COMP 110

Prasun Dewan[1]

6. Types

Perhaps the most important concept in programming is the notion of a type. We have already been using it – when we declared instance variables, we specified what type of variable they were – int or double. In this chapter, we will look more formally at the notion of a type. We will focus on a specific subset of Java types, called primitive types, most of which are derived from the works of Mathematics and logic. We will look both at the values defined for each type, and the operations that can be invoked on these values.

Types

In ordinary life, we tend to "type" people and things to get a better understanding of the world around us. For instance we, might distinguish the student type from the professor type. The distinction is based on the fact that, unlike a student, a professor teaches. Similarly, we might distinguish a bicycle from a motorcycle based on whether the vehicle moves automatically. In both of these examples, we typed the real-world entities based on what actions can be taken by them.

A programming-language type also defines a set of operations. A Java value is of a particular type if all operations defined by the type can be applied to it. For example, the values 3 and 4 are of the type, int, because we can perform int operations such as + and – on them. On the other hand, the strings "three" and "four" are not of this type, because we cannot do the int operations on them. For example, the following is illegal:

"four" – "three"

Therefore, these two values are of another type, String. The int and String values do, superficially, have a common operation, +. But recall that + is an overloaded symbol that stands for two different operations with very different behavior, addition and concatenation. Even the arithmetic operations are overloaded. For instance, the / operation has different meanings for int and double values: 3/4 is 0 whereas 3.0/4.0 is 0.75. It is because of such differences that double and int are different types.

By this definition, each class and interface is also a type, because it defines a set of operations. Let us consider typing three example instances of some of the classes we have created so far (Figure 1):

`ABMICalculator` instance, `ABMISpreadsheet` instance, `AnotherBMISpreadsheet` instance

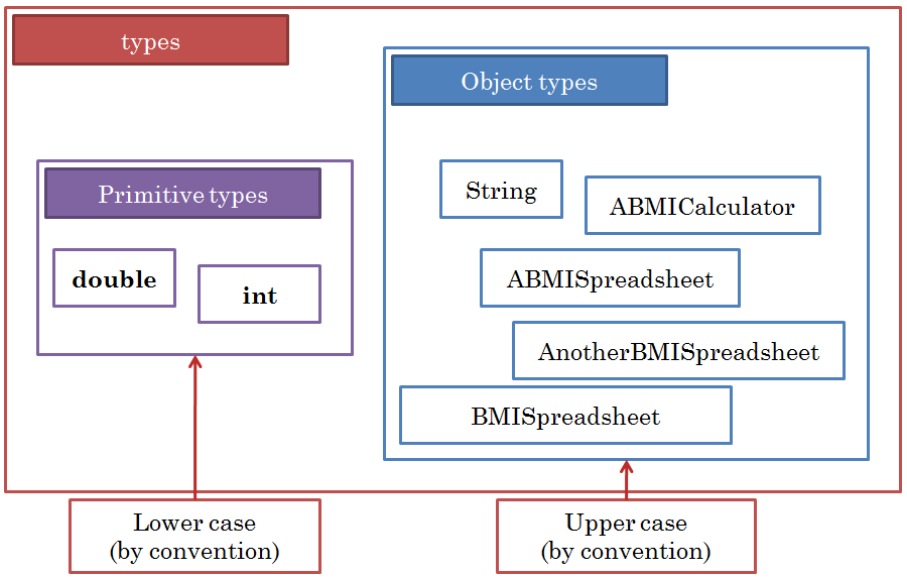

Figure 1. Examples of primitive and non-primitive (object) types

An instance of a `ABMICalculator` is different from the other two objects because we cannot invoke, for example, the operation `setWeight` on it. The other two objects, even though they are instances of different classes, can be considered to be of the same type, because the same set of operations, defined in the interface `BMISpreadsheet`, can be invoked on them.

In Java, not every type is a class or an interface. Some types such as int and double are special types that are neither classes nor interfaces. These types are called *basic* or *primitive* types (Figure 1), because they are building blocks for creating other types. To distinguish them from the other types, Java begins their names with a lowercase letter. Values of these types are not considered objects; instead, they are called *primitive values*. Variables/properties that are set to such values are called *primitive variables/properties*. Classes and interfaces are called object types, because they type objects, and variables/properties of these types are called *object variables/properties*.

In this chapter, we will look at primitive types provided by Java. Before we do so, we need to understand abstract values and operations, and their concrete representation in a program.

Abstract Value vs. Representation

It is important to distinguish between a value of a particular type and its program representation. A value is an abstract entity, while its representation is a concrete sequence of characters that identifies it in a program. A value may have multiple representations. For instance, the abstract double instance 2.2 can be represented by each of the following representations:

2.2 02.2 002.200 0.22E+1 LBS_IN_KG

For each of the primitive types we study, we will look at the predefined (literal and name) constants defined by the type to represent values of the type.

Table 1. Infix, Prefix, and Method Invocation

Operation	Kind	Usage
+	Binary Infix Operator	5 + 4
-	Unary Prefix Operator	-5
round	Method	Math.round(weight)

Syntax of Operation Invocation

Just as we must worry about the program representation of an abstract value, we need to also consider how we invoke abstract operations from the program. Consider the three operations shown in Table 1.

Each uses a different syntax for operation invocation. In the first one, the operation is binary and the operation name appears between the two operands. This syntax is called the *infix* syntax. In the second case, the operation is a unary operation and the operand appears after the operation name. This syntax is called the *prefix* syntax. In both cases, the operation specification is an *operator*, that is, consists of one or two symbols rather than a Java identifier. The third syntax is like the prefix syntax, but the operation is specified by a series of dot separated identifiers and operands (actual parameters) are enclosed within parentheses. This is the *method invocation syntax* we have used before.

As this discussion shows, there is no fundamental difference between operators such as + and methods such `Math.round`. Both take zero or more operands and perform some action on them. The differences are only syntactic. In fact, in some languages such as C++ allow programmer-defined operators. Therefore, we will use the term operation to refer to both operators and methods. In both cases, we will refer to the operands of the operations as actual parameters.

For each of the primitive types we study, we will study the abstract operations defined by it and the concrete syntax used to invoke these operations.

Operation Signature

When invoking an operation, it is necessary to know the types of the operands and result type. If the operation is a method, then the header of the method provides this information. On the other hand, if the operation is an operator such as + or –, there is no such information. Therefore, it is useful to use another formalism, called a *signature*, to specify this information. A signature of an operation has the form:

`<Operand 1 Type>, <Operand 2 Type>, … → <Result Type>`

It is, thus, a comma separated list of the types of the operands of the operation, followed by the symbol, →, followed by the type of the result type. For example, the signature of the int + is:

`int, int → int`

because it takes two int operands and produces an int result. Similarly, the signature of the function, `square`, of Chapter 2, is:

`int → int`

The list of operand types in a signature is null if the operation takes no arguments. For example, the signature of `getWeight` is:

→ `double`

Moreover, the result type is `void` in the case of procedures. For example, the signature of `setWeight` is:

`double` → `void`

Types and Assignment

A fundamental issue related to typing is what type of values can be assigned to a variable. To understand why this is an issue, consider the following assignments:

```
int weight = 2;
int weight = 2.5;
double height = 2;
int weight = "seventy";
```

Which of these do you think are legal? What is the general rule you used to determine the legality of an assignment? We will answer these questions for primitive types in this chapter, postponing assignment of object types for later.

