Chapter 2

Fundamental Data Types

Using BeanShell

Outline

- Fundamental data types and variables
- Declaration and initialization of variables
- Arithmetic operations and expressions
- Assignment statements
- Arithmetic functions from the Math class
- Using BeanShell to understand basic concepts
2.1 Fundamental data types and variables

Data comes in many types. There are numeric types for integers, characters and floating point numbers, and there are non-numeric types, such as the boolean type which represents the logical values true and false. These fundamental types are called primitive types. There are also object types, such as the string type for representing strings of characters. We can also define our own types. However, in this chapter we concentrate on the fundamental numeric data types.

Formally, a data type has two parts

- A set of values
- A set of operations on these values

In mathematics the most fundamental kinds of data are the integers and real numbers. We will start with these familiar types and see how they are represented as primitive types in Java.

2.1.1 Integer and floating point data types

In mathematics we define $\mathbb{Z}$ to be the set of all integers, and we can define subsets such as $\mathbb{N} = \{ n : n \in \mathbb{Z}, n \geq 0 \}$ containing only the non-negative integers (read this as the set of all $n$ such that $n$ belongs to $\mathbb{Z}$ and $n$ is greater than or equal to zero). From the integers we can then obtain the set of rational numbers (fractions), $\mathbb{Q} = \{ q : q = \frac{a}{b}, a, b \in \mathbb{Z}, b \neq 0 \}$. Finally, from the rationals we can obtain the real numbers $\mathbb{R}$.

In algebra, variables can be defined with values taken from some subset of the integers, rationals, or real numbers. Then, following the rules of algebra, operations on these values and variables can be defined.

You should be familiar with the standard operations of addition, subtraction, multiplication, and division, and with the rules of algebra for writing algebraic expressions involving variables, values, and operations. If $a$ and $b$ are integer or real variables we use $a + b$, $a - b$, $ab$, and $a/b$ to represent these operations. In mathematics variables are normally one letter symbols so the multiplication of $a$ and $b$ is implied by juxtaposition of the variables as in $ab$. We can also use $a \cdot b$ or $a \times b$ to denote multiplication and this is more appropriate for example in pseudo-code algorithms where variables often have multi letter names. In languages like Java $*$ is used to denote multiplication.

Division requires some care, since $a/b$ is undefined if $b = 0$. Also, if $a$ and $b$ are real numbers then $a/b$ is a real number, but if $a$ and $b$ are integers then $a/b$ need not be an integer. This means there are two kinds of division: real number division and integer division to produce a quotient and remainder. For integer division we can define the quotient and remainder using the

**Quotient-remainder theorem (Q-R theorem)** If $n$ (the numerator) and $d$ (the divisor or denominator) are non-negative integers and $d \neq 0$, there are unique integers $q$, called the quotient, and $r$, called the remainder, such that

$$n = d \cdot q + r, \quad \text{where } r \text{ satisfies } 0 \leq r < d$$

In mathematics the quotient $q$ is defined as the integer division $n \div d$, and the remainder $r$ is defined, using the modulus operator, as $n \mod d$. Do not confuse the integer division $n \div d$ with
the fraction \( \frac{n}{d} \) which is a rational number, or the real number \( \frac{a}{b} \) which in general is not an integer. For example \( 27 \div 5 \) is 5 (remainder is 2) and \( 27/5 \) is the real number 5.4. Using fractions we would write \( \frac{27}{5} = 5 + \frac{2}{5} \), so we have \( q = 5 \) and \( r = 2 = 27 \mod 5 \).

**From mathematical to computer data types**

We run into two problems when we try to make computer data types out of these mathematical ones:

- Mathematical sets, such as \( \mathbb{Z} \) and \( \mathbb{R} \), are infinite and we cannot represent an infinite number of integers or real numbers with a finite amount of computer memory.
- Real numbers are stored in computer memory in a binary form. Many real numbers and fractions, such as \( 1/3 \), \( \pi \), or \( \sqrt{2} \), have infinite decimal and binary representations, so they cannot be stored exactly in binary form with a finite amount of computer memory. Also, some numbers such as \( 1/10 = 0.1 \) have finite decimal representations but infinite binary representations.

The solution to the first problem is to restrict the ranges of the numbers to a finite subset of integers or real numbers, and for the second problem we must simply accept the fact that there is some small round-off error (loss of precision) incurred when the infinite decimal expansions of certain numbers such as \( \pi \) or \( \sqrt{2} \) are truncated to fit in a finite computer memory.

For storage, it is necessary to use a specific number of bits for each number. It is conventional to specify several kinds of numerical data types with different “sizes” (number of bits). The larger the number of bits, the larger the range of numerical data that can be represented. For integers, two common choices are 16-bit and 32-bit integers.

Similarly, real numbers are called floating point numbers and are commonly stored using 32 or 64 bits. The 32-bit numbers are called single precision floating point numbers. The 64-bit numbers are called double precision floating point numbers, since they can store numbers with approximately twice the precision (number of significant digits). As a rule of thumb, 32-bit floating point numbers have about 7 or 8 decimal digits of precision, and 64-bit numbers have about 16 decimal digits of precision. For example, in Java \( \pi \) has the approximate value 3.1415927 as a single precision number and 3.141592653589793 as a double precision number. The trade off is more precision at the expense of more storage space. Of course loss of precision (round-off error) does not occur for integer values; either the number fits exactly in memory or there is overflow.

In Java there are seven numeric data types. Five are integer data types, called byte, short, char, int, and long. The other two numeric types are floating point types, called float and double. These numeric types and the non-numeric boolean type are called primitive types since all data types are built from them. The seven numeric types, their sizes in bits, and their ranges, are shown in Table 2.1. From the table, the minimum and maximum values of the five integer types are \( -2^{n-1} \) and \( 2^{n-1} - 1 \), where \( n \) is the number of bits. We normally use the standard int and double types to represent integers and floating point numbers.

