Need of Data types

Data Types are used to store data *temporarily* in computer through a program.

In real world we have different types of data like integer, floating-point, character, boolean, and string, etc. To store all these types of data in program to perform business required calculation and validations we must use data types concept.

Definition of data type

Data type is something which gives information about

- Size of the memory location and range of data that can be accommodated inside that location.
- Possible legal operations those can be performed on that location.
  For Example: on *boolean* data we cannot perform addition operation.
- What type of result comes out from an expression when these types are used in side that expression.

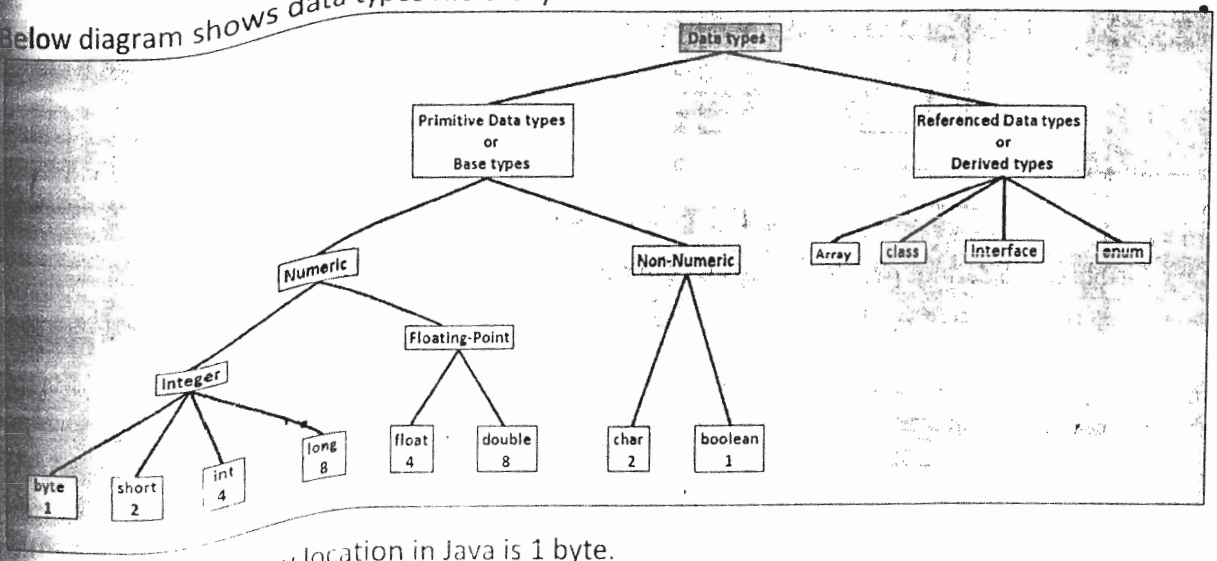
Whichever the keyword gives these semantics is treated as "data type".

Different Java data types

In Java mainly we have two types of data types

- Primitive types (8) - used to store single value at a time
- Referenced types (4) - used to collect multiple of values using primitive types.

Below diagram shows data types Hierarchy



The minimum memory location in Java is 1 byte.

Why do we have 8 primitive types in Java?

Based on *type and range* of data, primitive types are divided into 8 types.

Why do we have 4 referenced types?

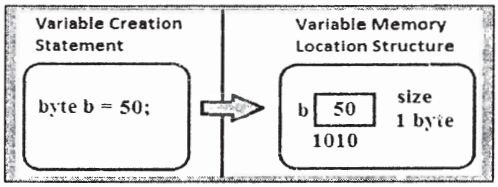
To collect *same type* of values Array is given and
To collect *different type* of values class, interface, enum are given

What JVM does when it encounters data type inside program?

It creates memory location based on the data type size, names that ML with the given name, and stores the assigned value in that memory location as show below. This named memory location is technically called *variable*.

In the below example, we have created a variable of type **byte** with the name **b** to store value **50**, so JVM creates memory location with **size 1 byte** at the address say 1010 and **named it as b** and stores the assigned value 50 in that memory location.

Below diagram shows variable creation and its memory location structure.



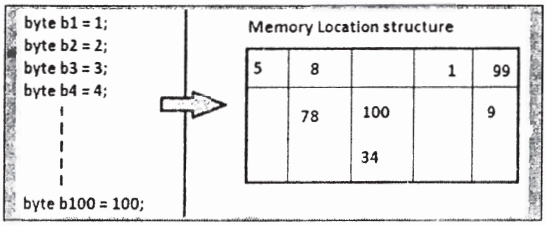
Limitations of primitive Data Types

Using primitive data types we cannot store multiple values in continuous memory locations.

Due to this limitation we face below two problems

Problem #1

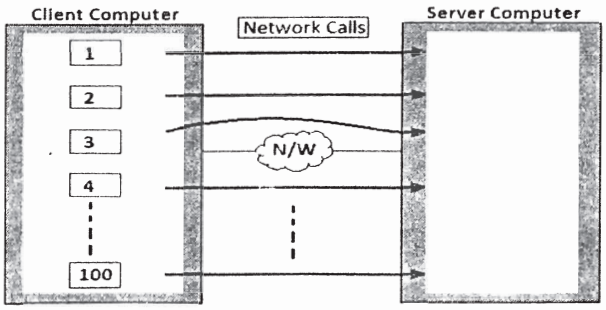
For instance if we want to store multiple values, say 1 to 100, we must create 100 variables. All those 100 variables are created across JVM at different locations as show below. Hence it takes more time to retrieve values.



Problem #2

Also using variables we cannot pass all values to the remote computer with single network call, which increases burden on network and also increase lines of code in program.

Below diagram shows the above limitation.



To pass 100 values it consumes 100 network calls.

Solution

To solve above two problems, we must group all values to send them as a single unit from one application to another application as method argument or return type. To group them as a single unit we must store them in continuous Memory Locations. This can be possible by using referenced datatypes array or class.

Why reference types are given, when we have 8 primitive types?

Referenced types are given to store multiple values in continuous memory locations to retrieve data in quick time and to pass all values with single network call.

Why four referenced types are given?

