The different *kinds* of variables in a Java program
The different kinds of *variables* in a Java program

Java has 4 different kinds of variables

- **Class** variables
- **Instance** variables
- **Local** variables
- **Parameter** variables

The kind of variable is determined by:

- *Where* the variable is defined
- Whether the keyword *static* was *used* in the variable definition.
Properties of variables (1)

Every variable has 2 properties:

• **life time** = the *duration* that a variable exists

• **scope** = the *region in the program* where the variable is accessible (can be used)
Properties of variables (2)

To accommodate different needs:

*Short term* information:
- Some *information* need *only* be retained *within a (one) method* (We call this *kind* of information *local information*)

*Long term* information:
- Some *information* must be *retained across several methods*
Local variables (1)

- A local variable is used to store information that is relevant for the duration of the execution of one method.

- A local variable will exist as long as the method in which they have been created is still running.

- As soon as the method terminates (i.e., returns), all the local variables defined inside the method are destroyed.
• A *local variable* is *defined* inside the *body* of a *method*
Local variables (3)

```java
public class MyProgram
{
    public static void main(String[] args)
    { // Body of method "main"

        double r; // +++ Local variable !!!

        r = MyProgram.min(1.0, 4.0);
        System.out.println(r);

        r = MyProgram.min(3.7, -2.9);
        System.out.println(r);

        r = MyProgram.min(-9.9, 3.8);
        System.out.println(r);
    }
}
```

```java
public class ToolBox
{
    public static double min(double a, double b)
    { // Body of method "min"

        double m = 0; // +++ Local variable !!!

        if (a < b)
        {
            m = a; // a is the smaller value
        }
        else
        {
            m = b; // b is the smaller value
        }

        return(m);
    }
}
```
Local variables (4)

- A local variable *only* exists between:
  - The **definition** of the *local variable*
  - The **end of the method** where the *local variable* is defined

```java
public class ToolBox {
    public static double min ( double a, double b ) {
        double m = 0;
        if ( a < b ) {
            m = a;
        } else {
            m = b;
        }
        return(m);
    }
}
```
Scope of a variable

- **Scope** of a *variable* = the *region* in the *program* where the *variable* is *accessible*

- Between the *2 location above*, the *scope* of a *variable* is *further limited* by the *block* in which it is *defined*
The scope of local variables (1)

Program "Scope1"

```java
public class Scope1 {
    public static void main(String[] args) {
        double r;
        r = 3.14;
        System.out.println(r);
    }
}
```

Program "Scope2"

```java
public class Scope2 {
    public static void main(String[] args) {
        double r;
        r = 3.14;
        System.out.println(r);
    }
}
```
The scope of *local variables* (2)

- The access of the variable *r* was *limited* to the block in which it was *defined* !!!

```java
public class Scope2 {
    public static void main(String[] args) {
        double r;
        r = 3.14;
        System.out.println(r);
    }
}
```

*A variable is accessible only within the block in which it is defined*

*Error:* *r* is *not accessible outside its block*
The scope of *local variables* (3)

```java
public class Scope3 {
    public static void main(String[] args) {
        {
            r = 1;   // (1)
            double r;
            r = r + 5;  // (2)
        }
        r = r + 2;  // (3)
    }
    r = r + 3;  // (4)
}
```
The scope of *local variables* (4)

- You *cannot* define *different* variables with the *same name* inside the *same scope*
Non-overlapping (or disjoint) scopes

- Disjoint scopes = 2 or more scopes that do not overlap with each other

```java
public class Scope4
{
    public static void main( String[] args )
    {
        double r;
        ...
        String s;
        ...
    }
}
```
Non-overlapping (or disjoint) scopes (2)

```java
public class MyProgram {
  public static void main(String[] args) {
    double m;
    m = Toolbox.min(1.0, 4.0);
    ...
  }
}

public class Toolbox {
  public static double min(double a, double b) {
    double m = 0;
    if (a < b) {
      m = a;
    } else {
      m = b;
    }
    return m;
  }
}
```

We can use the SAME name `m` for 2 different variables in different method scopes!
Scoping rule for *nested scope* (1)

```java
public class Scope5 {
    public static void main(String[] args) {
        double r;  // r is defined in outer scope
        {
            inner scope
        }
        double t;
    }
}
```

*Nested scoping rule:*
- `r` is accessible in inner scope that follows the definition.
Scoping rule for *nested scope* (2)

```java
class Scope5 {
    public static void main(String[] args) {
        double r;
        {
            double t;  // t is defined here in outer scope
            {
                // inner scope
            }
        }
        // t is NOT accessible in inner scope that precedes the definition
    }
}
```
Parameter Variables
A parameter variable is used to store information that is being passed from the location of the method call into the method that is called.

```java
public class Class1 {
    public static void method1() {
        double x; // x contains // some information
        ....
        Class2.method2(x);
        // Pass (give) information // stored in x to method2 !!!
        ....
    }
    ....
}

public class Class2 {
    public static void method2( double paramVar ) {
        statements that operate on paramVar ....
    }
    ....
}
```
Parameter variables (2)