Since our integers now have a limited range we sometimes need to be concerned with overflow after performing an arithmetic operation. For example, what happens if we add something to the largest int value? What happens if we add two int values and the sum is larger than the largest
Fundamental Data Types

<table>
<thead>
<tr>
<th>Type</th>
<th>bits</th>
<th>Minimum value</th>
<th>Maximum value</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>8</td>
<td>(-128)</td>
<td>127</td>
</tr>
<tr>
<td>short</td>
<td>16</td>
<td>(-32768)</td>
<td>32767</td>
</tr>
<tr>
<td>char</td>
<td>16</td>
<td>0</td>
<td>65535</td>
</tr>
<tr>
<td>int</td>
<td>32</td>
<td>(-2147483648)</td>
<td>2147483647</td>
</tr>
<tr>
<td>long</td>
<td>64</td>
<td>(-9223372036854775808)</td>
<td>9223372036854775807</td>
</tr>
<tr>
<td>float</td>
<td>32</td>
<td>(\approx \pm 1.40 \times 10^{-45})</td>
<td>(\approx \pm 3.40 \times 10^{38})</td>
</tr>
<tr>
<td>double</td>
<td>64</td>
<td>(\approx \pm 4.94 \times 10^{-324})</td>
<td>(\approx \pm 1.80 \times 10^{308})</td>
</tr>
</tbody>
</table>

Table 2.1: The Java integer and floating point primitive types.

An int value? These problems with integer operations are called overflow. In other words, performing arithmetic operations on int values can cause overflow and the result will be undefined.

A similar problem occurs for floating point numbers of type float or double. For example, the numbers \(3.4 \times 10^{45}\) and \(-5.65 \times 10^{56}\) have exponents that are outside the range of the float type but within the range of the double type. For floating point numbers there is also the problem of underflow. There are floating point numbers such as \(1.5 \times 10^{-400}\) that are not zero, but according to the table they are smaller than the smallest possible non-zero number. They are normally stored as zero. This is called underflow to 0.

It is not necessary to understand in detail how numbers are actually stored internally in binary form. It is only necessary to understand how overflow and round-off errors can occur.

The char data type

The char data type represents characters, such as the letters of the alphabet, punctuation, or digits. In Java a character literal is represented as a character enclosed in single quotes. For example, the letter “a” is denoted by ‘a’, the digit “3” is denoted by ‘3’, and the exclamation mark “!” is denoted by ‘!’.

The boolean data type

The only non-numeric primitive type is the boolean type which represents the two logical values for true and false, denoted in Java by the boolean literals true and false. We will not use this type until we discuss conditional statements in a later Chapter.

2.1.2 Integer and floating point literals

An integer value such as 0, 4, or 5434 is called an integer literal. Similarly, a floating point value with a decimal point such as \(3.1416\) is called a floating point literal. Scientific notation can also
2.1 Fundamental data types and variables

be used for a floating point literal. For example, \(6.023 \times 10^{23}\) denotes \(6.023 \times 10^2\) and uses an \(E\) to indicate the power of 10. Lowercase \(e\) can also be used to denote the exponent. Integer literals never have a decimal point and floating point literals always have one, except a decimal point is not required if an exponent is used. For example, \(3 \times 10^{-5} = 0.00003\).

2.1.3 Declaring and initializing variables in Java

The concept of a variable is fundamental to all programming languages. In Java, each variable has a name, a type (such as int or double), and it corresponds to a storage location in computer memory which can hold a value of the specified type. Think of a variable as a “named storage location”. The content of this storage location is the value of the variable. Thus, at any stage in the execution of a computer program, a variable has a name and a value. To declare a variable means to specify its type and its name in a declaration statement. This provides information to the compiler that is used to allocate enough storage space.

\[\text{Example 2.1 (variable declarations)}\]

The statements

\[
\text{int width; double area;}
\]

declare an int variable called width and a double variable called area.

The declarations in Example 2.1 allocate two storage locations: 32 bits for an integer value and 64 bits for a double precision value. They do not specify or define the content of the storage locations. The content of the storage location for a variable is called the value of the variable. In Example 2.1, width and area are uninitialized variables. We say that the value of an uninitialized variable is undefined. It is useful to have a pictorial representation of a variable as a “named storage location” in memory. This is illustrated in Figure 2.1 for the two uninitialized variables. The box represents the storage location; the variable name is written beside the box. The content of the box is the value of the variable. A question mark is shown to indicate that the two variables have not been initialized.

Before we can use these variables they need to be given values. There are two ways to do this: at the same time they are declared, or later in an assignment statement. We can give them values when we declare them as the following example shows.

\[\text{Example 2.2 (variable declaration and initialization)}\]

Instead of the declarations in the preceding example, we can use the initialized declarations:

\[
\text{int width = 5; double area = 3.1416;}
\]
Each declares a variable and gives it an initial value at the same time. The pictorial representation of these two variables, after initialization, is shown in Figure 2.2. The values in the boxes are the current values of the variables.

**Example 2.3** It is also possible to declare and initialize multiple variables of the same type in a single declaration:

```c
double radius, area, circumference;
```

Another possibility is

```c
double radius = 2.0, area, circumference;
```

which declares three variables and assigns the initial value 2.0 to `radius`.

Alternatively, when the variables are declared without initial values, we can initialize them later, as shown in the next example.

**Example 2.4** (assignment statements) If the variables `width` and `area` have been previously declared, as in Example 2.1, we can give them values using statements like:

```c
width = 5;
area = 3.1416;
```

Each of these statements is an example of an assignment statement.

**Example 2.5** (Multiple assignment statement) It is possible to give several variables of the same type a common value using a multiple assignment statement. Assuming that `a`, `b`, and `c` have been declared as variables of type `double`, the statement

```c
a = b = c = 0.0;
```

gives each the value 0.0 and is equivalent to

```c
a = 0.0;
b = 0.0;
c = 0.0;
```

which uses three separate assignment statements.
Assignment statements always have the name of a variable to the left of the = sign and the value to be given the variable on the right of the equal sign. You can read the statement “width = 5;” as “width gets or receives the value 5”. It is an error to use a variable on the left side of an assignment statement if it has not been previously declared in a declaration statement. You can always distinguish a declaration statement from an assignment statement because the former specifies the type and the latter does not specify it.