In the rest of the chapter, we will first look at the literals and named constants used to specify values of various primitive types, then the assignment rules for these types, and finally at the operations that can be applied values of these types.

int

This type corresponds to the set of integers we have studied in Mathematics. A Java value of type int is an integer. However, not every integer is a Java int. This is because integers are whole numbers in the range:

$$-\infty \quad .. \quad +\infty$$

However, a Java int must fit in memory for the computer to process it, and computer memory is finite (though very large). The range of integers mapped to int values, is, thus, determined by the size of the memory slot created for each int. In Java, this slot consists of 32 binary digits or *bits* Figure 2. This means that a range of 2^{32} integers can be mapped to instances of int. Half of the numbers in this range (2^{31}) are negative integers, one is zero, and remaining (2^{31} - 1) are positive integers. Thus, instances of int are integers in the range:

$$-2^{31} \; .. \; (2^{31} - 1)$$

or

-2147483648 … 2147483647

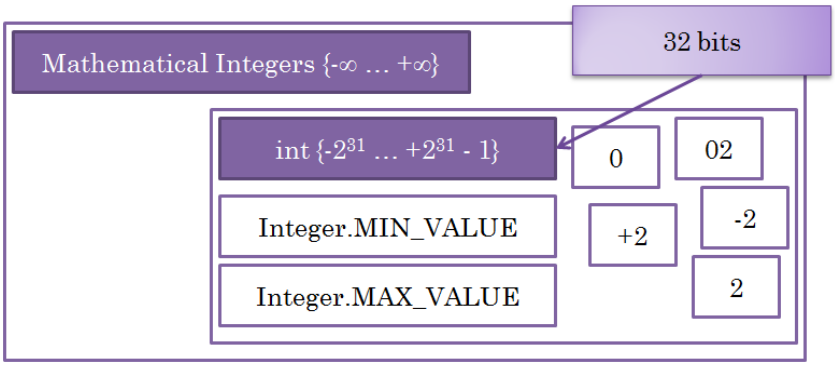

Figure 2. int literals and named constants

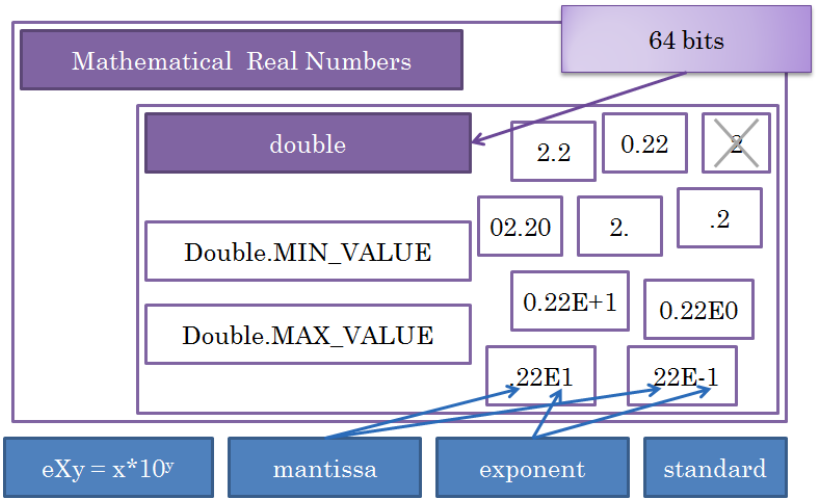

Figure 3. double literals and named constants

We do not have to memorize the values of the smallest and largest integers, since these are stored in the Java named constants, `Integer.MIN_VALUE` and `Integer.MAX_VALUE`, respectively.

We can specify an int values in program using their decimal representations:

2 02 1 234 -3 +10 -2147483648 2147483647

In other words, literals for int values are decimal representations of these values.

double

The type double describes a subset of the set of real numbers we have seen in Mathematics. Recall that a real number is a number that can have a fractional part. This type is named double to indicate that the memory slot created for its instances is twice the size of the memory slot created for an int, that is, it is 64 bits long. The Java predefined named constants, `Double.MIN_VALUE` and `Double.MAX_VALUE` define the absolute value of the smallest (non-zero) double value and largest double value, respectively.

In Java, in the standard representation, we can use a mantissa of up to seventeen significant decimal digits and exponents in the range -323..309. The following numbers are in range:

.12345678901234567
.12345678901234568
0.1E309
0.1E-322

but not the following:

. 123456789012345678
0.1E+310
0.1E-324

Java disallows us from entering exponents that are not in range. Moreover, it rounds off the mantissa to the first seventeen digits. Thus,

System.out.println (.123456789012345678)

prints:

.12345678901234568

As in the case of int values, decimal representations of double values serve as literals for them. However, not all decimal representations of these values are valid literals for them – only those that do not conflict with the literals of equivalent integers. We cannot use, for example, the representation:

2

for the abstract double value, two, since it denotes the int value, 2.The reason for this restriction is that Java must be able to uniquely type each value so that it knows what operations can be applied to it. Therefore, in the literal for a double value, a dot is required, even if there is no fractional part. The following are valid double literals:

2.2 2.0 0.22 02.20 -0.22 +2.2 .2 2.

We can also use scientific representations of these numbers as literals:

0.22E+1 2.2E0 22E-1 - 0.22E+1

where:

$$xEy = x * 10^{y}$$

As shown above, a double value can be represented in multiple ways. The standard or *floating point* representation is a scientific representation in which the number to the left of the exponent is a fraction beginning with a non-zero digit. Thus, the following is the standard representation of 2.2:

.22E1

The following is the standard representation of .022:

.22E-1

The digits to the left of the exponent are called the *mantissa* and to the right are called the *exponent* (Figure 3).

So far, we have looked only at the values of primitive types. To fully understand these types, we must also look at the operations that can be performed on these values.

Other Number Types

32 bits for an integer and 64 bits for a real number may be too little or too much for the integer and real numbers we wish to process in a particular program. Therefore, Java provides additional types with different number

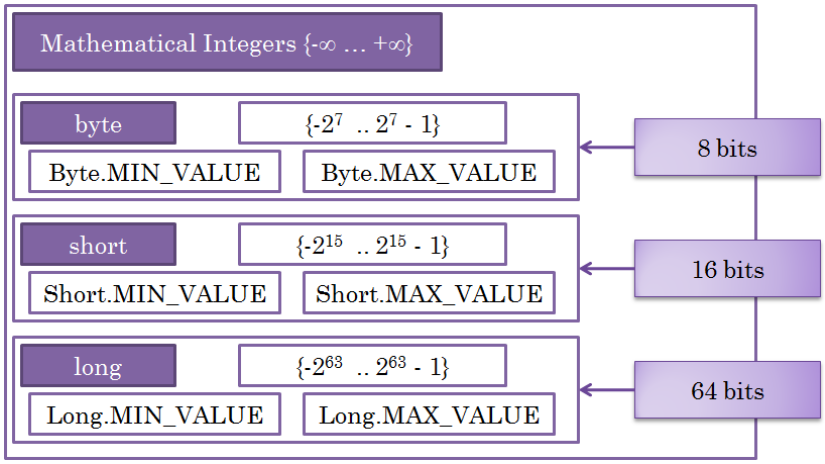

Figure 4. byte, short, and long named constants

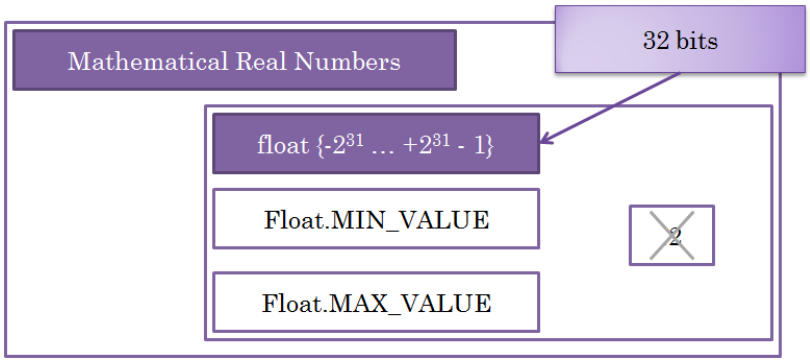

Figure 5. float named constants

of bits for these numbers. It defines the types, byte, short, and long for storing integer values in 8, 16, and 64 bits, respectively.