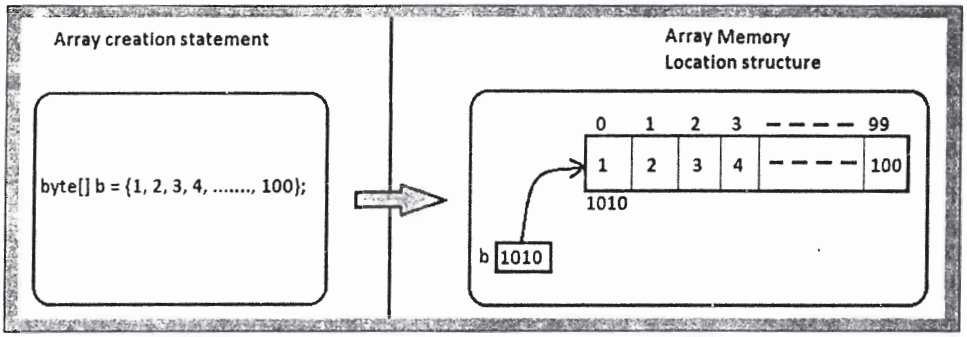
To collect only homogeneous and also heterogeneous type of values.

Understanding Array

In Java Array is a reference data type. It is used to store fixed number of multiple values *of same type* in continuous Memory locations.

Note: Like other data types Array is not a keyword rather it is a concept. It creates continuous memory locations using other primitive and reference types.

Below diagram shows array creation and its memory location structure to store 100 values of type byte.



In the above diagram array is created with 100 values. So JVM creates 100 continuous memory locations with each location of size 1 byte with some starting address, assume, 1010. Each location is created with an index starts with Zero. Finally the base address is stored in the referenced variable b to read and modify those values further.

As you noticed Memory location wise there is no difference in storing multiple values using variables and array, both consumed same size of memory in this case 100 bytes. The only difference is performance. Array, always, gives high performance than all other data types in storing multiple values.

Array Limitation:

Its size is fixed, means it will not allow us to store values more than its size. Also it will not allow different values.

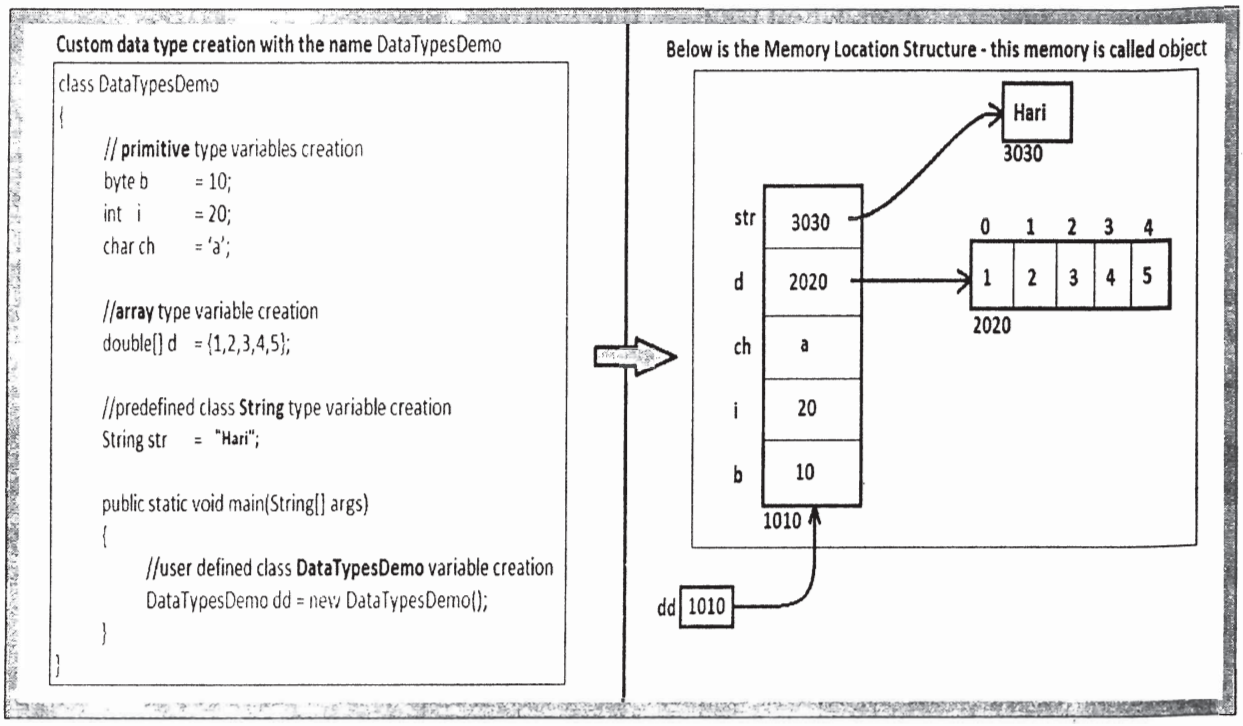
Solution:

Array limitation is solved using **class**.

class

Basically using the keyword *class* SUN provides a way to create new data types to store different types of values in continuous memory location using primitive and referenced types.

Below program shows creating a user defined data type with the name "DataTypesDemo" to group different types of values.



Save the above class with DataTypesDemo.java to compile and execute.

"***DataTypesDemo***" is the new datatype name, ***new*** keyword provides continuous memory locations for all variables created in that class, and that base address is stored in variable ***dd.***

So we can **define class** as - *it is a user defined data type used to store single and multiple values of same and different data type values in continuous memory locations.*

The special feature of class data type is - it also allows us to define methods to provide business logic, i.e. main logic of the application.

Why referenced types are called derived types and what is their size?

Referenced data types are called as derived data types because they are created using primitive types and its size is the addition of all primitive data type's size used in the class. Above class consumes 55 bytes of memory.

Below is the calculation

(byte + int + char + (5 * double) + 4 * char) = (1 + 4 + 2 + (5*8) + (4*2)) = 55 bytes.

Below table shows primitive data type's Size, Range and Default values

Data Type Name	Size [byte(s)]	Range	Default Value
byte	1	-128 to 127	0
short	2	-32,768 to 32,767	0
int	4	-2,147,483,648 to 2,147,483,647	0
long	8	-9,223,372,036,854,775,808 to +9,223,372,036,854,775,807	0
float	4	1.40129846432481707e-45 to 3.40282346638528860e+38	0.0
double	8	4.94065645841246544e-324 to 1.79769313486231570e+308	0.0
char	2	0 to 65,535	One white space
boolean	1	false or true	false
Reference	Depends on PDT	Depends on primitive data types	null

Default values are only applicable to class level variables. Class level variables are automatically initialized by JVM with default values based on its data type but where as local variables must be initialized by developer explicitly.

