# Chapter 2 Elementary Programming

# Identifiers



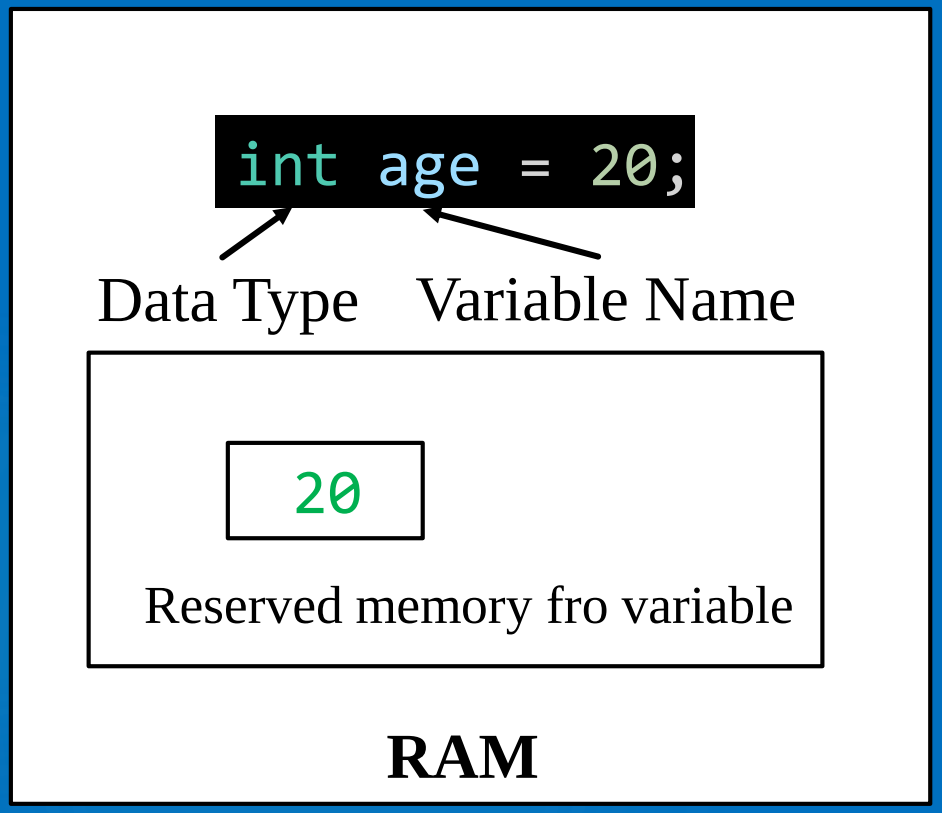
An identifier is *a sequence of characters* that consist of *letters*, *digits*, *underscores* (_), and *dollar signs* (*$*).

- An identifier must *start with* a letter, an underscore (_), or a dollar sign (*$*). It *cannot start with a digit*.
- An identifier cannot be a *reserved word*.
- An identifier cannot be *true*, *false*, or *null*.
- An identifier can be of *any length*.

```
import java.util.Scanner;
public class TestClass{
    public static void displayName(String name){
        String MyName = name;
        System.out.println("Your name: " + MyName);
    }
}
```

# Variables

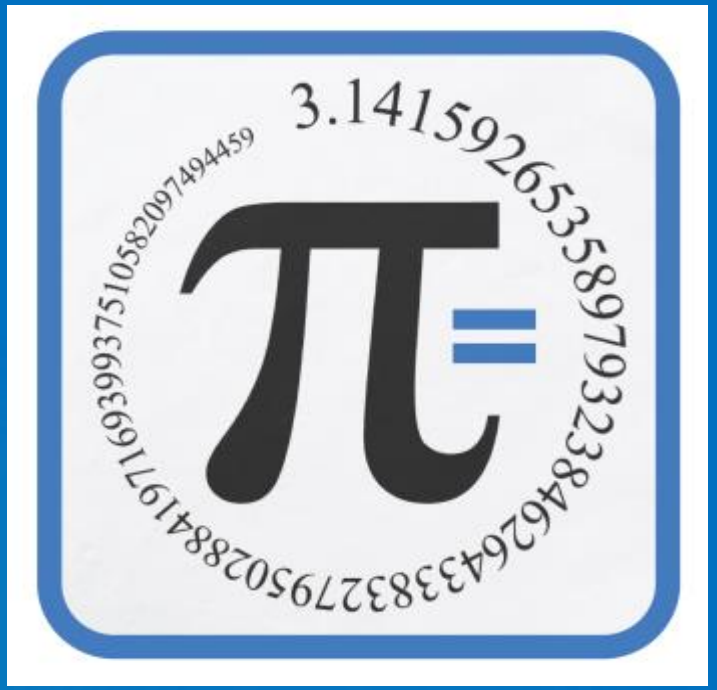





```java
// variable declaring
int a;
double b, d;
char c;
Object o1;
// declaring and initialization
String s = "Hi";
int i = 1, j = 2;
// Using
System.out.println(i * j);
```

# Constants






*final datatype* CONSTANT_NAME = VALUE;

```
final double PI = 3.14159;
final double E;
E = 2.718281;
System.out.printf("PI is %4.2f", PI);
System.out.printf("E is %4.2f", E);
//The constants PI and E cannot be modified
```

