Chapter 2:
Java Fundamentals

Chapter Topics

Chapter 2 discusses the following main topics:

- The Parts of a Java Program
- The `print` and `println` Methods
- The Java API
- Variables and Literals
- Primitive Data Types
- Arithmetic Operators
- Combined Assignment Operators

Chapter Topics (2)

- Creating named constants with final
- The `String` class
- Scope
- Comments
- Programming style
- Using the `Scanner` class for input
- Dialog boxes

Parts of a Java Program

- A Java source code file contains one or more Java classes.

- If more than one class is in a source code file, only one of them may be public.

- The public class and the filename of the source code file must match.

  ex: A class named *Simple* must be in a file named *Simple.java*

- Each Java class can be separated into parts.

Parts of a Java Program

- See example: <u>Simple.java</u>
- To compile the example:
  - **`javac Simple.java`**
    - Notice the `.java` file extension is needed.
    - This will result in a file named *Simple.class* being created.
- To run the example:
  - **`java Simple`**
    - Notice there is no file extension here.
    - The *java* command assumes the extension is `.class`.

Analyzing The Example

// This is a simple Java program. ← **This is a Java comment. It is ignored by the compiler. It is part of the program header**

public class Simple ← **This is the class definition for the class Simple**
{

This area is the body of the class Simple. All of the data and methods for this class will be between these curly braces.

}

Analyzing The Example

// This is a simple Java program.

public class Simple
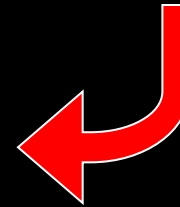{

    public static void main(String[] args)
    {

    }

}

This is the method definition for the main method. The main method is where a Java application begins.
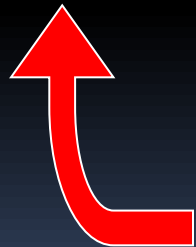
This area is the body of the main method. All of the actions to be completed during the main method will be between these curly braces.

Analyzing The Example

```java
// This is a simple Java program.

public class Simple
{
    public static void main(String [] args)
    {
        System.out.println("Programming is great fun!");
    }
}
```

This is the Java Statement that is executed when the program runs.

Parts of a Java Program

- Comments
  - The line is ignored by the compiler.
  - The comment in the example is a single-line comment.
- Class Declaration
  - The class declaration tells the compiler things about the class such as what other classes can use it (public) and that it is a Java class (class), and the name of that class (Simple).
- Curly Braces
  - When associated with the class , they define the scope of the class.
  - When associated with a method, they define the scope of the method.

Parts of a Java Program

- The `main` Method
  - This line must be exactly as shown in the example (except the *args* variable name can be programmer defined).
  - This is the line of code that the *java* command will run first.
  - This method starts the Java program.
  - Every Java *application* must have exactly one `main` method.
- Java Statements
  - When the program runs, the statements within the `main` method will be executed.
  - Can you see what the line in the example will do?

Java Statements

- If we look back at the previous example, we can see that there is only one line that ends with a semi-colon.

  ```
  System.out.println("Programming is great fun!");
  ```

- This is because it is the only Java statement in the program.

- The rest of the code is either a comment or other Java framework code.

Java Statements

- Comments are ignored by the Java compiler so they need no semi-colons.

- Other Java code elements that do not need semi colons include:

  - class headers
    - Terminated by the code within its curly braces.

  - method headers
    - Terminated by the code within its curly braces.

  - curly braces
    - Part of framework code that needs no semi-colon termination.

Short Review

- Java is a case-sensitive language.

- All Java programs must be stored in a file with a .java file extension.

- Comments are ignored by the compiler.

- A .java file may contain many classes but may only have one public class.

- If a .java file has a public class, the class must have the same name as the file.

Short Review

- Java applications must have one `main` method.

- For every left brace, or opening brace, there must be a corresponding right brace, or closing brace.

- Statements are terminated with semicolons.
  - Comments, class headers, method headers, and braces are not considered Java statements.

Special Characters

//	double slash	Marks the beginning of a single line comment.
( )	open and close parenthesis	Used in a method header to mark the *parameter list*.
{ }	open and close curly braces	Encloses a group of statements, such as the contents of a class or a method.
" "	quotation marks	Encloses a string of characters, such as a message that is to be printed on the screen
;	semi-colon	Marks the end of a complete programming statement

Console Output

- Many of the programs that you will write will run in a terminal or console window.
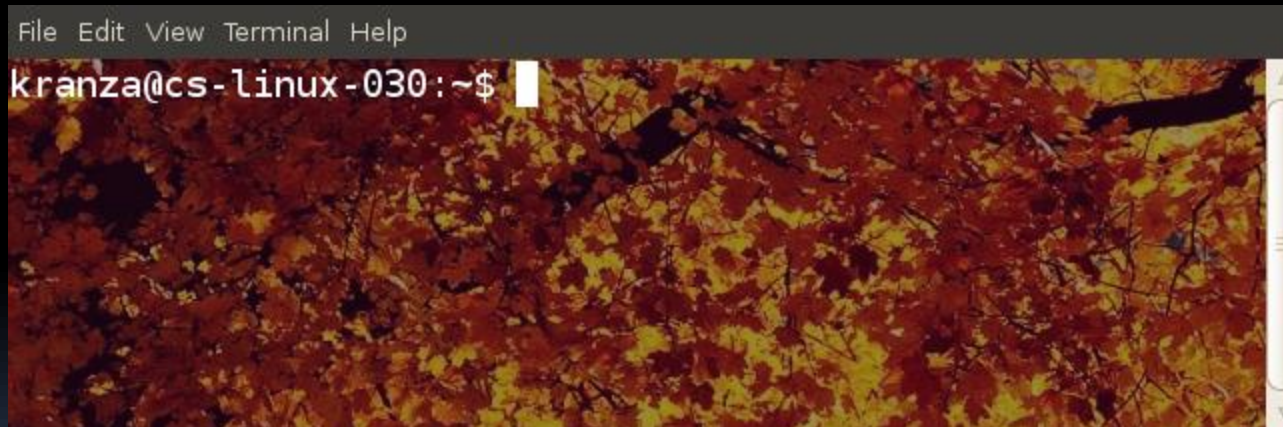
Console Output

- The console window that starts a Java application is typically known as the *standard output* device.

- The *standard input* device is typically the keyboard.

- Java sends information to the standard output device by using a Java class stored in the standard Java library.

