

Chapter 2

Primitive Types and Simple I/O

- Primitive Data types
- Strings: a class
- Assignment
- Expressions
- Keyboard and Screen I/O
- Documentation & Style

What is a program variable?

- A named location to store data
 - » a container for data
- It can hold **only one type** of data
 - » for example only integers, only floating point (real) numbers, or only characters

Creating Variables

- All program variables **must** be *declared* before using them
- A variable declaration associates a name with a storage location in memory and specifies the type of data it will store:

Type Variable_1, Variable_2, ...;

- For example, to create three integer variables to store the number of baskets, number of eggs per basket, and total number of eggs:

```
int numberOfBaskets, eggsPerBasket, totalEggs;
```

Assigning Values to Variables

- The *assignment* operator: “=”
 - » the “equals sign”
- Not the same as in algebra
- It means -
“Assign the value of the expression on the right side to the variable on the left side.”
- Can have the variable on both sides of the equals sign:



```
int count = 10; // initialize counter to ten
count = count - 1; // decrement counter
» new value of count = 10 - 1 = 9
```

Assigning Initial Values to Variables

Initial values *may or may not* be assigned when variables are declared:

```
//These are not initialized when declared
//and have unknown values
int totalEggs, numberOfBaskets, eggsPerBasket;

//These are initialized to 0 when declared
int totalEggs = 0;
int numberOfBaskets = 0;
int eggsPerBasket = 0;
```

- Programming tip: *it is good programming practice always to initialize variables.*

Changing the Value of a Variable

- Usually the value is changed (assigned a different value) somewhere in the program
- May be calculated from other values:

```
totalEggs = numberOfBaskets *
eggsPerBasket;
```

- or read from keyboard input:

```
totalEggs = SavitchIn.readLineInt();
```

Variable Names: *Identifiers*

Rules

- these **must** be obeyed
- *all* Java identifiers must follow the same rules
- must **not start with a digit**
- must contain only numbers, letters, underscore (_) and \$ (but avoid using \$, it is reserved for special purposes)
- names are **case-sensitive** (ThisName and thisName are two different variable names)

Good Programming Practice

- these **should** be obeyed
- always use meaningful names from the problem domain (for example, `eggsPerBasket` instead of `n`, which is meaningless, or `count`, which is not meaningful enough)
- start variable names with lower case
- capitalize interior words (use `eggsPerBasket` instead of `eggsperbasket`)
- avoid using \$ since it is reserved for special purposes

Two Main Kinds of *Types* in Java

primitive data types

- the simplest types
- cannot decompose into other types
- **values only**, no methods
- Examples:
 - `int` - integer
 - `double` - floating point (real)
 - `char` - character

class types

- more complex
- composed of other types (primitive or class types)
- both **data** and **methods**
- Examples:
 - `SavitchIn`
 - `String`

Primitive Data Types

Type Name	Kind of Value	Memory Used	Size Range
byte	integer	1 byte	-128 to 127
short	integer	2 bytes	-32768 to 32767
int	integer	4 bytes	-2,147,483,648 to 2,147,483,647
long	integer	8 bytes	-9,223,372,036,854,775,808 to 9,223,374,036,854,775,808
float	floating point	4 bytes	+/- 3.4028... x 10 ⁺³⁸ to +/- 1.4023... x 10 ⁻⁴⁵
double	floating point	8 bytes	+/- 1.767... x 10 ⁺³⁰⁸ to +/- 4.940... x 10 ⁻³²⁴
char	single character (Unicode)	2 bytes	all Unicode characters
boolean	<i>true</i> or <i>false</i>	1 bit	not applicable

Which Ones to Know for Now

Display in text is for reference; for now stick to these simple primitive types:

- **int**
 - » just whole numbers
 - » may be positive or negative
 - » no decimal point
- **char**
 - » just a single character
 - » uses ***single*** quotes
 - » for example

```
char letterGrade = 'A';
```
- **double**
 - » real numbers, both positive and negative
 - » has a decimal point (fractional part)
 - » two formats
 - number with decimal point, e.g. 514.061
 - e (or *scientific*, or *floating-point*) notation, e.g. 5.14061e2, which means 5.14061 x 10²