# Numerical Data Types



| Name | Range | Storage Size |
|---|---|---|
| byte | $-2^7$ (-128) to $2^7-1$ (127) | 8-bit signed |
| short | $-2^{15}$ (-32768) to $2^{15}-1$ (32767) | 16-bit signed |
| int | $-2^{31}$ (-2147483648) to $2^{31}-1$ (2147483647) | 32-bit signed |
| long | $-2^{63}$ to $2^{63}-1$ (i.e., -9223372036854775808 to 9223372036854775807) | 64-bit signed |
| float | Negative range: -3.4028235E+38 to -1.4E-45 Positive range: 1.4E-45 to 3.4028235E+38 | 32-bit IEEE 754 |
| double | Negative range: -1.7976931348623157E+308 to -4.9E-324 Positive range: 4.9E-324 to 1.7976931348623157E+308 | 64-bit IEEE 754 |

# Numeric Operators

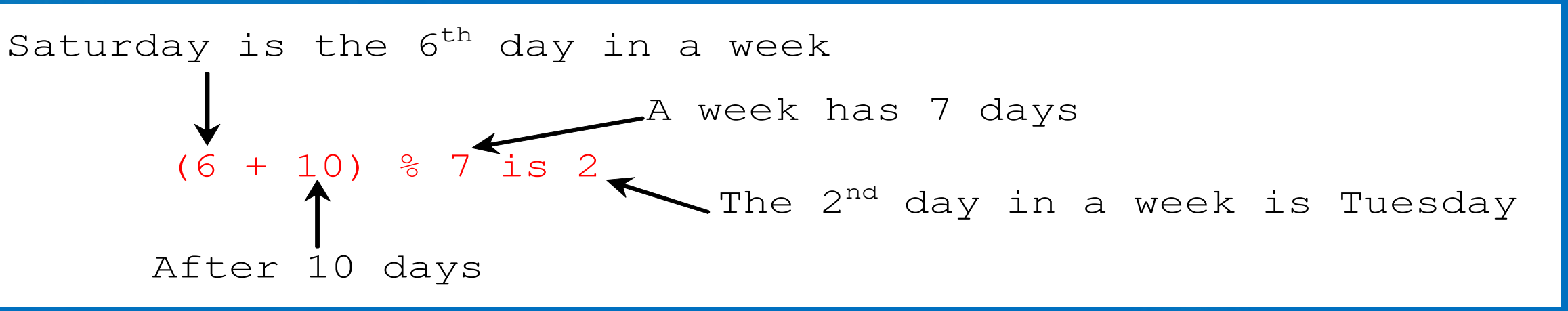


| Name | Meaning | Example | Result |
|---|---|---|---|
| + | Addition | 34 + 1 | 35 |
| - | Subtraction | 34.0 - 0.1 | 33.9 |
| * | Multiplication | 300 * 30 | 9000 |
| / | Division | 1.0 / 2.0 | 0.5 |
| % | Remainder | 20 % 3 | 2 |

# Remainder Operator



Remainder is very useful in programming. For example, an *even number %2* is always *0* and an *odd number* %2 is always *1*. So you can use this property to determine whether a number is even or odd. Suppose today is Saturday and you and your friends are going to meet in 10 days. *What day is in 10 days*? You can find that day is Tuesday using the following expression:



# Remainder Operator



```
public class TestClass {
    public static void main(String[] args) {
        int num = 20;
        if ( num % 2 == 0) {
            System.out.println("The number is even");
        }else {
            System.out.println("The number is odd");
        }
    }
}
```

# Reading Input from the Console



We can read input from the console using the Scanner class. Use the methods next(), nextByte(), nextShort(), nextInt(), nextLong(), nextFloat(), nextDouble(), or nextBoolean() to obtain to a string, byte, short, int, long, float, double, or boolean value.

```
import java.util.Scanner;
public class TestClass {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.print("Enter a integer:");
        int num = input.nextInt();
        System.out.println("The number entered is " + num);
    }
}
```

# Example: Displaying Time



Write a program that converts seconds to minutes. For example, 70 seconds = 1 minute and 10 seconds.

```
import java.util.Scanner;
public class TestClass {
    public static void main(String[] args) {
        System.out.print("Enter seconds: ");
        Scanner input = new Scanner(System.in);
        int seconds = input.nextInt();
        int minutes = seconds / 60;
        int remainSeconds = seconds % 60;
        System.out.printf("minutes:%d,seconds:%d",minutes,remainSeconds);
    }
}
```

```
Enter seconds: 800
minutes:13,seconds:20
```

# NOTE: Float Accuracy



Calculations involving floating-point numbers are *approximated* because these numbers are not stored with complete accuracy. For example:

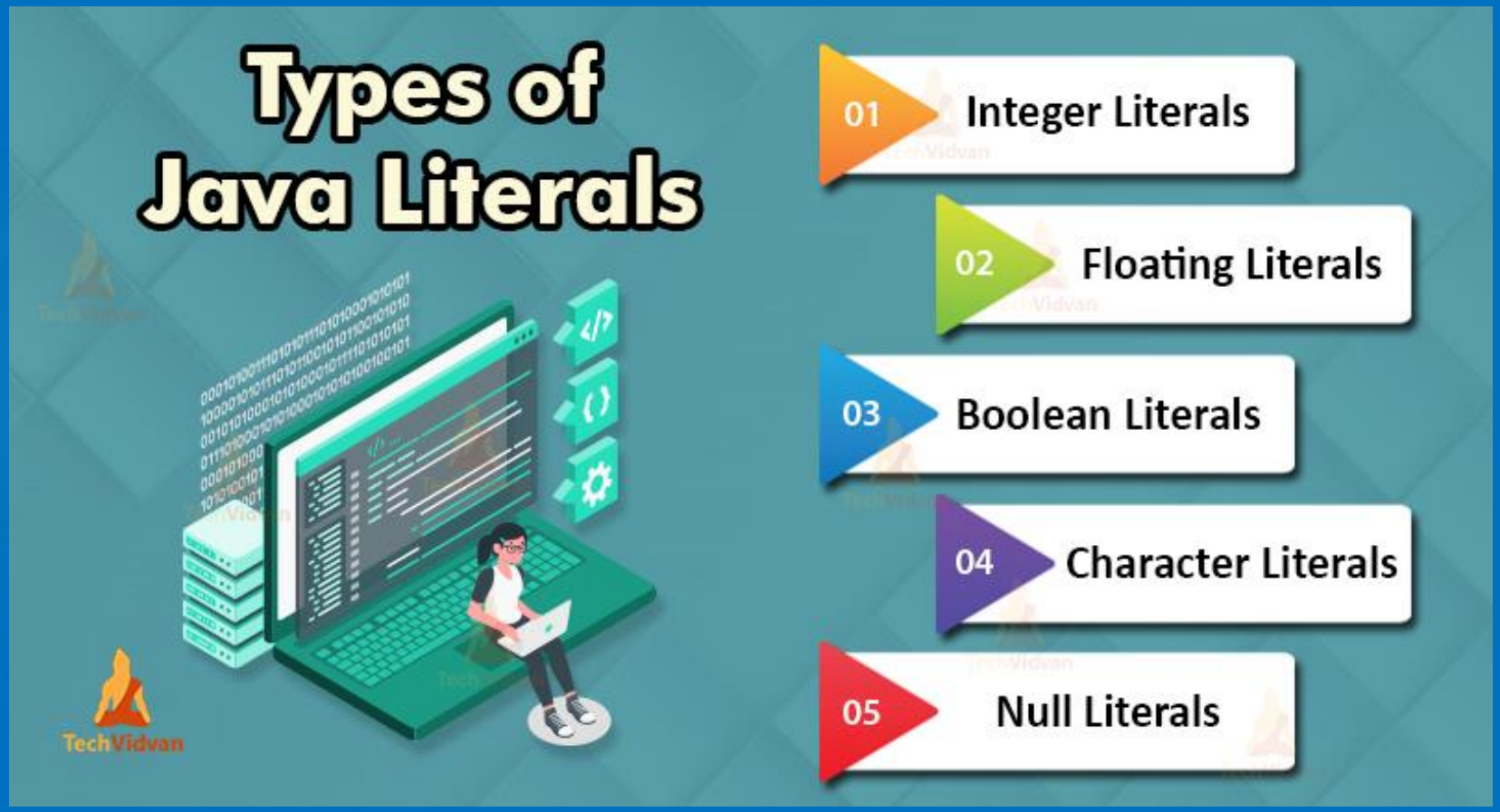
```
public class TestClass {
    public static void main(String[] args) {
        System.out.println(1.0-0.1-0.1-0.1-0.1-0.1);
        System.out.println(1.0-0.9);
    }
}
```

```
0.5000000000000001
0.09999999999999998
```

# Numeric Literals



Literals are the data items that have *fixed* or *constant values*. Literals can represent various types of values such as *numeric*, *character*, *boolean*, or *String* values.




```
int number = 20;
```

Variable Literal

# Integer Literals



```
public class TestClass {
    public static void main(String[] args) {
        int decimalLiteral = 561; // decimal literal
        int octalLiteral = 01204; // octal literal
        int hexLiteral = 0x1BfA; // Hexa-decimal literal
        System.out.println("Decimal-literal: " + decimalLiteral);
        System.out.println("Octal-literal: " + octalLiteral);
        System.out.println("Hex-literal: " +  hexLiteral);
    }
}
```

```
Decimal literal: 561
Octal literal: 644
Hex-literal: 7162
```

# Floating literals



```
public class TestClass {
    public static void main(String args[]) {
        double fl1 = 987.678; // fractional floating literal
        double fl2 = 089.0987; // fractional floating litera
        double fl3 = 1.234e20; // Exponential form
        System.out.println("fl1: " + fl1);
        System.out.println("fl2: " + fl2);
        System.out.println("fl3: " + fl3);
    }
}
```

```
fl1: 987.678
fl2: 89.0987
fl3: 1.234E20
```

# Boolean, Character and String Literals



```
public class TestClass {
    public static void main(String args[]) {
        boolean b1 = true;
        boolean b2 = false;
        char c = 'a';
        String s1 = "hi";
        String s2 = null;
    }
}
```

# Shortcut Assignment Operators



| Operator | Example | Equivalent |
|---|---|---|
| += | i += 8 | i = i + 8 |
| -= | f -= 8.0 | f = f - 8.0 |
| *= | i *= 8 | i = i * 8 |
| /= | i /= 8 | i = i / 8 |
| %= | i %= 8 | i = i % 8 |

```
public class TestClass {
    public static void main(String args[]) {
        int num = 0;
        num += 100; // num is 100
        num /= 2; // num is 50
        num -= 3; // num is 47
        num *= 2; // num is 94
        num %= 3; // num is 1
    }
}
```

