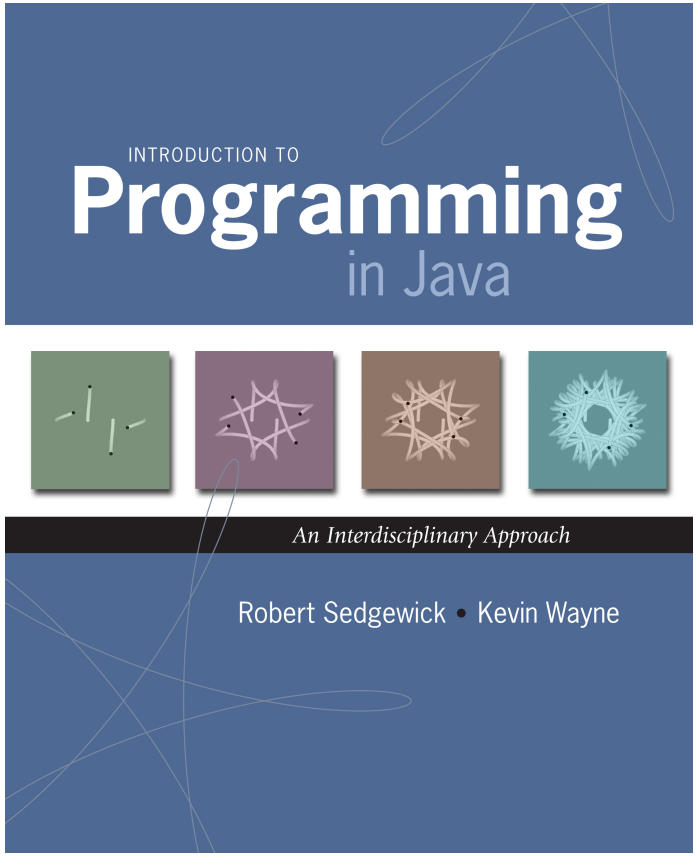
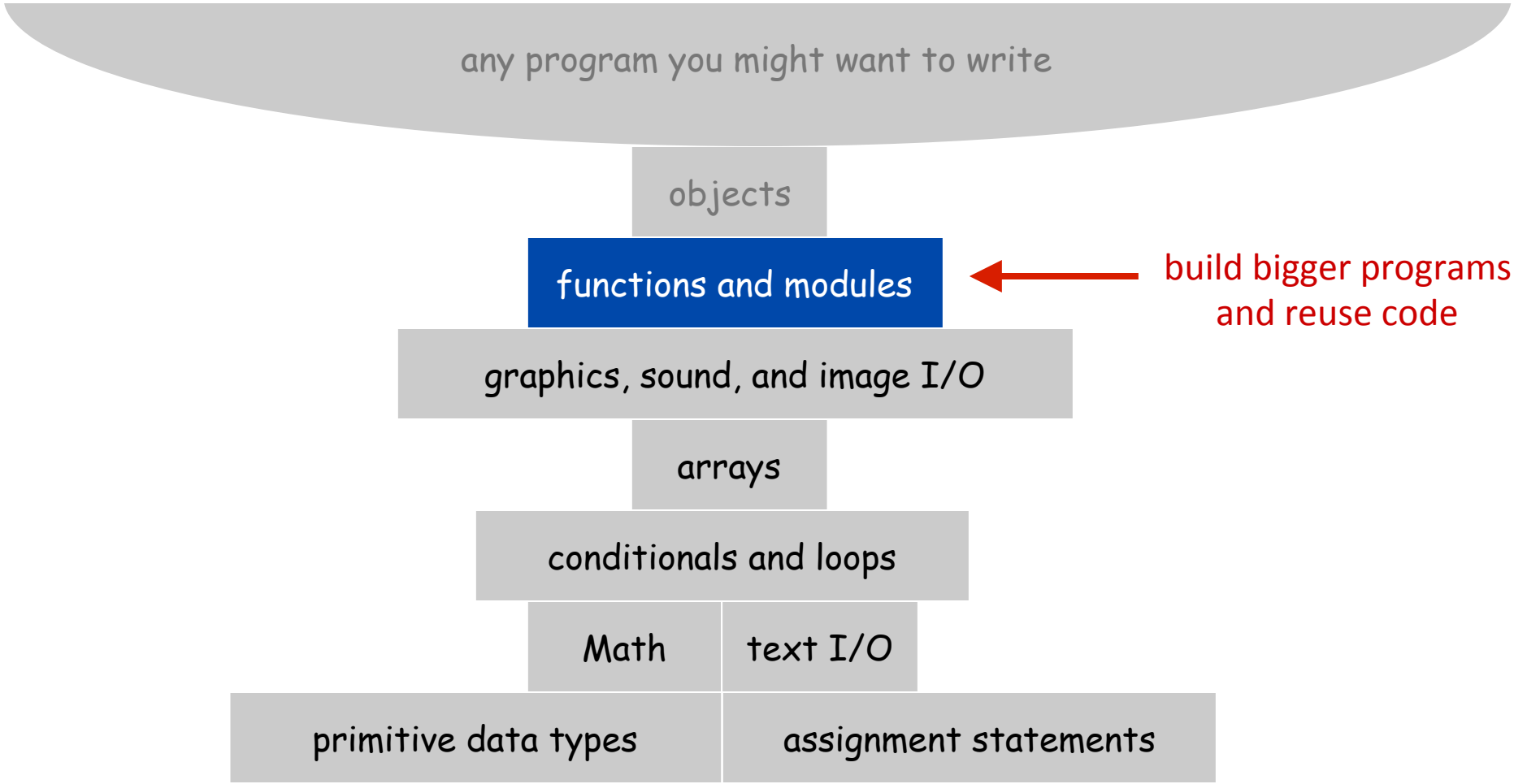
2.1 Functions