Specialized Assignment Operators

- A shorthand notation for performing an operation on and assigning a new value to a variable
- General form: `var <op>= expression;`
 - » equivalent to: `var = var <op> (expression);`
 - » <op> is **+**, **-**, *****, **/**, or **%**
- Examples:

```
amount += 5;
//amount = amount + 5;

amount *= 1 + interestRate;
//amount = amount * (1 + interestRate);
```
- Note that the **right side is treated as a unit** (put parentheses around the entire expression)

Returned Value

- Expressions *return* values: the number produced by an expression is “returned”, i.e. it is the “return value.”

```
int numberOfBaskets, eggsPerBasket, totalEggs;
numberOfBaskets = 5;
eggsPerBasket = 8;
totalEggs = numberOfBaskets * eggsPerBasket;
```

 - » in the last line `numberOfBaskets` returns the value 5 and `eggsPerBasket` returns the value 8
 - » `numberOfBaskets * eggsPerBasket` is an expression that returns the integer value 40
- Similarly, methods return values
`SavitchIn.readLine();` is a method that returns a string read from the keyboard

Casting: changing the data type of the *returned* value

- **Cast** means to change the data type of a value
- You cannot put a value of one type in a variable of a different type unless you convert it to match the type of the variable
- Casting only changes the type of the *returned value* (the single instance where the cast is done), not the type of the variable
- For example:

```
double x;  
int n = 5;  
x = n;
```
- Since `n` is an integer and `x` is a double, the value returned by `n` must be converted to type double before it is assigned to `x`

Implicit Casting

- Casting is done **implicitly** (automatically) when a “lower” type is assigned to a “higher” type
- The data type hierarchy (from lowest to highest):
byte --> short --> int --> long --> float --> double
- For example:

```
double x;  
int n = 5;  
x = n;
```

 - » the value returned by `n` is *cast* to a double, then assigned to `x`
 - » `x` contains 5.000... (as accurately as it can be encoded as a floating point number)
 - » This is called *implicit* casting because it is done automatically
 - » The data type of the variable `n` is unchanged; is still an int

Data Types in an Expression: More Implicit Casting

- Some expressions have a mix of data types
- All values are automatically advanced (implicitly cast) to the highest level before the calculation
- For example:

```
double a;  
int n = 2;  
float x = 5.1;  
double y = 1.33;  
a = (n * x) / y;
```

 - » `n` and `x` are automatically cast to type `double` before performing the multiplication and division

Explicit Casting

- Explicit casting changes the data type of the value for a single use of the variable
- Precede the variable name with the new data type in parentheses:

```
(<data type>) variableName
```

 - » the type is changed to `<data type>` only for the single use of the returned value where it is cast
- For example:

```
int n;  
double x = 2.0;  
n = (int)x
```

 - » the value of `x` is converted from `double` to `integer` before assigning the value to `n`

Explicit casting is required to assign a higher type to a lower

- **ILLEGAL:** *Implicit* casting to a *lower* data type

```
int n;  
double x = 2.1;  
n = x; //illegal in java
```

It is illegal since `x` is double, `n` is an int, and double is a higher data type than integer

- **LEGAL:** *Explicit* casting to a lower data type

```
int n;  
double x = 2.1;  
n = (int)x; //legal in java
```

- You can always use an explicit cast where an implicit one will be done automatically, but it is not necessary

Truncation When Casting a **double** to an Integer

- Converting (casting) a double to integer does not round; it *truncates*
 - » the fractional part is lost (discarded, ignored, thrown away)

- For example:

```
int n;  
double x = 2.99999;  
n = (int)x; //cast is required, x is truncated  
» the value of n is now 2
```

- This behavior is useful for some calculations, as demonstrated in *Case Study: Vending Machine Change*