A characteristic feature of a variable is that it has a type and a value. Java is a **strongly typed** language. This means that once a variable has been declared, its type can never be changed. However, when the program is running the value of a variable can be changed at any time using an assignment statement.

If you forget to give a value to a variable and later attempt to use its value, the Java compiler will complain with an error message saying that the variable has not been initialized.

### Rules for naming variables

In Java, the simplest kind of name is called an **identifier**. An identifier must begin with a letter, the dollar sign character ($), or the underscore character (_). Any remaining characters can also be one of these, or any digit character (0 to 9). It is recommended that you not use the dollar sign or underscore in variable names. Some identifiers, such as the type names `int` and `double`, are reserved words called **keywords**. They cannot be used for other purposes, such as variable names.

Java is also **case-sensitive**. This means that the case (uppercase or lowercase) is significant. For example, `width`, `Width`, and `WIDTH`, would be the names of three different variables. It is conventional to begin variable names with a lowercase letter and you should always follow this convention, even though it is not a language requirement.

### Constants

In addition to variables, Java also has **constants**. Like variables they have names, a type, and a value. However once a value has been assigned it can never be changed. Constant declarations are distinguished from variable declarations by using the strange keyword `final`: once you have specified a value its final! The Java compiler will complain if you attempt to change the value of a constant. The main purpose of a constant is to give a meaningful name to a literal, such as an integer or floating point literal, as the following example shows:

**Example 2.6** (declaring constants) The declarations

```java
final double CM_PER_INCH = 2.54;
final int MARGIN_WIDTH = 5;
```

define a `double` constant for a conversion factor from centimeters to inches, and an `int` constant that might represent the default width of the margin in a typesetting program.

Effective use of constants improves readability and makes it easier to modify programs. For example, if the number 5 appears in several places, it might mean the margin width in one place and something else in another place. Then, changing the margin width is difficult. It is easy if a constant is used.
It is also conventional to use all uppercase letters for constants with the underscore character simulating a space.

2.2 Arithmetic operations and expressions

2.2.1 Basic arithmetic operations

In Java the addition and subtraction operations are denoted as in mathematics by + and -. Juxtaposition cannot be used for multiplication since we want to have variable names longer than one letter: ab is a variable, not the product of a and b. Therefore, most programming languages use the asterisk * to denote multiplication. For example, ab is a variable but a*b is the product of variables a and b. Division is denoted by /. As in mathematics there are two kinds of division: integer division to obtain the quotient, and the real, or floating point, division. In pseudo-code div is often used for integer division and / is used for floating point division.

Unfortunately in Java / is used for both kinds of division: a/b is an integer division only if both a and b have integer values. If either or both of a and b have floating point values it is a floating point division. In Java, the modulus operation a mod b, giving the remainder when a is divided by b, is denoted by % . The % operator is called the modulus operator or the remainder operator.

2.2.2 Arithmetic expressions and precedence rules

Mathematical expressions involving the basic operations can easily be translated to Java using the same well known mathematical precedence rules (or order of operations):

1. *, %, and / have the same precedence. They have a higher precedence than + and - so they are done first, in the left to right order in which they appear. This is called left associativity. Example: In the expression a + b*c + d the multiplication is done first.

2. + and - have the same precedence and are done next in the left to right order in which they appear. They are also left associative. Example: In the expression a + b - c, the addition is done first, followed by the subtraction.

3. Parentheses have the highest precedence of all and can change the precedence of the other operators. For example, in the expression a + b*c + d the multiplication is done first, but in the expression (a + b)* (c + d), the multiplication is done last.

There are many other operators so this table is not complete.

Example 2.7 (unary and binary operators) The arithmetic operators +, -, *, /, and % are binary operators. For example, in an expression such as a + b, the values a and b are said to be the operands of the + operator. Since there are two operands, + is called a binary operator. Unlike the binary * and / operators, the + and - operators can also be used in expressions such as -b, or +b, where they have only one operand, the value of b. In this context they are called unary operators.
2.3 Assignment statements

**Example 2.8** (mathematical expressions) Table 2.2 shows some mathematical expressions and their translations to Java, assuming that all variables are real and represented as type double. You must be careful with division operations. If \( \frac{9}{5} \) and \( \frac{5}{9} \) had been translated as \( 9/5 \) and \( 5/9 \), the wrong results would be obtained, since \( 9/5 \) is an integer division with value 1, and \( 5/9 \) is an integer division with value 0.

<table>
<thead>
<tr>
<th>Mathematical Expression</th>
<th>Java Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a + bc - 4 )</td>
<td>( a + b* c - 4.0 )</td>
</tr>
<tr>
<td>( \frac{1}{2} (a + b)(c - 7) )</td>
<td>((a + b)*(c - 7.0)/2.0 )</td>
</tr>
<tr>
<td>( \frac{9}{3} c + 32 )</td>
<td>( (9.0/5.0)*c + 32.0 )</td>
</tr>
<tr>
<td>( \frac{5}{7} (f - 32) )</td>
<td>( (5.0/9.0)*(f - 32.0) )</td>
</tr>
<tr>
<td>( a^2 b^2 + \frac{c^3}{a+b} )</td>
<td>( a<em>a</em>b<em>b + c</em>c*c/(a + b) )</td>
</tr>
<tr>
<td>( 3x^2 - 2x + 4 )</td>
<td>( 3.0<em>x</em>x - 2.0*x + 4.0 )</td>
</tr>
<tr>
<td>( 1.3 + x(3.4 - x(2.5 + 4.2x)) )</td>
<td>( 1.3 + x*(3.4 - x*(2.5 + 4.2*x)) )</td>
</tr>
<tr>
<td>( s(s-a)(s-b)(s-c) )</td>
<td>( s*(s-a)<em>(s-b)</em>(s-c) )</td>
</tr>
</tbody>
</table>

Table 2.2: Translation of mathematical expressions to Java.