# Increment and Decrement Operators



| Operator | Name | Description |
|---|---|---|
| **++var** | preincrement | Increment ***var*** by ***1*** and use the new ***var*** value |
| **var++** | postincrement | Increment ***var*** by ***1***, but use the original ***var*** value |
| **--var** | predecrement | Decrement ***var*** by ***1*** and use the new ***var*** value |
| **var--** | postdecrement | Decrement ***var*** by ***1*** and use the original ***var*** value |

```
int i = 10;
int newNum = 10 * i++;
```

Same effect as

```
int newNum = 10 * i;
i = i + 1;
```

```
int i = 10;
int newNum = 10 * (++i);
```

Same effect as

```
i = i + 1;
int newNum = 10 * i;
```

# Numeric Type Conversion



Casting a type with a small range to a type with a larger range is known as ***widening*** a type. Casting a type with a large range to a type with a smaller range is known as ***narrowing*** a type. Java will ***automatically widen*** a type, but you must ***narrow*** a type ***explicitly***.

```
public class TestClass {
    public static void main(String args[]) {
        long a = 12;
        int b = (int) 12L;
        double c = 12.12F;
        float d = (float) 12.12;
        float e = a;
        double f = b;
    }
}
```

# Numeric Type Conversion



When performing a *binary operation* involving two operands of *different types*, Java *automatically* converts the operand based on the following rules: If one of the operands is *double*, the other is converted into double. Otherwise, if one of the operands is *float*, the other is converted into float. Otherwise, if one of the operands is *long*, the other is converted into long. Otherwise, both operands are converted into *int*.

```
public class TestClass {
    public static void main(String args[]) {
        System.out.println(36.88f * 12);
        System.out.println(36.88 * 12);
    }
}
```

```
442.56
442.56000000000006
```

# The Math Class



The class *Math* contains methods for performing *basic numeric operations* such as the elementary exponential, logarithm, square root, and trigonometric functions.
https://docs.oracle.com/javase/8/docs/api/java/lang/Math.html

```
package java.lang;
import java.util.Random;

public final class Math {
    private Math() {}
    public static final double E = 2.718281828459045235 4;
    public static final double PI = 3.14159265358979323846;
    // other methods
}
```

# Trigonometric Methods



sin(double a), cos(double a), tan(double a), acos(double a), asin(double a), atan(double a)

```
public class TestClass {
    public static void main(String args[]) {
        System.out.println(Math.sin(Math.PI / 6));
        System.out.println(Math.cos(0));
        System.out.println(Math.tan(Math.PI / 4));
        System.out.println(Math.asin(0.5));
        System.out.println(Math.acos(1.0));
        System.out.println(Math.atan(1.0));
    }
}
```

```
0.4999999999999999
4
1.0
0.9999999999999999
0.5235987755982989
0.0
0.7853981633974483
```

# Exponent Methods



exp(double a), log(double a), log10(double a), pow(double a, double b), sqrt(double a)

```
public class TestClass {
    public static void main(String args[]) {
        System.out.println(Math.exp(1));
        System.out.println(Math.log(Math.E));
        System.out.println(Math.log10(10));
        System.out.println(Math.pow(2, 3));
        System.out.println(Math.sqrt(2));
    }
}
```

```
2.718281828459045
1.0
1.0
8.0
1.4142135623730951
```

# Rounding Methods



double ceil(double x): x rounded *up* to its nearest integer.

double floor(double x): x is rounded *down* to its nearest integer.

double rint(double x): x is rounded to its *nearest integer*. If x is *equally* close to two integers, the *even one* is returned as a double.

int round(float x): Return (int)Math.floor(x+0.5).

long round(double x): Return (long)Math.floor(x+0.5).

# Rounding Methods



```
public class TestClass {
    public static void main(String[] args) {
        System.out.println(Math.ceil(2.1));
        System.out.println(Math.ceil(2.0));
        System.out.println(Math.ceil(-2.0));
        System.out.println(Math.ceil(-2.1));
        System.out.println(Math.floor(2.1));
        System.out.println(Math.rint(2.1));
        System.out.println(Math.rint(-2.1));
        System.out.println(Math.rint(2.5));
        System.out.println(Math.round(2.6f));
        System.out.println(Math.round(-2.6));
    }
}
```

```
3.0
2.0
-2.0
-2.0
2.0
2.0
-2.0
2.0
3
-3
```

# Random Generating



double Math.random(): Returns a random double value in the range [0.0, 1.0).

```
public class TestClass {
    public static void main(String[] args) {
        int num = (int)(Math.random() * 10);
        System.out.println("Generate a random number (0~9): " + num);
        char c = (char) ('A' + (int)(Math.random() * 26));
        System.out.println("Generate a random letter (A~Z): " + c);
    }
}
```

```
Generate a random number (0~9): 5
Generate a random letter (A~Z): G
```

# Other Methods



max(a, b) and min(a, b): Returns the maximum or minimum of two parameters.

abs(a): Returns the absolute value of the parameter.

```
public class TestClass {
    public static void main(String[] args) {
        int a = 20;
        int b = 30;
        System.out.println(Math.max(a, b));
        System.out.println(Math.max(a, b));
        System.out.println(Math.abs(-128));
    }
}
```

```
30
30
128
```

# Character Data Type（16bits）



```
public class TestClass {
    public static void main(String[] args) {
        char letter1 = 'A';
        char numChar1 = '4';
        char letter2 = '\u0041'; //Unicode
        char numChar2 = '\u0034';
        System.out.println(letter1); //A
        System.out.println(letter2++); //B
        System.out.println(numChar1); //4
        System.out.println(numChar2); //4
    }
}
```

NOTE: The *increment* and *decrement operators* can also be used on char variables to get the next or preceding Unicode character.

