Lecture 8

A series on variable types
This episode starring
Parameter Variables

Declaring an Integer Variable

Declaring an integer variable:

```
int counter = 0;
```

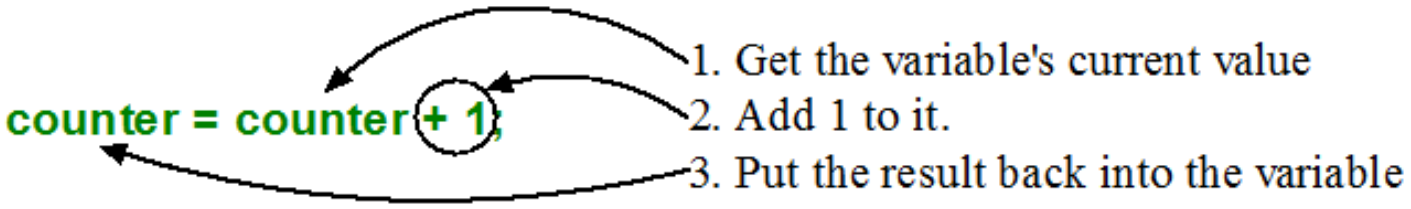
Using a variable's value:

```
if (counter == 0)
{ ...
}
```

```
while (counter > 0)
{ ...
}
```

```
this.myMethod(counter);
```

Changing a variable's value:



Variable Types

Variable Type	How it is defined	Scope (how long it lasts)	Uses
Local Variable	<type> Name = value; In function or method code block.	Until the closing bracket for the clause it is in.	Temporary use in a method, or loop.
Parameters	<type> Name in method declaration. Value is passed when method is called	For the duration of the Method. Copies value inside of variable.	Passing information to a Method.
Instance Variables (Non-static Fields)	<type> Name during class definition	For the lifetime of an object (an instance of the class)	Values that help define an instance of the class and persist through multiple service calls
Class Variables (Static Fields)	Static <type> Name during class definition	For the life time of the class (can be accessed through class definition)	Values that are the same for EVERY instance of the class

More Flexible Code

The Way We Originally Learned ("Hard Coded")

```
public void move2()
{   this.move();
    this.move();
}
```

```
public void move3()
{   this.move();
    this.move();
    this.move();
}
```

```
public void move4()
{   this.move();
    this.move();
    this.move();
    this.move();
}
```

A More Flexible Way ("Argument Coded")

```
public void howManyMoves(int numMoves)
{   int counter = 0;
    while(counter < numMoves)
    {   this.move();
        counter++;
    } // Note: This method has no error handling
}
```

← New Flexible **Method**

How it might be called in **main** →
```
karel.howManyMoves(2);
```
or →
```
karel.howManyMoves(3);
```
or →
```
karel.howManyMoves(4);
karel.howManyMoves(12);
```

Why use parameters

You'll notice that every new method (service) we've created does something:

Each time we call upon one of those services, it'll always do the same thing

```
public void turnAround()
{ this.turnLeft();
  this.turnLeft();
}

public void move3()
{ this.move();
  this.move();
  this.move();
}

public void turnRight()
{ this.turnAround();
  this.turnLeft();
}
```

We could have asked for user input, but then the service would fill **TWO** roles

1. **Doing** <something>
2. **Interacting** with the user

We'd like each service to have a single, well-defined, and easy to summarize **role**

Thus, things like **moveMultiple** should move the robot through multiple intersections

Things like **main** should run the part of the program that relays instructions from the user to the robot(s)

Using Parameters

We need a way to pass information to a service.

We'd like to be able to say:

```
rob.moveMultiple(3);
```

or

```
rob.moveMultiple(7);
```

or even

```
Scanner keyboard = new Scanner(System.in);
int howFar = keyboard.nextInt();
rob.moveMultiple(howFar);
```

The actual info being passed to the service is called an ARGUMENT

The service must also be told to expect this information *AND make room* for it to be stored somewhere in memory, so instead of:

```
public void moveMultiple()
```

we'll write

```
public void moveMultiple(int numberOfIntersections)
```

Makes a COPY of the value that is passed. Now numberOfIntersections can be used like any other variable: Use the value, print it out, assign a new value, etc.