Console Output

- Java classes in the standard Java library are accessed using the Java Applications Programming Interface (API).

- The standard Java library is commonly referred to as the *Java API*.

Console Output

- The previous example uses the line:

  `System.out.println("Programming is great fun!");`

- This line uses the `System` class from the standard Java library.

- The `System` class contains methods and objects that perform system level tasks.

- The `out` object, a member of the `System` class, contains the methods `print` and `println`.

Console Output

- The `print` and `println` methods actually perform the task of sending characters to the output device.

- The line:

  ```
  System.out.println("Programming is great fun!");
  ```

  is pronounced: System dot out dot println

  ...

- The value inside the parenthesis will be sent to the output device (in this case, a string).

Console Output

- The `println` method places a newline character at the end of whatever is being printed out.

- The following lines:

  ```
  System.out.println("This is being printed out");
  System.out.println("on two separate lines.");
  ```

  Would be printed out on separate lines since the first statement sends a newline command to the screen.

Console Output

- The `print` statement works very similarly to the `println` statement.

- However, the `print` statement does not put a newline character at the end of the output.

- The lines:

```
System.out.print("These lines will be");
System.out.print("printed on");
System.out.println("the same line.");
```

Will output:

```
These lines will beprinted onthe same line.
```

Notice the odd spacing? Why are some words run together?

Console Output

- For all of the previous examples, we have been printing out strings of characters.

- Later, we will see that much more can be printed.

- There are some special characters that can be put into the output.
  ```
  System.out.print("This line will have a newline at the end.\n");
  ```

- The `\n` in the string is an escape sequence that represents the newline character.

- Escape sequences allow the programmer to print characters that otherwise would be unprintable.

Java Escape Sequences

`\n`	newline	Advances the cursor to the next line for subsequent printing
`\t`	tab	Causes the cursor to skip over to the next tab stop
`\b`	backspace	Causes the cursor to back up, or move left, one position
`\r`	carriage return	Causes the cursor to go to the beginning of the current line, not the next line
`\\`	backslash	Causes a backslash to be printed
`\'`	single quote	Causes a single quotation mark to be printed
`\"`	double quote	Causes a double quotation mark to be printed

Java Escape Sequences

- Even though the escape sequences are comprised of two characters, they are treated by the compiler as a single character.

```
System.out.print("These are our top sellers:\n");
System.out.print("\tComputer games\n\tCoffee\n ");
System.out.println("\tAspirin");
```

Would result in the following output:

```
These are our top seller:
        Computer games
        Coffee
        Asprin
```

- With these escape sequences, complex text output can be achieved.

Variables and Literals

- A variable is a named storage location in the computer's memory.

- A literal is a value that is written into the code of a program.

- Programmers determine the number and type of variables a program will need.

- See example:`Variable.java`

Variables and Literals

This line is called a *variable declaration.*
```
int value;
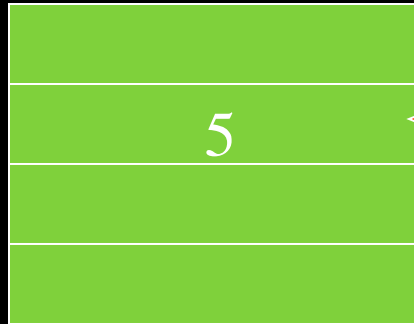```

The following line is known as an assignment statement.
```
value = 5;
```

0x000

0x001    5

0x002

0x003

The value 5 is stored in memory.

This is a string *literal*. It will be printed as is.

```
System.out.print("The value is ");
System.out.println(value);
```

The integer 5 will be printed out here.
Notice no quote marks?

The + Operator

- The + operator can be used in two ways.
  - as a concatenation operator
  - as an addition operator
- If either side of the + operator is a string, the result will be a string.

```
System.out.println("Hello " + "World");
System.out.println("The value is: " + 5);
System.out.println("The value is: " + value);
System.out.println("The value is: " + '/n' + 5);
```

String Concatenation

- Java commands that have string literals must be treated with care.

- A string literal value cannot span lines in a Java source code file.

```
System.out.println("This line is too long and now it
has spanned more than one line, which will cause a
syntax error to be generated by the compiler. ");
```

String Concatenation

- The String concatenation operator can be used to fix this problem.

```
System.out.println("These lines are " +
                    "are now ok and will not " +
                    "cause the error as before.");
```

- String concatenation can join various data types.

```
System.out.println("We can join a string to " +
                    "a number like this: " + 5);
```

String Concatenation

- The Concatenation operator can be used to format complex String objects.

```
System.out.println("The following will be printed " +
                   "in a tabbed format: " +
                   \n\tFirst = " + 5 * 6 + ", " +
                   "\n\tSecond = " (6 + 4) + "," +
                   "\n\tThird = " + 16.7 + ".");
```

- Notice that if an addition operation is also needed, it must be put in parenthesis.

Identifiers

- Identifiers are programmer-defined names for:
  - classes
  - variables
  - methods
- Identifiers may not be any of the Java reserved keywords.

Identifiers

- Identifiers must follow certain rules:
  - An identifier may only contain:
    - letters a–z or A–Z,
    - the digits 0–9,
    - underscores (_), or
    - the dollar sign ($)
  - The first character may not be a digit.
  - Identifiers are case sensitive.
    - `itemsOrdered` is not the same as `itemsordered`.
  - Identifiers cannot include spaces.

Java Reserved Keywords

The following character sequences, formed from ASCII letters, are reserved for use as keywords and cannot be used as identifiers:

abstract	continue	for	new	switch
assert	default	if	package	synchronized
boolean	do	goto	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

The keywords `const` and `goto` are reserved, even though they are not currently used. This may allow a Java compiler to produce better error messages if these C++ keywords incorrectly appear in programs.

While **true** and **false** might appear to be keywords, they are technically Boolean literals. Similarly, while null might appear to be a keyword, it is technically the null literal.

Variable Names

- Variable names should be descriptive.

- Descriptive names allow the code to be more readable; therefore, the code is more maintainable.

- Which of the following is more descriptive?