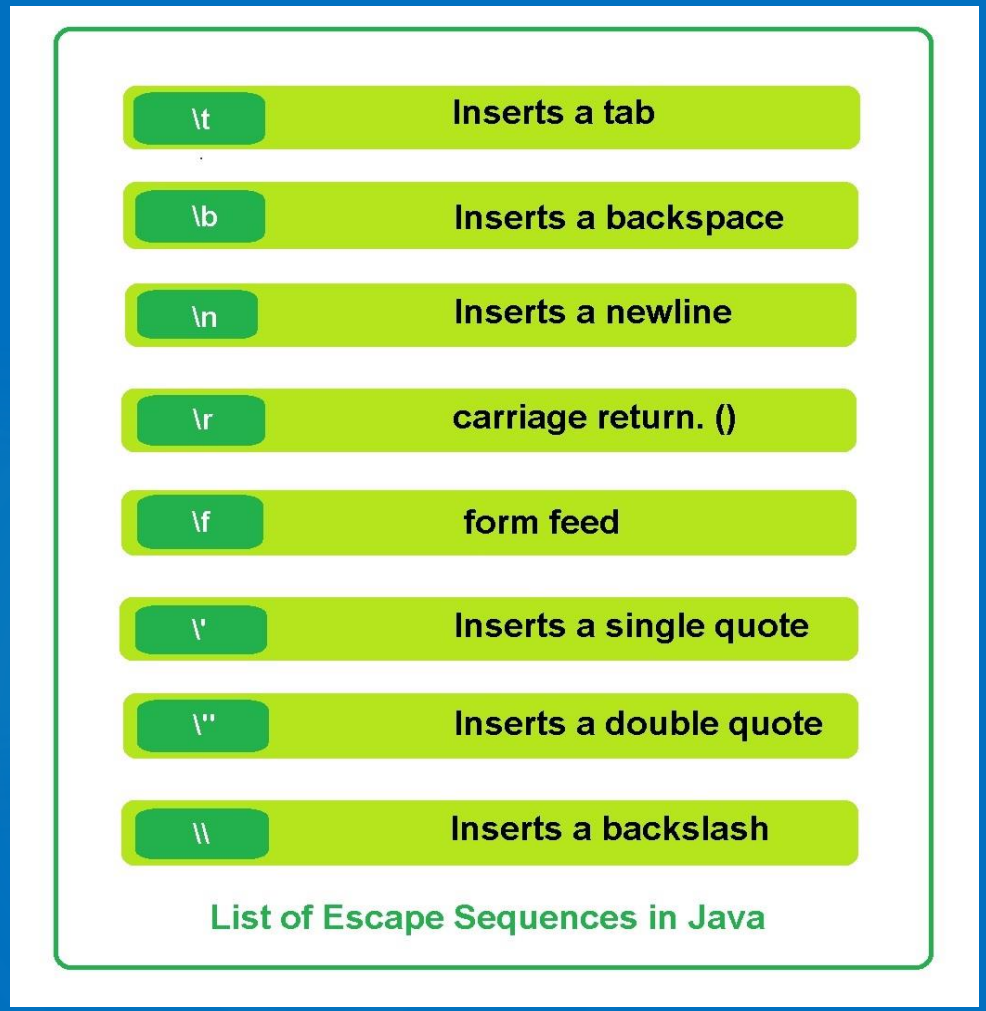
# ASCII Character Set is a subset of the Unicode from \u0000 to \u007f

TABLE B.1 ASCII Character Set in the Decimal Index

| | *0* | *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* | *9* |
|---|---|---|---|---|---|---|---|---|---|---|
| **0** | nul | soh | stx | etx | eot | enq | ack | bel | bs | ht |
| **1** | nl | vt | ff | cr | so | si | dle | dcl | dc2 | dc3 |
| **2** | dc4 | nak | syn | etb | can | em | sub | esc | fs | gs |
| **3** | rs | us | sp | ! | ″ | # | $ | % | & | ’ |
| **4** | ( | ) | * | + | , | - | . | / | 0 | 1 |
| **5** | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | : | ; |
| **6** | < | = | > | ? | @ | A | B | C | D | E |
| **7** | F | G | H | I | J | K | L | M | N | O |
| **8** | P | Q | R | S | T | U | V | W | X | Y |
| **9** | Z | [ | \ | ] | ^ | _ | ‘ | a | b | c |
| **10** | d | e | f | g | h | i | j | k | l | m |
| **11** | n | o | p | q | r | s | t | u | v | w |
| **12** | x | y | z | { | \| | } | ~ | del | | |

# Escape Sequences






```java
public class TestClass {
    public static void main(String[] args) {
        System.out.println("10\t20\t30");
        System.out.println("40\t50\t60");
        System.out.println("70\t80\t90");
    }
}
```

```
10 20 30
40 50 60
70 80 90
```