**Example 2.9** (integer vs floating point division) Let totalCents be a variable of type int having the value 3527. Then totalCents / 100 is an integer division having the value 35 and totalCents % 100 gives the remainder 27. On the other hand, totalCents / 100.0 is interpreted as a double floating point result, so its value is 35.27 since 100.0 is a double literal constant.

In these examples of expressions, as a matter of style, we have surrounded the binary operators + and − by a single space, but we have not done this for the binary operators * and /. This is a common convention to emphasize that a typical arithmetic expression is composed of terms and factors. The terms are separated by the binary + and − operators, and the factors are separated by *, /, and %. In the first example in Table 2.2, the terms are \( a \), \( b*c \), and 4.0. The term \( b*c \) is composed of two factors \( b \) and \( c \). The extra space around terms makes them stand out. If you prefer, you can surround all binary operators by a single space.

### 2.3 Assignment statements

Assignment statements are used to give values to variables whose type has already been declared. In Java the = sign is used to indicate an assignment. The left side is the name of the variable and the right side is an expression. In the simplest cases an expression may be a literal, or a variable, or an expression involving arithmetic operators. When an assignment statement is executed, the value of the expression on the right side is evaluated and assigned as the value of the variable on the left side.
Example 2.10 (does not denote mathematical equality) An assignment statement should never be confused with an equation or an equality. There is a big difference between the assignment statement

\[ x = x + 1; \]

and the mathematical equation \[ x = x + 1. \] In the assignment statement the right side is evaluated by adding one to the current value of \( x \) and assigning the result as the new value of \( x \). The mathematical equation is meaningless since it implies that \( 0 = 1 \).

Example 2.11 (special combination operators) There are also special combination operators such as \( += \), \( -= \), \( *= \), and \( /= \), which combine an arithmetic operation and assignment. For example, the assignment statement

\[ \text{totalArea} += \text{area}; \]

is just shorthand for

\[ \text{totalArea} = \text{totalArea} + \text{area}; \]

We will try to avoid using these combination operators since they make programs harder to read.

Example 2.12 (assignment statements) The following statements declare three variables of type \texttt{double}, using one combined declaration instead of three, and use three assignment statements to specify the radius and calculate the area and circumference of a circle having this radius:

\begin{verbatim}
double radius, area, circ;
radius = 3.0;
area = Math.PI * radius * radius;
circ = 2.0 * Math.PI * radius;
\end{verbatim}

It is not necessary to remember the double precision value of \( \pi \), since it is available as the constant \texttt{Math.PI} in a special built-in Java class called \texttt{Math}. We will see that there are many useful mathematics functions in the \texttt{Math} class. The name \texttt{Math.PI} is an example of a qualified name (a name with a dot in it).

2.3.1 Try it with BeanShell

You can test your understanding of simple examples like this using the interactive \texttt{BeanShell} program: a scripting shell for Java. With it you can execute Java statements immediately.

In Figure 2.3 we show \texttt{BeanShell} in action on Example 2.12. In Figure 2.3(a) \texttt{BeanShell}'s special \texttt{print} statement is used to see the value of a variable or expression. In Figure 2.3(b) \texttt{BeanShell}'s special \texttt{show()} function is used to automatically show the result of each assignment statement in angle brackets. This function acts like a “toggle” that turns on or off the displaying of intermediate results from assignment statements. This is the most useful way to get output for simple examples. If show is “on” then you can see the value of a variable at any time by simply typing its name followed by a semicolon.
2.3 Assignment statements

The `print` and `show` functions are not part of Java. They are simply provided by BeanShell to control and view output. Later we will see how to do output in Java.

**EXAMPLE 2.13 (dollars and cents)** Continuing with Example 2.9 and assuming that `show()` is ‘on’, try the following statements in BeanShell:

```plaintext
bsh % int totalCents, cents, dollars;
bsh % totalCents = 3527;
<3527>
bsh % dollars = totalCents / 100;
<35>
bsh % cents = totalCents % 100;
<27>
```

Here `bsh %` is the BeanShell prompt and is not typed.

**EXAMPLE 2.14 (integer division and remainder)** Try the following assignment statements in BeanShell, assuming that `show()` is in ‘on’.

```plaintext
bsh % int n = 123, remainder, hundreds, tens, units;
bsh % hundreds = n / 100;
<1>
bsh % remainder = n % 100;
<23>
bsh % tens = remainder / 10;
<2>
bsh % units = remainder % 10;
<3>
```
Here a multiple declaration statement is used to declare five variables and initialize one of them. Then the hundreds, tens, and units digits are extracted from the 3-digit integer 123.

**Example 2.15** (special increment and decrement operators) In Java there is a special increment operator, denoted by `++`, for adding one to the value of a variable, and a special decrement operator, denoted by `--`, for subtracting one from the value of a variable. Try the following statements in BeanShell, assuming `show()` is 'on'.

```bash
bsh % int i = 3;
bsh % int j = 4;
bsh % i++;
<3>
bsh % print(i);
4
bsh % j--;
<4>
bsh % print(j);
3
bsh %
```

Notice that the automatically displayed values are the values before the increment and decrement is applied. The BeanShell `print` statement shows the final values. The statements

```bash
i++;
j--;
```

are equivalent to the assignment statements

```bash
i = i + 1;
j = j - 1;
```

We will not use the increment and decrement operators too often, since they can make programs harder to read. There are also `--j` and `++j` forms of these operators. Also in some cases `++j` and `j++` have different effects and similarly for `--j` and `j--`.

