a plus sign for positive numbers	+1.23
(	enclose negative numbers in parenthesis	(1.23)
,	show decimal separators	12,300

Formatting Numbers

Format Flags		
flag	meaning	example
-	left-justification	`123 followed by spaces`
0	show leading zeros	`001.23`
+	show a plus sign for positive numbers	`+1.23`
(	enclose negative numbers in parenthesis	`(1.23)`
,	show decimal separators	`12,300`

The following shows a more complicated example:

```
System.out.printf("%-6s%5.2f%n", "Tax:", tax);
```

The third format specifier is `%n` which represents a newline character. So if the variable `tax` has a value of `0.2975` and if the character `x` is used to represent a space, then the above line has output of `Tax:xxx0.30` .

Formatting Numbers

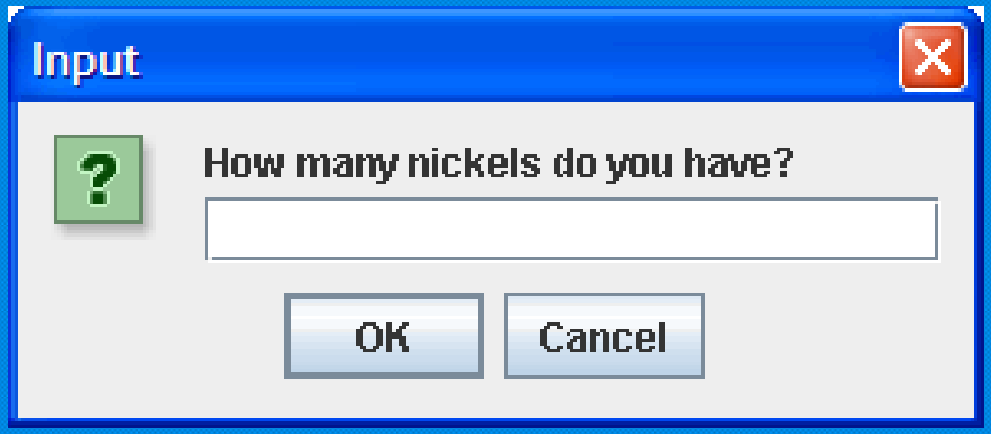
The `format` method of the `String` class is similar to the `printf` method. However, it returns a string instead of producing output. For example, the call

```
String message = String.format("Total:%5.2f", total);
```

sets the message variable to the string `"Total: 3.50"`.

Reading Input using a Dialog Box

In this section, we will learn about reading user input. The JOptionPane class has a static method showInputDialog that displays an input dialog:



The user can type any String into the input field and click the OK button. Then the showInputDialog method returns the String that the user entered.

You should capture this input with a String variable:

```
String input =
   JOptionPane.showInputDialog("How many nickels do you have?");
```

Reading Input using a Dialog Box

Often you want the input as a number, not a `String`. Use the `Integer.parseInt` and `Double.parseDouble` methods to convert the string to a number:

```
int count = Integer.parseInt(input);
```

If the user doesn't enter a number, then the `parseInt` method throws an exception. An exception is a way for a method to indicate an error condition. This will terminate the program with an error message.

Reading Input using a Dialog Box

Finally, when you call `JOptionPane.showInputDialog` in your programs, you need to add a line: `System.exit(0);`

Otherwise, your program will not exit automatically. The number 0 is the error code. A code of 0 indicates successful completion of the program. Nonzero codes indicate an error condition.

An example of a test class that takes user input is shown on your paper.

Reading Console Input

Finally, in Java 5.0, you can read keyboard input in a convenient manner using the `Scanner` class. To construct a `Scanner` object so that you can read from keyboard input, pass the `System.in` object to the `Scanner` constructor:

```
Scanner console = new Scanner(System.in);
```

Once you have a scanner, you use the `nextInt` or `nextDouble` methods to read the next integer or floating-point number.

```
System.out.print("Enter quantity: ");
int quantity = console.nextInt( );

System.out.print("Enter price: ");
double price = console.nextDouble( );
```

Notice that each method call is preceded by a prompt.

Reading Console Input

The `nextLine` method returns the next line of input while the `next` method returns the next word .

```
System.out.print("Enter city: ");
String city = console.nextLine( );

System.out.print("Enter state code: ");
String state = console.next( );
```

Here we use the `nextLine` method to read a city name that may consist of multiple words, such as San Francisco . We use the `next` method to read the state code (such as CA) which consists of a single word.

Characters

Strings are composed of characters . Characters are values of the `char` type. A variable of type `char` can hold a single character.

