

Types, Values and Variables (Chapter 4, JLS)

Primitive Types	Values	Representation
boolean	{false, true}	1-bit (possibly padded to 1 byte)
Numeric Types		
Integral Types	$\subseteq \mathbf{Z}$	
byte	$-2^7 \dots (2^7 - 1)$	8-bit signed two's complement
short	$-2^{15} \dots (2^{15} - 1)$	16-bit signed two's complement
int	$-2^{31} \dots (2^{31} - 1)$	32-bit signed two's complement
long	$-2^{63} \dots (2^{63} - 1)$	64-bit signed two's complement
char	$0..(2^{16} - 1)$	16-bit Unicode (www.unicode.org)
Floating-Point Types	$\subseteq \mathbf{R}$	
float		32-bit ANSI/IEEE Standard 754
double		64-bit ANSI/IEEE Standard 754
Reference Types		
Class Type	user-defined objects	
e.g. Object		
e.g. String		
Array Type	sequence of <i>Type</i>	
e.g. int[]		
e.g. String[]		

Variables

Primitive Types

- Value stored in variable
 - e.g. `boolean x = true;`
 - e.g. `int y = 5;`
 - e.g. `char z = 'A';`
- Default value stored is false or 0
 - e.g. `boolean x;`
 - e.g. `int y;`
- final variables can be assigned at most once
 - e.g. `final int SIZE = 10;`
 - e.g. `final char EXCELLENT = 'A';`
 - e.g. `final float PI = 3.14159;`

Reference Types

- Reference (i.e., pointer) to value (i.e., object) stored in variable
 - e.g. `String x = "hello";`
 - e.g. `Widget y = new Widget();`
 - e.g. `int[] z = {3, 2, 4, 5};`
- Default reference stored is null, which refers to nothing
 - e.g. `String x;`
 - e.g. `int[] y;`
- final variables can be assigned at most once; however, objects to which they refer can be modified
 - e.g. `final String GREETING = "hello";`
 - e.g. `final int[] CUTOFFS = {90, 80, 70, 60};`

Expressions (Chapter 15, JLS)

Literals

Variables

Operations:

Prec./Assoc.	Category	Operator	Operand(s)	Result	
high	n/a	Postfix	++ --	numeric variable	numeric value, before side effect
↓	n/a	Prefix	++ --	numeric variable	numeric value, after side effect
↓	←	Unary	+ - ~ !	$\left\{ \begin{array}{l} \text{numeric (+ -)} \\ \text{integral (~)} \\ \text{boolean (!)} \end{array} \right.$	$\left\{ \begin{array}{l} \geq 32\text{-bit numeric value} \\ \geq 32\text{-bit integral value} \\ \text{boolean value} \end{array} \right.$
↓	→	Multiplicative	* / %	numeric, numeric	$\geq 32\text{-bit numeric value}$
↓	→	Additive	+ -	numeric, numeric	$\geq 32\text{-bit numeric value}$
↓	→	Shift	<< >> >>>	integral, integral	$\geq 32\text{-bit integral value}$
↓	n/a	Relational	< > <= >=	numeric, numeric	boolean value
↓	→	Equality	== !=	$\left\{ \begin{array}{l} \text{boolean, boolean} \\ \text{numeric, numeric} \\ \text{reference, reference} \end{array} \right.$	boolean value
↓	→	Bitwise	$\left\{ \begin{array}{l} \& \\ \wedge \\ \end{array} \right.$	$\left\{ \begin{array}{l} \text{boolean, boolean} \\ \text{integral, integral} \end{array} \right.$	$\left\{ \begin{array}{l} \text{boolean value} \\ \geq 32\text{-bit integral value} \end{array} \right.$
↓	→	Boolean	$\left\{ \begin{array}{l} \&\& \\ \end{array} \right.$	boolean, boolean	boolean value, short-circuit
↓	←	Conditional	?:	boolean, any, any	value, short-circuit
low	←	Assignment	= *= /= %= += -= <<= >>= >>>= &= ^= =	$\left\{ \begin{array}{l} \text{variable, any (=)} \\ \text{numeric variable, numeric (*= /= %= += -=)} \\ \text{integral variable, integral (<<= >>= >>>= \&= ^= =)} \\ \text{boolean variable, boolean (\&= ^= =)} \end{array} \right.$	value, after side effect

- Order of operations determined by
 - 1) parentheses
 - 2) precedence
 - 3) associativity
- Evaluation: result (and possible side effect) obtained by
 - 1) evaluating left operand
 - 2) evaluating right operand (unless short-circuit)
 - 3) applying operation

Statements (Chapter 14, JLS)

- Block

{ Statement }*

- used to execute a sequence of statements

- Empty statement

;