# Casting between char and Numeric



```
public class TestClass {
    public static void main(String[] args) {
        for(int i = 0; i < 100; i++) {
            char c = (char)('A' + (char)(Math.random() * 26));
            System.out.print(c);
            if(i != 0 && (i + 1) % 20 == 0)
                System.out.print("\n");
        }
    }
}
```

```
IXGUOQIAEAMQLAFHPKEC
DYPSXVNKFKAILAUTRRCK
HXAXHZGRERVQOLUTCWYU
NBXOCJNOQYSSSTCBHSXT
XOVPIJACSOZLILTJGTLJ
```

# Comparing and Testing Characters



```
public class TestClass {
    public static void main(String[] args) {
        char ch = 'Y';
        if (ch >= 'A' && ch <= 'Z')
              System.out.println(ch + " is an uppercase letter");
        else if (ch >= 'a' && ch <= 'z')
          System.out.println(ch + " is a lowercase letter");
        else if (ch >= '0' && ch <= '9')
          System.out.println(ch + " is a numeric character");
    }
}
```

```
Y is an uppercase letter
```

# The Character Class



The *Character* class *wraps* a value of the primitive type *char* in an object. In addition, this class provides a large number of *static methods* for determining a *character's category* (lowercase letter, digit, etc.) and for *converting characters* from uppercase to lowercase and vice versa.

```
public class TestClass {
    public static void main(String[] args) {
        char ch = 'Y';
        System.out.println(Character.toLowerCase(ch)); //y
    }
}
```

# The String Type



The *String* class represents character strings. All *string literals* in Java programs, such as "abc", are implemented as *instances* of this class. The String type is *not a primitive type*. It is known as a *reference* type.

```
public class TestClass {
    public static void main(String[] args) {
        String s1 = "Hello"; //String Creation
        String s2 = new String("Xiaoming"); //String Creation
        String s = s1 + " " + s2 + "!"; //String Concatenation
        System.out.println(s);
    }
}
```

```
Hello Xiaoming!
```

# Simple Methods for String Objects



```
public class TestClass {
    public static void main(String[] args) {
        System.out.println("abcd".length());
        System.out.println("abcd".charAt(1));
        System.out.println("abcd".concat("efg"));
        System.out.println("abcd".toUpperCase());
        System.out.println("ABCD".toLowerCase());
        System.out.println(" abcd".trim());
    }
}
```

```
4
b
abcdefg
ABCD
abcd
abcd
```

# Comparing Strings



```
public class TestClass {
    public static void main(String[] args) {
        System.out.println("abcd".equals("abcd"));
        System.out.println("abcd".equalsIgnoreCase("ABCD"));
        System.out.println("abcd".compareTo("bbcd"));
        System.out.println("abcd".startsWith("ab"));
        System.out.println("abcd".endsWith("cd"));
    }
}
```

```
true
true
-1
true
true
```

# Obtaining Substrings



| Method | Description |
|---|---|
| `substring(beginIndex)` | Returns this string's substring that begins with the character at the specified `beginIndex` and extends to the end of the string |
| `substring(beginIndex, endIndex)` | Returns this string's substring that begins at the specified `beginIndex` and extends to the character at index `endIndex - 1` |

```
public class TestClass {
    public static void main(String[] args) {
        System.out.println("abcdefghi".substring(2));
        System.out.println("abcdefghi".substring(3, 6));
    }
}
```

```
cdefghi
def
```

# Finding a Character or a Substring



| Method | Description |
|---|---|
| `indexOf(ch)` | Returns the index of the first occurrence of `ch` in the string. Returns `-1` if not matched. |
| `indexOf(ch, fromIndex)` | Returns the index of the first occurrence of `ch` after `fromIndex` in the string. Returns `-1` if not matched. |
| `indexOf(s)` | Returns the index of the first occurrence of string `s` in this string. Returns `-1` if not matched. |
| `indexOf(s, fromIndex)` | Returns the index of the first occurrence of string `s` in this string after `fromIndex`. Returns `-1` if not matched. |
| `lastIndexOf(ch)` | Returns the index of the last occurrence of `ch` in the string. Returns `-1` if not matched. |
| `lastIndexOf(ch, fromIndex)` | Returns the index of the last occurrence of `ch` before `fromIndex` in this string. Returns `-1` if not matched. |
| `lastIndexOf(s)` | Returns the index of the last occurrence of string `s`. Returns `-1` if not matched. |
| `lastIndexOf(s, fromIndex)` | Returns the index of the last occurrence of string `s` before `fromIndex`. Returns `-1` if not matched. |

# Formatting Output



```
public class TestClass {
    public static void main(String[] args) {
        int x = 100;
        System.out.printf("Printing simple integer: x = %d\n", x);
        System.out.printf("Formatted precison: PI = %.2f\n", Math.PI);
        float n = 5.2f;
        System.out.printf("Formatted width: n = %.4f\n", n);
        n = 2324435.3f;
        System.out.printf("Formatted margin: n = %20.4f\n", n);
    }
}
```

```
Printing simple integer: x = 100
Formatted precison: PI = 3.14
Formatted width: n = 5.2000
Formatted margin: n =         2324435.2500
```

# Formatting Output



| Specifier | Output | Example |
|---|---|---|
| %b | a boolean value | true or false |
| %c | a character | 'a' |
| %d | a decimal integer | 200 |
| %f | a floating-point number | 45.460000 |
| %e | a number in standard scientific notation | 4.556000e+01 |
| %s | a string | "Java is cool" |

# Programming Style and Documentation



- Appropriate Comments
- Naming Conventions
- Proper Indentation and Spacing Lines
- Block Styles

# Appropriate Comments



Include a *summary* at the beginning of the program to explain *what the program does*, its *key features*, its *supporting data structures*, and any *unique techniques* it uses. Include your name, class section, instructor, date, and a brief description at the beginning of the program.

