Mr G's Java Jive

#6: Idiot-Proofing and the Gatling Class

With this handout you'll start using the **gatling** class to idiot-proof your programs.

Java Made Easy

As you've probably already figured out, Java is a very powerful language with many different commands in many different classes that allow you to do all kinds of things. The problem is that some of the simplest things you might want to do have commands that aren't very easy to figure out. In a course that insisted on you learning how to use Java the way it is "out of the box," you'd probably be made to figure out how to write the incredibly complex code needed to perform these simple tasks. However, since the point of this course is to introduce you to Java with as little pain as possible and get you writing useful programs as quickly as possible, I've taken a different position.

You should remember that there are two main types of classes: **standalone** and **helper**. So far we've been writing **standalone** classes (and remember, standalone classes must have one and only one **main** method). Now it's time to take a look at a **helper** class. Helper classes are like **libraries** of code written by you or someone else, that have easy to use methods that you can use in your **standalone** programs. Far from being a case of cheating or the "easy way out," this is actually the way that Java was designed to be used.

With that in mind, I've created the **gatling** class for you to use. But before you use it, we have to take a look at some of the many things that can go wrong.

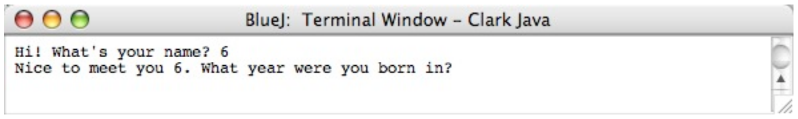
Users and Idiots

It can safely be said that many (if not most) users are idiots. No matter how well-designed a program might be, there is sure to be a few bonehead users who either don't understand how it's supposed to be run, or who delight in specifically ignoring the instructions to see what happens. It is for these people that the saying "Make it idiot-proof, and they'll build a better idiot" was coined.

We're going to take a few minutes to act like idiots, and give the converse program input that it wasn't designed to accept, so we can see what happens. Note that in this case we're not being idiots, but simply acting like them in the name of research.

Bad Input

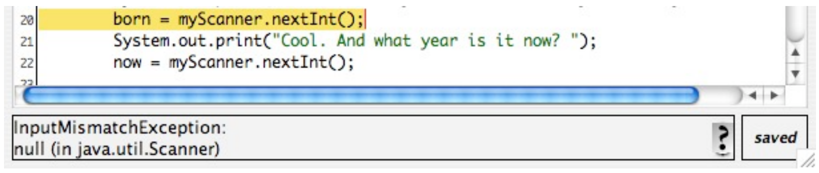
Let's start out with a simple piece of bad input: with a nod to the old Patrick McGoohan TV show **The Prisoner**, we'll put in the user's name as **6**. That's right, the number **6**. Check out what happened below:



```
Hi! What's your name? 6
Nice to meet you 6. What year were you born in?
```

Well, that seems to have run just fine, and now it wants to know what year the user was born in. Let's put in **nineteen sixty-five**. Notice that we're going to input the words **nineteen sixty-five**, and not the number **1965**. What happens now?

Ouch! When we did that, we got kicked right back to the class window, which gave us the error message shown below. But what does it mean?



```
        born = myScanner.nextInt();
        System.out.print("Cool. And what year is it now? ");
        now = myScanner.nextInt();
```

```
InputMismatchException:
null (in java.util.Scanner)
```

Type Mismatch Explained

The program crashed when you tried to put in **words** where it expected a **number**, or rather a **String** where it expected an **int**. Because they were two different **types** it's called a **type mismatch**. BlueJ even highlighted the line in the program that generated the error (even though the error was really user generated).

But wait a minute! You're probably wondering why the program didn't crash and give us a **type mismatch** error the first time, when we put in the user's name as **6**. After all, that's an **int** and not a **String**, right?

Well, actually, **String** is probably the most lenient data type there is. Just about everything can be considered a String. So when we entered the number **6**, Java was just fine with it. It read it as the **digit** 6 rather than the **number** 6. So no **type mismatch**.

On the other hand, the **int** type is very restrictive. If what you enter there doesn't match the **int** type, it'll crash and burn. Let's try another example to check that out. Enter any name you want, but enter the number **3.14159** as the year. You entered a number, what was the problem this time?

The problem was that your number was a **real number** and the program was looking for an **integer** (remember, that's what **int** means). When given a real number where an integer is expected, Java doesn't automatic **truncate** or **round** the number because it doesn't know which of the two you want to do. Instead, it just gives you an error message that says that you should either rewrite the program or give it the appropriate input.

