Click here to view code image

```
public static void main(String[] args) {
    int itemsBought = 0;
    int funds = 100;
    for (int price = 10; funds >= price; price += 10) {
        funds -= price;
        itemsBought++;
    }
    System.out.println(itemsBought + " items bought.");
    System.out.println("Cash left over: " + funds + " cents");
}
```

In summary, don't use `float` or `double` for any calculations that require an exact answer. Use `BigDecimal` if you want the system to keep track of the decimal point and you don't mind the inconvenience and cost of not using a primitive type. Using `BigDecimal` has the added advantage that it gives you full control over rounding, letting you select from eight rounding modes whenever an operation that entails rounding is performed. This comes in handy if you're performing business calculations with legally mandated rounding behavior. If performance is of the essence, you don't mind keeping track of the decimal point yourself, and the quantities aren't too big, use `int` or `long`. If the quantities don't exceed nine decimal digits, you can use `int`; if they don't exceed eighteen digits, you can use `long`. If the quantities might exceed eighteen digits, use `BigDecimal`.

Item 61: Prefer primitive types to boxed primitives

Java has a two-part type system, consisting of *primitives*, such as `int`, `double`, and `boolean`, and *reference types*, such as `String` and `List`. Every primitive type has a corresponding reference type, called a *boxed primitive*. The boxed primitives corresponding to `int`, `double`, and `boolean` are `Integer`, `Double`, and `Boolean`.

As mentioned in **Item 6**, autoboxing and auto-unboxing blur but do not erase the distinction between the primitive and boxed primitive types. There are real differences between the two, and it's important that you remain aware of which you are using and that you choose carefully between them.

There are three major differences between primitives and boxed primitives. First, primitives have only their values, whereas boxed primitives have identities distinct from their values. In other words, two boxed primitive instances can have the same value and different identities. Second, primitive types have only fully functional values, whereas each boxed primitive type has one nonfunctional value, which is `null`, in addition to all the functional values of the corresponding primitive type. Last, primitives are more time- and space-efficient than boxed primitives. All three of these differences can get you into real trouble if you aren't careful.

Consider the following comparator, which is designed to represent ascending numerical order on `Integer` values. (Recall that a comparator's `compare` method returns a number that is negative, zero, or positive, depending on whether its first argument is less than, equal to, or greater than its second.) You wouldn't need to write this comparator in practice because it implements the natural ordering on `Integer`, but it makes for an interesting example:

Click here to view code image

```
// Broken comparator - can you spot the flaw?
Comparator<Integer> naturalOrder =
    (i, j) -> (i < j) ? -1 : (i == j ? 0 : 1);
```

This comparator looks like it ought to work, and it will pass many tests. For example, it can be used with `Collections.sort` to correctly sort a million-element list, whether or not the list contains duplicate elements. But the comparator is deeply flawed. To convince yourself of this, merely print the value of `naturalOrder.compare(new Integer(42), new Integer(42))`. Both `Integer` instances represent the same value (42), so the value of this expression should be 0, but it's 1, which indicates that the first `Integer` value is greater than the second!

So what's the problem? The first test in `naturalOrder` works fine. Evaluating the expression `i < j` causes the `Integer` instances referred to by `i` and `j` to be *auto-unboxed*; that is, it extracts their primitive values. The evaluation proceeds to check if the first of the resulting `int` values is less than the second. But suppose it is not. Then the next test evaluates the expression `i==j`, which performs an *identity comparison* on the two object references. If `i` and `j` refer to distinct `Integer` instances that represent the same `int` value, this comparison will return `false`, and the comparator will incorrectly return 1, indicating that the first `Integer` value is greater than the second. **Applying the `==` operator to boxed primitives is almost always wrong.**

In practice, if you need a comparator to describe a type's natural order, you should simply call `Comparator.naturalOrder()`, and if you write a comparator yourself, you should use the comparator construction methods, or the static compare methods on primitive types (Item 14). That said, you could fix the problem in the broken comparator by adding two local variables to store the primitive `int` values corresponding to the boxed `Integer` parameters, and performing all of the comparisons on these variables. This avoids the erroneous identity comparison:

Click here to view code image

```
Comparator<Integer> naturalOrder = (iBoxed, jBoxed) -> {
    int i = iBoxed, j = jBoxed; // Auto-unboxing
    return i < j ? -1 : (i == j ? 0 : 1);
};
```

Next, consider this delightful little program:

Click here to view code image

```
public class Unbelievable {
    static Integer i;

    public static void main(String[] args) {
        if (i == 42)
            System.out.println("Unbelievable");
    }
}
```

No, it doesn’t print `Unbelievable`—but what it does is almost as strange. It throws a `NullPointerException` when evaluating the expression `i==42`. The problem is that `i` is an `Integer`, not an `int`, and like all nonconstant object reference fields, its initial value is `null`. When the program evaluates the expression `i==42`, it is comparing an `Integer` to an `int`. In nearly every case **when you mix primitives and boxed primitives in an operation, the boxed primitive is auto-unboxed.** If a null object reference is auto-unboxed, you get a `NullPointerException`. As this program demonstrates, it can happen almost anywhere. Fixing the problem is as simple as declaring `i` to be an `int` instead of an `Integer`.

Finally, consider the program from page 24 in **Item 6**:

Click here to view code image

```
// Hideously slow program! Can you spot the object creation?
public static void main(String[] args) {
    Long sum = 0L;
    for (long i = 0; i < Integer.MAX_VALUE; i++) {
        sum += i;
    }
    System.out.println(sum);
}
```

This program is much slower than it should be because it accidentally declares a local variable (`sum`) to be of the boxed primitive type `Long` instead of the primitive type `long`. The program compiles without error or warning, and the variable is repeatedly boxed and unboxed, causing the observed performance degradation.

In all three of the programs discussed in this item, the problem was the same: the programmer ignored the distinction between primitives and boxed primitives and suffered the consequences. In the first two programs, the consequences were outright failure; in the third, severe performance problems.

So when should you use boxed primitives? They have several legitimate uses. The first is as elements, keys, and values in collections. You can't put primitives in collections, so you're forced to use boxed primitives. This is a special case of a more general one. You must use boxed primitives as type parameters in parameterized types and methods (Chapter 5), because the language does not permit you to use primitives. For example, you cannot declare a variable to be of type `ThreadLocal<int>`, so you must use `ThreadLocal<Integer>` instead. Finally, you must use boxed primitives when making reflective method invocations (Item 65).

In summary, use primitives in preference to boxed primitives whenever you have the choice. Primitive types are simpler and faster. If you must use boxed primitives, be careful! **Autoboxing reduces the verbosity, but not the danger, of using boxed primitives.** When your program compares two boxed primitives with the `==` operator, it does an identity comparison, which is almost certainly *not* what you want. When your program does mixed-type computations involving boxed and unboxed primitives, it does unboxing, and **when your program does unboxing, it can throw a `NullPointerException`.** Finally, when your program boxes primitive values, it can result in costly and unnecessary object creations.

Item 62: Avoid strings where other types are more appropriate

Strings are designed to represent text, and they do a fine job of it. Because strings are so common and so well supported by the language, there is a natural tenden-