Rule: So the rule is we must initialize local variables below accessing them

Q) What is the output from the below program?

```
class A{
        static int a;
        public static void main(String[] args){
                int p;
                System.out.println(a);
                System.out.println(p);
        }
}
```

Literals

Literal is a constant.

Ex: 10, 20.4, 'a', "abc", true

Types of literals and their default datatypes

In Java we have below types of literals

- **Integral Literals:** All integer type literals are called integral literals. By default they are of type int. If we want to represent them as long we must suffix literal with 'l' or 'L'. We do not have byte or short type literals.

- **Floating-point Literals:** All floating point literals are of type double. If we want to represent them as float we must suffix literal with 'f' or 'F'. double literals can also be suffixed with 'd' or 'D'.

- **Character literal**: The single character placed inside single quote is considered as character literal. All character literals are of type char.
  Rule: In single quote we are not allowed to place more than one character.
  In single quote we can place either single space or ONE character.

- **String literal:** Characters placed inside double quote is considered as string literal. All string literals are of type java.lang.String
  Note: In double quote we can place ZERO to 'n' number of characters.

Below table shows Types of literals, their data type and its default types

Type of Literal	Datatype	default value	Sample values
Integral	int	0	10, 20, 30, 40, ...
	long	0	10L, 20L,30L, 40L, ...
Floating-Point	float	0.0	10.0f, 20.0f, 30.34f, 40.3f, ...
	double	0.0	10.0, 20.0, 30.34, 40.3, ...
Character	char	one space	'a', 'b', 'c', '1', '@', '#', '\n', ...
Boolean	boolean	false	true, false
String	java.lang.String	null	"abc", "bbc", "a", "10", "@#$", ...

Identify valid literals from the below list

- 10
- 10.345
- 53.67f
- 2345L
- 3.45d
- 45D
- 20b
- 34s
- 'a'
- a
- '#'
- '1'
- '10'
- ''
- ""
- "abc"
- "10"
- "1"
- "a"
- hi

Primitive Data type conversion

The process of changing one type of value to another type of value is called type conversion.
We develop type conversion by assigning a value of one variable to a variable of another type.
For example:

Below example shows assigning integer value to float variable

```
int a = 10;

//type conversion
float f = a;
```

a 10

f 10.0

Primitive type conversion

We have two types of conversions, they are:

1. Implicit type conversion / Automatic type conversion / widening
2. Explicit type conversion / Casting / narrowing.

Automatic or Implicit Conversion

If STR <= DTR then that conversion is called automatic type conversion. In this conversion there is no loss of data hence compiler compiles the given class.

This conversion is also called *widening, because* if we create destination variable with highest range data type the data size is increased to destination data type size.

Below example shows **implicit conversion** or **widdening**

```
int  a = 10;      a 10      size 4 bytes
long l = a ;      l 10      size 8 bytes - widdening
```

Casting or **Explicit Conversion** or **Narrowing**

Performing type conversion from highest range datatype variable to lowest range datatype variable by using *cast operator* is called type conversion. It is also called explicit conversion because this conversion performed by developer explicitly by using *cast operator*.

Cast operator is a data type placed in parenthesis after *"= " operator* and before *source variable*.

Syntax: <Destination data type> <variable name> = *(data type)* <source type>;

By using cast operation

- We are convincing compiler that the value stored in source type variable is within the range of cast operator type and
- We are allowing JVM to reduce source value to cast operator type if its range is greater than cast operator type

Sometimes it is necessary to perform type conversion between highest range data type variable to lowest range datatype variable, in this case developer must do conversion explicitly by using cast operator as shown below.

Below example shows **casting**

```
long  L = 10;      ✓ L 10    size 8 bytes
int   i = L ;      ✗ CE: possible loss of precession
//casting
int   i = (int)L ; ✓ i 10    size 4 bytes - Narrowing
```

Here we are telling to compiler that the value stored in **L** is with in the range of **int**, so compiler allows this conversion as it assumes there is no loss of data.

Casting is also called *narrowing* because data size is decreased.

In casting, what will happen if source variable has value greater than destination type range?
No CE, No RE, assignment is performed by reducing its value to the cast operator range by using 2's compliment and stores that result in the destination variable.

We can use below **short-cut formula** to know the reduced value
[minRange + (result - maxRange - 1)]

Below program shows applying casting short-cut formula

```
int  i = 254;
byte b1 = (byte) i;
```

i 254

b1 -2

```
minRange + (result - maxRange - 1)
=> -128 + (254 - 127 -1);
=> -128 + (254 - 128)
=> -128 + (126)
=> -2
```

Rules in Primitive Type Conversion

Rule #1: Source and destination data types must be compatible; otherwise it leads to compile time error "**incompatible types**". Except boolean all primitive data types are compatible. It means boolean value or variable cannot be assigned to any other data type variable.

Below example shows **incompatible types** error

```
int   a = 10;     ✓
float f = a;      ✓
//boolean b = a;  ✗ CE: incompatible types
                        found   : int
                        required: boolean
```

Rule #2: Source type range must be less than or equals to destination type range, otherwise it leads to CE: "**possible loss of precision**"

Below example shows **possible loss of precision** error

```
float  f1 = 10;   ✓
float  f2 = f1;   ✓
double d1 = f1;   ✓
int    i  = f1;   ✗ CE: possible loss of precision
                        found   : float
                        required: int
```

Rule on cast operator:

- *cast operator* data type must be compatible with source type else it leads to CE: "**inconvertible types**" and also
- It should be compatible with destination type else it leads to CE: "**incompatible types**" and also
- its range must be <= destination type range else it leads to CE: "**possible loss of precession**".

