REFERENCE TYPES AND PARAMETERS

DR. KRISHNENDU GUHA

TEACHING ASSISTANTS

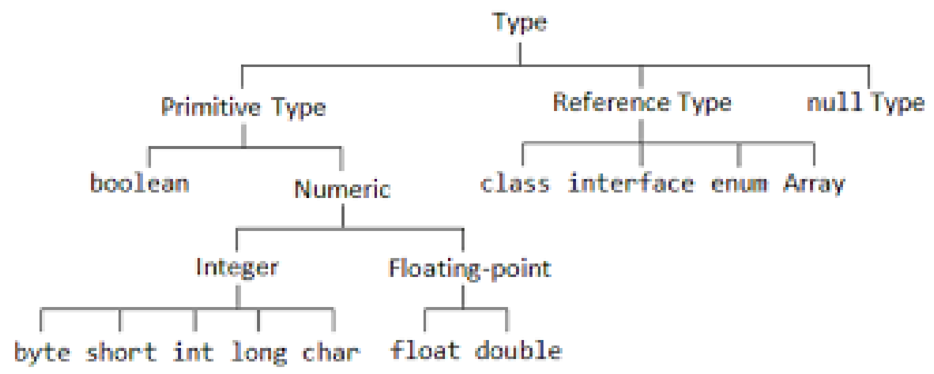
ZARRAR HAIDER (123120583@UMAIL.UCC.IE)

RHEENA BLAS (120347046@UMAIL.UCC.IE)

VALUE VS REFERENCE TYPES

- A variable may have one of Java’s **primitive types (value type)**:
  - these are types that **are built into the language**

    int, double, boolean, short, char, etc.

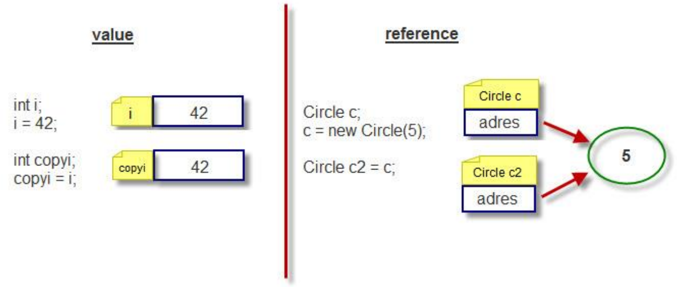


- Or, a variable may have a **reference type:**
  - with **one exception**, they **are not built into the language**:

    each **class that a programmer defines** gives us one such type, e.g. String, Scanner, Random, Dog,...
  - the one that is **built-in is arrays**
  - to *create a ’value’ belonging to a reference type*, you use **new** (except when exploiting the short-cut for String)
  - **variables of these types hold**

    either null or

    references to objects (i.e. ’pointers’ or memory addresses)



```
int i;
i = 42;
int copyi;
copyi = i;
```

```
Circle c;
c = new Circle(5);
Circle c2 = c;
```

MEMORY SPACE

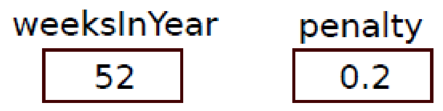
- When you **create a primitive type**, a **single space in memory is allocated to store its value** and that variable directly holds a value.

  The **value is stored in the stack** in memory.
- **Reference types** represent the **address of the variable rather than the data itself**, assigning a reference variable to another doesn't copy the data.

```
// primitive types
int weeksInYear = 52;
double penalty = 0.2;
// reference types
String firstName = "Brian"
Person p1 = null;
String lastName = null;
Person p2 = new Person("Tom", 50, "Co. Clare");
```

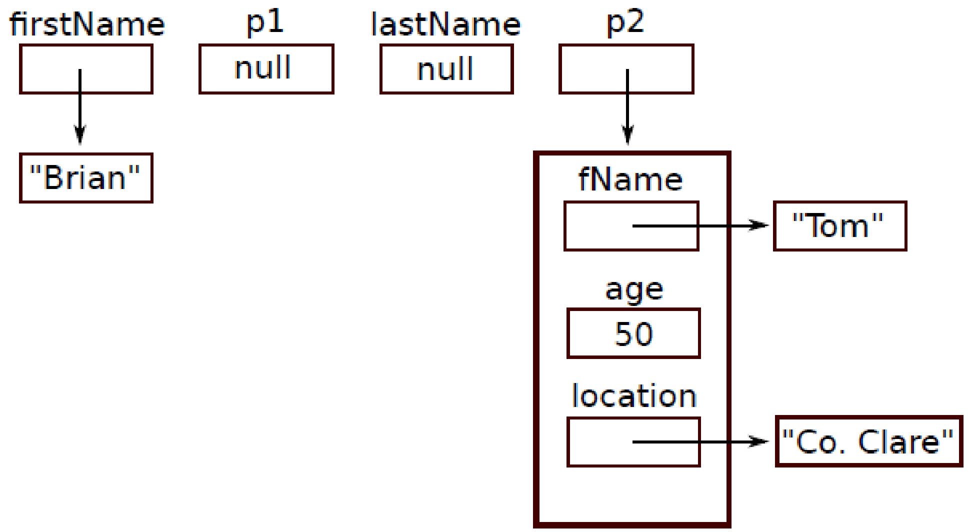
VALUE TYPE REPRESENTATION

- // primitive types
- int weeksInYear = 52;
- double penalty = 0.2;