  ```
  double tr = 0.0725;

  double salesTaxRate = 0.0725;
  ```

- Java programs should try be *self-documenting*.

Java Naming Conventions

- Variable names should begin with a lower case letter and then switch to title case thereafter:

    Ex: `int` `caTaxRate`

- Class names should be all title case.

    Ex: `public class` `BigLittle`

- More Java naming conventions can be found at:

    http://java.sun.com/docs/codeconv/html/CodeConventions.doc8.html

- A general rule of thumb about naming variables and classes are that, with some exceptions, their names tend to be nouns or noun phrases.

Primitive Data Types

- Primitive data types are built into the Java language and are not derived from classes.

- There are 8 Java primitive data types.

  - byte
  - short
  - int
  - long

  - float
  - double
  - boolean
  - char

Numeric Data Types

byte	1 byte	Integers in the range -128 to +127
short	2 bytes	Integers in the range of -32,768 to +32,767
int	4 bytes	Integers in the range of -2,147,483,648 to +2,147,483,647
long	8 bytes	Integers in the range of -9,223,372,036,854,775,808 to +9,223,372,036,854,775,807
float	4 bytes	Floating-point numbers in the range of $\pm3.4\times10^{-38}$ to $\pm3.4\times10^{+38}$, with 7 digits of accuracy
double	8 bytes	Floating-point numbers in the range of $\pm1.7\times10^{-308}$ to $\pm1.7\times10^{+308}$, with 15 digits of accuracy

Variable Declarations

- Variable Declarations take the following form:
  - *DataType VariableName;*
    - `byte inches;`
    - `short month;`
    - `int speed;`
    - `long timeStamp;`
    - `float salesCommission;`
    - `double distance;`

Integer Data Types

- `byte`, `short`, `int`, and `long` are all integer data types.
- They can hold whole numbers such as 5, 10, 23, 89, etc.
- Integer data types cannot hold numbers that have a decimal point in them.
- Integers embedded into Java source code are called *integer literals*.