Class1

method1

\( x \)

123

pass information

Class2

method2

paramVar

123

Passing information into a parameter variable
public class MyProgram
{
    public static void main(String[] args)
    {
        double r;
        r = Toolbox.min(1.0, 4.0); // Passing information into the min method
        ....
    }
}

public class Toolbox
{
    public static double min(double a, double b)
    {
        double m = 0;
        if (a < b)
        {
            m = a;
        }
        else
        {
            m = b;
        }
        return(m);
    }
}
Defining *parameter variables* (1)

- A *parameter variable* is defined inside the *brackets* ( ... ) of the header of a method.

```java
public class SomeClass {
    
    public static void SomeMethod(...) {
        ...
    }

    }
```
Defining *parameter variables* (2)

```java
public class MyProgram
{
    public static void main(String[] args)
    { // Body of method "main"

        double r; // *** Local variable
        r = MyProgram.min(1.0, 4.0);
        System.out.println(r);

        r = MyProgram.min(3.7, -2.9);
        System.out.println(r);

        r = MyProgram.min(-9.9, 3.8);
        System.out.println(r);
    }
}
```

```java
public class ToolBox
{
    public static double min(double a, double b) // Body of method "min"
    {
        double m = 0; // *** Local variable
        if (a < b)
        {
            m = a; // a is the smaller value
        }
        else
        {
            m = b; // b is the smaller value
        }
        return m;
    }
}
```
The *life time* and *scope* of parameter variables (1)

- The **life time** of a *parameter variable* is the **entire body** of the method.
- A parameter variable *behaves* like a **local variable** that is **defined** at the *start* of the method.

```java
public class ToolBox {
    public static double min ( double a, double b ) {
        double m = 0;
        if ( a < b ) {
            m = a;
        } else {
            m = b;
        }
        return(m);
    }
}
```
The *life time* and *scope* of parameter variables (2)

- You *cannot* define *different* variables with the *same name* inside an *outer scope* and inside an *inner scope*.
- In Java, you *cannot* define a *local* variable inside a method that has the *same name* as a *parameter variable*.

```java
public class ToolBox {
    public static double min(double a, double b) {
        double m = 0;
        if (a < b) {
            m = a;
        } else {
            m = b;
        }
        return m;
    }
}
```
Class Variables
Class Variables (1)

- A class variable is used to store long term information that are used by many different methods in the program.

- Can have the public access.

- Must use the keyword static.

- Inside the braces { ... } of a class.

- Outside the braces { ... } of every method.
Class Variables (2)

Class variables are defined here

```java
public class SomeClass {
    public static int a;

    public static void SomeMethod( ... )
    {
        ...
    }

    public static double b;

    public static void SomeMethod( ... )
    {
        ...
    }

    public static double[ ] c;
}
```
Accessing a class variable (1)

- A class variable is accessed using its complete name:

  `nameOfClass.nameOfClassVariable`

```java
public class ClassVar1
{
    public static double a;  // <----- Class variable

    public static void main(String[] args)
    {
        // Body of method "main"

        ClassVar1.a = 3.1415;  // Accessing a class variable
        System.out.println(a);
    }
}
```
Accessing a class variable (2)

- We can reference a class variable defined inside the same class without using the class name.

```java
public class ClassVar2 {
    public static double a; // <----- Class variable

    public static void main(String[] args) {
        // Body of method "main"
        a = 3.1415; // We can omit the classname in this method
        System.out.println(a);
    }
}
```
The *life time and scope* of (public) class variables (1)

- **Class variables** exist for the *entire execution* of the Java program
- **Class variables** are *accessible* in *every method* of every class

```java
public class ClassVar3x
{
    public static void print()
    {
        System.out.println(ClassVar3x.a); // Can be access here, even though
        // the definition appears below!
    }

    public static double a = 3.1415; // Class variable "ClassVar3x.a"
}

public class ClassVar3
{
    public static void main(String[] args)
    {
        System.out.println(ClassVar3x.a); // Can be accessed here

        ClassVar3x.print(); // Invoke ClassVar3x.print to show that
        // It too can access the class variable
    }
}
```
The *life time and scope* of (public) class variables (2)

- When the method has a *local variable* or a *parameter variable* is *equal* to the name of the class variable, then:

  The *class variable* with the *same name* can *no longer accessible* with the *short hand notation* in that scope !!!
The *life time* and *scope* of (public) class variables (3)

```java
public class ClassVar4
{
    public static double a = 3.1415;

    public static void main(String[] args)
    {
        System.out.println(a);   // prints 3.1415     ... (1)

        System.out.println(a); // prints "abc"        ... (2)
            // inside inner scope
        System.out.println(a); // prints "abc"        ... (3)

        System.out.println(ClassVar4.a); // prints 3.1415 ... (4)
            ... (5)

        System.out.println(a);   // prints 3.1415     ... (6)

        boolean a = true;        // Class var a is not accessible ...(7)
            // in main any longer !!!
        System.out.println(a);   // prints true       ... (8)

        System.out.println(ClassVar4.a); // prints 3.1415 ... (9)
    }
}
```