### 2.4 Conversion between numeric types (type casting)

It is often necessary to convert a value of one numeric type to a value of another type. Sometimes the compiler will automatically do this and sometimes it will complain and produce an error message. The basic rule can be obtained from Table 2.1 in the column that indicates how many bits of storage are required for the values of each type. If you attempt to convert to a type which requires a smaller number of storage bits, the compiler will issue an error message because information may be lost in converting to a value with a smaller number of bits. However we can force the conversion with a technique called **type casting**, illustrated in the examples below. Of course this is normally useful only if information is not lost in converting to the smaller size so numeric type casts should be used with care.
Example 2.16 (valid implicit type conversions) Assume that \( d \) and \( e \) have type \texttt{double} and \( i \) has type \texttt{int}. The two assignment statements in

\begin{verbatim}
  bsh % int i = 1;
  bsh % double d, e;
  bsh % d = i;
  <1>
  bsh % e = i + 3.55;
  <4.55>
\end{verbatim}

do not cause any problems. In the first assignment the integer \( i \) is being converted to a larger size (any \texttt{int} will fit in a \texttt{double}). In the second assignment, to add the 32-bit integer value of \( i \) to the 64-bit double value 3.55, the value with the smaller size is first converted to a value of the larger size (so the value of \( i \) is converted to a \texttt{double}), with no loss of information, then the addition is performed as a 64-bit addition and the result is assigned to \( e \).

Example 2.17 (invalid implicit type conversions) Continuing the previous example the assignment statements in

\begin{verbatim}
  bsh % int j;
  bsh % i = e;
  // Error: Typed variable: i: Can’t assign double to int: ...
  bsh % j = i + e;
  // Error: Typed variable: j: Can’t assign double to int: ...
\end{verbatim}

each result in an error message. The first statement is an attempt to assign a 64-bit number as the value of a 32-bit variable. This cannot always be done without losing information. The second statement is an attempt to add the 32-bit integer \( i \) and the 64-bit double number \( d \), and assign the result as the value of a 32-bit integer \( j \). The expression on the right side does not cause problems: the 32-bit value of \( i \) can be converted to a 64-bit double precision number and then added to the value of \( d \). However, the attempt to assign this double precision result to \( j \) may cause a loss of information so the result is a compiler error.

2.4.1 Truncation of floating point numbers

A type cast can be used to truncate a floating point number. This has the affect of discarding the fractional part of the number to produce an integer result. Even though there is a loss of information, this is often a useful operation.

Example 2.18 (explicit type conversion as truncation) Continuing with the previous example try

\begin{verbatim}
  bsh % i = (int) e;
  <4>
  bsh % j = i + (int) e;
  <8>
\end{verbatim}
and the compiler does not produce any errors. The use of an explicit type name in parentheses in front of an expression is called a **type cast**. It tells the compiler to do the conversion anyway even if data could be lost. A type cast is a unary operator that converts the expression to which it is applied, to the specified type. Therefore, \((\text{int}) \ e\) causes the double number \(e\) to be converted to an \(\text{int}\) type. This means that the fractional part of \(e\), if any, is thrown away and the resulting integer is kept as the value. This is called **truncation** and is a useful operation if we want the integer part of a real number. If the integer is too big to store in an \(\text{int}\) then garbage is produced. For example, in BeanShell try

```java
bsh % i = (int)12345.5434;
<12345>
bsh % i = (int) 12345678912343.5;
<2147483647>
```

and observe that the value 12345.5434 is truncated to 12345, which is not too large for a 32-bit integer. However, the double value 12345678912343.5 would be truncated to 12345678912343 which is too large to hold in a 32-bit integer and overflow occurs with a meaningless result. In fact the result here is the largest integer value (see Table 2.1).

### 2.4.2 Loss of precision in floating point conversions

A loss of precision can also occur when trying to convert values of type \(\text{double}\) to type \(\text{float}\) since the size is reduced from 64 bits to 32 bits.

**Example 2.19** (explicit type conversion as loss of precision) If \(d\) has type \(\text{double}\) and \(f\) has type \(\text{float}\) then the statement

```
f = d;
```

results in a compiler error. We can use the typecast

```
f = (float) d;
```

The result will generally be a loss of precision in the conversion (see Table 2.1) which may be acceptable in some applications. Consider the BeanShell example

```java
bsh % double d = 1.11111111111111;
bsh % float f;
bsh % f = d;
// Error: Typed variable: f: Can’t assign double to float: ...
bsh % f = (float) d;
<1.11111111111111>
bsh % d = 1e-66;
<1.0E-66>
bsh % f = (float) d;
<0.0>
bsh % d = 1e66;
```
2.5 Arithmetic functions from the Math class

We will see that Java programs are made up of one or more classes each of which contains methods (functions are called methods in Java). Java has many built-in libraries of useful classes and methods. For example, the Math library contains many standard mathematical functions as well as two constants. We have already used the constant Math.PI to represent the double precision value of \( \pi \). There is also Math.E which represents the double precision value of \( e \).

There is a function Math.sqrt(x) for computing \( \sqrt{x} \); a power function Math.pow(x,y) for computing \( x^y \); the trigonometric functions Math.sin(x), Math.cos(x), and Math.tan(x); the inverse trigonometric functions Math.asin(x), Math.acos(x), and Math.atan(x); the exponential and log functions Math.exp(x) and Math.log(x), and several other functions.

2.5.1 Examples of mathematical functions

Here are some examples that use mathematical functions. BeanShell can be used to try them.

**Example 2.20** (square root function) The statements (assuming show is ‘on’)

```bash
bsh % double x = 1.0, y = 2.0;
bsh % double x1 = 1.0, y1 = 2.0, x2 = 2.0, y2 = 3.0;
bsh % double distance1 = Math.sqrt(x*x + y*y);
bsh % double dx = x2 - x1;
bsh % double dy = y2 - y1;
bsh % double distance2 = Math.sqrt(dx*dx + dy*dy);
bsh % print(distance1);
2.2360677974999997
bsh % print (distance2);
1.4142135623705999
```

compute the distance \( \sqrt{x^2 + y^2} \) of the point \((x,y)\) from the origin, and the distance \( \sqrt{(x_2-x_1)^2 + (y_2-y_1)^2} \) between the points \((x_1,y_1)\) and \((x_2,y_2)\) using the specific points \((x,y)=(1,2), (x_1,y_1)=(1,2), \) and \((x_2,y_2)=(2,3)\).

