So far we have looked mostly at simple cases in which all the numbers involved in a calculation were either all integers or all doubles.

Here, we will see what happens when we mix these types in calculations.
Here is an important principle to remember: Java will not normally store information in a variable if in doing so it would lose information. Consider the following two examples:
Java doesn’t like to lose data – Example 1

• An example of when we would lose information:

```java
double d = 29.78;
int i = d;
//won’t compile since i is an integer and it would have to chop-off
// the .78 and store just 29 in i….thus, it would lose information.
```

• There is a way to make the above code work. We can force compilation and therefore result in 29.78 being “stored” in i as follows (actually, just 29 is stored since i can only hold integers):

```java
int i = (int)d;
//(int) “casts” d as an integer… It converts d to integer form.
```
Java doesn’t like to lose data – Example 2

- An example of when we would **not** lose information:

```java
int j = 105;
double d = j;
//legal, because no information is lost in storing 105 in the
// double variable d.
```
The most precise

- In a math operation involving two different data types, the result is given in terms of the more precise of those two types...as in the following example:

```java
int i = 4;
double d = 3;
double ans = i/d;
//ans will be 1.3333333333333333...the result is double precision
```

20 + 5 * 6.0 returns a double. The 6.0 might look like an integer to us, but because it’s written with a decimal point, it is considered to be a floating point number...a double.
Don’t be fooled

- Consider the following two examples that are very similar...but have different answers:

```java
double d = (double)5/4; //same as 5.0 / 4...(double) only applies to the 5
System.out.println(d); //1.25

int j = 5;
int k = 4;
double d = (double)(j / k); //(j / k) is in its own little “world” and performs
//integer division yielding 1 which is then cast as
//a double, 1.0
System.out.println(d); //1.0
```
Constants

- Constants follow all the rules of variables; however, once initialized, they **cannot** be changed. Use the keyword `final` to indicate a constant. Conventionally, constant names have all capital letters. The rules for legal constant names are the same as for variable names. Following is an example of a constant:

  ```java
  final double PI = 3.14159;
  ```
The following illustrates that constants can’t be changed:

```java
final double PI = 3.14159;
PI = 3.7789; //illegal
```

When in a method, constants may be initialized after they are declared.

```java
final double PI; //legal
PI = 3.14159;
```

Constants can also be of type *String, int and other types.*

```java
final String NAME= "Peewee Herman";
final int LUNCH_COUNT = 122;
```
In the previous lesson we learned that the compound operator expression `j+= x;` was equivalent to `j = j + x;`. Actually, for **all compound operators** there is also an implied cast to the type of `j`.

For example, if `j` is of type `int`, the real meaning of

```
j+= x; is:
j = (int)(j + x);  
```

is:

```
j+= x;  
j = (int)(j + x);  
```