Penn Engineering

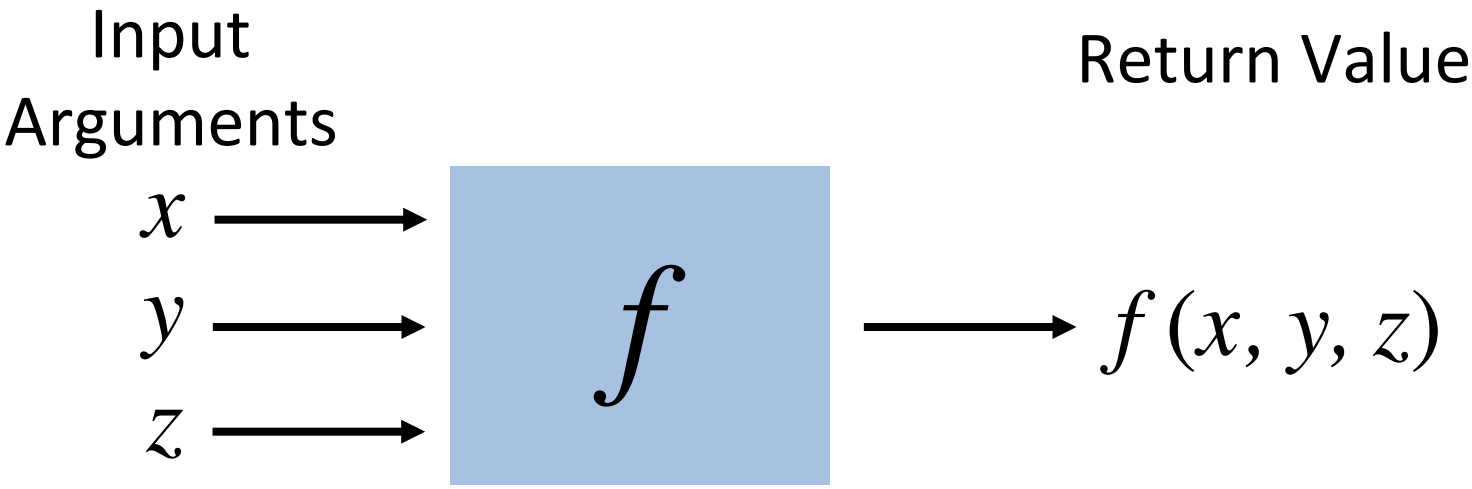
A Foundation for Programming



Penn Engineering

Functions

- Take in input arguments (zero or more)
- Perform some computation
  - May have side-effects (such as drawing)
- Return one output value