Truncation When Doing Integer Division

- No truncation occurs if at least one of the values in a division is type float or double (all values are promoted to the highest data type)
- Truncation occurs if **all** the values in a division are integers
- For example:

```
int a = 4, b =5, c;  
double x = 1.5, y;  
y = b/x;//value returned by b is cast to double  
        //value of y is approximately 3.33333  
c = b/a;//all values are ints so the division  
        //truncates: the value of c is 1!
```

The **char** Data Type

- The `char` data type stores a single “printable” character
- For example:

```
char answer = `y`;  
System.out.println(answer);  
» prints (displays) the letter y
```

Characters as Integers

- Characters are actually stored as integers according to a special code
 - » each printable character (letter, number, punctuation mark, space, and tab) is assigned a different integer code
 - » the codes are different for upper and lower case
 - » for example 97 may be the integer value for 'a' and 65 for 'A'
- ASCII (Appendix 3) and Unicode are common character codes
- Unicode includes all the ASCII codes plus additional ones for languages with an alphabet other than English
- Java uses Unicode

Casting a **char** to an **int**

- Casting a char value to int produces the ASCII/Unicode value
- For example, what would the following display?

```
char answer = `y`;
System.out.println(answer);
System.out.println((int)answer);
```
- Answer: the letter 'y' on one line followed by the ASCII code for 'y' (lower case) on the next line:

```
>y
>89
>
```

GOTCHA: Imprecision of Floating Point Numbers

- Computers store numbers using a fixed number of bits, so not every real (floating point) number can be encoded precisely
 - » an infinite number of bits would be required to precisely represent any real number
- For example, if a computer can represent up to 10 decimal digits, the number 2.5 may be stored as 2.499999999 if that is the closest it can come to 2.5
- Integers, on the other hand, are encoded precisely
 - » if the value 2 is assigned to an int variable, its value is precisely 2
- This is important in programming situations you will see later in the course

The Modulo Operator: **a%b**

- Used with integer types
- Returns the remainder of the division of b by a
- For example:

```
int a = 57; b = 16, c;  
c = a%b;
```

 - » c now has the value 9, the remainder when 57 is divided by 16
- A very useful operation: see *Case Study: Vending Machine Change*

Vending Machine Change

Excerpt from the `ChangeMaker.java` program:

```
int amount, originalAmount,
    quarters, dimes, nickels, pennies;
... // code that gets amount from user not shown
originalAmount = amount;
quarters = amount/25;
amount = amount%25;
dimes = amount/10;
amount = amount%10;
nickels = amount/5;
amount = amount%5;
pennies = amount;
```

If amount is 90 then there 90/25 will be 3, so there are three quarters.

If amount is 90 then the remainder of 90/25 will be 15, so 15 cents change is made up of other coins.

Arithmetic Operator Precedence and Parentheses

- Java expressions follow rules similar to real-number algebra
- Use parentheses to force precedence
- Do not clutter expressions with parentheses when the precedence is correct and obvious

Examples of Expressions

Ordinary Math Expression	Java Expression (preferred form)	Java Fully Parenthesized Expression
$\text{rate}^2 + \text{delta}$	<code>rate*rate + delta</code>	<code>(rate*rate) + delta</code>
$2(\text{salary} + \text{bonus})$	<code>2 * (salary + bonus)</code>	<code>2 * (salary + bonus)</code>
$\frac{1}{\text{time} + 3\text{mass}}$	<code>1/(time + 3 * mass)</code>	<code>1/(time + (3 * mass))</code>
$\frac{a - 7}{t + 9v}$	<code>(a - 7) / (t + 9 * v)</code>	<code>(a - 7) / (t +(9 * v))</code>

Increment and Decrement Operators

- Shorthand notation for common arithmetic operations on variables used for counting
- Some counters count up, some count down, but they are integer variables
- The counter can be incremented (or decremented) before or after using its current value

```
int count;
```

```
...
```

```
++count preincrement count: count = count + 1 before using it
```

```
count++ postincrement count: count = count + 1 after using it
```

```
--count predecrement count: count = count -1 before using it
```

```
count-- postdecrement count: count = count -1 after using it
```

Increment and Decrement Operator Examples

common code

```
int n = 3;  
int m = 4;  
int result;
```

What will be the value of `m` and `result` after each of these executes?