While Java does provide predefined constants for determining the range of values that can be represented by instances of these additional types, it does not define any literals to represent these values. This may seem strange. Why not, for example, define 2 as a long literal, and 2.2 as a float literal? The reason is the same as the one for not defining 2 as a double literal. Each literal must be typed uniquely, and the literals `2` and `2.2` have already been taken by int and double respectively.[2]

Assignment Rules and Casting

If these additional number types define no literals, how can we assign values to variables of these types? For example, how can we store the value `70` in the long variable, `weight`? Actually, we can do the obvious:

[2] It would be possible to define long literals for those integers that are not int literals, such as 214748364812. However, Java does not do this, defining no literals at all for long.

```
long l  = 70;
```

Java has the ability to convert a value of any number type to a corresponding value of another number type. For example, in the assignment above, it converts the int 70 to a corresponding long value and stores this in the long variable weight. Similarly, in the assignment:

```
double d = 70;
```

Java automatically converts the int 70 to the corresponding double 70.0 and assigns this value to the double variable:

```
double d = 70.0;
```

These two assignments are safe because for every int value there is a corresponding long or double value that represents the same abstract number. However, the reverse is not always true. Therefore, the following assignment is not safe

```
int i = 70.6;
```

and will not be allowed by Java. On the other hand, the following assignment will be allowed by Java:

```
int i =  (int) 70.6;
```

The code fragment, (int), asks Java to cast or coerce the double 70.6 to an int value. When we cast a double as an int, Java truncates the double, that is, gets rid of the fractional part.3 Thus, the above assignment is equivalent to:

```
int i =  70;
```

When we want to deal with whole numbers, truncation is desirable, but in other situations it is not. It is for this reason that Java does not automatically do the corecion for us. By explicitly casting, we are telling it that we know what we are doing and are willing to accept any negative consequences.

At this point, we are ready to understand the general assignment rules for primitive types. We can assign a primitive expression `E` to a primitive variable `v`:

- if the type of expression `E` is the same as or narrower than the type of `v` – otherwise it cannot be stored in `v`.
- if the type of `v` is narrower than the type of E, and we have explicitly cast the type of `E` to the type of `v`.

[3] It may also have to truncate the whole number part since the whole number part in a double can exceed (be smaller than) the maximum (minimum) int. Thus:

```
(int) (Integer.MAX_VALUE + 1.0)  == Integer.MAX_VALUE
```

A type `T1` is *narrower* than another type `T2` if:

$$\text{Set of instances of T1} \subseteq \text{Set of instances of T2}$$

or, in other words, if every instance of T1 can be mapped to an instance of T2. By this definition, byte is narrower than short, which is narrower than int, which is narrower than long. We will call a type T2 wider than T1 if T1 is narrower than T2.

Between some primitive types, no narrow relationship may be defined. For example, there is no narrow relationship between any of the number types and the type boolean discussed below. Thus, the following statement is illegal:

```
boolean b =  (boolean) 1;
```

We will look the assignment rules for object types later. For now, we will assume that an exact type match is required for non-primitive types such as String. Later we will relax these rules as we better understand non-primitive types.

Rules for Parameter and Return Types

Consider the following call:

```
setWeight(70);
```

It is legal, even though the formal parameter of the method is a double and the actual parameter is an int. Recall that passing an actual parameter in a method call is, in fact, an assignment, because the actual parameter is implicitly assigned to the corresponding formal parameter. Thus, in the call above, an int value is assigned to a double variable, which is allowed without a cast. On the other hand, in the case of a method with an int formal parameter:

```
public void setIntWeight(int newWeight) {
      weight = newWeight;
}
```

we would have to cast a double actual parameter:

```
setIntWeight((int) 77.77);
```

Returning a value in a function is also, in fact, an assignment, to an internal variable whose type is the return type declared in the header of the function. Think of the name of the variable being the same as the name of the function. In fact, in some languages such as Pascal this variable is visible to the programmer. A value returned by a return statement is assigned to this variable.

Consider the following method returning an int value for the weight:

```
double weight;
public int getIntWeight() {
      return weight;
}
```

Table 2. Summary of primitive types

Primitive Type	Values Stored	Size in bits
int	integers	32
long	integers	64
short	integers	16
byte	integers	8
double	real numbers	64
float	real numbers	32
boolean	true & false	8

If the internal variable were exposed, this method would be equivalent to:

```
double weight;
public int getIntWeight() {
      int getIntWeight = weight;
}
```

Now, according to our assignment rules, the method is illegal because a double value cannot be assigned to an int variable. Put in other words, this method is illegal, because Java cannot automatically convert the return value, the double value, weight, to a value of the return type, int.

Therefore, we must use a cast:

```
public int getIntWeight() {
      return (int) weight;
}
```

boolean

Often our code must test if some condition has been met. For instance, it may wish to test if our BMI is within the recommended range of BMI values. The boolean type defines two values, denoted by the literals true and false, for indicating the success and failure of such a test.

Summary of Primitive Types

Table 2 summarizes the names of the Java primitive types, the kinds of values they store, and the size of the memory slot allocated to their instances. So far, we have looked only at the values of primitive types. To fully understand these types, we must also look at the operations that can be performed on these values.

int Arithmetic Operations

These are the operations we use in Mathematics for adding, subtracting, multiplying, and dividing integers. The following table describes the symbol used for naming each operation, the action it takes, and the types of the operand and result.

In Table 3:

Table 3. Arithmetic operations on ints

Name	Action	Operands & Result Type (Signature)
+	add	int, int → int
-	subtract	int, int →int
-	negate	int → int
*	multiply	int, int → int
/	int quotient	int, int → int
%	int remainder	int, int → int

Table 4. Arithmetic operations on doubles

Name	Action	Operands & Result Type (Signature)
+	add	double, double →double
-	subtract (unary)	→double
-	subtract	double, double → double
*	multiply	double, double → double
/	divide	double, double → double

```
int, int → int
```

indicates that the operation takes two int operands and produces one int result. A description such as this of the types of the operands and result of an operation is called the *signature* of the operation. It is like the header of a method, except that the names of the operation and its formal parameters are not included in it.

The difference between the two "–" operations is that one is a unary minus, returning the result of subtracting its single operand from 0; while the other is the regular, binary minus, subtracting its second operand from the first operand. Thus:

```
-i
```

Is the same as:

```
0 - i
```

where i is an integer variable.