**Example 2.21** (distances using the square root and cosine functions) Given that a and b, are two sides of a triangle, and \( \gamma \) is the contained angle in degrees, the declaration statements
compute the length of the third side and the perimeter and area of the triangle with side lengths 1 and contained angle 90 degrees. Note that the trigonometric functions sin, cos, and tan require angles in radians so we have use the function Math.toRadians to do the conversion. There is also a function Math.toDegrees to convert radians to degrees.

**BeanShell tip:** If you are typing more than a few statements into the BeanShell workspace, as in the previous two examples, and make a mistake then you may have to start over. To avoid this open a workspace editor from the “File menu”. A mini-editor appears and you can type your statements here and edit them. When you want to execute the statements just select “Eval in Workspace” from the “Evaluate” menu and the results will appear in the workspace.

---

**EXAMPLE 2.22** *(calculating windchill)* Given the wind speed \(v\) in kilometers per hour and the air temperature \(t\) in degrees Celsius, the statement

\[
double wc = 0.045*(5.27*Math.sqrt(v) + 10.45 - 0.28*v)*(t - 33.0) + 33.0;
\]

computes the wind chill temperature in kilometers per hour. If you are using miles per hour for \(v\) and degrees Fahrenheit for \(t\) then

\[
double wc = 0.0817*(3.71*Math.sqrt(v) + 5.81 - 0.25*v)*(t - 91.4) + 91.4;
\]

would be the appropriate statement.

**EXAMPLE 2.23** *(a heat loss formula)* Another formula that computes heat loss instead of windchill is given by

\[
double h = (10.45 + 10.0*Math.sqrt(v) - v)*(33.0 - t);
\]

where \(v\) is the wind speed in meters per second, \(t\) is the temperature in degrees Celsius, and \(h\) is the heat loss in kilo calories per square meter per hour.

**EXAMPLE 2.24** *(the power function)* The expression \(x^{2/3} + y^{2/3}\) can be computed with the declaration statement
double d = Math.pow(x, 2.0/3.0) + Math.pow(y, 2.0/3.0);

assuming that the double variables x and y have been assigned values.

**Example 2.25** (investment example using the power function) If $r$ is the interest rate in percent per year, $m$ is the number of times interest is compounded per year, $a$ is the initial investment, and $n$ is the number of years, then the Java translation of the value

$$v = a \left(1 + \frac{r}{100m}\right)^{mn}$$

of the investment after $n$ years is

```java
double v = a * Math.pow(1.0 + r / (100.0*m), m*n);
```

using the Math.pow function in the Math class. The inner parentheses are very important here. Without them it would be a division by 100.0 followed by a multiplication by $m$, and not a division by 100.0*$m$ as required. Also both arguments of the pow function are of type double. We specified $m*n$, which is an int, as second argument and the compiler does an implicit type cast to convert it to a double argument value.

**Example 2.26** (using exp, sin, and cos) You may have seen expressions such as $e^{-3x}\cos x - 2e^{-2x}\sin x$ in calculus. The Java translation is

```java
Math.exp(-3.0*x)*Math.cos(x) - 2.0*Math.exp(-2.0*x)*Math.sin(x)
```

assuming that the double variable $x$ has been assigned a value.

**Example 2.27** (generating random integers) The Math.random() method returns a random double precision number $r$ such that $0 \leq r < 1$. For example the following statement

```java
int number = (int) (10 * Math.random()) + 1;
```

generates a random integer in the range 1 to 10 and assigns it as the value of number.

First the integer 10 is converted to a double value, and the double precision multiplication is performed to give a value $r$ in the range $0 \leq r < 10$. Then the type cast converts this by truncation to an integer $i$, in the range $0 \leq i \leq 9$, and 1 is added to give an integer in the range in the range $1 \leq i \leq 10$, which is assigned to number. The parentheses around $10 * Math.random()$ are necessary since the type cast (int) is always applied to the value on its immediate right, which would be 10 without the parentheses.

### 2.5.2 Rounding floating point numbers

We have seen how to truncate floating point numbers to obtain integers using a type cast in Example 2.18 and Example 2.27. Sometimes it is necessary to round floating point numbers to the nearest integer. The Math.round function will round a double number to the nearest integer. The return type is long not int since rounding a double may produce a value that will not fit in a 32-bit integer but will always fit in a 64-bit long integer.
**Example 2.28** (rounding to an integer) Consider the following statements in BeanShell:

```java
bsh % int i, j, k;
bsh % i = (int) Math.round(123.45);
<123>
bsh % j = (int) Math.round(123.56);
<124>
bsh % k = (int) Math.round(-123.56);
<-124>
bsh % k = Math.round(123.56);
// Error: Typed variable: k: Can't assign long to int: ...
```

The rounded value of 123.45 is 123 as a 64-bit integer, the rounded value of 123.56 is 124 as a 64-bit integer, and the rounded value of -123.56 is -124 as a 64-bit integer. The final assignment statement shows that it is necessary to use the typecast `(int)` to force conversion from `long` to `int`.

**Example 2.29** (rounding to two decimal places) A double precision variable `x` can be rounded to two digits after the decimal point using a declaration statement such as:

```java
double x2 = Math.round(x * 100.0) / 100.0;
```

If `x` has the value 123.4567 then `x * 100.0` has the value 12345.67, and `Math.round` converts this to the long integer 12346, and division by 100.0 produces the double number 123.46.

### 2.5.3 Mathematical function prototypes

In Java functions are called methods. To understand how to use one of the mathematical functions in the `Math` class, we need to know (1) the name of the method, (2) the formal arguments and their types, if any, and (3) the type of value that is computed and returned by the function.