- has no effect

- Labeled statement

Identifier : Statement

- used with labeled break and continue statements

- Local variable declaration

[final] Type Identifier [= Expression](, Identifier [= Expression]) ;*

- used to create local variable(s) and optionally assign initial value(s)
- scope of variable is from point of declaration to end of block in which declared

- Expression

Expression ;

- *Expression* must have a side effect

- if statement

if (Expression) Statement [else Statement]

- condition *Expression* must be of type boolean

- switch statement

switch (Expression) { ((case Expression :)⁺ Statement⁺)[default : Statement⁺] }*

- condition *Expression* must be integral
- case *Expression*(s) must be integral constant (literal or final variable)

- while statement
 - while (*Expression*) *Statement*
 - condition *Expression* must be of type boolean
- do statement
 - do *Statement* while (*Expression*) ;
 - condition *Expression* must be of type boolean
- for statement
 - for ([*Expression*(, *Expression*)*]; [*Expression*]; [*Expression*(, *Expression*)*]) *Statement*
 - for (*LocalVariableDeclaration*; [*Expression*]; [*Expression*(, *Expression*)*]) *Statement*
 - initial *Expression*(s) must have a side effect
 - condition *Expression* must be of type boolean
 - update *Expression*(s) must have a side effect
- break statement
 - break [*Identifier*] ;
 - must occur inside a switch, while, do or for statement
- continue statement
 - continue [*Identifier*] ;
 - must occur inside a while, do or for statement

Arrays (Chapter 10, Sections 15.10 and 15.13, JLS)

- Arrays are objects that contain zero or more component variables.
- Arrays are homogeneous—components have the same type.
- Arrays are created dynamically (as the program executes).
- The length of an array is the number of components therein; it does not change after creation.

- Array type

Type (*[]*)⁺

- component type may be any *Type* (including array types)
- dimension is number of pairs of *[]*

- Array initializer

{ [*Expression* (, *Expression*)^{*}] [,] }

{ [*ArrayInitializer* (, *ArrayInitializer*)^{*}] [,] }

- creates an array
- components are assigned from expression values
- length is number of expressions or array initializers

- Array creation expression

new ArrayType ArrayInitializer

- creates an array having a specific component type
- dimension, length and compatible component values can be given

new Type ([*Expression*])⁺ (*[]*)^{*}

- dimension and lengths (in some dimensions) can be given by expression values

- Array type local variable declaration

[*final*] *ArrayType Identifier* [= *Expression*] (, *Identifier* [= *Expression*])^{*} ;

[*final*] *ArrayType Identifier* [= *ArrayInitializer*] (, *Identifier* [= *ArrayInitializer*])^{*} ;

- used to create local array variable(s) and optionally assign initial component value(s)
- expressions must be array creation expressions
- scope of variable is from point of declaration to end of block in which declared

- Array access expression

Expression [*Expression*]

- array reference expression must be array type and have non-null value
- index expression must be integral type and have value between 0 and length-1
- first, second, third, . . . components are accessed by index expression values 0, 1, 2, . . .

- Array length expression

Identifier . *length*

```

class Sorting {

    public static void main(String[] arguments) {
        int[] v;
//-----
        System.out.println("Bubble sorting...");
        println(v = new int[]{8, 2, 5, 3, 9, 4, 6, 1, 10, 7});

        for (int i = v.length-1; i > 0; i--)

            for (int j = 0; j < i; j++)

                if (v[j] > v[j+1]) swap(v, j, j+1);
        println(v);
//-----
        System.out.println("Selection sorting...");
        println(v = new int[]{8, 2, 5, 3, 9, 4, 6, 1, 10, 7});

        for (int i = 0; i < v.length-1; i++) {
            int lowIndex = i;

            for (int j = i+1; j < v.length; j++)

                if (v[j] < v[lowIndex]) lowIndex = j;

            swap(v, lowIndex, i);
        }
        println(v);
//-----
        System.out.println("Insertion sorting...");
        println(v = new int[]{8, 2, 5, 3, 9, 4, 6, 1, 10, 7});

        for (int i = 1; i < v.length; i++)

            for (int j = i; j > 0; j--)

                if (v[j] < v[j-1]) swap(v, j, j-1);
        println(v);
//-----
    }

    private static void swap(int[] v, int i, int j) { // swap v[i] and v[j]
        int temp = v[i];
        v[i] = v[j];
        v[j] = temp;
    }

    private static void println(int[] v) { // print array v

        for (int i = 0; i < v.length; i++)
            System.out.print(" " + v[i]);
        System.out.println();
    }
}

```

Methods (Sections 8.4 and 15.12, JLS)

- A method declares an encapsulated sequence of statements that can be invoked.
- On invocation, a fixed number of argument values are passed into a method as parameters.
- On completion, either nothing (`void`) or one value is returned.