Find compilation errors in the below program

```
int    i     = 10 ;
byte   b1    = i ;
byte   b2    = (byte) i;
byte   b3    = (int)  i;
byte   b4    = (boolean) i;
```

Self practice bits

Find out compilation errors if any

1. Automatic / implicit / widdening conversion
2. Casting / explicit/ narrowing conversion

Ex:

```
int a = 10;

float f = a;

boolean bo = a;
```

Ex:

```
int a = 10;

int b = a;

//widdening
long l = a;

//narrowing
byte b = (byte)a;
```

a [10] 4 bytes

b [10] 4 bytes

l [10] 8 bytes

b [10] 1 byte

```
int a = 10;
byte b = a;
byte b = (byte)a;
boolean  bo = a;
boolean  bo = (boolean)a;
boolean  bo = (byte)a;
byte b = (short) a;
short s = (byte)a;
byte b = (short)(byte)a;
byte b = (byte)(short)a;
```

```
int a = 254;

byte b1 = (byte) a;
short s1 = (short) a;

short s2 = (short)(byte) a;

System.out.println(a);
System.out.println(b1);
System.out.println(s1);
System.out.println(s2);
```

Special cases

Case #1: Long value can be assigned to float variable, but float variable cannot be assigned to long variable, because below two reasons

- float range is greater that logn range, check ranges table
- floating point value cannot be stored in long variable.

Find out CEs in the below assignments

```
long L = 10;
int i =L;
float f =L;
float f1= 10;
long b = f1;
long L2 = (long)f1;
```

```
float f1 = 254.345f;
byte b1 = (byte) f1;

System.out.println(f1);
System.out.println(b1);
```

Case #2: char and number types are compatible types
So we can assign

- char literal to number variable
- int literal to char variable

In the above assignment JVM performs required conversions
If we assign char literal to number variable, JVM stores that character's ASCII number
If we assign number literal to char variable, JVM stores that number's ASCII character

Below is the list of characters and their ASCII numbers

Character	'0'	'1'	'2'	'3'	'4'	'5'	'6'	'7'	'8'	'9'
ASCII Number	48	49	50	51	52	53	54	55	56	57

Character	'A'	'B'	'C'	.....	.....	....	....	....	....	'Z'
ASCII Number	65	66	67	.....	.....	....	....	....	....	90

Character	'a'	'b'	'c'	.....	.....	....	....	....	....	'z'
ASCII Number	97	98	99	.....	.....	....	....	....	....	122

Java supports Unicode character set, it is a super set of ASCII character set.

- Unicode character set range is "0 to 65535" and
- ASCII character set range is "0 to 255"

So, the rule is: The number we are assigning to char variable must be in range of 0 to 65535, else it leads to CE: *possible loss of precision*.

What is the output from the below program?

```
char ch1 = 'a';      System.out.println(ch1);      char ch3 = 255;
char ch2 = 97;       System.out.println(ch2);      char ch4 = 65000;
                                                   char ch5 = 65535;
int i1 = 98;         System.out.println(i1);       char ch6 = 66000;
int i2 = 'b';        System.out.println(i2);
```

The special character "?"

- If the assigned number's corresponing character is not supporting by your computer JVM internally stores the special character "?"
- By default Windows OS supports only ASCII character set, so if we assign number value beyond 0 – 255 range JVM internally stores the "?" character

What is the out from the below program?

```
char ch1 = 98;
char ch2 = 250;

System.out.println(ch1);
System.out.println(ch2);
```

Q) We know we can assign int literal to char variable, the same way can we assign *int variable* to char variable?

No, because int datatype range is greater than char datatype range

What is the output from the below program?

```
int i1 = 98
char ch2 = i1;

char ch1 = 98;
```

But we can assign int variable to char variable through cast operator

What is the output from the below program?

```
int i1 = 98
char ch2 = i1;
char ch2 = (char)i1;
```

Q) Also think, can we assign char variable to byte, short variables, and byte variables to char variable?

- We can assign char literal to byte, short variables if the assigned char literal ASCII number is within the range of byte or short datatypes.
- But we cannot assign char variable to byte or short variables also we cannot assign byte or short variables to char variable, because their range has negative number.

What is the output from the below program?

```
byte b = 97;
char ch = b; ✗ CE: possible loss of precision

char ch = (char)b; ✓
System.out.println(ch); //a
```

Find out compile time errors in the below program, comment them and print value of remaining variables.

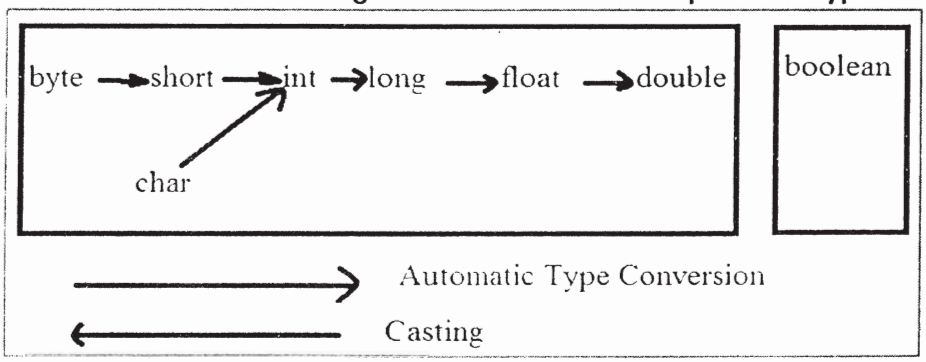
```
char ch1 = 'a';                 System.out.println(ch1);
char ch2 = '1';                 System.out.println(ch2);
char ch3 = '10';                System.out.println(ch3);
char ch4 = 97;                  System.out.println(ch4);
char ch5 = 49;                  System.out.println(ch5);
char ch6 = 1;                   System.out.println(ch6);
char ch7 = 255;                 System.out.println(ch7);
char ch8 = 65000;               System.out.println(ch8);
char ch9 = 65535;               System.out.println(ch9);
char ch10 = 66000;              System.out.println(ch10);
char ch11 = -97;                System.out.println(ch11);

int i1 = 97;                    System.out.println(i1);
int i2 = 'a';                   System.out.println(i2);

char ch12 = i1;                 System.out.println(ch12);
char ch13 = (char)i1;           System.out.println(ch13);

int i3 = -97;                   System.out.println(i3);
char ch14 = i3;                 System.out.println(ch14);
char ch15 = (char)i3;           System.out.println(ch15);
float  f = `a';                 System.out.println(f);
```

Below is the conclusion diagram we should follow in primitive types variables assignment



Q) What type of destination variable we should create to store all primitive types of values, except boolean?

A) double type variable.

Q) In how many ways we can initialize a variable?

There are 4 ways to initialize a variable

1. **Using literal** -> int i = 10;
2. **Using another variable** -> int j = i;
3. **Using expression** -> int k = i + j;
4. **Using non-void method** -> int x = m1();

Case #3:

In all above four ways of assignment compiler *checks only source data type and its range but not its value*, but in case of *int and char literals assignment it checks type, range and also its value*. If the value is within the range of destination type variable compiler allows assignment else it throws PLP error.