- (a) `result = n * ++m; //preincrement m`
- (b) `result = n * m++; //postincrement m`
- (c) `result = n * --m; //predecrement m`
- (d) `result = n * m--; //postdecrement m`

Answers to Increment/Decrement Operator Questions

- (a) 1) `m = m + 1; //m = 4 + 1 = 5`
2) `result = n * m; //result = 3 * 5 = 15`
- (b) 1) `result = n * m; //result = 3 * 4 = 12`
2) `m = m + 1; //m = 4 + 1 = 5`
- (c) 1) `m = m - 1; //m = 4 - 1 = 3`
2) `result = n * m; //result = 3 * 3 = 9`
- (d) 1) `result = n * m; //result = 3 * 4 = 12`
2) `m = m - 1; //m = 4 - 1 = 3`

The *String* Class

- A string is a sequence of characters
- The **String** class is used to store strings
- The String class has methods to operate on strings
- String constant: one or more characters in *double* quotes
- Examples:

```
char charVariable = `a`; //single quotes
String stringVariable = "a"; //double quotes
String sentence = "Hello, world";
```

String Variables

- Declare a String variable:
`String greeting;`
- Assign a value to the variable
`greeting = "Hello!";`
- Use the variable as a String argument in a method:
`System.out.println(greeting);`
 - » causes the string `Hello!` to be displayed on the screen

Indexing Characters within a String

- The index of a character within a string is an integer starting at **0** for the first character and gives the position of the character
- The `charAt(Position)` method returns the char at the specified position
- `substring(Start, End)` method returns the string from position *Start* to position *End*
- For example:

```
String greeting = "Hi, there!";  
greeting.charAt(0) returns H  
greeting.charAt(2) returns ,  
greeting.substring(4,6) returns the
```

H	i	,		t	h	e	r	e	!
0	1	2	3	4	5	6	7	8	9

Concatenating (Appending) Strings

Stringing together strings - the "+" operator for Strings:

```
String name = "Mondo";  
String greeting = "Hi, there!";  
System.out.println(greeting + name + "Welcome");
```

causes the following to display on the screen:

```
>Hi, there!MondoWelcome
```

```
>
```

» Note that you have to remember to include spaces if you want it to look right:

```
System.out.println(greeting + " " + name  
                    + " Welcome");
```

» causes the following to display on the screen:

```
>Hi, there! Mondo Welcome
```

```
>
```

Escape Characters

- How do you print characters that have special meaning?
- For example, how do you print the following string?
The word "hard"
- Would this do it?

```
System.out.println("The word "hard");
```

 - » No, it would give a **compiler error** - it sees the string The word between the first set of double quotes and is confused by what comes after
- **Use the backslash character**, "\", to escape the special meaning of the internal double quotes:

```
System.out.println("The word \"hard\"); //this works
```

More Escape Characters

Use the following escape characters to include the character listed in a quoted string:

- \ " Double quote.
- \ ' Single quote.
- \\ Backslash.
- \n New line. Go to the beginning of the next line.
- \r carriage return. Go to the beginning of the current line.
- \t Tab. White space up to the next tab stop.