- See Example: IntegerVariables.java

Floating Point Data Types

- Data types that allow fractional values are called *floating-point* numbers.

  - 1.7 and -45.316 are floating-point numbers.

- In Java there are two data types that can represent floating-point numbers.

  - `float` - also called *single precision* (7 decimal points).

  - `double` - also called *double precision* (15 decimal points).

Floating Point Literals

- When floating point numbers are embedded into Java source code they are called *floating point literals*.

- The default type for floating point literals is `double.`
  - 29.75, 1.76, and 31.51 are `double` data types.

- Java is a *strongly-typed* language.

- See example: <u>Sale.java</u>

Floating Point Literals

- A `double` value is not compatible with a `float` variable because of its size and precision.
  - `float number;`
  - `number = 23.5; // Error!`
- A `double` can be forced into a `float` by appending the letter F or f to the literal.
  - `float number;`
  - `number = 23.5F; // This will work.`

Floating Point Literals

- Literals cannot contain embedded currency symbols or commas.

  - `grossPay = $1,257.00; // ERROR!`
  - `grossPay = 1257.00;    // Correct.`

- Floating-point literals can be represented in *scientific notation*.

  - $47{,}281.97 == 4.728197 \times 10^4$.

- Java uses *E notation* to represent values in scientific notation.

  - $4.728197 \times 10^4 == 4.728197\text{E}4$.

Scientific and E Notation

Decimal Notation	Scientific Notation	E Notation
247.91	2.4791×10^2	2.4791E2
0.00072	7.2×10^{-4}	7.2E-4
2,900,000	2.9×10^6	2.9E6

See example: SunFacts.java

The `boolean` Data Type

- The Java `boolean` data type can have two possible values.
  - `true`
  - `false`
- The value of a `boolean` variable may only be copied into a `boolean` variable.

See example: TrueFalse.java

The `char` Data Type