Find out compilation errors in the below assignments

```
byte b = 10;
short s = 300;
char ch = 108;

int i = 10;
int i = 10L;

float f = 10;
float f = 10L;
float f = 10.0;
float f = 10.0f;

double d = 10.0;
double d = 10.0f;

double d = 10L;
double d = 'a';
double d = 10;
```

Answer below question

```
1. byte b1 = 10;
2. int i = 10;
3. byte b2 = i;

4. Sopln(b1);
5. Sopln(i);
6. Sopln(b2);

choose one option
1. CE at line 1
2. CE at line 3
3. 10 10 10
4. CE at line 6
5. none of the above
```

Reference Types conversion

Like primitive types, we can also perform type conversion operation in between reference types. Assigning one class object to another class referenced variable is called referenced type conversion. To perform referenced type conversion the two classes should be compatible. The classes become compatible only if they are developed with inheritance. Inheritance relation is also called IS-A relation.

So the rule we should check in reference type conversion is:

Source type *IS-A* destination types or not.

How can we develop inheritance relation between two types?

By using either *extends* or *implements* keywords

"*extends*" is used for developing inheritance between two classes or two interfaces, and "*implements*" is used for developing inheritance between interface, class.

Syntax:

```
class Example{}
class Sample extends Example{}
```

```
interface A{}
interface B extends A{}
class C implements A{}
```

The class that is *placed after extends keyword* is called *super class*, here Example is super class, and the class that is *placed before extends keyword* is called *sub class*, here Sample is sub class. The classes created *without inheritance* relationship are called *siblings*

Q) Which are the classes called compatible?

- Subclass is compatible with Superclass
- Super class is not compatible with subclass
- Siblings are not compatible.

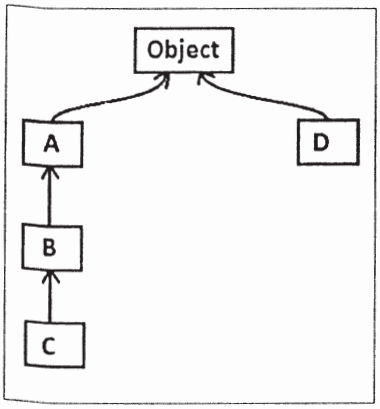
Q) What is the right assignment between referenced types?

The rule we should check in referenced type conversion is *Source Type* "IS-A" *Destination Type* or not. So we can assign subclass object reference to a superclass referenced variable. But super class object reference cannot be assigned to a subclass referenced variable. Also sibling object reference cannot be assigned to another sibling class referenced variable.

Q) What type of referenced variable we must create to store all types of objects?

A) java.lang.Object class type, because it is the super class of all types of classes.

Consider below inheritance hierarchy, and identify relations between these classes



- Object class is the super class of all user defined and predefined classes
- class A "IS-A" subclass of Object
- class B "IS-A" subclass of A, Object
- class C "IS-A" subclass of B, A, Object
- class D "IS-A" subclass of Object, and is sibling of A, B, C

So, we can store

- A class object reference in Object class refrenced variable
- B class object reference in A or Object class refrenced variable
- C class object reference in B or A or Object class refrenced variable
- D class object reference in Object class reference

```
/*
Below program shows implementing
above inheritance hierarchy.
*/

class A{}
class B extends A{}
class C extends B{}
class D{}
```

Identify compile time errors in the below assignments?

```
class ReferenceTypeConversion {
   public static void main(String[] args){

        Object obj1 = new Object();
        Object obj2 = new A();
        Object obj3 = new B();
        Object obj4 = new C();
        Object obj5 = new D();

        A a1 = new A();          B b1 = new A();
        A a2 = new B();          B b2 = new B();
        A a3 = new C();          B b3 = new C();
        A a4 = new D();          B b4 = new D();

   }
}
```

Types of referenced type conversion:
Java supports two types of reference type conversions

- Upcasting / automatic conversion
- Downcasting / casting

Upcasting: It is the implicit reference type conversion.
The process of *storing subclass object reference in super class reference* variable is called upcasting.
For Example:

```
A a = new B();
```

Note: It is not possible to store super class object in sub class reference variable, it leads to "**incompatible types**" compile time error.
For Example:

```
A a = new A();          ✓

B  b = a;               ✗ CE: incompatible types
B  b = new A();         ✗ CE: incompatible types
```

Downcasting: it is the explicit reference type conversion, casting.
Retrieving subclass object reference from super class referenced variable and storing it in the same sub class referenced variable is called downcasting.
For Example:

```
A a = new B();          ✓
B b = (A) a;            ✓
```

Here, in casting we are informing to compiler that the object stored in **e** variable is **B** type object. Hence compiler allows compiling the program as they are having IS-A relation.

Rule in using cast operator:
The cast operator type and source type should have inheritance relation else it leads to Compile time error "**inconvertible types**"
For Example:

```
A a = new A();
D d = (D) a;        ✗ CE: inconvertible types
```

java.lang.ClassCastException:
In casting the object coming from source variable, if it is not compatible with cast operator type JVM throws above exception "*ClassCastException*".

In casting compiler cannot identify the object coming from the source variable, because it checks only source variable type and cast operator type has IS-A relation or not. So when the source variable is superclass type and cast operator is subclass type compiler always compiles

this casting. But if the object coming from the referenced variable is sibling type of cast operator type then JVM throws ClassCastException

Below code throws CCE

```
Object obj    = new A();        ✓
D      d      = (D) obj;        ✗RE: ClassCastException
```

In the above statement **D d = (D)obj;** compiler only checks **obj** and **D** has inheritance relation or not, it does not check the **object stored** in **obj** variable because compiler checks only type of the variable. Since both types have inheritance relation compiler allows above conversion. But at runtime JVM identifies the **object coming** from **obj** variable is of type **A** which is **not compatible** with **D**, hence JVM terminates program execution by throwing **ClassCastException.**

Q) What Compiler and JVM do in reference type conversion?

Compiler checks the source variable type & cast operator type has inheritance relation or not.

In the above example, compiler

- first checks obj type "IS-A" D => No
- then it checks D "IS-A" obj type => Yes

Then it compiles this casting statement

JVM checks the source type object "IS-A" cast operator type or not.