One more thing about screen output: how to get or avoid a new line

- Sometimes you want to print part of a line and not go to the next line when you print again

- Two methods, one that goes to a new line and one that does not

```
System.out.println(...); //ends with a new line
```

```
System.out.print(...); //stays on the same line
```

- For example:

```
System.out.print("This will all ");
```

```
System.out.println("appear on one line");
```

- System.out.print() works similar to the "+" operator:

```
System.out.println("This will all "
```

```
    + "appear on one line, too");
```

Program I/O

- I/O - Input/Output
- Keyboard is the normal input device
- Screen is the normal output device
- Classes are used for I/O
- They are generally add-on classes (not actually part of Java)
- Some I/O classes are always provided with Java, others are not

I/O Classes

- We have been using an output method from a class that automatically comes with Java:
`System.out.println()`
- But **Java does not automatically have an input class**, so one must be added
 - » ***SavitchIn* is a class specially written to do keyboard input**
- `SavitchIn.java` is provided with the text - see Appendix 2
- Examples of `SavitchIn` methods for keyboard input:
`readLineInt()`
`readLineDouble()`
`readLineNonwhiteChar()`
- Gotcha: remember Java is case sensitive, for example `readLineNonWhiteChar()` will not work

Input Example from Vending Machine Change Program

Excerpt from the `ChangeMaker.java` program:

```
int amount, originalAmount,  
        quarters, dimes, nickels pennies;  
System.out.println("Enter a whole number...");  
System.out.println("I will output ... coins");  
System.out.println("that equals that amount ...");
```

```
amount = SavitchIn.readLineInt();  
originalAmount = amount;
```

Prompts so that user knows what they need to type.

Lets the user type in an integer and stores the number in amount.

Keyboard Input Gotchas

Note the two variations for reading each type of number

readLine variation

- reads a whole line
- asks the user to reenter if it is not the right format
- Try to use these
- Examples:
`readLineInt()`
`readLineDouble()`

read variation

- reads just the number
- aborts the program if it is not the right format
- Avoid using these
- Examples:
`readInt()`
`readDouble()`

A little practical matter: If the screen goes away too quickly ...

If the output (screen display) of your programs does not stay on the screen, use this code:

```
System.out.println("Press any key to end program.")  
String junk;  
junk = SavitchIn.readLine();
```

- The display stops until you enter something
- Whatever you enter is stored in variable `junk` but is never used
- it is "thrown away"

Documentation and Style

- Use **meaningful names** for variables, classes, etc.
- Use **indentation** and **line spacing** as shown in the examples in the text
- Always include a “prologue” (an brief explanation of the program at the beginning of the file)
- Use all **lower case** for variables, except capitalize internal words (`eggsPerBasket`)
- Use all **upper case** for variables that have a constant value, `PI` for the value of pi (3.14159...) (see text for more examples)

Named Constants

- *Named constant*—using a name instead of a value
- Example: use `MORTGAGE_INTEREST_RATE` instead of `8.5`
- Advantages of using named constants
 - » Easier to understand program because reader can tell how the value is being used
 - » Easier to modify program because value can be changed in one place (the definition) instead of being changed everywhere in the program.
 - » Avoids mistake of changing same value used for a different purpose

Defining Named Constants

```
public static final double PI = 3.14159;
```

public—no restrictions on where this name can be used

static—must be included, but explanation has to wait

final—the program is not allowed to change the value

- The remainder of the definition is similar to a variable declaration and gives the type, name, and initial value.
- A declaration like this is usually at the beginning of the file and **is not inside the main method** definition.

Comments

- *Comment*—text in a program that the compiler ignores
- Does not change what the program does, only explains the program
- Write meaningful and useful comments
- Comment the *non-obvious*
- Assume a *reasonably* knowledgeable reader
- `//` for single-line comments
- `/* ... */` for multi-line comments

Summary

Part 1

- Variables hold values and have a *type*
 - » The type of a Java variable is either a primitive type or a class
 - » Common primitive types in Java include `int`, `double`, and `char`
 - » A common class type in Java is `String`
 - » Variables must be declared
- Parentheses in arithmetic expressions ensure correct execution order
- Use `SavitchIn` methods for keyboard input
 - » `SavitchIn` is not part of standard Java

Summary

Part 2

- Good programming practice:
 - » Use meaningful names for variables
 - » Initialize variables
 - » Use variable names (in upper case) for constants
 - » Use comments sparingly but wisely, e.g. to explain non-obvious code
 - » Output a prompt when the user is expected to enter data from the keyboard
 - » Echo the data entered by the user