# Naming Conventions



Choose *meaningful* and *descriptive* names.

- Variables and method names: Use *lowercase*. If the name consists of several words, concatenate all in one, use *lowercase for the first word*, and *capitalize* the first letter of each subsequent word in the name.
- Class names: *Capitalize* the first letter of each word in the name. For example, the class name *ComputeArea*.
- Constants: *Capitalize all letters* in constants, and use *underscores* to connect words. For example, the constant PI and *MAX_VALUE*

# Proper Indentation and Spacing



A single space line should be used to separate segments of the code to make the program easier to read. For example:

```
int i = 3+4 * 4;  //Bad style
int i = 3 + 4 * 4; //Good style
```

# Block Styles



*Next-line style*

```
public class Test
{
  public static void main(String[] args)
  {
    System.out.println("Block Styles");
  }
}
```

*End-of-line style*

```
public class Test {
  public static void main(String[] args) {
    System.out.println("Block Styles");
  }
}
```

# Programming Errors



- Syntax Errors: Detected by the *compiler*
- Runtime Errors: Causes the program to *abort*
- Logic Errors: Produces *incorrect result*

# Syntax Errors



```
public class TestClass {
    public static void main(String[] args) {
        i = 30;
        System.out.println(i + 4);
        int j;
        System.out.println(j + 4);
    }
}
```

# Runtime Errors



```
public class TestClass {
    public static void main(String[] args) {
        int i = 1 / 0;
    }
}
```

# Logic Errors



```
//Print 1 to 10 integers
public class TestClass {
    public static void main(String[] args) {
        for(int i = 0; i <= 10; i++){
            System.out.print(i);
        }
    }
}
```

# Debugging



Logic errors are called *bugs*. The process of *finding* and *correcting errors* is called *debugging*. A common approach to debugging is to use a combination of methods to *narrow down* to the part of the program where the bug is located. You can *hand-trace* the program (i.e., catch errors by *reading the program*), or you can *insert print statements* in order to show the values of the variables or the *execution flow* of the program. This approach might work for a short, simple program. But for a large, complex program, the most effective approach for debugging is to use a *debugger utility*.