Functions (Static Methods)

- Applications:
  - Use mathematical functions to calculate formulas
  - Use functions to build modular programs

- Examples:
  - Built-in functions:

    ```
    Math.random(), Math.abs(), Integer.parseInt()
    ```

  - I/O libraries:

    ```
    PennDraw.circle(), PennDraw.setPenColor()
    ```

  - User-defined functions:

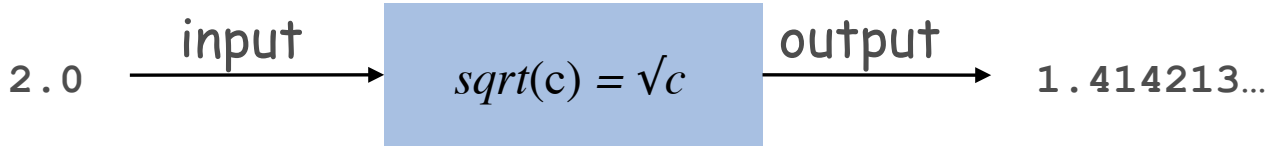
    ```
    main()
    ```

Why do we need functions?

- Break code down into logical sub-steps
- Readability of the code improves
- Testability - focus on getting each individual function correct

Anatomy of a Java Function

- Java functions – It is easy to write your own
  - Example: `double sqrt(double c)`




```
public static double sqrt(double c) {
    ...
}
```

return type — `double`

method name — `sqrt`

arguments — `(double c)`

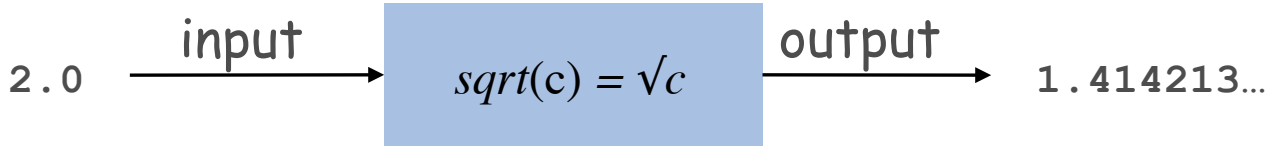
method signature (excludes return type) — `sqrt(double c)`

Please note that the method signature is defined incorrectly in the figure on pg 188 of your textbook

Penn Engineering

Anatomy of a Java Function

- Java functions – It is easy to write your own
  - Example: `double sqrt(double c)`




```
public static double sqrt(double c)
{
   if (c < 0) return Double.NaN;
   double err = 1e-15;
   double t = c;
   while (Math.abs(t - c/t) > err * t)
      t = (c/t + t) / 2.0;
   return t;
}
```