The only non-intuitive operation here is /. In Java:

```
5/2 → 2
```

This is because, as the signature of the operation shows, the type of the result of the operation is an int and not a double. Specifically, the result is the int quotient we get from the division. To find the remainder from the division, we use %:

```
5 % 2 → 1
```

Thus, the following equality does *not* hold true:

```
x == (x/y)*y
```

Instead, the following equality holds true:

```
x == (x / y) * y + (x % y)
```

double Arithmetic Operations

These are like the previous operations, except that they take double operands and produce double results Table 4.

Thus, for every double operation here, there is a corresponding int operation that has the same name and performs "same" computation. For instance, the + operation in both cases adds the two operands.

Why not treat each pair of corresponding int and double operations as a single operation? For example, why not treat the int + and the double + as a single + operation that works for both types of values? These are different computer operations in that different CPU instructions are executed to perform them. For instance, the two + operations process different representations of their operands and results and detect *overflow* differently. An overflow occurs when the result of an operation is too big or too small to fit in the slot allocated for it. To determine if an overflow occurs, the int + must compare the result with `Integer.MAX_VALUE` and `Integer.MIN_VALUE`, while the double + must compare it with `Double.MAX_VALUE` and `Double.MIN_VALUE`. The differences are even more evident when we consider the two / operations. The double operation does a (more) precise division.

```
5/2 →  2
```

5.0/2.0 → 2.5

There is no % operation for doubles.

Thus, each of + ,-, *, and / is an overloaded operators with different implementations for int and double values rather than a single operator that handles both types.

Overflow

In most cases, the value of an overflown result is set to the nearest legal value of that type. Thus,

```
Integer.MAX_VALUE + 1 → Integer.MAX_VALUE
Integer.MIN_VALUE – 1 → Integer.MIN_VALUE
 Double.MAX_VALUE + 1 → Double.MAX_VALUE
```

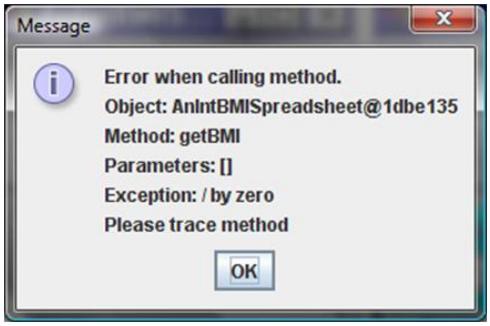

Figure 6. ObjectEditor behavior when dividing an int by 0

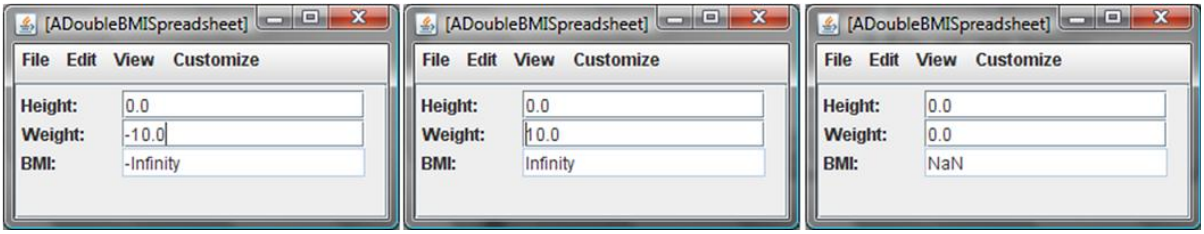

Figure 7. ObjectEditor behavior when dividing a double by 0

Overflow also occurs when a value is divided by zero. Figure 6 shows the ObjectEditor behavior when integer division with a positive, negative, and zero numerator is used. Java behaves somewhat differently than ObjectEditor, In particular, dividing a positive (negative) int by zero results in `Integer.MAX_VALUE` (`Integer.MIN_VALUE`). Dividing int zero by zero evaluates to zero.

Figure 7 shows the ObjectEditor behavior when double division is used. Dividing a positive (negative) double with 0.0 evaluates to a special value, `Double.POSITIVE_INFINITY` (`Double.NEGATIVE_INFINITY`). Dividing a double zero by zero evaluates to another special value, `Double.NaN`. Figure 7 shows the print representation of these three values.

Automatic Conversion and Explicit Casting

What if we wanted to invoke an arithmetic operation on an int and a double:

```
5 / 2.0
```

Since the arithmetic operations provided by Java do not accept mixed types, one of the two operands must be converted to the type of the other. Java automatically converts the operand whose type is *narrower*.

Thus, in the example above, Java would coerce the int 5 to the double 5.0 and call the double / on the converted expression:

```
5.0 / 2.0
```

What if we wanted integer division instead, that is, wanted to convert the double to the narrower type, int? In this case, we have to explicitly cast the double to an int:

```
5 / (int) 2.0
```

To better understand mixing of types, consider the following assignment statement:

```
int i =  (int) (5/2.0)
```

After the division, this statement becomes:

```
int i =  (int) 2.5
```

Next the cast is done, resulting in the following assignment:

```
int i = 2;
```

Now consider the following assignment statement:

```
double d = 5/ (int) 2.0;
```

After the cast, it becomes:

```
double d =  5/ 2;
```

Because both operands are int values, int division is used, even though the expression is being assigned to a double variable:

```
double d  = 2;
```

Finally, the RHS is automatically converted to the wider type:

```
double d = 2.0;
```

Even though the LHS of the assignment is a double, the truncated result is stored in the variable. When evaluating the RHS, Java does not look at the LHS.

To further understand assignment of mixed types, suppose we wished the weight property to be an int rather than a double.

We may consider changing just the type of the instance variable, `weight`. However, Java will complain when compiling `setWeight`:

```
int weight;
public void setWeight(double newWeight) {
      weight = newWeight;
}
```

because the method assigns the double `newWeight` to the int instance variable. The problem of course is that the type of the variable has been changed without changing the type of the property. What we should do then, is to make the weight property also an int by changing the return type of the getter method and the formal parameter of the setter method:

```
public int getWeight() {
	return weight;
}
public void setWeight(int newWeight) {
	weight = newWeight;
}
```

Now consider what happens if we made the property an int, by keeping the getter and setter methods above, but made the instance variable a double again:

```
double weight;
```

The setter method compiles fine this time, because we can always store an int value in a double variable. However, the getter method does not compile this time, because it tries to automatically convert the double variable to an int return value. It will not, however, complain if we explicitly convert the variable to the return value:

```
public int getWeight() {
	return (int) weight;
}
```

This cast is always safe, that is, we are guaranteed it will result in no loss of information. The setter method never stores a fractional value in the instance variable; thus there is no fractional part to truncate when the value is converted back to an int.

This is not the case, however, if we cast the return value of BMI to an int

```
public double getBMI() {
	return (int) weight/(height*height);
}
```

Here the double division operation is used, because the divisor is a double. This value is then truncated when it is cast to an int, and then the truncated value is converted back to a double when it is returned as a double.

What if we had cast the divisor rather than the result of the division?

```
public double getBMI() {
	return (weight/(int)(height*height));
}
```

The cast truncates the divisor in the process of converting it into an int. Since both the numerator and denominator are int values, integer division is used, which truncates the result also, before returning it as a double. Thus, this time, truncation happens before and after the division, yielding a different result with no fractional part.

Strong vs. Weak Typing

As we have seen in this chapter, not all expressions or assignments are legal. For instance, we cannot assign a String to an int variable. Not all languages have the notion of an illegal assignment. For instance, in C-based languages, we can assign any expression to any variable. Thus, the following is legal in C:

```
"hello" - 1
```

There is typing in these languages to allow programmers to indicate what kind of values can be assigned to a variable if the program is to work correctly (that is, have implementation-independent semantics). But the language allows arbitrary values to be assigned to a variable.