- Method modifier

- `public`
 - `protected`
 - `private`

- determines how widely a method is visible/accessible (default is package private)

- `abstract`

- determines if a method has no body (`abstract`)

- `static`

- determines if a method is a class method (`static`) or an instance method (`non-static`)
 - a class method is associated (statically) with its class
 - an instance method is associated (dynamically) with every object (instance) of its class type

- `final`

- determines if a method cannot be overridden by a subclass of its class (`final`)

- Formal parameters

- `[final] Type Identifier (, [final] Type Identifier)*`

- determine number, type and order of values passed into a method
 - viewed as local variable declarations by encapsulated statements
 - assigned incoming values on invocation

- Method header

- `MethodModifier* Type Identifier ([FormalParameters]) [throws Type (, Type)*]`
`MethodModifier* void Identifier ([FormalParameters]) [throws Type (, Type)*]`

- specifies modifiers, return type, name, formal parameters, thrown exception types
 - method signature consists of method name and formal parameters

- Method declaration

- `MethodHeader { Statement* }`

- occurs within a class declaration
 - non-abstract method body must be a sequence of statements (i.e., block)

- `MethodHeader ;`

- abstract method body must be omitted

- Argument list

*Expression (, Expression)**

- list of expressions evaluated from left-to-right
- expression values are assigned to corresponding formal parameters

- Method invocation expression

Identifier (ArgumentList)

- invokes the method with matching signature
- causes evaluation of arguments, assignment to formal parameters, execution of body
- can have side effects (caused by argument evaluation or body execution)

- return statement

return [Expression] ;

- causes abrupt completion of encapsulated statements
- optionally contains an expression whose value is returned by a method
- non-void method bodies must contain (and eventually execute) a return statement
- non-void method bodies may not contain return statements without expressions
- void method bodies may not contain return statements with expressions

Classes (Chapter 8, Section 15.9 and 15.11, JLS)

- A class declares encapsulated members—fields (i.e., variables), methods or other classes.
- A class declares a user-defined reference type; i.e., a class type.
- Objects (instances) of the class type can be created.
- Encapsulated members may be class members (`static`) or instance members (`non-static`).
- Class members are associated (statically) with the class.
- Instance members are associated (dynamically) with every object (instance) of the class type.

- Field modifier

- `public`
 - `protected`
 - `private`

- determines how widely a field is visible/accessible (default is package private)

- `static`

- determines if a field is a class variable (`static`) or an instance variable (`non-static`)
 - a class variable is associated (statically) with the class
 - an instance variable is associated (dynamically) with every object (instance) of the class type

- `final`

- determines if a field can be assigned at most once (`final`)

- Field declaration

- FieldModifier** *Type Identifier* [= *Expression*](, *Identifier* [= *Expression*])* ;

- used to create class or instance variable(s) and optionally assign initial value(s)
 - scope of variable depends on visibility/accessibility field modifiers

- Field access expression

- Type* . *Identifier*

- *Type* must be a class type
 - *Identifier* must be a class variable in class *Type*
 - result is class variable in class *Type*

- Expression* . *Identifier*

- *Expression* must be of class type
 - *Identifier* must be an instance variable in class
 - result is instance variable in object referred to by *Expression* value

- Static initializer

- `static` *Block*

- block of statements initializes class variables before class methods are invoked

- Class Instance Creation Expression

`new Type ([ArgumentList])`

- creates an object (instance) of class type *Type*
- object created contains its own (non-static) instance variables and instance methods
- object created does not contain (static) class variables or class methods
- invokes instance initializer in class *Type*, if any
- invokes the constructor with matching signature in class *Type*
- result is a reference to the object created

- Instance initializer

Block

- block of statements initializes instance variables before constructors or instance methods are invoked

- Constructor modifier

`public`
`protected`
`private`

- determines how widely a constructor is visible/accessible (default is package private)

- Constructor header

`ConstructorModifier* Type ([FormalParameters]) [throws Type (, Type)*]`

- specifies modifiers, name (same as class name), formal parameters, thrown exception types
- constructor signature consists of class name and formal parameters

- Constructor declaration

- like an instance method whose name is the class name
- invoked only by a class instance creation (`new`) expression
- block of statements initializes instance variables depending on formal parameters

- `this` expression

`this`

- must occur in the body of an instance initializer, constructor or instance method
- result is a reference to object whose instance initializer, constructor or instance method is executing

- Class modifier

- public
 - protected
 - private

- determines how widely a class type is visible/accessible (default is package private)

- abstract

- determines if a class contains methods with no body (abstract)

- static

- determines if a class is static or an inner class (non-static)

- a static class is associated (statically) with the class

- an inner class is associated (dynamically) with every object (instance) of the class type

- final

- determines if a class cannot have subclasses (final)