Solving the Problem

What you need are some methods that get the type of information you want, and only the type of information you want. You need a method that will get an **int** and won't accept anything else from the user. Java doesn't have this natively, but that doesn't mean that it can't be done. It just means that someone has to write the code for it. Fortunately for you, someone has, and that someone is me. The code you need for the methods you crave is all in the **helper class** (remember that term) **gatling.java**.

Getting the File

You'll find this file at the URL **www.gatling.us/keith/class/gatling.java**. Once you get there, you'll want to **save** this into your project folder by using the **Save As** command. When you do this, the Mac operating system will ask you if you want to append the extension **.txt** to the file (because it thinks it's looking at a text file). Just tell it "no."

Looking Under the Hood - I

OK, the good news here is that I'm not going to make you examine and take apart every line of code that's in this class. I am, however, going to make you take a quick look so that you get an idea of what's in there and what it does. Let's start off with the comments, and even then, I'm not going to make you look at everything (mainly because there's not enough room on this page).

```
//includes the following methods:
//  int getInt()
//    gets an integer value from the keyboard with idiot checking
//  float getFloat()
//    gets a float value from the keyboard with idiot checking
//  double getDouble()
//    gets a double value from the keyboard with idiot checking
//  char getChar()
//    gets a single character from the keyboard, with idiot checking
//     String getLine()
//    gets an entire line of text from the keyboard
//  String getNext()
//    get next word from the keyboard
```

From this you can see that there are a whole bunch of **get** methods. In my little world, the word **get** always begins any method whose job it is to get information from somewhere. Hence **getInt()**, **getFloat()**, **getDouble()**, and all the other members of the **get** family.

Now let's take a look at one specific method, the one we're going to be using, **getInt()**.

Looking Under the Hood - II

This is where we dissect the **getInt()** method to see how it works. Don't worry, you won't have to write anything like this (yet), but taking a close look at it helps you to understand not only how this method works, but gives you a sneak peek at some other programming concepts. Let's take this section by section, starting with the **instance variables**.

```
public static int getInt()
{//start getInt
   //instance variables
     String mystring;
     Scanner myScanner = new Scanner(System.in);
     int mynum = 0;
     boolean validnum = false;
```

First of all, what's up with calling these **instance variables**? This comes from a 10-dollar word that has to do with how **objects** are created. Don't worry about it yet.

As far as **mystring**, **myScanner**, and **mynum** go, you've dealt with all three of these types already. They should be old friends. What's new here is the **boolean** variable **validnum**. I've set this to **false** because while I'm looking for a valid integer, I'm going to assume that the number <u>isn't</u> valid until proven otherwise. You'll see why in a minute.

```
//do the work
     while (!validnum)
     {//start the loop
        mystring=myScanner.nextLine();
```

In the section shown above, I've started a **while loop**. A while loop does something over and over as long as a certain condition is true. In this case the condition that needs to be true is for **validnum** to be **false**. That's what **!validnum** means. It means "not validnum," and is shorthand for **validnum is false**.

The next line gets a line of text from the keyboard and puts it into a String called **mystring**.

```
        try
        {
            mynum=Integer.parseInt(mystring);
            validnum=true;
        }
```

Here I've used a **try** statement. A **try** is like a fancy **if**. Oh wait, we haven't talked about **if** statements yet. Anyway, a **try** statement tries something to see if it will work. In this case, what I'm trying is to see if the String **mystring** contains a valid integer value that we can put into the int **mynum**. This is done using a method from the **Integer** helper class called **parseInt**. The **parseInt** method takes a look at the string and tries to convert it into an int value. If it's successful, and can put that value into **mynum**, then the value of **validnum** changes to **true**.

If it's not successful, then it **throws** an error exception.

```
        catch (NumberFormatException x)
        {
            validnum=false;
            System.out.print("  That was not a valid number. Please try again: ");
        }
        }//end the loop
```

When someone **throws** something at you, you usually try to **catch** it, and that's what happens next. A **try** statement is always followed by a **catch** statement so that any error messages go to the **catch** instead of to the user.

Here I'm looking for a **NumberFormatException** that I'm calling **x**. If **x** is **thrown**, then the **catch** statement **catches** it and does two things as a result. The first is to say that **validnum** is **false** (strictly speaking, this may not be necessary, since it started out as **false**, but we'll leave it this way just to be safe). The second thing it does is to tell the user that they entered bad data and need to try again. This is a much better way to handle the situation than simply crashing.

The loop continues to run until the user enters a string that this method can parse as an integer and **validnum** changes to **true**.

```
        return mynum;
  }//end getInt
```

Finally, once a valid value is entered, **getInt** returns this to whatever method called it in the first place.