- The Java `char` data type provides access to single characters.

- `char` literals are enclosed in single quote marks.
  - 'a', 'Z', '\n', '1'

- Don't confuse `char` literals with string literals.
  - `char` literals are enclosed in single quotes.
  - String literals are enclosed in double quotes.

See example: Letters.java

Unicode

- Internally, characters are stored as numbers.

- Character data in Java is stored as Unicode characters.

- The Unicode character set can consist of 65536 (2^{16}) individual characters.

- This means that each character takes up 2 bytes in memory.

- The first 256 characters in the Unicode character set are compatible with the ASCII* character set.
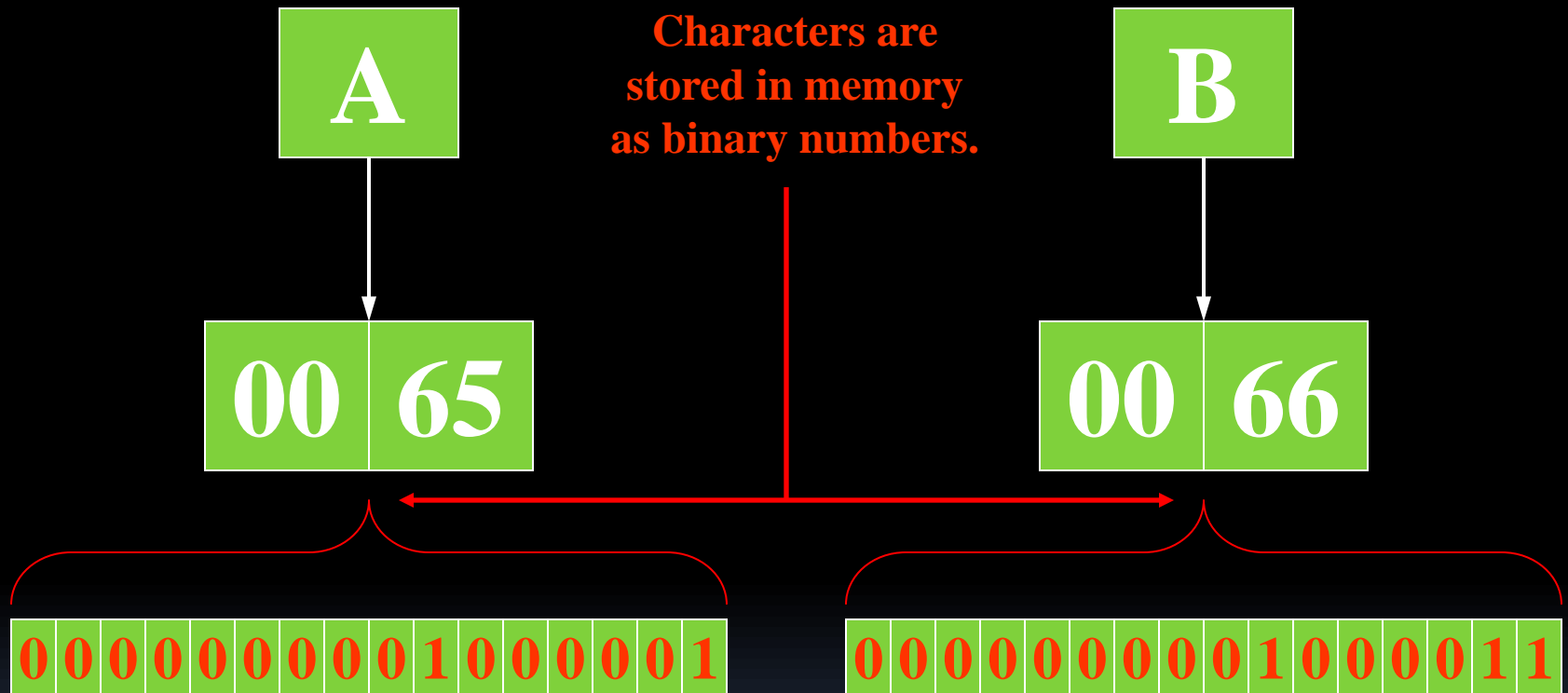
See example: Letters2.java
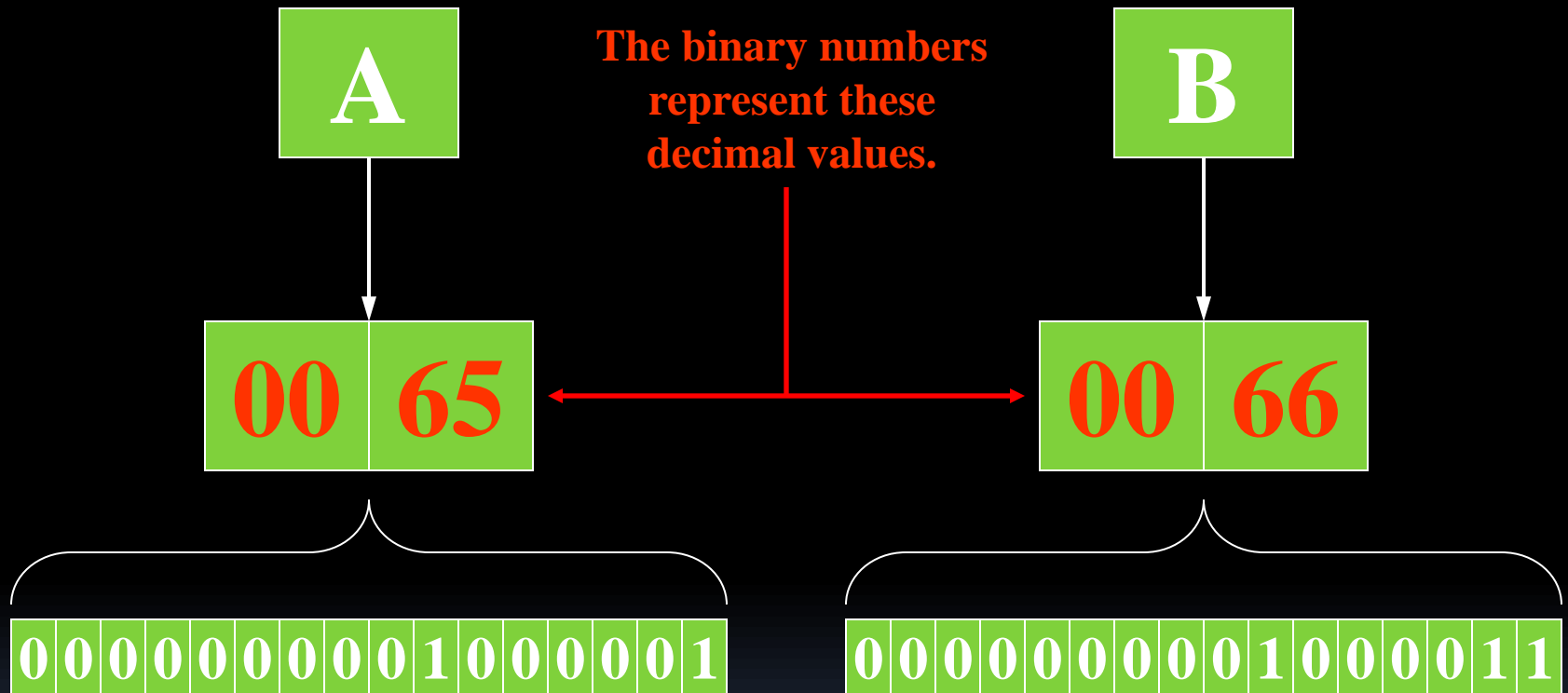
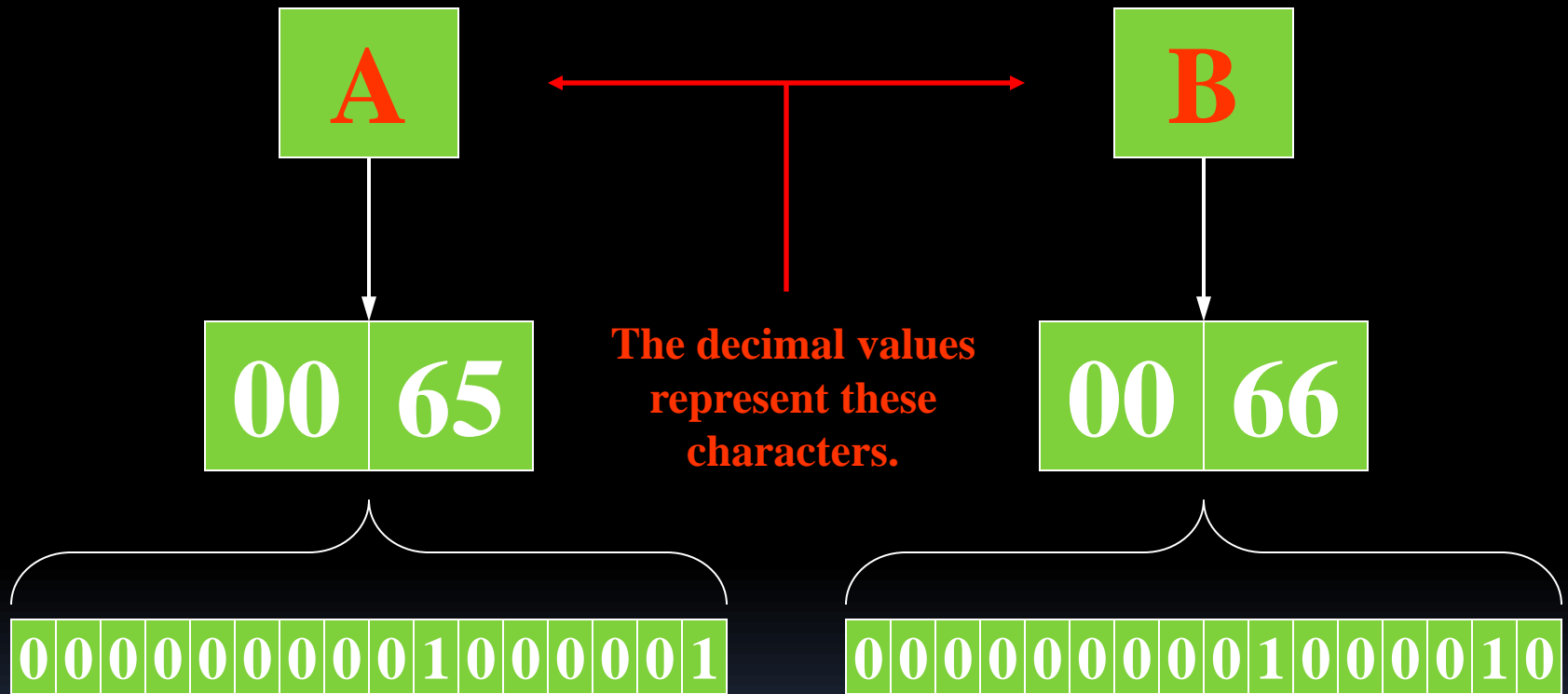*American Standard Code for Information Interchange

Unicode

Unicode

A

Characters are stored in memory as binary numbers.

B

| 00 | 65 |

| 00 | 66 |

| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |

| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 |

Unicode

A

The binary numbers represent these decimal values.

B

| 00 | 65 | ← → | 00 | 66 |

000000001000001

000000001000011
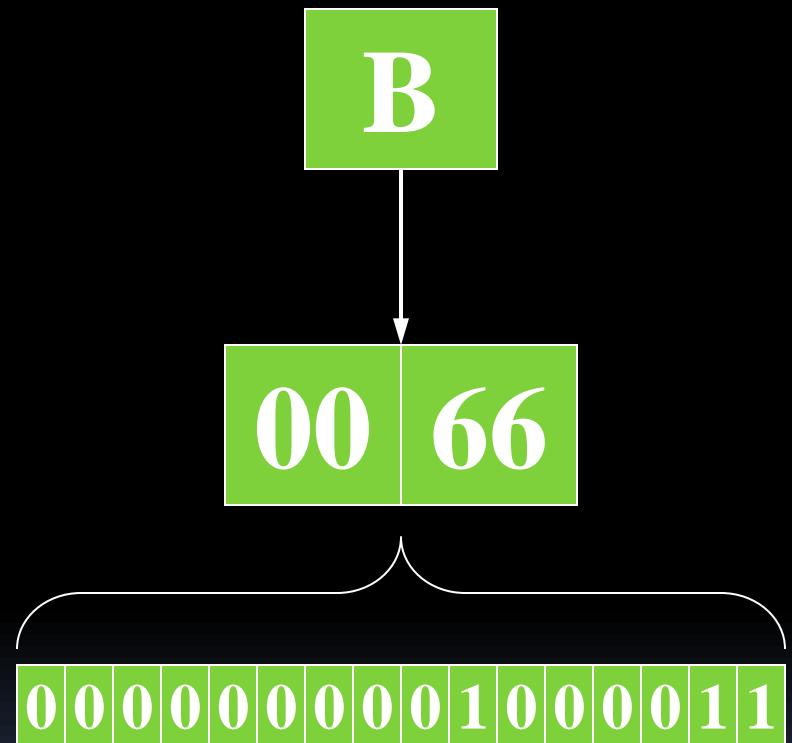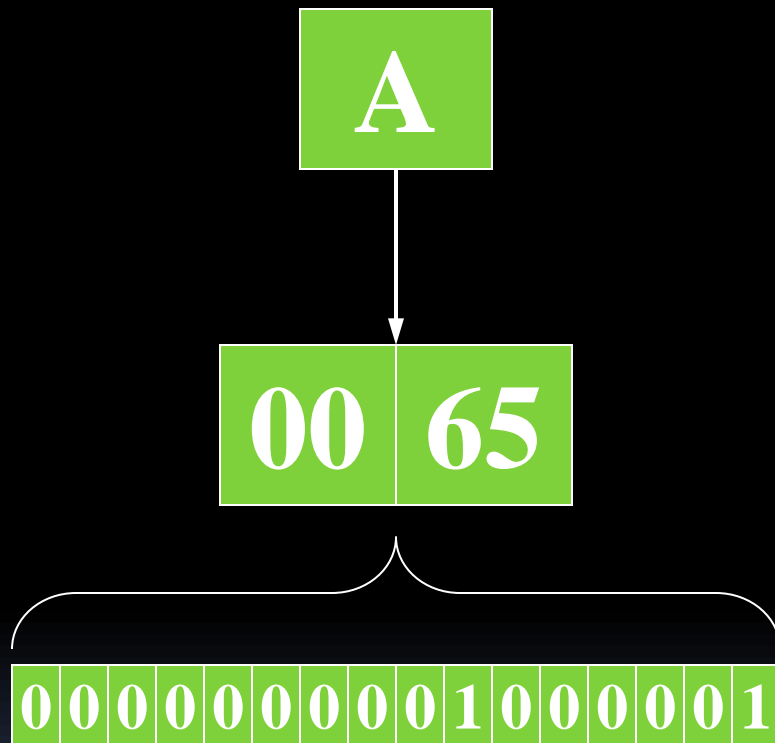
Unicode

Some more Unicode

龜 ←——————————→ औ

The decimal values represent these characters.

20096   2324

0100111010000000   0000100100010100
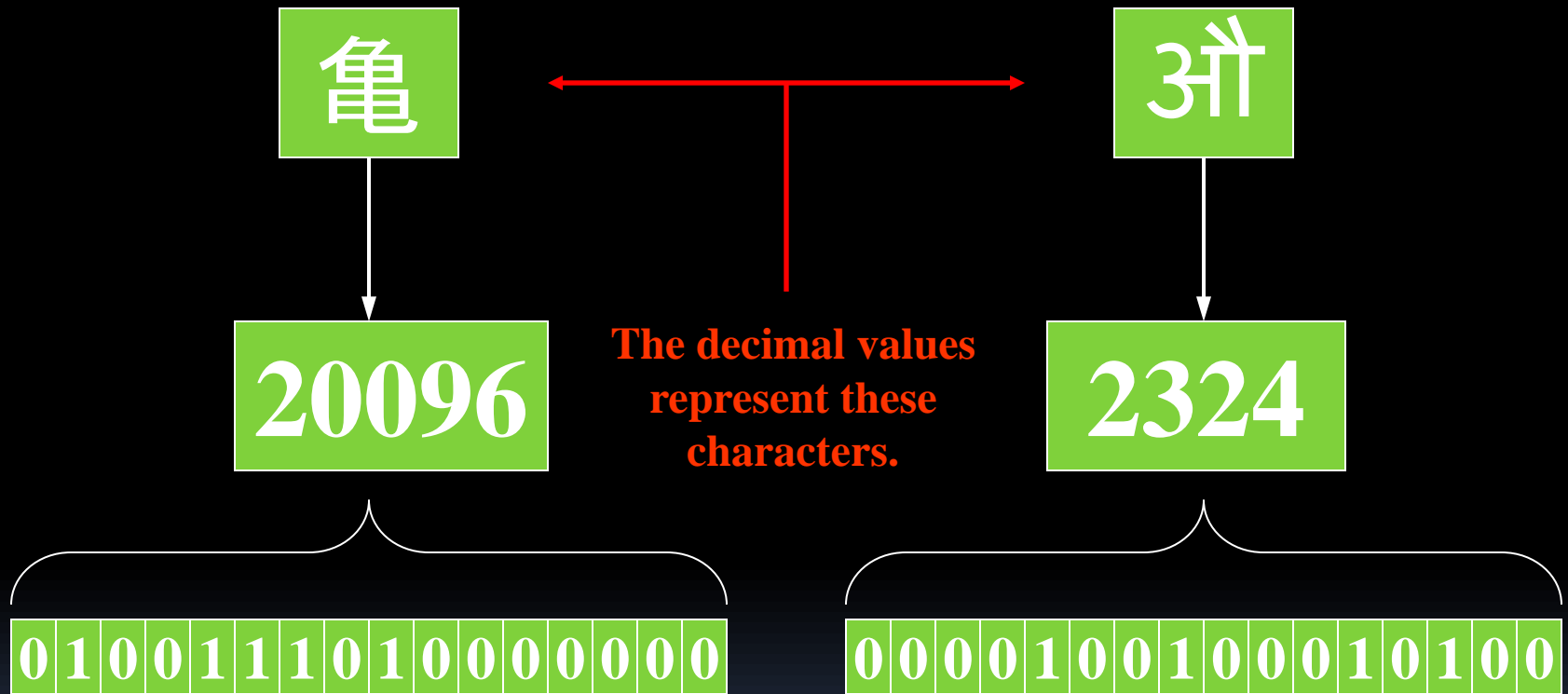
Variable Assignment and Initialization

- In order to store a value in a variable, an *assignment statement* must be used.

- The *assignment operator* is the equal (=) sign.

- The operand on the left side of the assignment operator must be a variable name.

- The operand on the right side must be either a literal or expression that evaluates to a type that is compatible with the type of the variable.

Variable Assignment and Initialization