In the above example source type object is "A" and cast operator type is "D", they are siblings so JVM throws CCE.

How can we solve ClassCastException?

To solve CCE exception we should use **instanceof** operator.

It returns *boolean* value by checking source type object with the given class. It works exactly as like cast operator, the only difference is for siblings comparison *cast operator* throws ClassCastException where as *instance operator* returns false.

So to solve CCE before downcasting we should check object type using **instanceof** operator. Below is the syntax to use instanceof operator

Syntax: **referenced variable** instanceof **classname**

Here *referenced variable* is the source variable and *class name* is the cast operator type name It returns true, if referenced variable contains object **IS-A** class type. Else returns false.

Below code shows doing downcasting with instanceof operator condition

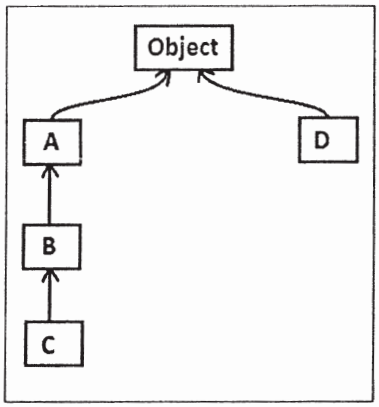
```
Object obj    = new A();
if(obj instanceof D){
        D  d  = (D) obj;
}
```

Now we do not get CCE. JVM does not execute casting since instanceof operator returns false because the source variable object is A.

Rule of instanceof operator:

Source variable type and *classname* type should have IS-A relation else compiler throws CE: inconvertible types. If the object coming from referenced variable is same class or subclass object then instanceof operator returns true. If it is super class or sibling type object instanceof operator returns false,

Consider below inheritance hierarchy, and Find out what is the output in the below statements?




```
A a = new B();
System.out.println( a instanceof Object);
System.out.println( a instanceof A);
System.out.println( a instanceof B);
System.out.println( a instanceof C);
System.out.println( a instanceof D);

Object obj = new B();
System.out.println( a instanceof Object);
System.out.println( a instanceof A);
System.out.println( a instanceof B);
System.out.println( a instanceof C);
System.out.println( a instanceof D);
```

When we should implement upcasting and downcasting in Project:

Need of upcasting

In projects upcasting is implemented to develop loosely coupled runtime polymorphic applications means to store all subclasses objects into a single referenced variable and further to execute the invoked method from different subclasses based on the object stored in the referenced variable. You will understand more about upcasting in OOPS chapter

Limitation in Upcasting

In upcasting compiler allows us to invoke only superclass members, we cannot invoke subclass specific members with superclass type variable, it leads to **CE: cannot find symbol**, because compiler checks only variable type but not object stored in superclass reference variable.

Check below example

```
class Example{
        void m1(){
                Sopln("m1");
        }
}

class Sample extends Example{
        void m2(){
                Sopln("m2");
        }
}
```

```
class Test{
        public static void main(String[] args){

                Example e = new Sample();
                e.m1();   ✓
                e.m2();   ✗CE: c f s
        }
}
```

In this example, e.m2() method call leads to CE because m2() method definition is not available in superclass Example.

Need of downcasting

We should implement downcasting to invoke subclass specific members, because when we store subclass object in superclass referenced variable we cannot invoke that subclass specific member using super class referenced variable. In the above example in Test class we must implement downcasting to call m2() method from Sample class as shown below:

```
class Test{
        public static void main(String[] args){

                Example e = new Sample();
                e.m1();

                Sample s = (Sample)obj;
                e.m2();
        }
}
```

SCJP Question:

Given:
10. interface Foo {}
11. class Alpha implements Foo {}
12. class Beta extends Alpha {}
13. class Delta extends Beta {
14. public static void main(String[] args) {
15. Beta x = new Beta();
16. // insert code here
17. }
18. }

Which code, inserted at line 16, will cause a java.lang.ClassCastException?

A. Alpha a = x;
B. Foo f = (Delta)x;
C. Foo f = (Alpha)x;
D. Beta b = (Beta)(Alpha)x;

need of upcasting

objects are created and passed as arguments at run-time, these objects are of like Person, Animal, Vehicle, Shape etc... We process all these objects in a by taking them as argument. To make sure that we are reading and processing only a particular type of objects we must define method with parameter of type which can hold only that particular type of objects (super class type).

In this case Upcasting should be implemented; it means we should take method parameter of type which is a super class of a group of sub classes.

What is the output from the below programs?

```
class Example{
        void m1(){
                Sopln("m1");
        }
}
class Sample extends Example{
        void m2(){
                Sopln("m2");
        }
}
```

```
class Test{
        static void m3(Example e){
                e.m1();
                e.m2();
        }
}
```

```
class Test{
    static void m3(Example e){
          e.m1();

          Sample s = (Sample)e;
          s.m2();
    }

    public static void main(String[] args){
          Test.m3(new Sample());

          Test.m3(new Example());
    }
}
```

```
class Test{
    static void m3(Example e){
          e.m1();

          if(e instanceof Sample){
            Sample s = (Sample)e;
            s.m2();
          }
    }
    public static void main(String[] args){
          Test.m3(new Sample());

          Test.m3(new Example());
    }
}
```

```
class Test{
  static void m3(Object obj){
        if (obj instanceof Example)
        {
          Example e = (Example)obj;
          e.m1();

        }
        else if (obj instanceof Sample)
        {
          Sample s = (Sample)obj;
          s.m1();
          s.m2();
        }
  }
```

```
  public static void main(String[] args)
  {
        Test.m3(new Example());

        Test.m3(new Sample());

        Test.m3(new Object());
    }
}
```

Automatic Type Promotions in an Expression

If an expression has arithmetic operators with different data types, its result type is the highest range data type used in that expression.

JVM follows below standard rules in evaluating an expression

1. It replaces all variables and method calls with their values
2. It promotes all lesser range data type values to the highest range variable type that is used in that expression, then it starts calculation

   For example:

```
int     i1 = 10;
float   f1 = 20;

float   f2 = i1 + f1;
```

The above expression result type is float. So the destination variable type should be either float or its next range data type.

Q) Can we store the above expression value in int variable?