For example, the function `Math.pow` computes $x^y$, so it needs two arguments which we can call `x` and `y`, they are both double numbers, and the value returned has type `double`. This tells us immediately that the assignment statement in Example 2.25 has the correct form (syntax), recalling that the compiler will implicitly convert the `int` argument $m*n$ to a `double` value.

In Java the **method prototype** is used to give a compact description of the rules for using the method. For the power function the prototype is:

```java
static double pow(double x, double y)
```

The word `static` means that this method is not associated with any object (see Chapter 3). Next we have the return type (`double`), and then the method name (`pow`). Inside the parentheses there is a list of argument types followed by some names (`x` and `y`) to refer to the arguments.

Each type-name pair such as `double x` or `double y`, which appear in the method prototype, is called a **formal argument** (the word parameter is often used as a synonym for argument).

Each value supplied when the method is called is referred to as an **actual argument**. The process of using a method prototype to determine how to use a method is illustrated in Figure 2.4 for the declaration statement in Example 2.25. Here the rightmost two arrows show the corre-
double v = a * Math.pow(1.0 + r/(100.0*m), m*n );

Figure 2.4: Matching actual and formal arguments

spondence between the formal arguments and the actual arguments. The actual argument, $1.0 + \frac{r}{(100.0*m)}$, is an expression that corresponds to the formal argument $x$, and the actual argument, $m*n$, is an expression that corresponds to the formal argument $y$. The leftmost arrow shows that the value of the expression

$$\text{Math.pow}(1.0 + \frac{r}{(100.0*m)}, m*n)$$

is a double number, so it makes sense to multiply it by the double number $a$.

The important idea here is that we can look at a statement, such as the one in Example 2.25, and immediately see, by looking at the prototype, that it is a valid use of the method. Each of the methods in the Math class has a prototype.

**Example 2.30** (some math function prototypes) Here are prototypes for some of the arithmetic functions we have used in the preceding examples:

```
static double sqrt(double x)
static double pow(double x, double y)
static double sin(double x)
static double cos(double x)
static double tan(double x)
static double exp(double x)
static double random()
static long round(double x)
```

Most of these functions take a single formal argument of type double and return a double value, except for pow, random and round. The random function takes no arguments, but the empty set of parentheses is still needed when calling the function (See Example 2.27).

The prototype for round clearly shows, as mentioned above, that the return value is of type long, not int, so that no information is lost in the rounding.

### 2.6 Terminology introduced in this chapter

In this section we give simple definitions of the most important concepts introduced in the Chapter.
simple identifier
A sequence of one or more letters, digits and underscores such that the first character is not a letter. Identifiers are used to give names to variables, classes and other entities.

An almost universal convention is to begin the name of a class with an upper case letter. All other identifiers begin with a lower case letter. In either case capitalize the beginning letter of each interior word.

Identifiers are case sensitive. The only example of a class we have seen so far is the Math class.

Examples: radius, numberOfStudents, Math

numeric literal
A value representing a number such as an integer or a floating point constant in fixed or scientific form

Examples: 1, -34 are literals of type int.
Examples: 1L, -3456789212231L are literals of type long.
Examples: 1.0, -3.4, -3.4D, 4.5, 4.5D, 4.5d, 1.23E-04 are double literals. The suffix d or D is optional. An exponent (power of 10) is denoted by e or E.
Examples: 1.0f, -3.4f, -3.4F are literals of type float. The suffix f or F must be present.

variable
A named storage location that can hold a value of some type.

Examples: radius

type
A specific kind of data such as the set of all integers or the set of all real numbers.

Examples: int, float, double

variable declaration
A statement having one of the forms

```java
typeName identifier;
typeName identifier = expression;
```

where identifier is the name of the variable, typeName is the variable type and expression is an expression that evaluates to a value that can be assigned to the variable.
2.6 Terminology introduced in this chapter

Example: double radius;
Example: double radius = 2.0;
Example: double area = Math.PI * radius * radius;
Example: int n=123, remainder, hundreds, tens, units;
Example: double area, circumference;
Example: double radius = 3.0, area;

The final three examples show that multiple variables of the same type can be declared and optionally initialized in a single declaration.

**constant declaration**

A constant has the form

```
static final typeName identifier = expression;
```

The strange keyword `static` indicates that constants are associated with the class, not the objects of the class. The equally strange keyword `final` distinguishes a constant declaration from an initialized variable declaration.

It is conventional to name constants using upper case letters and the underscore to simulate a space.

**Example:** final double CM_PER_INCH = 2.54;

**arithmetic expression**

An expression involving variables and operators that evaluates to a numeric value.

**Example:** radius
**Example:** 2.0 * Math.PI * radius
**Example:** remainder % 10

**assignment statement**

A statement of the form

```
identifier = expression;
```

where `identifier` is the name of a variable that has already been declared and `expression` is an expression whose value is assigned to the variable.

**Example:** radius = 2.0;
**Example:** area = Math.PI * radius * radius;
**Example:** a = b = c = 0.0;

The last example shows that several variables can be assigned the same value in a multiple assignment statement.
2.7 Review exercises

- **Review Exercise 2.1** Define the following terms and give examples of each.

<table>
<thead>
<tr>
<th>Term</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>data type</td>
<td>Q-R theorem</td>
</tr>
<tr>
<td>mod</td>
<td>floating point number</td>
</tr>
<tr>
<td>single precision</td>
<td>primitive types</td>
</tr>
<tr>
<td>round-off error</td>
<td>double precision</td>
</tr>
<tr>
<td>scientific format</td>
<td>underflow</td>
</tr>
<tr>
<td>rounding operation</td>
<td>fixed point format</td>
</tr>
<tr>
<td>uninitialized variable</td>
<td>variable</td>
</tr>
<tr>
<td>identifier</td>
<td>keyword</td>
</tr>
<tr>
<td>precedence rules</td>
<td>increment operator</td>
</tr>
<tr>
<td>type casting</td>
<td>implicit type conversion</td>
</tr>
<tr>
<td>strongly typed</td>
<td>numeric literal</td>
</tr>
<tr>
<td>constant declaration</td>
<td>arithmetic expression</td>
</tr>
<tr>
<td>character literal</td>
<td>boolean literal</td>
</tr>
<tr>
<td>binary operator</td>
<td>integer division</td>
</tr>
<tr>
<td>Math class</td>
<td>method prototype</td>
</tr>
<tr>
<td>actual argument</td>
<td>formal argument</td>
</tr>
</tbody>
</table>