```java
// This program shows variable assignment.

public class Initialize
{
  public static void main(String[] args)
  {
    int month, days;

    month = 2;
    days = 28;
    System.out.println("Month " + month + " has " +
                       days + " Days.");
  }
}
```

The variables must be declared before they can be used.

Variable Assignment and Initialization

```java
// This program shows variable assignment.

public class Initialize
{
  public static void main(String[] args)
  {
    int month, days;

    month = 2;
    days = 28;
    System.out.println("Month " + month + " has " +
                          days + " Days.");
  }
}
```

Once declared, they can then receive a value (initialization); however the value must be compatible with the variable's declared type.

Variable Assignment and Initialization

```java
// This program shows variable assignment.

public class Initialize
{
  public static void main(String[] args)
  {
    int month, days;

    month = 2;
    days = 28;
    System.out.println("Month " + month + " has " +
                       days + " Days.");
  }
}
```

After receiving a value, the variables can then be used in output statements or in other calculations.

Variable Assignment and Initialization

```java
// This program shows variable initialization.

public class Initialize
{
   public static void main(String[] args)
   {
      int month = 2, days = 28;
      System.out.println("Month " + month + " has " +
                           days + " Days.");
   }
}
```

Local variables can be declared and initialized on the same line.

Variable Assignment and Initialization

- Variables can only hold one value at a time.

- Local variables do not receive a default value.

- Local variables must have a valid type in order to be used.

```
public static void main(String [] args)
{
    int month, days; //No value given…

   System.out.println("Month " + month + " has " +
                      days + " Days.");
}
```

Trying to use uninitialized variables will generate a Syntax Error when the code is compiled.

Arithmetic Operators

- Java has five (5) arithmetic operators.

Operator	Meaning	Type	Example
+	Addition	Binary	`total = cost + tax;`
-	Subtraction	Binary	`cost = total - tax;`
*	Multiplication	Binary	`tax = cost * rate;`
/	Division	Binary	`salePrice = original / 2;`
%	Modulus	Binary	`remainder = value % 5;`

Arithmetic Operators

- The operators are called binary operators because they must have two operands.

- Each operator must have a left and right operator.

See example: Wages.java

- The arithmetic operators work as one would expect.

- It is an error to try to divide any number by zero.

- When working with two integer operands, the division operator requires special attention.

Integer Division

- Division can be tricky.

  In a Java program, what is the value of 1/2?

- You might think the answer is 0.5…

- But, that's wrong.

- The answer is simply 0.

- Integer division will truncate any decimal remainder.

Operator Precedence

- Mathematical expressions can be very complex.
- There is a set order in which arithmetic operations will be carried out.

	Operator	Associativity	Example	Result
Higher Priority	− (unary negation)	Right to left	`x = -4 + 3;`	-1
	`*  /  %`	Left to right	`x = -4 + 4 % 3 * 13 + 2;`	11
Lower Priority	`+  -`	Left to right	`x = 6 + 3 - 4 + 6 * 3;`	23

Grouping with Parenthesis

- When parenthesis are used in an expression, the inner most parenthesis are processed first.
- If two sets of parenthesis are at the same level, they are processed left to right.