local variables

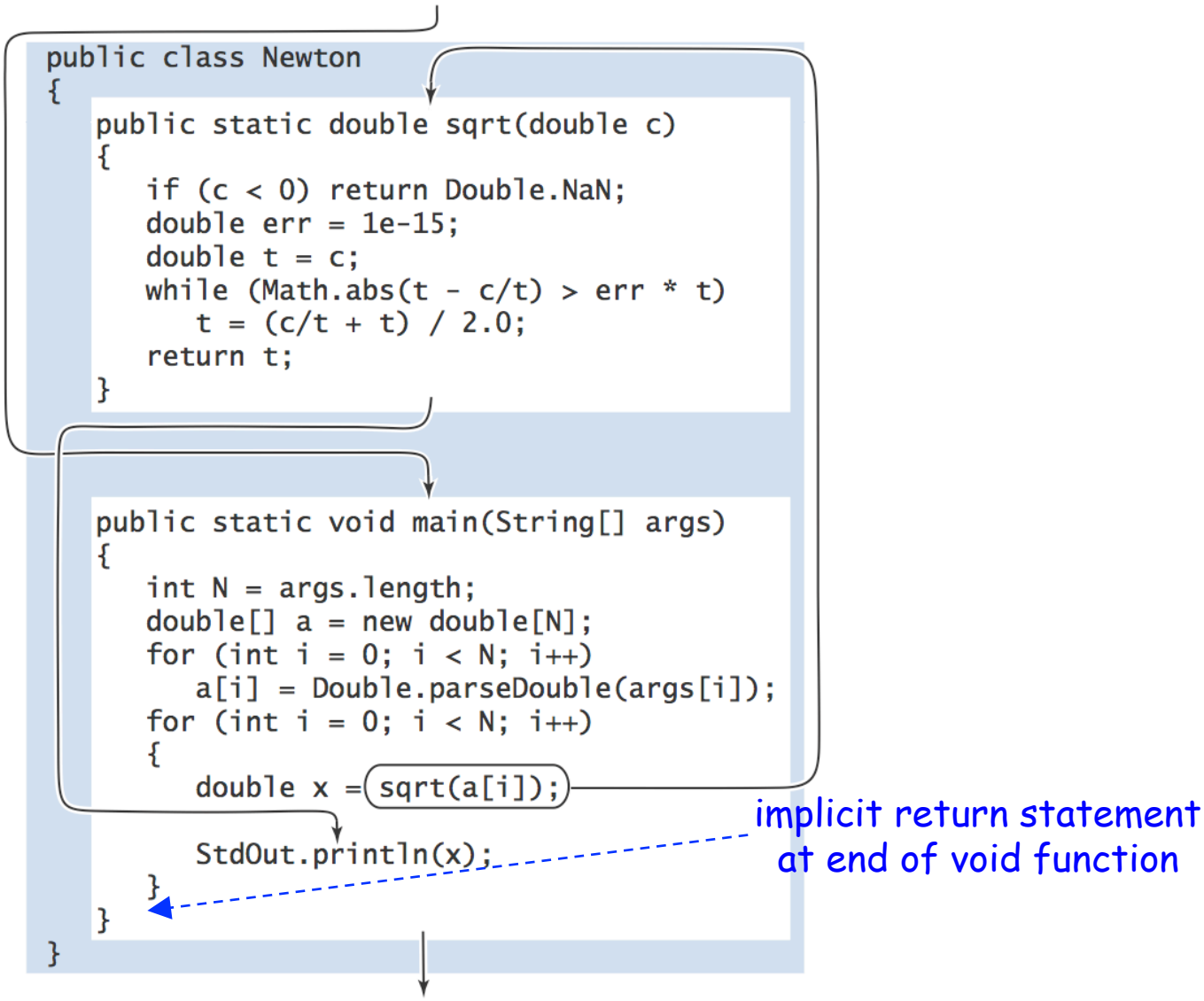
method body

return statement

call on another method

Flow of Control

Functions provide a new way to control the flow of execution

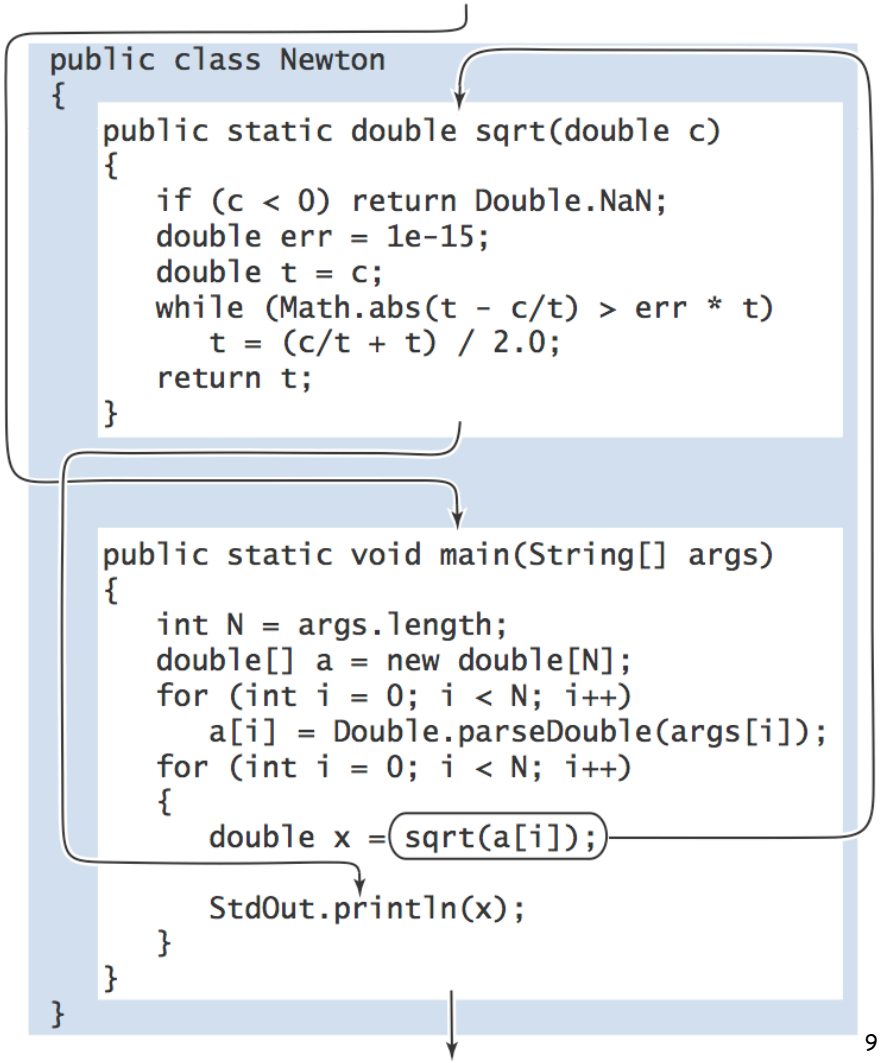


Flow of Control

What happens when a function is called:

- Control transfers to the function
- Argument variables are assigned the values given in the call
- Function code is executed
- Return value is substituted in place of the function call in the calling code
- Control transfers back to the calling code

Note: This is known as "pass by value"



```
public class Newton
{
   public static double sqrt(double c)
   {
      if (c < 0) return Double.NaN;
      double err = 1e-15;
      double t = c;
      while (Math.abs(t - c/t) > err * t)
         t = (c/t + t) / 2.0;
      return t;
   }

   public static void main(String[] args)
   {
      int N = args.length;
      double[] a = new double[N];
      for (int i = 0; i < N; i++)
         a[i] = Double.parseDouble(args[i]);
      for (int i = 0; i < N; i++)
      {
         double x = sqrt(a[i]);

         StdOut.println(x);
      }
   }
}
```

Penn Engineering

Organizing Your Program

- Functions help you organize your program by breaking it down into a series of steps
  - Each function represents some abstract step or calculation
  - Arguments let you make the function have different behaviors

- **Key Idea**: write something ONCE as a function then reuse it many times

Scope

Scope: the code that can refer to a particular variable

- A variable's scope is the entire code block (any any nested blocks) after its declaration

Simple example:

```
int count = 1;
for (int i = 0; i < 10; i++) {
      count *= 2;
}
// using 'i' here generates
// a compiler error
```