Calling a Method with Parameters

Notice how the parameters that we declare are matched up against the "hard-coded" arguments that we give it:

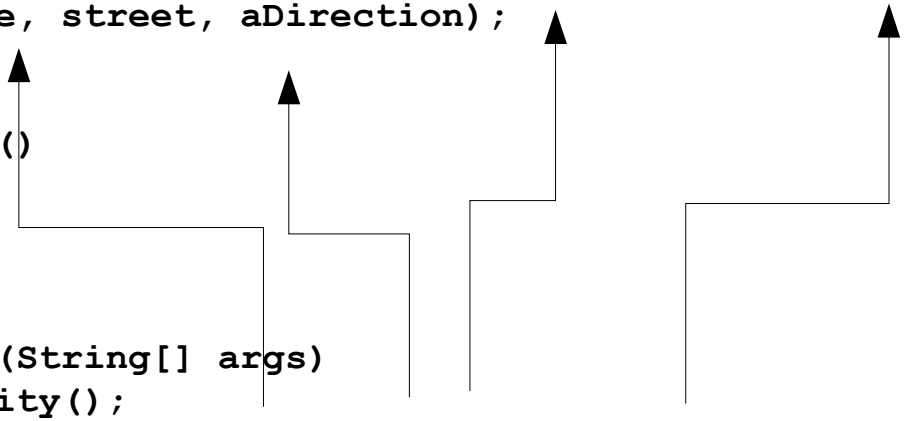
```
import becker.robots.*;

public class MrRoboto extends Robot
{
   public MrRoboto(City theCity, int avenue, int street, Direction aDirection)
   {  super(theCity, avenue, street, aDirection);
   }

    public void turnAround()
   {  this.turnLeft();
      this.turnLeft();
   }

   public static void main(String[] args)
   {  City bothell = new City();
      MrRoboto lisa = new MrRoboto(bothell, 3, 2, Direction.SOUTH);

      lisa.turnAround();
      lisa.move();
      lisa.turnAround();
   }
}
```



Using a parameter for a while loop

The following method moves a robot east to Avenue 18.

```
public void moveToAvenue18() // <-- no parameter
{  while(this.getAvenue() < 18)
   {  this.move();
   }
}
```

This is very limited, and only useful to move the robot specifically to Avenue 18 since it was "hard-coded" to do so. With a parameter, however, it can be used to move the robot to any avenue east of its current location.

```
public void moveToAvenue(int destAve) // with parameter
{ while(this.getAvenue() < destAve)
  { this.move();
  }
}
```

In main you'd call it this way: `this.moveToAvenue(destAve);`

Parameters v. Temporary Variables

In this section, we will show how parameter variables are closely related to temporary variables, explore using parameters with constructors, and discuss overloading.

Parameter Variables versus Temporary Variables

```
public void moveTheBot()
{   int howFar = 2; // <-- temporary variable
    this.street = this.street + howFar;
}
```

In main you'd call it this way: `this.moveTheBot();`

```
public void moveTheBot(int howFar) // <-- parameter variable
{   this.street = this.street + howFar;
}
```

In main you'd call it this way: `this.moveTheBot(howFar);`

In main you'd call it this way: `this.moveTheBot(2);`

Overloading Methods

In Java it is possible to define two or more methods within the **same class** that share the **same name**, as long as their parameter declarations are different. When this is the case, the methods are said to be **overloaded**, and the process is referred to as **method overloading**. Method overloading is one of the ways that Java implements **polymorphism**. *Polymorphism is the ability of an object to take on many forms, and uses the "is a" test determine* multiple inheritance *through from different classes, subclasses, etc.*

Method overloading is one of Java's most exciting and useful features. When an overloaded method is invoked, Java uses the **type** and/or **number of arguments** as its guide to determine which version of the overloaded method to actually call.

Thus, overloaded methods *must differ* in the **type** and/or **number of their parameters**.

While overloaded methods may have different return types, the return type alone is insufficient to distinguish two versions of a method. When Java encounters a call to an overloaded method, it simply executes the version of the method whose parameters match the arguments used in the call.

Let's have a look-see - - -

Overloading Methods

```
public class MethodOverloading extends Object
{
  public void test(int a) {
    System.out.println("a: " + a);
  }

  public void test(int a, int b) {
    System.out.println("a and b: " + a + "," + b);
  }

  public double test(double a) {
    System.out.println("double a: " + a);
    return a*a;
  }

  public static void main(String args[])
  {
    MethodOverloading MethodOverloading = new MethodOverloading();
    double result;

    MethodOverloading.test(10);
    MethodOverloading.test(10, 20);
    result = MethodOverloading.test(5.5);
    System.out.println("Result : " + result);
  }
}
```