- **Review Exercise 2.2** Express the following numbers as Java literals of type float:

  (a) 1234567,          (b) \(1.9 \times 10^{-37}\),          (c) 0.000045659043,   (d) 3.14159

- **Review Exercise 2.3** Express the following numbers as Java literals in scientific format of type double:

  (a) 1234567,          (b) \(1.9 \times 10^{-37}\),          (c) 0.000045659043,   (d) 3.14159

- **Review Exercise 2.4** Translate the following mathematical expressions or formulas into Java

  (a) \(\sqrt{x^{3/2} + y^{3/2}}\)  
  (b) \(x^2 + y^2 \left(\frac{1}{1+x^2}\right)^{1/2}\)  
  (c) \(\pi a \sqrt{a^2 + h^2}\)  
  (d) \(3.2 + 4.7x + 3.2x^2 - 7.5x^3\)  
  (e) \(e^{3x+2y} \sin(x + 4y)\)  
  (f) \(\frac{x^2}{1 + \sqrt{1+x^2}}\)  
  (g) \(\frac{\tan x + \tan y}{1 - \tan x \tan y}\)  
  (h) \(m \left(\frac{y^2}{L} + g \cos \theta\right)\)  
  (i) \(c \left(\frac{x}{|x|} - \frac{x}{(x^2+y^2)^{1/2}}\right)\)

- **Review Exercise 2.5** What are the differences among the three statements

  ```java
  int width;
  int width = 5;
  width = 5;
  ```

- **Review Exercise 2.6** Why would the compiler complain about the statements
int totalCents = 3527;
int dollars = totalCents / 100.0;

Write the correct statements.

**Review Exercise 2.7** Give short answers to the following questions.

(a) What is the purpose of an assignment statement?

(b) What is the difference between the equal sign in an assignment statement and the equal sign that is often used to denote an equation?

(c) When does \( x = x + 1 \) make sense?

**Review Exercise 2.8** Give short answers to the following questions.

(a) Why do round-off errors occur?

(b) What is overflow and how does it occur?

(c) What is underflow and how does it occur?

(d) What are the differences in Java among 0.1F, 0.1, and 0.1D?

(e) If no suffix is specified what does the compiler assume about a number’s type?

(f) In a mathematical expression, how would you change the order of precedence so that an addition operation is performed before a multiplication operation?

(g) In an expression in which an integer is multiplied by a double number, what type does the compiler assign to the result?

(h) Why does the compiler not automatically convert a double value to an int value?

(i) When truncating a value of type double by type casting to a value of type int, what error can occur?

(j) The prototype for the sqrt method indicates that the argument is of type double. Why does the method call expression Math.sqrt(2), which uses an int argument not cause a compiler error?

(k) Give some examples of implicit type conversions.

(l) What is the difference between explicit type conversion and implicit type conversion?

**Review Exercise 2.9** What is the largest number that can be correctly multiplied by itself before integer overflow occurs?
2.8 BeanShell exercises

- **BeanShell Exercise 2.1 (Evaluating mathematics formulas)**
  For each expression in Review Exercise 2.4 pick some values for the variables and evaluate the expression.

- **BeanShell Exercise 2.2 (Converting inches to feet and inches)**
  Write some statements that define a number of inches as an int variable totalInches and convert this number of inches to feet and inches and print the result. For example, 67 inches is 5 feet and 7 inches.

- **BeanShell Exercise 2.3 (Converting floating point hours to hours, minutes, and seconds)**
  Write some statements that define a double variable totalHours initialized to some floating point number of hours such as 3.245, convert it to hours, minutes and the nearest second, and display the results. For example, 3.245 hours is 3 hours, 14 minutes, and 42 seconds.

- **BeanShell Exercise 2.4 (Astronomy calculations)**
  In astronomy, angles are measured in degrees, minutes (1/60 degree), and seconds (1/60 minute). Write some statements that define a given angle specified by three integers (degrees, minutes and seconds), convert it to a floating point angle, calculate the sine of this angle, and display the results. For example, for integer values 3, 15, and 45 the floating point angle is 3.2625 and the sine of this angle is 0.056910601485907715.

- **BeanShell Exercise 2.5 (Celsius to Fahrenheit temperature conversion)**
  Write some statements that define a temperature in degrees Celsius as a double value, convert it to degrees Fahrenheit, and display the converted temperature.

- **BeanShell Exercise 2.6 (Fahrenheit to Celsius temperature conversion)**
  Write some statements that define a temperature in degrees Fahrenheit as a double value, convert it to degrees Celsius, and display the converted temperature.

- **BeanShell Exercise 2.7 (Pythagorean theorem)**
  Write some statements that define the coordinates of two points \((x_1, y_1)\) and \((x_2, y_2)\) as double values, compute the distance between the points, and display the result.

- **BeanShell Exercise 2.8 (Height in metric units)**
  Write some statements that define two int variables for the height of a person in feet and inches, convert the height to centimeters and display the result. For example, someone who is 5 feet 10 inches tall is 177.8 cm tall.

- **BeanShell Exercise 2.9 (Heat loss calculator)**
  Write some statements that define a temperature in degrees Celsius and a wind speed in kilometers per hour as double variables and use the formulas in Example 2.22 and Example 2.23 to compute and display the wind chill temperature and the heat loss. To use the heat loss formula in Example 2.23 you will have to convert the input wind speed from kilometers per hour to meters per second.

  For example, if the wind speed is 30 km/hr and the temperature is -15 C then the windchill is -33.77635416592807 and the heat loss is 1487.240646055102