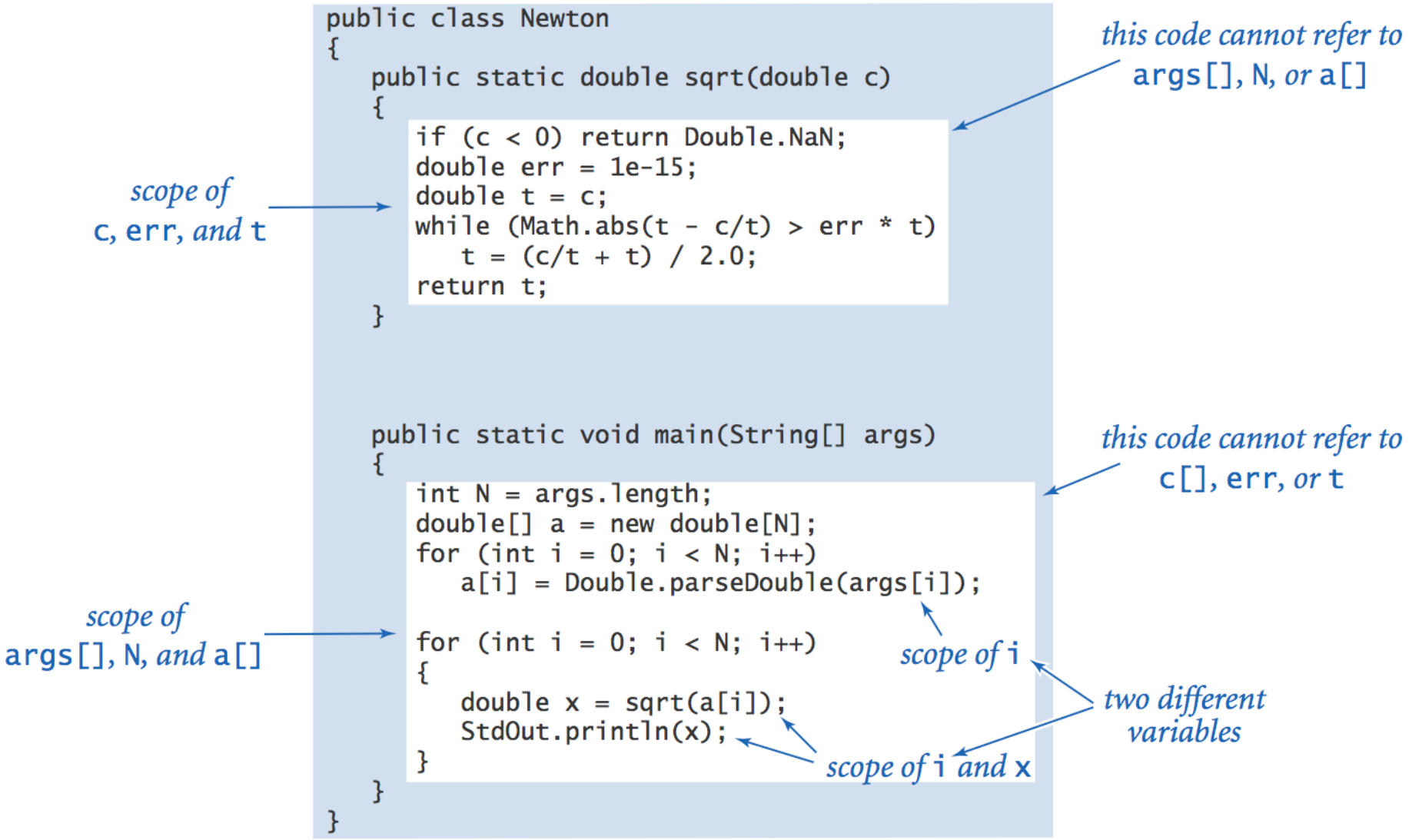
Best practice: declare variables to limit their scope

Function Challenge 1

Q. What happens when you compile and run the following code?

```
public class Cubes1 {

   public static int cube(int i) {
      int j = i * i * i;
      return j;
   }

   public static void main(String[] args) {
      int N = Integer.parseInt(args[0]);
      for (int i = 1; i <= N; i++)
         System.out.println(i + " " + cube(i));
   }
}
```