Using type information to prevent certain assignments is called strong typing, and allowing arbitrary assignments is called weak typing. Weak typing is analogous to saying that second-hand smoking is harmful but is allowed, or not wearing seat belts can cause injuries but is allowed. Strong typing is analogous to banning smoking or requiring seat belts.

Languages tend to support weak typing because ``real programmers'' need it. As it turns out, you can do some tricky, implementation- dependent, things in weakly typed languages. (Just as you can hang yourself out of a window or change seats in mid journey if you do not keep seat belts on all the time.) In these languages expressions such as

```
"hello world" - 1
```

have obscure, implementation-dependent semantics.

Such uses, however, are often mistakes, and Java supports strong typing so that it can automatically catch such mistakes as semantic errors before the program is executed. In a weakly typed language such as C, typing errors such as the ones above have to be caught as runtime logic errors during debugging. Preventing first-time programmers from such errors is one of the main reasons we chose Java as the first language.

Miscellaneous Math Operations

Java provides several additional useful operations, some of which are given in Table 5. They compute several standard mathematical functions, which are invoked on `Math`. Thus, the following statement prints the value of π:

```
System.out.println(Math.pi());
```

Similarly, the following statement raises 5 to the power 3:

```
System.out.println(Math.power(5, 3));
```

It is important to understand the difference between truncation of double values, which we saw earlier, and the round function provided by Math:

```
(int) 5.9          →    5
Math.round(5.9)    →    6
```

Table 5. Miscellaneous mathematical operations

Operations (invoked on Math)	Signature
abs()	double →double, int →int
acos(), asin(), atan(), cos(), sin(), tan()	double →double
pow ()	double, double →double
exp(), log()	double →double
round()	double →long
random(), pi()	→double
sqrt()	double →double

Table 6. Relational Operations on ints and doubles

Name	Action	Signature of int implementation	Signature of double implementation
==	equal?	int, int →boolean	double, double →boolean
!=	not equal?	int, int →boolean	double, double →boolean
>	greater than?	int, int →boolean	double, double →boolean
<	less than?	int, int →boolean	double, double →boolean
>=	greater than or equal?	int, int →boolean	double, double →boolean
<=	less than or equal?	int, int →boolean	double, double →boolean

Table 7. Generalized Signatures

Name	Action	Generalized Signature
==	equal?	T, T →boolean
!=	not equal?	T, T →boolean
>	greater than?	OrderedT, OrderedT → boolean
<	less than?	OrderedT, OrderedT → boolean
>=	greater than or equal?	OrderedT, OrderedT →boolean
<=	less than or equal?	OrderedT, OrderedT → boolean

Since the result of a round is a long, we must cast it before using it as an int:

```
int i  = (int) Math.round (5.9)
```

The reason Math.round returns a long is that an int (32 bits)may not be big enough to fit all the digits of the whole number part of a double (64 bits) For example:

```
(int) Math.round (Integer.MAX_VALUE + 1.0) → (int) Integer.MAX_VALUE
```

On the other hand, a long is as big as a double, and therefore, is guaranteed to fit the result of rounding.

As we need the remaining functions of Math, we will explain exactly what they compute.

Relational Operations

In addition to the overloaded arithmetic operations, Java defines a set of overloaded *relational operations* with both int and double implementations Table 5. Miscellaneous mathematical operations

Operations (invoked on Math)	Signature

abs()	double →double, int →int
acos(), asin(), atan(), cos(), sin(), tan()	double →double
pow ()	double, double →double
exp(), log()	double →double
round()	double →long
random(), pi()	→double
sqrt()	double →double

Table 6. Each of these operations does a test involving two int or double operands and returns a boolean indicating the result of the test.

Thus:

```
5 == 5      →    true
5 == 4      →    false
5 >= 4      →    true
5 != 5      →    false
5 <= 5      →    true
```

Note that we use the two equal symbols, pronounced "equal equal", for equality rather than a single equal because the latter has been reserved for assignment. It is very easy to confuse them; so we must be careful to use the right one. Many subtle errors can be attributed to using = instead of ==. Fortunately, Java catches many of them.

The relational operations can be applied to values of types other than int and boolean, as shown below.

The == and != operations can be applied to values of any primitive type. Thus, the following are legal:

```
true == true       →    true
false != false     →    false
```

The result is always a boolean but both operands have to be of the same (arbitrary) primitive type[4], T, as indicated by the signature:

```
T, T -> boolean
```

We cannot apply other relational operations to arbitrary types, since their operands must be ordered. Thus:

```
true > false
```

[4] In fact, these operations are also applicable to instances of non-primitive types such as String, but their semantics are not very intuitive or useful. When we study non-primitive types in more detail, we will discuss values of these types can be compared. For instance, on a value of type String, which is a non primitive type, we can invoke the equals method to determine if it is the same as its argument string. Thus, after the assignment:

```
String s = "hello world";
```

the expression

```
s.equals("hello world") == true
```

Table 8. Logical Operations on boolean Values

Name(s)	Action	Signature
!	not	boolean →boolean
&&, &	and	boolean, boolean→boolean
||,		or	boolean, boolean→boolean

Table 9. Generalized Signatures

Name	Action	Generalized Signature
==	equal?	T, T →boolean
!=	not equal?	T, T →boolean
>	greater than?	OrderedT, OrderedT → boolean
<	less than?	OrderedT, OrderedT → boolean
>=	greater than or equal?	OrderedT, OrderedT →boolean
<=	less than or equal?	OrderedT, OrderedT → boolean

is illegal since boolean values in Java are not ordered.[5] Values of all other primitive types are considered ordered.

Logical Operations

These are operations on boolean values that produce boolean values Table 8. The `!` operation is the logical *not* operation. It returns the opposite or negation of its operand. Thus:

```
!true  →  false
!false  →  true
```

The && and & operations implement the logical and operation, which returns true if both of its operands are true. Thus:

```
true && true    →  true
true && false   →  false
false && true   →  false
false && false  →  false
```

Similarly, the || and |operation is the logical *or* operation, which returns true if either of its operands is true. Thus:

```
true || true    →  true
true || false   →  true
false || true   →  true
false || false  →  false
```

Short Circuit Evaluation

[5] In some languages such as Pascal booleans are ordered with true > false.

Why does Java provide & and |? When both operands have to be evaluated, the short-circuit evaluation is slower because it does the extra check to see if it can bypass the second evaluation. We recommend that you always use the short circuit operations because we are more interested in preventing errors rather than increasing performance; and in many cases it will indeed be faster.

Another reason for supporting & and | is that, unlike && and ||, they can take int operands, doing a "bit-wise" operation on the binary representation of the operands, with 1 behaving as true and 0 as false. Thus:

101 & 110 == 100

and

101 | 110 = 111.

Consider the following expression:

```
false && (9654.34/323.13  >  32.34)
```

We do not have to evaluate the second boolean operand to determine the result, because if the first operand is false, the result is false.

Now consider:

```
true || (9654.34/323.13 >  32.34)
```

Again, we do not have to evaluate the second boolean operand to determine the result since if the first operand is true, the result is true.