Using getInt

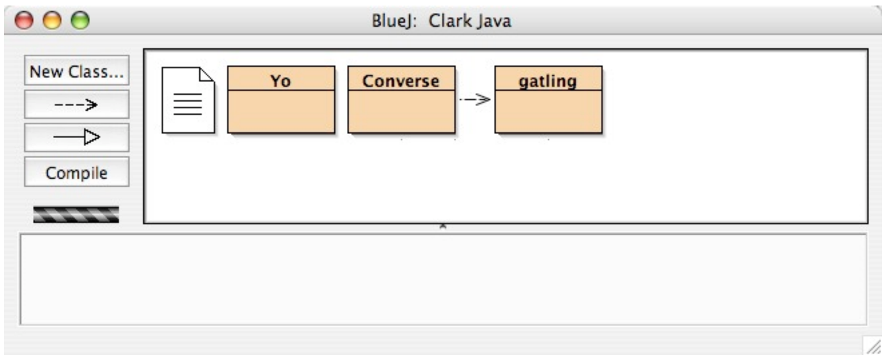
In order for this to work, we need to change two lines of code in **Converse**. Take a look at the example below.

```
//get input
  System.out.print("Hi! What's your name? ");
  name=myScanner.nextLine();
  System.out.print("Nice to meet you "+name+". What year were you born in? ");
  born = ~~myScanner.nextInt()~~ gatling.getInt();
  System.out.print("Cool. And what year is it now? ");
  now = ~~myScanner.nextInt()~~ gatling.getInt();
```

There we've changed **myScanner.nextInt()** to **gatling.getInt()**. Compile it, run it, and see what happens.

Oops!

Once again, this doesn't stand for **object-oriented programming**. It won't compile because it can't find **gatling.java**. Why can't it find it if we've put it into our project folder? Because while the computer knows it's there, BlueJ doesn't. To solve this problem, simply quit out of BlueJ and open it back up again. Now you should see a new item in your project window: the **gatling** file.

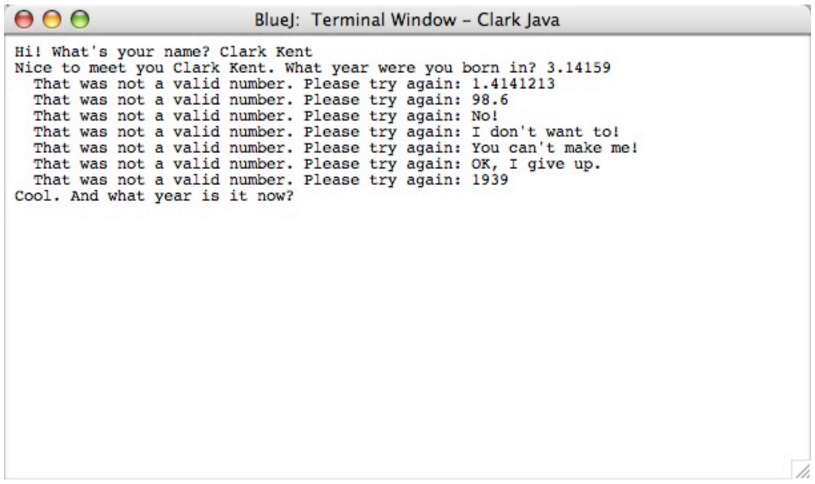


Not only does your project now include **gatling.java**, but you should see an arrow pointing from **Converse** to **gatling**. This is to show the **relationship** between them. **Converse** is using code from inside of **gatling**.

Now you're ready to compile and run **Converse** again.

Success

Now, if Clark enters some bad data, specifically, a non-integer response when an integer has been asked for, the results will be a little different. Check out the example below:



```
Hi! What's your name? Clark Kent
Nice to meet you Clark Kent. What year were you born in? 3.14159
  That was not a valid number. Please try again: 1.4141213
  That was not a valid number. Please try again: 98.6
  That was not a valid number. Please try again: No!
  That was not a valid number. Please try again: I don't want to!
  That was not a valid number. Please try again: You can't make me!
  That was not a valid number. Please try again: OK, I give up.
  That was not a valid number. Please try again: 1939
Cool. And what year is it now?
```

As you can see, this program will not let Clark go until he puts in a valid response to what year he was born in. The program doesn't spew error messages all over the screen, it just politely and firmly keeps telling Clark that he's entered bad data and asking him to try again.

But we haven't totally idiot-proofed this program. There are other ways for users to be stupid here, and we'll explore them, and the solutions, in handout #7.

This page intentionally left **almost** blank.