```
x = ((4*5) / (5-2) ) - 25;   // result = -19
```

Combined Assignment Operators

- Java has some combined assignment operators.

- These operators allow the programmer to perform an arithmetic operation and assignment with a single operator.

- Although not required, these operators are popular since they shorten simple equations.

Combined Assignment Operators

Operator	Example	Equivalent	Value of variable after operation
+=	x += 5;	x = x + 5;	The old value of **x** plus 5.
-=	y -= 2;	y = y - 2;	The old value of **y** minus 2
*=	z *= 10;	z = z * 10;	The old value of **z** times 10
/=	a /= b;	a = a / b;	The old value of **a** divided by **b**.
%=	c %= 3;	c = c % 3;	The remainder of the division of the old value of **c** divided by 3.

Creating Constants

- Many programs have data that does not need to be changed.

- Littering programs with literal values can make the program hard do read and maintain.

- Replacing literal values with constants remedies this problem.

- Constants allow the programmer to use a name rather than a value throughout the program.

- Constants also give a singular point for changing those values when needed.

Creating Constants

- Constants keep the program organized and easier to maintain.

- Constants are identifiers that can hold only a single value.

- Constants are declared using the keyword `final`.

- Constants need not be initialized when declared; however, they must be initialized before they are used or a compiler error will be generated.

Creating Constants

- Once initialized with a value, constants cannot be changed programmatically.

- By convention, constants are all upper case and words are separated by the underscore character.

  ```
  final double CAL_SALES_TAX = 0.725;
  ```

The `String` Class

- Java has no primitive data type that holds a series of characters.

- The `String` class from the Java standard library is used for this purpose.

- In order to be useful, the a variable must be created to reference a `String` object.

  `String number = "25";`

- Notice the `S` in `String` is upper case.

- By convention, class names should always begin with an upper case character.

Primitive vs. Reference Variables

- Primitive variables actually contain the value that they have been assigned.

  ```
  int number = 25;
  ```

- The value 25 will be stored in the memory location associated with the variable `number`.

- Objects are not stored in variables, however. Objects are *referenced* by variables.

Primitive vs. Reference Variables

- When a variable references an object, it contains the memory address of the object's location.

- Then it is said that the variable *references* the object.

```
String cityName = "Charleston";
```

The object that contains the character string "Charleston"

cityName | Address to the object | → | Charleston |
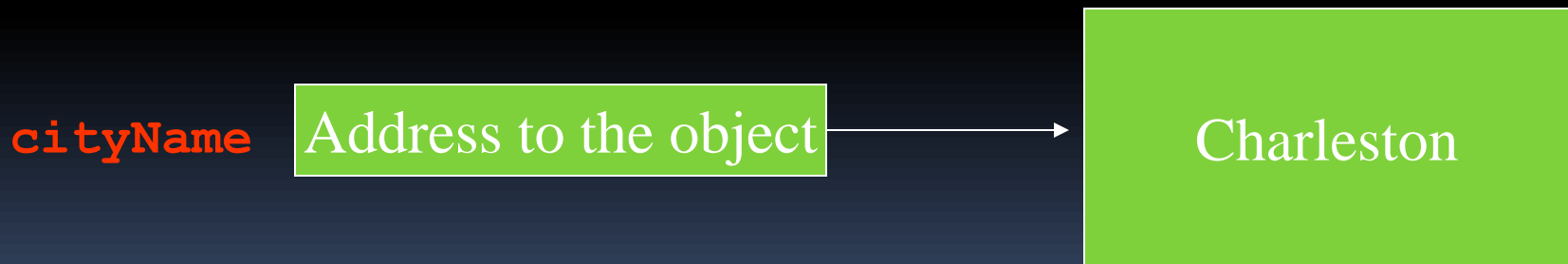
String Objects

- A variable can be assigned a String literal.

  `String value = "Hello";`

- `Strings` are the only objects that can be created in this way.

- A variable can be created using the *new* keyword.

  `String value = new String("Hello");`

- This is the method that all other objects must use when they are created.

See example: StringDemo.java

The `String` Methods

- Since `String` is a class, objects that are instances of it have methods.

- One of those methods is the `length` method.

  `stringSize = value.length();`

- This statement runs the `length` method on the object pointed to by the `value` variable.

See example: StringLength.java

`String` Methods

- The `String` class contains many methods that help with the manipulation of `String` objects.

- `String` objects are *immutable*, meaning that they cannot be changed.

- Many of the methods of a `String` object can create new versions of the object.

See example: StringMethods.java

Scope

- *Scope* refers to the part of a program that has access to a variable's contents.

- Variables declared inside a method (like the main method) are called *local variables*.

- Local variables' scope begins at the declaration of the variable and ends at the end of the method in which it was declared.

See example: Scope.java (This program contains an intentional error.)

Commenting Code

- Java provides three methods for commenting code.

Comment Style	Description
`//`	Single line comment. Anything after the // on the line will be ignored by the compiler.
`/* ... */`	Block comment. Everything beginning with /* and ending with the first */ will be ignored by the compiler. This comment type cannot be nested.
`/** ... */`	Javadoc comment. This is a special version of the previous block comment that allows comments to be documented by the javadoc utility program. Everything beginning with the /** and ending with the first */ will be ignored by the compiler. This comment type cannot be nested.

Commenting Code

- Javadoc comments can be built into HTML documentation.

- See example: <u>Comment3.java</u>

- To create the documentation:
    - Run the `javadoc` program with the source file as an argument
    - Ex: **`javadoc Comment3.java`**

- The `javadoc` program will create `index.html` and several other documentation files in the same directory as the input file.

Commenting Code

- Example index.html:

Programming Style

- Although Java has a strict syntax, whitespace characters are ignored by the compiler.

- The Java whitespace characters are:
  - space
  - tab
  - newline
  - carriage return
  - form feed

See example: Compact.java

Indentation

- Programs should use proper indentation.
- Each block of code should be indented a few spaces from its surrounding block.
- Two to four spaces are sufficient.
- Tab characters should be avoided.
  - Tabs can vary in size between applications and devices.
  - Most programming text editors allow the user to replace the tab with spaces.

See example: Readable.java

The `Scanner` Class

- To read input from the keyboard we can use the `Scanner` class.

- The `Scanner` class is defined in `java.util`, so we will use the following statement at the top of our programs:

```
import java.util.Scanner;
```

The `Scanner` Class

- Scanner objects work with `System.in`

- To create a `Scanner` object:
  ```
  Scanner keyboard = new Scanner (System.in);
  ```

- `Scanner` class methods are listed in Table 2-18 in the text.

- See example: Payroll.java

Dialog Boxes

- A *dialog box* is a small graphical window that displays a message to the user or requests input.
- A variety of dialog boxes can be displayed using the `JOptionPane` class.
- Two of the dialog boxes are:
  - Message Dialog - a dialog box that displays a message.
  - Input Dialog - a dialog box that prompts the user for input.

The `JOptionPane` Class

- The `JOptionPane` class is not automatically available to your Java programs.
- The following statement must be before the program's class header:

  `import javax.swing.JOptionPane;`

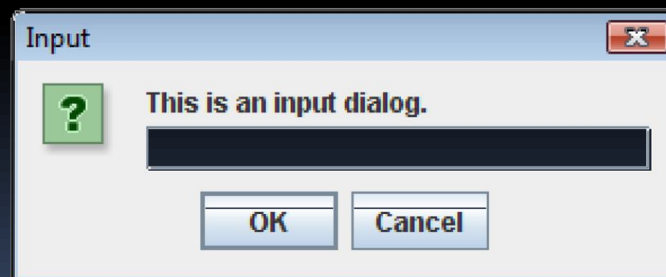- This statement tells the compiler where to find the `JOptionPane` class.

The `JOptionPane` Class

The `JOptionPane` class provides methods to display each type of dialog box.
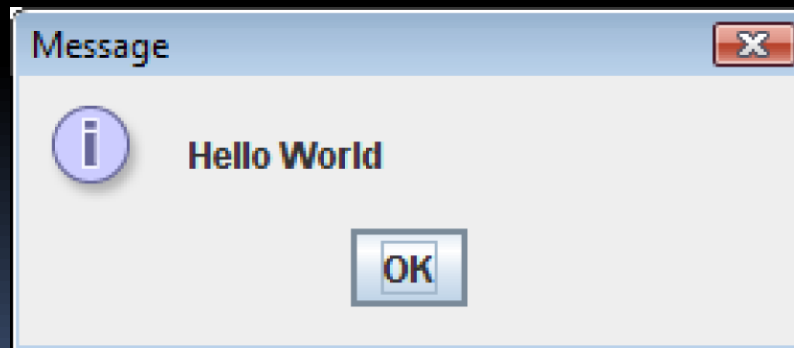
Message dialog
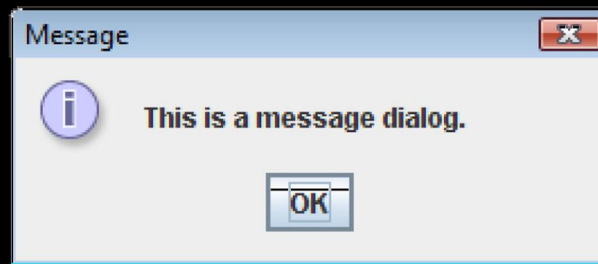
Input dialog

Message Dialogs

- `JOptionPane.showMessageDialog` method is used to display a message dialog.

  `JOptionPane.showMessageDialog(null, "Hello World");`

- The first argument will be discussed in Chapter 7.

- The second argument is the message that is to be displayed.

Input Dialogs

- An input dialog is a quick and simple way to ask the user to enter data.

- The dialog displays a text field, an Ok button and a Cancel button.

- If Ok is pressed, the dialog returns the user's input.

- If Cancel is pressed, the dialog returns null.

Input Dialogs

```
String name;
name = JOptionPane.showInputDialog(
        "Enter your name.");
```

- The argument passed to the method is the message to display.

- If the user clicks on the OK button, `name` references the string entered by the user.

- If the user clicks on the Cancel button, `name` references `null`.
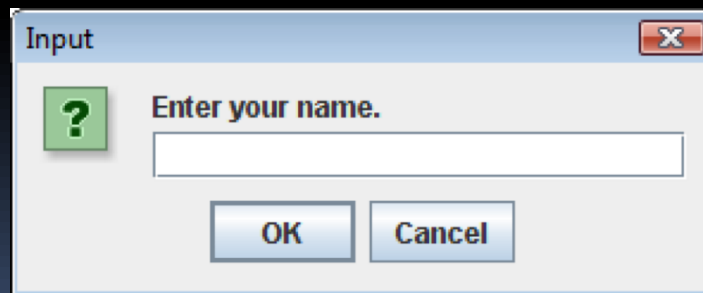
The `System.exit` Method

- A program that uses `JOptionPane` does not automatically stop executing when the end of the main method is reached.

- Java generates a *thread*, which is a process running in the computer, when a `JOptionPane` is created.

- If the `System.exit` method is not called, this thread continues to execute.

The `System.exit` Method

- The `System.exit` method requires an integer argument.

  System.exit(0);

- This argument is an *exit code* that is passed back to the operating system.

- This code is usually ignored, however, it can be used outside the program:

  - to indicate whether the program ended successfully or as the result of a failure.

  - The value o traditionally indicates that the program ended successfully.

Converting a String to a Number

- The `JOptionPane`'s `showInputDialog` method always returns the user's input as a `String`

- A `String` containing a number, such as "127.89, can be converted to a numeric data type.

The Parse Methods

- Each of the numeric wrapper classes, (covered in Chapter 10) has a method that converts a string to a number.

  - The `Integer` class has a method that converts a string to an `int`,

  - The `Double` class has a method that converts a string to a `double`, and

  - etc.

- These methods are known as *parse methods* because their names begin with the word "parse."

The Parse Methods

```
// Store 1 in bVar.
byte bVar = Byte.parseByte("1");

// Store 2599 in iVar.
Sting someRandomVariableName = "2599";
int iVar = Integer.parseInt(somerandomvariablename);

// Store 10 in sVar.
short sVar = Short.parseShort("10");

// Store 15908 in lVar.
long lVar = Long.parseLong("15908");

// Store 12.3 in fVar.
float fVar = Float.parseFloat("12.3");

// Store 7945.6 in dVar.
double dVar = Double.parseDouble("7945.6");
```

Reading an Integer with an Input Dialog

```
int number;
String str;
str = JOptionPane.showInputDialog(
    "Enter a number.");
number = Integer.parseInt(str);
```

Reading a `double` with an Input Dialog

```
double price;
String str;
str = JOptionPane.showInputDialog(
    "Enter the retail price.");
price = Double.parseDouble(str);
```

See example: PayrollDialog.java