Now we can explain the difference between the two implementations of *and* and *or*. The & and | operations always evaluate both operations; while the && and || operations, called *short-circuit* operations, sometimes only evaluate the first operand. The && operation does not evaluate is second operand if the first one is false, and the || operation does not evaluate the second operand if the first one is true. Which operation is used can influence the result if the second operation raises an exception. Consider:

```
true || (10/0 > 1)
true | (10/0 > 1)
```

In the second case, in some languages, we will get an error because we divided by 0 but in the first case we will not. (In Java, as we saw before, the result of dividing by zero is not an error but instead the maximum or minimum value.) We recommend that you always do short-circuit evaluation.

Creating Expressions

Recall that an expression is a program fragment or "program phrase" that produces a value. As our examples have shown, it can be:

1. A literal such as 1.77.
2. A variable such as `weight`.
3. An invocation of a function, such as `getBMI`
4. An application of predefined Java operators such as + and -.

In this chapter, we have seen several new ways of creating expressions. We have seen new kinds of literal expressions such as:

```
1.6
'A'
true
```

Moreover, we have seen new kinds of operations such as ||, !, <, and %:

```
false || true
```

```
! (true && false)
```

These, and all of the expressions we have seen in this chapter, involved only literals. Expressions are more interesting when they also involve variables:

```
hourlyWage*hoursWorked + BONUS
```

```
(hourlyWage < <MAXIMUM_HOURLY_WAGE) || (hoursWorked < MAX_HOURS)
```

```
! (0 == (i % 3))
```

Notice, that the last expression is different from others in that it involves the use of three different operations: !, <, and %. The result of applying the % operation is used as an operand to the < operation, which in turn provides its result to the ! operation. In this example:

```
(i % 3)
```

is a sub-expression of

```
(1 < (i % 3))
```

which, in turn, is a sub-expression of:

```
! (1 < (i % 3))
```

An expression that can be decomposed into sub-expressions is called a *compound expression*, and one that cannot is called a *simple expression*. Thus:

```
(1 < (i % 3))
```

is a compound expression, whereas

```
1
```

and

```
(i % 3)
```

are simple expressions.

In general, there can be arbitrary sub expression levels in an expression. Here is an example of a compound expression with a particularly large number of levels:

```
!(Math.round(5.5 + (7.4*3.4))) > ((- 'a') + ('A' + 'I'))
```

To evaluate an expression, we should be able to type the results of its subexpressions. For most of the expressions we have seen so far, our mathematical intuition works fairly well in typing them. However, in general, we need to know the signatures of all the operations involved in the expression. For example, given an expression such as:

```
5 < 3
```

we know the type of the result is `boolean` because of the signature:

```
int, int -> boolean
```

Boolean Expressions

An expression such as the one above that evaluates to either true or false is called a Boolean expression. Like arithmetic expressions, Boolean expressions can involve variables:

```
hoursWorked > MAX_HOURS
```

This expression returns true if the value of `hoursWorked` is greater than `MAX_HOURS` and false otherwise. For those of you who know about if statements, notice that this code can return a true or false value without using an if statement. In general, beginning programmers tend to overuse if statements - when you are tempted to use one, see if a simpler boolean expression can suffice.

Like arithmetic expressions, boolean expressions can also be assigned to variables. Thus, we can execute the following statement:

```
boolean  overWorked = hoursWorked > MAX_HOURS;
```

Assigning a Boolean expression may seem unintuitive. While we are used to assigning arithmetic expressions in formulas:

```
bmi = weight / (height * height);
```

we are not used to doing so for Boolean expressions. Assigning a Boolean expression to a variable essentially stores the result of a test in a variable, which can then be looked up later. To use a real-world analogy, the assignment above corresponds to a person writing down the result on a piece of paper, which is later looked up to, for instance, to determine if overtime should be paid.

To further illustrate the use of Boolean expressions, assume we wish to extend our BMI spreadsheet as shown in Figure 8.

Here, we have added a new property, OverWeight, of type boolean, that tells us if our weight is high compared to the height. A boolean value is displayed by ObjectEditor as a checkbox – the box is checked if the property is **true** and not checked it the property is **false**. Let us assume that we are overweight if our BMI is higher than the some named constant, HIGH_BMI, defined by us::

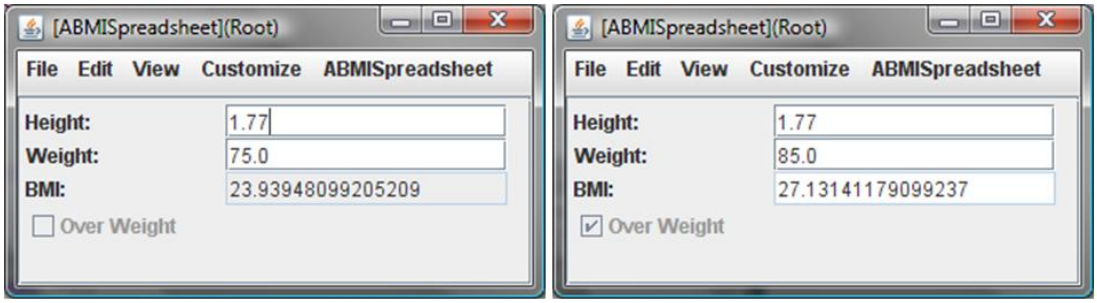

Figure 8. Adding a `boolean` property to ABMISpreadsheet

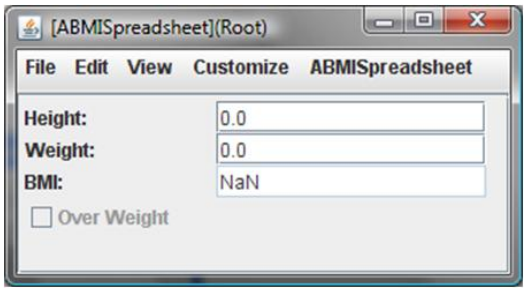

Figure 9. Inconsistent values in ABMISpreadsheet

```
public final double HIGH_BMI == 25;
```

The constant would be defined in the BMI spreadsheet interface, because it is implementation-independent.

The getter method for the property returns the value of a simple relational expression:

```
public boolean isOverWeight() {
            return getBMI() > HIGH_BMI;
}
```

Assertions

To better understand Boolean expressions and an important Java construct called assertion, consider again our original `BMISpreadsheet` shown in Figure 9.

It has the problem that the height and weight have illegal values. As a result, the BMI value makes no sense. We saw that we could use constructors to prevent the initial values of these variables from being inconsistent. But how can we ensure that subsequent illegal changes to these values result in an error message?

Java provides a special statement of the form:

assert <BooleanExpression>

which can be called to tell Java we expect some condition, expressed as a Boolean expression, to hold true. If the condition is false, Java stops the program and tells the user that the assertion is not true. You

might have seen such a message before because it is not uncommon for some product to fail with the message “internal assertion failed”.

The following code shows how assertions can be used to avoid illegal values of BMI:

```
public class ABMISpreadsheet {
      double height, weight;

      public ABMISpreadsheet(
            double theInitialHeight, double theInitialWeight) {
            setHeight( theInitialHeight);
            setWeight( theInitialWeight);
      }

      public double getHeight() { return height; }
      public void setHeight(double newHeight) { height = newHeight; }
      public double getWeight() {return weight; }
      public void setWeight(double newWeight) {weight = newWeight; }
      public boolean preGetBMI() { return weight >= 0 && height >= 0;}
      public double getBMI() {
            assert (preGetBMI());
            return weight/(height*height);
      }
}
```

Here `getBMI` calls a method called, `preGetBMI`, which, in turn, makes asserts the following Boolean expression:

```
weight > 0 && height > 0
```

If this condition is false, the assertion fails and the Java program terminates. An expression such as this that must be true at the start of the method is called a precondition of the method. A method precondition is much like a course pre-requisite. If you don’t meet a course prerequisite, you are not able to completely understand the course. Similarly, if a method prerequisite is not met, the method will not execute correctly.