Scope with Functions



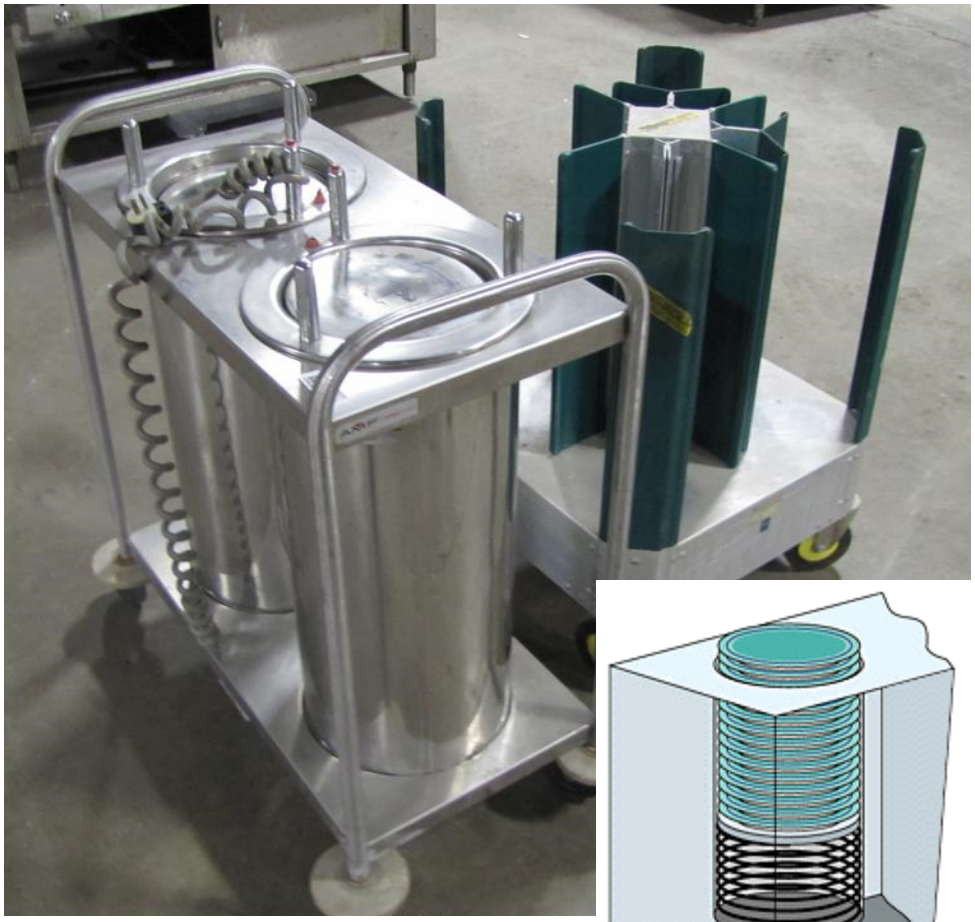
Tracing Functions

```
public class Cubes1 {

   public static int cube(int i) {
      int j = i * i * i;
      return j;
   }

   public static void main(String[] args) {
      int N = Integer.parseInt(args[0]);
      for (int i = 1; i <= N; i++)
         System.out.println(i + " " + cube(i));
   }
}
```

```
% javac Cubes1.java
% java Cubes1 6
1 1
2 8
3 27
4 64
5 125
6 216
```

Penn Engineering

Last In First Out (LIFO) Stack of Plates



Penn Engineering

Method Overloading

- Two or more methods *in the same class* may also have the same name
- This is called ***method overloading***

absolute value of an `int` *value*

```
public static int abs(int x)
{
   if (x < 0) return -x;
   else       return  x;
}
```

absolute value of a `double` *value*

```
public static double abs(double x)
{
   if (x < 0.0) return -x;
   else         return  x;
}
```

Penn Engineering

Method Signature

- A method is uniquely identified by
  - its **name** and
  - **its parameter list** (parameter types and their order)
- This is known as its ***signature***

Examples:

```
static    int min(int a, int b)
static double min(double a, double b)
static  float min(float a, float b)
```

Penn Engineering

Return Type is Not Enough

- Suppose we attempt to create an overloaded `circle(double x, double y, double r)` method by using different return types:

```
static void    circle(double x, double y, double r) {...}

//returns true if circle is entirely onscreen, false otherwise
static boolean circle(double x, double y, double r) {...}
```

- This is NOT valid method overloading because the code that calls the function can ignore the return value

```
circle(50, 50, 10);
```

  - The compiler can't tell which `circle()` method to invoke
  - Just because a method returns a value doesn't mean the calling code has to use it

Too Much of a Good Thing

Automatic type promotion and overloading can sometimes interact in ways that confuse the compiler

For example:

```
// version 1
static void printAverage(int a, double b) {
    ...
}

// version 2
static void printAverage(double a, int b) {
    ...
}
```

Why might this be problematic?