# Debugger

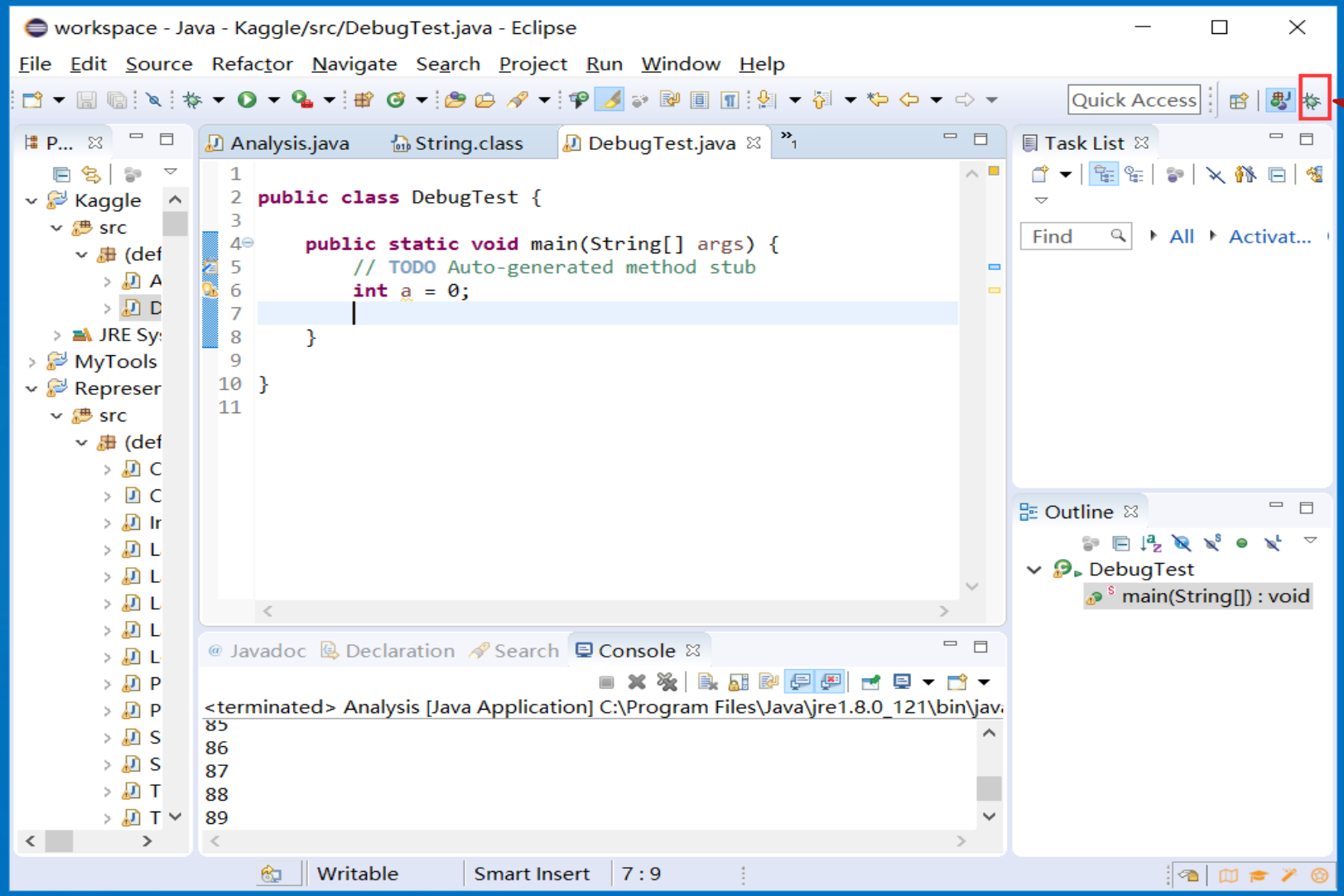


Debugger is a program that facilitates debugging. You can use a debugger to

- Execute a single statement at a time.
- Trace into or stepping over a method.
- Set breakpoints.
- Display variables.
- Display call stack.
- Modify variables.

# Eclipse Debugger

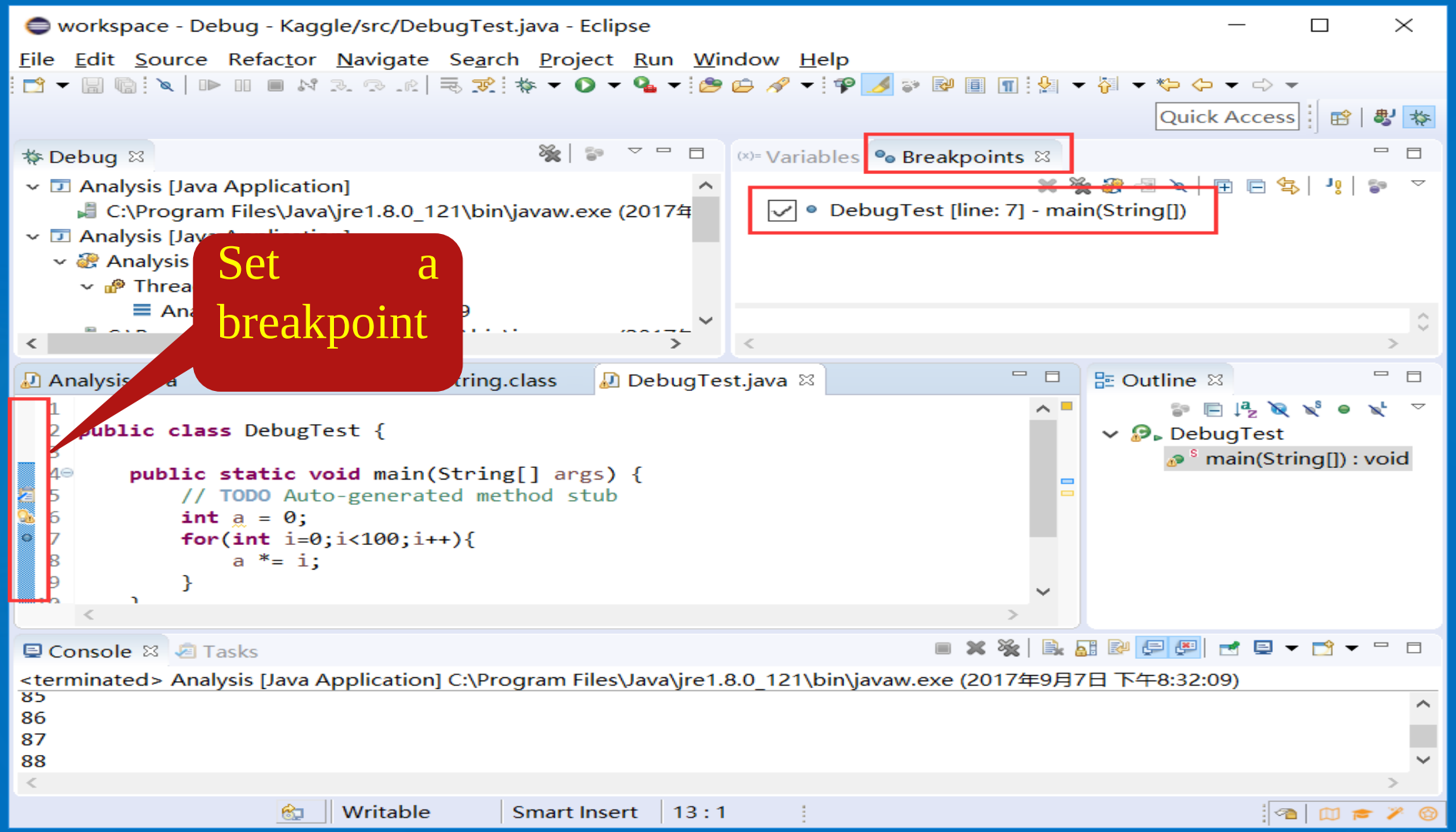





Click this button and change to debugging view

# Eclipse Debugger





# Eclipse Debugger