It is a good idea, as we have done above, to check the precondition of some method `M`, in a separate Boolean public method, called `preM`. This way, another method, `P`, that wishes to call `M` can first call `preM` first to see if the precondition of M is met. If `preM` returns false, then P should not call `M` to halt the program. Of course, you can give any name to the method that checks the precondition. However, a standard naming strategy such as the one we have used above helps makes it possible to find the

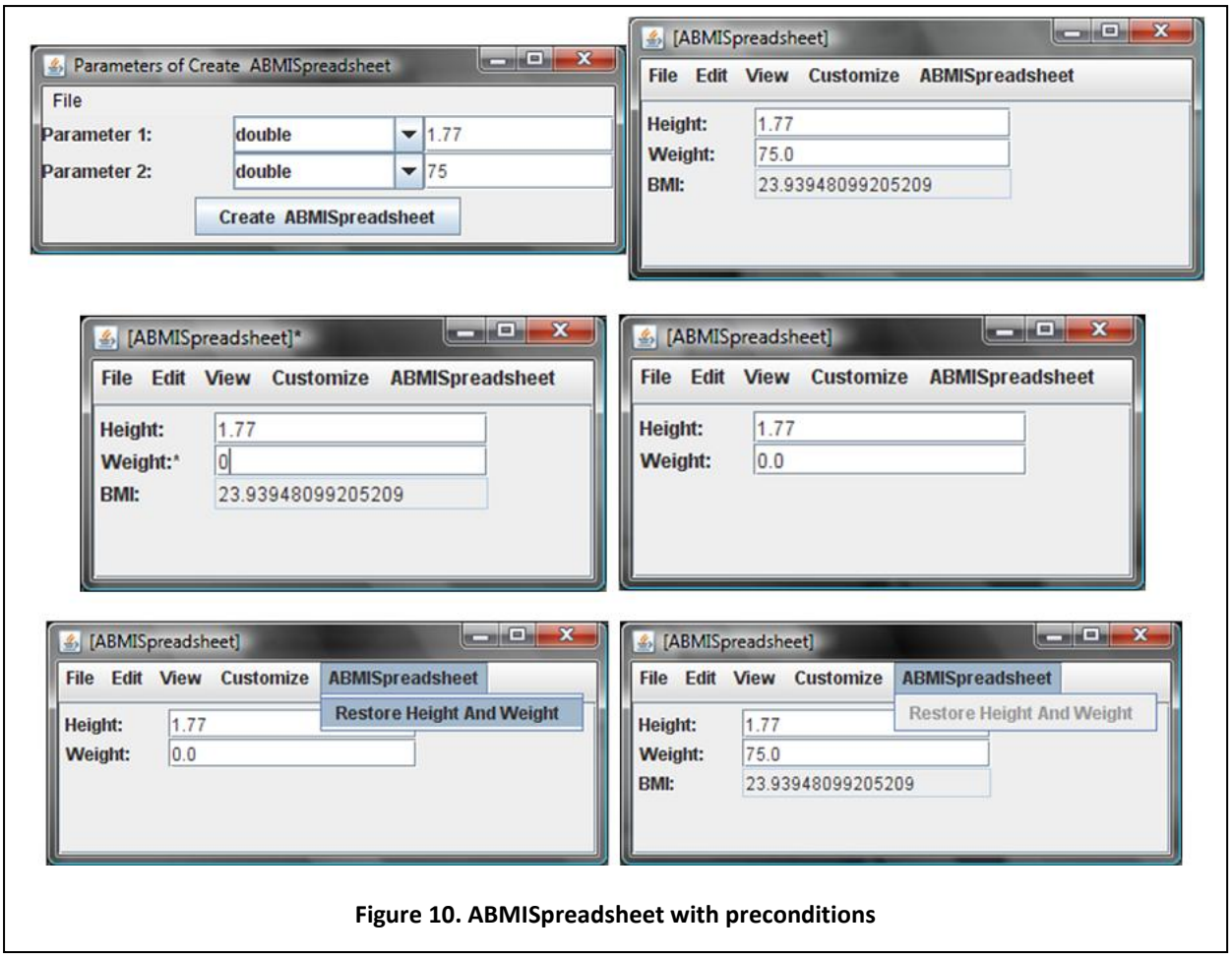

Figure 10. ABMISpreadsheet with preconditions

precondition method without reading documentation. Such a naming strategy is consistent with the idea of using Bean conventions to name getters and setters.

Like Bean conventions, the above naming strategy is used by ObjectEditor to determine if a getter method should be called. This is illustrated in the interaction shown in Figure 10. Suppose we create a new instance of `ABMISpreadsheet` with legal values (Figure 10 top left and right) and then try to enter the illegal value, 0.0, for the weight (Figure 10 middle left). If we had used the previous version of the class, ObjectEditor would have called `getBMI` to refresh the display of BMI. However, as we now have a precondition method for `getBMI`. ObjectEditor calls this method before it tried to call `getBMI`. As this method returns false, it does not call `getBMI`. Since it does not have a legal value for the BMI field, it simply removes the field from the display (Figure 10 middle right). It will redisplay the field when we enter a legal value for weight.

As the above example illustrates, at times we might want to restore the values of instance variables to their original values. We can provide a special method to do so, as shown in (Figure 10 bottom left). Notice that after we execute the command to invoke the method, it is disabled by ObjectEditor (Figure 10 bottom right). This makes sense because the variables already have their initial values. In general, ObjectEditor disables the menu command for a method whenever its precondition is not met. This implies that the newly added method has a precondition method that checks if the current values of the

instance variables are their initial values. This, in turn, implies that the object stores the initial values of the variables. As initial values are given in the constructor, we must change this method to store these values.

The following code gives the new version of the class:

```
public class ABMISpreadsheet {
	double height, weight;
	double initialHeight, initialWeight;
	public ABMISpreadsheet(
		double theInitialHeight, double theInitialWeight) {
		setHeight( theInitialHeight);
		setWeight( theInitialWeight);
		initialHeight = theInitialHeight;
		initialWeight = theInitialWeight;
	}
	public double getHeight() {return height; }
	public void setHeight(double newHeight) { height = newHeight; }
	public double getWeight() {return weight; }
	public void setWeight(double newWeight) {weight = newWeight; }
	public boolean preGetBMI() { return weight > 0 && height > 0;}
	public double getBMI() {
		assert preGetBMI();
		return weight/(height*height);
	}
	public boolean preRestoreHeightAndWeight() {
		return height != initialHeight || weight != initialWeight;
	}
	public void restoreHeightAndWeight() {
		assert preRestoreHeightAndWeight();
		height = initialHeight;
		weight = initialWeight;
	}
}
```

In the above code, we have predefined preconditions for only two of the methods. Should the other methods also have preconditions?