- Class Body Declaration

- FieldDeclaration*

- MethodDeclaration*

- StaticInitializer*

- InstanceInitializer*

- ConstructorDeclaration*

- ClassDeclaration*

- InterfaceDeclaration*

- a member encapsulated within a class declaration

- Class Declaration

- ClassModifier** class *Type* [*extends Type*] [*implements Type*(, *Type*)*] {
 *ClassBodyDeclaration**
}

- type extended is called direct superclass (i.e. parent type), Object if omitted

- type declared is a subclass of its direct superclass (i.e. parent type)

- types implemented must be interfaces, called direct superinterfaces

Exceptions (Chapter 11, Sections 15.6, 14.9 and 14.7, JLS)

- An exception signals that a running program has violated a semantic constraint
- An exception transfers control abruptly from point *thrown* to point *caught*
- An exception is represented by an instance of class `Throwable` or its subclasses
- An exception can be thrown during expression evaluation:

<i>Expression</i>	Potential Exception
<i>/ %</i>	<code>ArithmeticException</code>
<i>ArrayCreationExpression</i>	<code>OutOfMemoryError</code> <code>NegativeArraySizeException</code>
<i>ArrayAccessExpression</i>	<code>NullPointerException</code> <code>ArrayIndexOutOfBoundsException</code>
<i>MethodInvocationExpression</i>	<code>NullPointerException</code> <code>StackOverflowError</code> <code>Throwable</code> (from method body)
<i>ClassInstanceCreationExpression</i>	<code>OutOfMemoryError</code> <code>Throwable</code> (from constructor body)
<i>FieldAccessExpression</i>	<code>NullPointerException</code>

- An exception can be thrown by a `throw` statement
- An exception can be caught by a `try` statement or propagated “upward” to caller
 - `try` statement

```
try Block (catch ( FormalParameter ) Block)+
```

 - catch clause parameter must be of type `Throwable` or its subclass types
 - `try` block is executed
 - exception thrown within `try` block is caught by first catch clause to which it can be passed
 - catch clause that catches the exception (if any) is executed
 - `throw` statement

```
throw Expression ;
```

 - causes an exception to be thrown
 - expression must refer to an exception that
 - 1) is unchecked—of type `Error` or `RuntimeException` or their subclass types, or
 - 2) is checked and
 - 2a) is thrown within a `try` block having a catch clause to which it can be passed, or
 - 2b) is thrown within a method or constructor whose `throws` clause lists its type

Interfaces (Chapter 9, Section 8.4.3.1, JLS)

- An abstract method is a public instance method with no body
- An abstract method declares publicized object behavior (not yet implemented)
- An abstract method cannot be invoked
- An abstract method must be contained in an abstract class

- Abstract method modifier

`public`

- must be `public`, implied if omitted
- cannot be `protected` or `private`

`abstract`

- determines if a method has no body (`abstract`)
- cannot be `static`, must be an instance method

- Abstract method header

*AbstractMethodModifier** *Type Identifier* ([*FormalParameters*]) [*throws Type* (, *Type*)*]

*AbstractMethodModifier** *void Identifier* ([*FormalParameters*]) [*throws Type* (, *Type*)*]

- specifies modifiers, return type, name, formal parameters, thrown exception types
- method signature consists of method name and formal parameters

- Abstract method declaration

AbstractMethodHeader ;

- An abstract class may contain abstract methods
- An interface is an abstract class that contains only constants and abstract methods

- Interface modifier

- `public`

- must be `public`, implied if omitted

- `abstract`

- must be `abstract`, implied if omitted
 - cannot be `static` or `final`

- Interface Member Declaration

- ConstantDeclaration* (`public static final`, implied if omitted)

- AbstractMethodDeclaration*

- ClassDeclaration*

- InterfaceDeclaration*

- ;

- a member encapsulated within an interface declaration

- Interface Declaration

- InterfaceModifier** `interface Type` [`extends Type+`] {

- InterfaceMemberDeclaration**

- }

- types extended must be interfaces, called direct superinterfaces

Inheritance (Sections 8.2 and 8.4.6, JLS)

Recall class declarations:

- Class Declaration

```
ClassModifier* class Type [extends Type] [implements Type(, Type)*] {  
    ClassBodyDeclaration*  
}
```

- type extended is called direct superclass (i.e. parent type), Object if omitted
 - type declared is called a subclass of its direct superclass (i.e. parent type)
 - types implemented must be interfaces, called direct superinterfaces
- A class inherits all non-private members of its direct superclass and superinterfaces
 - A class does not inherit static initializers, instance initializers or constructors
 - Instance methods declared in the class override inherited ones with the same signature
 - An overridden method can be invoked using a method invocation expression qualified with `super`.
 - A method name shared by multiple methods with different signatures is called overloaded