No, it leads to CE: possible loss of precision

```
int     i2 = i1 + f1;  ✗ CE:  possible loss of precision
```

We can store this result in int variable with cast operator as show below

```
int     i2 = (int) (i1 + f1);
```

Q) Check if below statements are compiled fine?

```
int     i2 = (int) i1 + f1;
int     i2 =  i1 + (int)f1;
```

Rule #1: In an expression both operands must be compatible types else it leads to compile time error: "**operator cannot be applied .....**"

Ex:

```
int       i1  = 10;
boolean bo = true;

int       i2  = i1 + bo;  ✗ CE: operator + cannot be applied between int, boolean
```

In an expression "byte, short, and char" variables are automatically promoted to int because the minimum memory location used to calculate an expression is 4 bytes. ...ls check JVM architecture chapter.

...statement leads to Compilation error

```
...b1= 50;
...b2= 100;
...b3= b1+ b2; >< CE: possible loss of precision
                  found: int
                  required: byte
```

We should use cast operator to store the result in byte variable, as shown below

```
byte b3 = (byte) (b1 + b2);
```

Self practice bits:

Rule: All operands in an expression must be compatible, else it leads to CE: operator can not be applied .....

JVM performs below steps in evaluating an expression

It replaces

1. all variables with thier values.
2. all method calls with their returned values
3. promotes lowest range data type values to highest range data type values that is used in that expression.

ex:

case 1:

```
int a = 10;
int b = 20;

int c = a + b;
```

same types so no promotion

case2:

```
int a = 20;

float f = 30;

float f = a + f;
```

```
=> 20 + 30.0
=> 20.0 + 30.0
=> 50.0
```

int is promotted to float

case 3:

```
int a = 50;
blooean b = true;

int c = a + b;
```

CE: operator + cannot be applied to int,boolean

incompatible types, So CE

In an expression "byte, short, and char" are automatically promotted to int.
Why? this is the secret of JVM, check JVM architecture.

case1:

```
1. byte b1 = 10;
2. byte b2 = 20;
3. byte b3 = b1 + b2 ;

4. Sopln(b3);

byte b3 = b1 + b2 ;
=> byte b3 = 10 + 20 ;
=> byte b3 = int + int ;
=> byte b3 = int  ;
```

case2:

```
1. short s1 = 10;
2. short s2 = 20;
3. short s3 = s1 + s2 ;

4. Sopln(s3);

=>short s3 = s1 + s2 ;
=>short s3 = 10 + 20 ;
=>short s3 = int + int ;
=>short s3 = int ;
```

case3:

```
1. char ch1 = 'a';
2. char ch2 = 'b';
3. char ch3 = ch1 + ch2 ;

4. Sopln(ch3);

=>char ch3 = ch1 + ch2 ;
=>char ch3 = 'a' + 'b' ;
=>char ch3 = 97 + 98 ;
=>char ch3 = int + int ;
=>char ch3 = int ;
```

Note: Compiler does not check values range in an expression, it checks only type.
Hence all above cases expression assignment leads to CE because those expressions results **int** type value.

hence compiler think it can not stored in lesser range variable types.

Conditions apply

it is true only if expression contains variables, if it contains int literals directly it evalues expression and checks the result is with in the range of destination type. if that reslut is with in the range of desination type,assignment is possible else it throws CE: PLP

Ex:

```
byte b3 = 10 + 20;

byte b3 = 126 +1;

byte b3 = 126 +2;
```

```
byte b1 =  10L + 20;
byte b2 =  (byte)10L + 20;
byte b3 =  (int)10L + 20;
```

```
char ch1 = 'a' + 'b';
char ch2 =  ch1 + 'c';
```

Working with *System.out.println*

Using *System.out.println* statement we can print all types of literals

- Directly
- Using variable
- Using expression
- Using non-void methods

Write program to add two integer numbers and print the result as

The addition of 10 and 20 is 30

```
class Addition{
        public static void main(String[] args){
                int a = 10;
                int b = 20;
                int c = a  + b;
                System.out.println(c);
                System.out.println("The addition of " + a +" and " + b + " is " + c);
        }
}
```

Evaluation process of above statement

```
=> System.out.println("The addition of " + 10 +" and " + 20 + " is " + 30);
=> System.out.println("The addition of " + "10" +" and " + 20 + " is " + 30);
=> System.out.println("The addition of 10" +" and " + 20 + " is " + 30);
=> System.out.println("The addition of 10 and " + 20 + " is " + 30);
=> System.out.println("The addition of 10 and " + "20" + " is " + 30);
=> System.out.println("The addition of 10 and 20" + " is " + 30);
=> System.out.println("The addition of 10 and 20 is " + 30);
=> System.out.println("The addition of 10 and 20 is " + "30");
=> System.out.println("The addition of 10 and 20 is 30");
```

Arithmetic Operators

Java supports 5 Arithmetic operators

1. Addition +
2. Subtraction -
3. Multiplication *
4. Division /
5. Reminder %

In Java all operators in an expression are executed from LEFT to RIGHT means from "=" operator to ";" according to their precedence.

For operators precedence order check Operators chapter

Arithmetic Operators precedence

- *, /, % operators have highest and same precedence
- +, - operators have next and same and precedence

What is the output from the below expression

System.out.println (4 * 2 + 8 / 2 – 3 * 3 + 4 / 2);

Evaluation:

- ⇨ System.out.println (4 * 2 + 8 / 2 – 3 * 3 + 6 / 3);
- ⇨ System.out.println (8 + 8 / 2 – 3 * 3 + 6 / 3);
- ⇨ System.out.println (8 + 4 – 3 * 3 + 6 / 3);
- ⇨ System.out.println (8 + 4 – 9 + 6 / 3);
- ⇨ System.out.println (8 + 4 – 9 + 2);
- ⇨ System.out.println (12 – 9 + 2);
- ⇨ System.out.println (3 + 2);
- ⇨ System.out.println (5);

Addition "+" Operator: A overloaded operator

In Java, + is the only overloaded operator.
It can be used as both

1. Addition operator
2. Concatenation operator

If + operator has it's both operands as numbers, it acts as an addition operator.
If one of the operand is String, it acts as concatenation operator

For Example

int i1 = 10 + 20; <= in this expression it act as *addition* operator
System.out.println(i1); => **30**

String s = "a"+"b"; <= in this expression it act as *concatenation* operator
System.out.println(s); => **ab**

Concatenation means appending both operands as single value

String s = "a" + 10; <= in this expression it act as *concatenation* operator
System.out.println(s); => **a10**

Division "/" Operator

Just calculate and tell what is the output from the below statement

System.out.println(22/7 * 10 * 10);

Your favorite expression (**PI * r^2**), answer is **3.14** right ☺ It is ***wrong answer***,
It is Java mathematics, not your general mathematics, and Answer is ***300.***

$\frac{22}{7} = 3$ because $\frac{\text{int}}{\text{int}} = \text{int}$ So, result of above expression is **300**

Some other important point of / operator

1. We cannot divide an integer number by ZERO, it leads to RE: ***java.lang.ArithmeticException***, not Infinity
2. But we can divide a floating point number by ZERO, its output is **Infinity**
3. We cannot divide a Integer zero by a ZERO it also leads to RE: *ArithmeticException*
4. We can divide a floating point ZERO by ZERO its output is **NaN**

Answer Below questions, find out if there are any CEs.