Too Much of a Good Thing

```
static void average(int a, double b) { /*code*/ }
static void average(double a, int b) { /*code*/ }
```

- Consider if we do this

```
public static void main (String[] args) {
    ...
    average(4, 8);
    ...
}
```

- The Java compiler can't decide whether to:
  - promote 7 to 7.0 and invoke the first version of `average()`, or
  - promote 5 to 5.0 and invoke the second version
- Take-home lesson: don't be too clever with method overloading

Penn Engineering

Documentation

Method-level Documentation

- Method header format:

```
/**
 * Name: circleArea
 * PreCondition: the radius is greater than zero
 * PostCondition: none
 * @param radius - the radius of the circle
 * @return the calculated area of the circle
 */
static double circleArea (double radius) {
    // handle unmet precondition
    if (radius < 0.0) {
        return 0.0;
    } else {
        return Math.PI * radius * radius;
    }
}
```

Penn Engineering

Method Documentation

- Clear communication with the class user is of paramount importance so that he can
  - use the appropriate method, and
  - use class methods properly.
- Method comments:
  - explain what the method does, and
  - describe how to use the method.
- Two important types of method comments:
  - ***precondition*** comments
  - ***post-conditions*** comments

Preconditions and Postconditions

- Precondition
  - What is assumed to be true when a method is called
  - If any pre-condition is not met, the method may not correctly perform its function.

- Postcondition
  - States what will be true after the method executes (assuming all pre-conditions are met)
  - Describes the side-effect of the method

An Example of Pre/Post-conditions

Very often the precondition specifies the limits of the parameters and the postcondition says something about the return value.

```
/*Prints the specified date in a long format
      e.g.  1/1/2000 -> January 1, 2000
  Inputs:  the month, day, and year
  Pre-condition:
       1 <= month <= 12
       day appropriate for the month
       1000 <= year <= 9999
  Post-condition:
       Prints the date in long format
*/
public static void printDate(int month, int day, int year)
{
  // code here
}
```

FUNCTION EXAMPLES

Function Examples

absolute value of an `int` *value*

```
public static int abs(int x)
{
   if (x < 0) return -x;
   else       return  x;
}
```

overloading

absolute value of a `double` *value*

```
public static double abs(double x)
{
   if (x < 0.0) return -x;
   else         return  x;
}
```

primality test

```
public static boolean isPrime(int N)
{
   if (N < 2) return false;
   for (int i = 2; i <= N/i; i++)
      if (N % i == 0) return false;
   return true;
}
```

multiple arguments

hypotenuse of a right triangle

```
public static double hypotenuse(double a, double b)
{  return Math.sqrt(a*a + b*b);  }
```

Penn Engineering

Function Challenge 2

Q. What happens when you compile and run the following code?

```
public class Cubes2 {

   public static int cube(int i) {
      int i = i * i * i;
      return i;
   }

   public static void main(String[] args) {
      int N = Integer.parseInt(args[0]);
      for (int i = 1; i <= N; i++)
         System.out.println(i + " " + cube(i));
   }
}
```

Penn Engineering

Function Challenge 3

Q. What happens when you compile and run the following code?

```
public class Cubes3 {

   public static int cube(int i) {
      i = i * i * i;
   }


   public static void main(String[] args) {
      int N = Integer.parseInt(args[0]);
      for (int i = 1; i <= N; i++)
         System.out.println(i + " " + cube(i));
   }
}
```

Function Challenge 4

Q. What happens when you compile and run the following code?

```
public class Cubes4 {

   public static int cube(int i) {
      i = i * i * i;
      return i;
   }

   public static void main(String[] args) {
      int N = Integer.parseInt(args[0]);
      for (int i = 1; i <= N; i++)
         System.out.println(i + " " + cube(i));
   }
}
```

Penn Engineering

Function Challenge 5

Q. What happens when you compile and run the following code?

```
public class Cubes5 {

   public static int cube(int i) {
      return i * i * i;
   }


   public static void main(String[] args) {
      int N = Integer.parseInt(args[0]);
      for (int i = 1; i <= N; i++)
         System.out.println(i + " " + cube(i));
   }
}
```