Character constants look like string constants except that character constants are delimited by single quotes `'` . For example, `"H"` is a string containing a single character. But `'H'` is a character constant.

You can use escape sequences inside character constants. For example, `'\n'` is the newline character and `'\u00E9'` is the character é.

Characters

Characters have numerical values. These values are shown in appendix A5. For example, the character `'H'` is encoded as 72.

The `charAt` method of the `String` class returns a character from a string. As with the `substring` method, the positions in the string are counted starting at 0.

For example, the statement

```
String greeting = "Hello";
char ch = greeting.charAt(0);
```

sets `ch` to the character `'H'`.

Comparing Primitive Types and Objects

In Java, every value is either a _primitive type_ or an _object reference_. Primitive types are numbers (such as `int`, `double`, `char`, and the `boolean` type you will encounter in chapter 5). The table shown on the handout summarizes the primitive types available for use in Java.

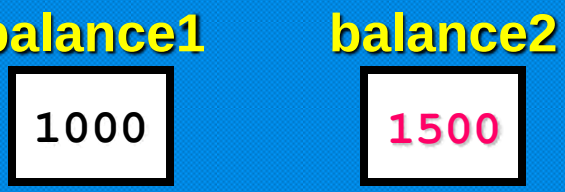
There is an important difference between primitive types and objects in Java. Primitive types hold _values_ but object variables don't hold objects -- they hold _references to objects_.

When you copy a primitive type value, the original and the copy are _independent_ values. But when you copy an object reference, both the original and the copy are references to the _same object_.

Comparing Primitive Types and Objects

Consider the following code using primitive type variables:

```
double balance1 = 1000;
double balance2 = balance1;
balance2 = balance2 + 500;
```

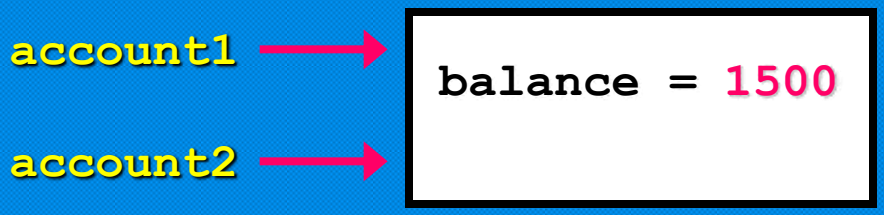


Now the variable `balance1` contains 1000 and `balance2` contains 1500 .

Comparing Primitive Types and Objects

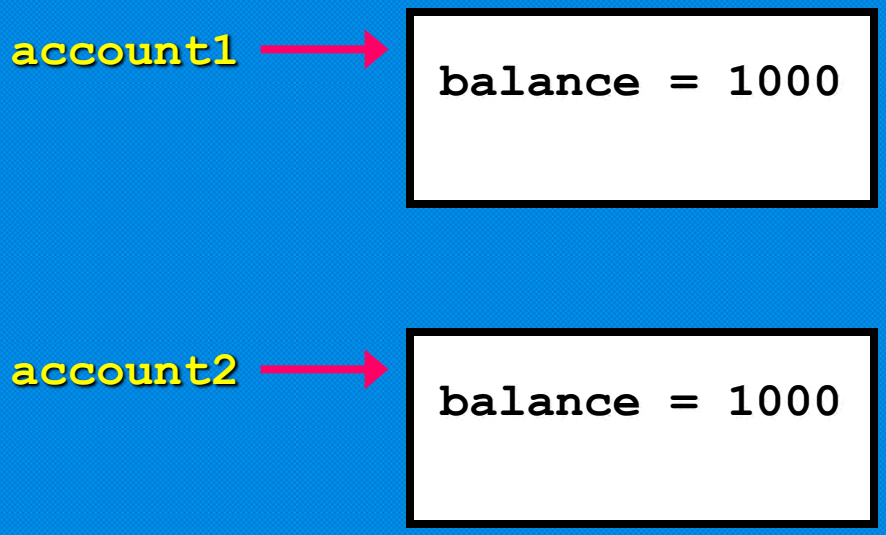
Now consider similar code for `BankAccount` objects:

```
BankAccount account1 = new BankAccount(1000);
BankAccount account2 = account1;
account2.deposit(500);
```



Since both `account1` and `account2` refer to the same object, when `account2` changed to $1500 `account1` also changed to $1500.

Comparing Primitive Types and Objects



If you need to make a copy of an object, you will need to construct a new object:

```
account2 = new BankAccount(account1.getBalance( ));
```