//TypeConversion.java

```
class TypeConversion{
        public static void main(String[] args) {
        byte b1 = 10;
        int i1 = b1;

        byte b2 = i1;
        byte b2 = (byte)i1;

        int i = true;

        int i = (int)true;

        int i2 = 254;
        byte b3 = i2;
        byte b3 = (byte)i2;

        char ch1 = 'a';
        int i3 = ch1;

        int i4 = 97;
        char ch2 = i4;
        char ch2 = (char)i4;

        long l1 = 10;
        float f1 = l1;

        long l2 = f1;
        long l2 = (long)f1;

        System.out.println("b1: "+b1);
        System.out.println("i1: "+i1);
        System.out.println("b2: "+b2);
        System.out.println("b3: "+b3);
        System.out.println("ch1: "+ch1);
        System.out.println("i3: "+i3);
        System.out.println("ch2: "+ch2);
        System.out.println("i4: "+i4);
        System.out.println("l1: "+l1);
        System.out.println("f1: "+f1);
        System.out.println("l2: "+l2);
    }
}
```

// ThinkAsCompilerAndJVM.java

```
class ThinkAsCompilerAndJVM {
        public static void main(String[] args) {

        System.out.println( 10        );
        System.out.println( 'a'       );
        System.out.println( "a"       );
        System.out.println( 10.0      );
        System.out.println( 10.345f   );
        System.out.println( 30L       );
        System.out.println( 30l       );
        System.out.println( 50 + 20   );
        int a = 30;   int b = 40;
        System.out.println( a + b     );
        System.out.println( "a + b"   );

        System.out.println( "a + b"+a+b      );
        System.out.println( "a + b"+ ( a+b)   );

        System.out.println( "a - b"+ a – b     );
        System.out.println( "a - b"+ ( a – b ) );

        System.out.println( "a * b"+a*b      );
        System.out.println( "a * b"+ ( a*b )  );

        System.out.println( ""+10 + 20        );
        System.out.println( 10 + ""+ 20       );
        System.out.println( 10 + 20 +""       );

        System.out.println( 22/7 * 10 * 10   );
        System.out.println( 22F/7 * 10 * 10);
        System.out.println( 22D/7 * 10 * 10);

        System.out.println( 10 / 0    );
        System.out.println( 10.0 / 0  );
        System.out.println( -10.0 / 0 );

        System.out.println( 0 / 0     );
        System.out.println( 0.0 / 0   );
        System.out.println( -0.0 / 0  );

    }
}
```

Equality Operators

We have two equality operators to compare two values

1. == equals
2. != Not Equals

Both operators compare values of the variables.
If both variables have *same values* == operator returns *true, and* != operator returns *false*
If both variables have *different values* == operator returns *false,* != operator returns *true*

For example

```
int a = 50;
int b = 50;
System.out.println( a == b );
System.out.println( a != b );
System.out.println( a = b);
System.out.println( a = b == b);
System.out.println( (a = b) == b);
```

Q) What is the output from below comparison?

```
System.out.println ( 10 == 10.0 );
```

A) **true**
Reason: int value 10 promoted to double 10.0

Special case:
Comparing *float* value with *double* value

We get true only if we compare round floating point "float" and "double" values. In remaining all other floating point float and double values comparison returns false.

```
System.out.println( 3.0f == 3.0 ); //true
System.out.println( 3.5f == 3.5 ); //true

System.out.println( 3.3f == 3.3 ); //false
System.out.println( 3.7f == 3.7 ); //false
```

Types of Languages based on types conversion

Based on the types conversion / typing concept the entire available languages can be broadly classified in to three categories

1. **Weakly typed**
   - The languages which allow us to assign big range variable value to small range variable without any restriction up to greater extent are called weakly typed programming languages.
   - Such types of languages are "C, C++", and most other high-level languages.
   - For example:
     - We can assign a "float" variable to "int" variable.
     - But we can't assign "structure" variable to "int" variable.

2. **Strongly/strictly typed**
   - In this type of languages we are not allowed to perform the operations like above. It leads to compile time error "possible loss of precision".
   - In compilation phase compiler of this type of languages does type checking thoroughly.
   - Such type of language is "JAVA".
     - for example:
       - We can't assign a "float" variable to an "int" variable directly.

- Before doing type conversion in java, java compiler checks below semantics:
  - *Every variable and expression should have a type, and every type is strictly defined*, else leads to CE.
  - All assignments, whether explicit or via parameter passing in method calls, re checked for type compatibility.
  - Java compiler checks all expressions and parameters to ensure that the types are compatible.
  - Any type mismatches are errors. That must be corrected to finish the compiler compiling the class.

3. **Untyped**:
   - The languages which do not use primitive variables to store data, instead if they used objects to store data, are called **untyped** programming languages.
   - These languages are also called as pure object-oriented programming languages.
     - Ex: Smalltalk
   - In these languages everything is an object.

Is Java 100% pure object-oriented programming language?
It is 99.9999999% object-oriented, remaining 0.00000001% not object-oriented because it uses primitive types to store data.

In Java 5 this limitation is solved by introducing a concept called
Auto Boxing and **Auto Unboxing.**

So from Java 5 onwards we can declare *JAVA also pure* object oriented programming language.

By using this concept we can avoid using primitive variables for storing data, instead we can use objects directly. But still primitive variables support is also available to provide backward compatibility.

Using auto boxing and unboxing we can perform calculation as shown below

```
Integer io1 = 10;
Integer io2 = 20;
Integer io3 = io1 + io2;
System.out.println("Result: "+io3);
```

Note: If you use jdk1.4 compiler, this code error out at compilation.

Check next chapter for details on Autoboxing and unboxing