Consider first `setWeight` and `setHeight`. It does not make sense to give a negative value to the weight and height, hence we can assert preconditions that check this condition as illustrated below:

```
		public boolean preSetWeight (double newWeight) {
			return newWeight > 0;
		}
		public void setWeight(double newWeight) {
			assert preSetWeight(newWeight);
			weight = newWeight;
		}
```

```
public boolean preSetHeight (double newHeight) {
      return newHeight > 0;
}
public void setHeight(double newHeight) {
      assert preSetHeight(newHeight);
      height = newHeight;
}
```

We can similarly add preconditions for the getters:

```
public boolean preGetWeight () {
      return weight > 0;
}
public double getWeight() {
      assert preGetWeight();
      return weight
}
public boolean preGetHeight () {
      return weight > 0;
}
public double getHeight() {
      assert preGetHeight();
      return height
}
```

However, once we have checked the preconditions for the two setter methods, we don't need to write preconditions for any of the three getter methods. This is because the setters have made sure that illegal values cannot be assigned to the variables.

Printing Different Types

The `println` method can be used to print values of arbitrary types:

```
System.out.println (2)          output:     2
System.out.println (2.0)              output:     2.0
System.out.println ((int) 2.0)        output:     2
System.out.println ('2')              output:     2
System.out.println ((int) '2')        output:     50
System.out.println (char) 51) output:     3
System.out.println (5 > 0)    output:     true
```

`println` is an overloaded method, and different implementations of it are used for the different types of values, much as + is an overloaded operator, and different implementations of it are used to add int and double values.

Notice the use of the explicit cast:

```
(int) '2'
```

Contrast this with some of the previous that also converted chars to their integer codes:

Table 10. Precedence Levels

! - (T)
* / %
+ -
> < <= >=
== !=
&
|
&&
||

```
int i =  'A'
  'B' - 'A'
```

In the previous examples, Java automatically does the cast for us since otherwise the program fragments would not be legal. It cannot do so in the `println` case, since a `println` without the cast is legal, as shown above. `println` is an overloaded procedure defined for both character and integer arguments. The cast explicitly indicates which definition should be used.

Operator Precedence

In general, it is a good idea to use parenthesis to clearly indicate the sub expressions of an expression. Otherwise, what we type can be ambiguous. For instance, does

```
5 + 5 * 3
```

mean

```
(5 + 5) * 3
```

or

```
5 + (5 * 3)
```

Java has default rules for disambiguating such ambiguous specifications. It divides operators into *precedence levels*, applies operators in a higher precedence levels before operators in lower precedence levels, and applies operators in the same precedence level in a left to right fashion. For instance, it puts * at a higher precedence level than +. As a result

```
(5 + 5 * 3)  == (5 + (5 * 3))
```

Table 10 describes the precedence levels of the operators we have seen so far. In this table, in (T), T is a type name. Thus, (T) is the casting of an expression to type T. Moreover, in this table, the minus in the top level stands for the *unary* minus, while the one in the third level stands for the binary minus. Thus

```
-5 - 4
```

is

```
(-5) - 4
```

and not

```
-(5 - 4)
```

because the unary minus has a higher precedence than the binary minus.

Similarly

```
 !true && false
```

is

```
(!true) && false
```

and not

```
! (true && false)
```

because ! has a higher precedence than &&. Moreover

```
5 / 4 * 3
```

is

```
(5 / 4) * 3
```

and not

```
 5 / (4 * 3)
```

because both / and * have the same precedence.

The following, particularly tricky expression:

```
true || false == false || true
```

is

```
true || (false == false) || true
```

and not:

```
(true || false) == (false || true)
```[6]

because == has higher precedence than ||.

[6] In Pascal, this would be the valid interpretation!

Now, consider the expression:

```
(int) 5 / 2.0
```

The cast applies to the literal, 5, not the result of the division, because casting, (T), has higher precedence than division, /. Casting the `int` to an `int` leaves the value unchanged. Thus, this expression is equivalent to:

```
5/2.0;
```

To cast the result of the division, we should have enclosed it in parentheses:

```
(int) (5/2.0);
```

Finally, consider the expression:

```
i < j < k
```

where i, j, k are int variables.Based on your mathematics intuition, you might translate this into English as:

```
i is less than j, which is less than k
```

However, this is not how Java interprets this expression. It sees two occurrences of the binary operation, <, and uses left to right precedence in parsing them:

```
(i < j) < k
```

The first evaluation of < returns a boolean value, true or false. Thus, the second < is essentially asked to compare a boolean value with an int value, which is illegal. The English assertion above is translated in Java as:

```
i < j && j < k
```

Summary

- The values a program manipulates can be classified into multiple types.
- A type can be an object or primitive type, and an object type can be an interface or class.
- Each primitive type defines literals and named constants to represent its values of the type in a program, and a set of operations on these values.
- The types int, short, and long define different subsets of integers, byte defines a subset of whole numbers (unsigned integers), and double defines a subset of real numbers. The type boolean defines the logic values, true and false.
- Java provides arithmetic, relational, and logic operations to manipulate these values. Arithmetic operations apply to the number types (int, double, short, byte), relational operations apply to the ordered types (the number types), and logical operations apply to boolean.

- Java allows both implicit and explicit conversion of a value of a particular type to an equivalent value of another type.
- It also provides us methods to output values of different types.

Exercises

1. What are the types of the following literals?

```
2   2.0 true true 2E+1
```

2. Evaluate the following expressions, giving their type:

	Expression	Type	Value
(e.g.)	5 + 3	int	8
(a) 5 / 2	_____	_____	_____
(b) 5 % 2	_____	_____	_____
(c) 5.0 / 2.0	_____	_____	_____
(d) ((int) 5.8) / 2.0	_____	_____	_____

3. Write boolean expressions that are equivalent to the English assertions given below. That is, they should return true if the assertions are true, and false otherwise. An example assertion and corresponding expression are given below. Assume i is an int variable, c1 and c2 are char variables, and b1 and b2 are boolean variables.

   (e.g.) i is 2.

   Answer: i == 2

   (a) i is greater than 5

   (b) i is in the range 2 to 5 inclusive.

   (c) i is an odd number.

   (d) exactly one of b1 and b2 is true.

4. Use the precedence levels of Figure 8, parenthesize the following expressions to show how they will be evaluated:

   (e.g.) 3 + 4 –2

   Answer: (3 + 4) - 2

   a) 9 – 4 / 2
   b) 9 / 4 – 2
   c) 9 < 4 == 4 > 9
   d) true && false != false & true
   e) true == false == false == true
   f) 75.0 / 1.77*1.77

5. Explain the errors in the following expressions:

   (e.g.) “hello” - 1

   Answer: Cannot subtract an int from a String.

   a) true + false
   b) 9 < (4 == 4) > 9
   c) (int) true
   d) 6.0 % 2

6. Which of the following expressions are true?

   a) Integer.MIN_VALUE - 1.0 == Integer.MIN_VALUE

   b) (int) (Integer.MIN_VALUE - 1.0) == Integer.MIN_VALUE

   c) Integer.MIN_VALUE - 1.0 == (double) Integer.MIN_VALUE

7. Suppose we made the instance variable, `weight`, an int. Will the following getter method compile correctly?

```
public double getWeight() {
      return weight;
}
```

8. Assuming `weight` is an int variable assigned the value 75 and height is a double with value 1.77, what value is returned by the following definition of getBMI.

```
public double getBMI() {
      return (weight/((int) height* (int) height));
}
```

9. Add a read-only boolean property, `UnderWeight`, to the BMI example, that is true if the BMI is less than 21 and false otherwise.