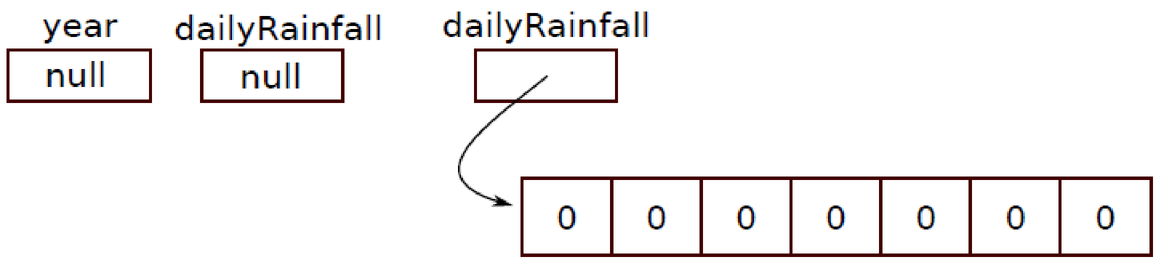
REFERENCE TYPE REPRESENTATIO

- // reference types
- String firstName = "Brian"
- Person p1 = null;
- String lastName = null;
- Person p2 = new Person("Tom", 50, "Co. Clare");

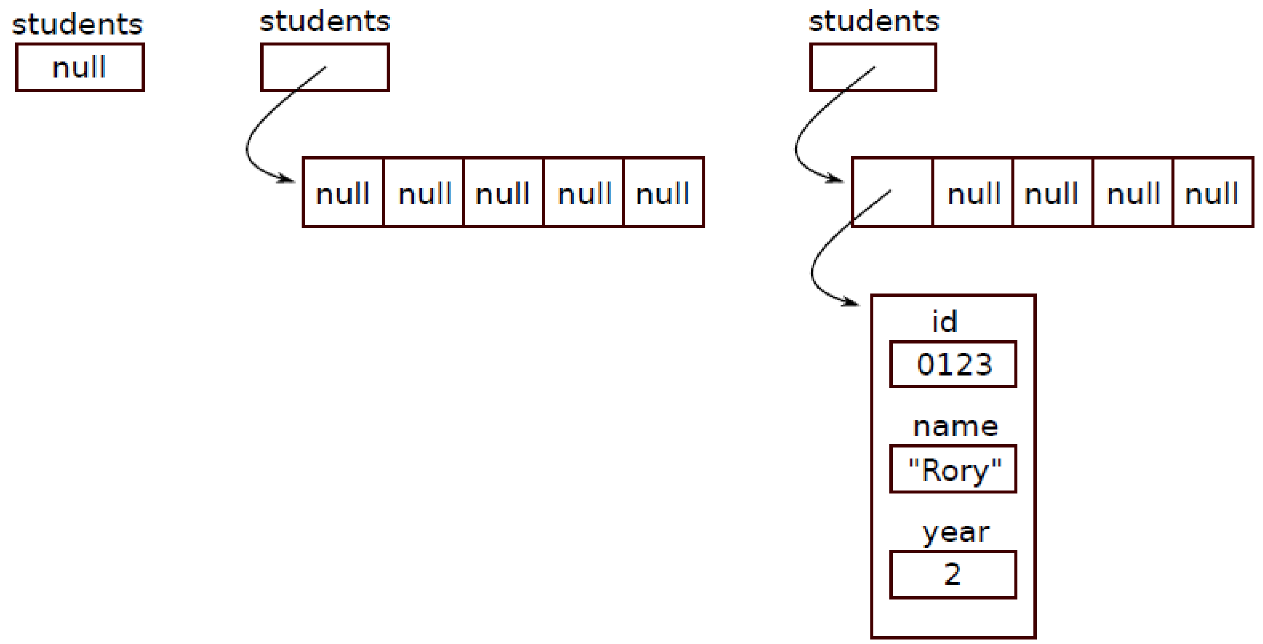


REFERENCE TYPE: ARRAY

- **Arrays** are **built-in reference types**.
- Java allows you **to create an array of references to any type of object** (to instances of any class).
- So, **array variables hold null or a reference** ('pointer').
  - // reference types
  - Year year = null;
  - int[] dailyRainfall;
  - dailyRainfall = new int[7];



- Student[] students = null;
- students = new Student[5];
- students[0] = new Student(0123, "Rory", 2);



Note: "Rory" is actually referenced with a pointer!

GOOD PRACTICE

- Avoid creating unnecessary objects! Creating an object is expensive.
  - String s = new String("apple"); // do not do this!!
  - String s = "apple";

- Autoboxing allows programmer to mix primitive (int, double, long, etc.) and boxed primitive types (Integer, Double, Long, etc.) automatically as needed.
  - int x = 3;
  - Integer y = new Integer(3);

- Watch out for unintentional autoboxing.

```
private static long sum(){
        Long sum = 0L;
        for(long i = 0; i <= Integer.MAX_VALUE; i++)
                sum += i;
        return sum;
}
```

Declaring the variable sum as an object results in constructing 2^31 unnecessary Long instances. Using primitive type can reduce the run-time from 6.3 seconds to 0.59 seconds :)

OBJECT CLASS

- Object class in Java is a concrete class that is **primarily designed for extension**.
- It contains a number of **nonfinal methods**: equals, hashCode, toString, clone, finalize.
- It is the **responsibility of any class overriding these methods** to **obey their general contracts**.
- If contracts are not obeyed, the **functions will not work properly**.

METHODS COMMON TO ALL OBJECTS

- toString()

- A great idea: **always override toString**.
- If you **don't override it, it returns you the class name followed by an "at" sign**: Student@1234b3.
- A good toString implementation makes your class much more pleasant to use and makes

systems using the class easier to debug.

```
@Override
public String toString(){
	return name + " is a " + age + " year old " + breed;
}
```

Then, you can easily print an object (they are equivalent):

```
System.out.println(dog); or
System.out.println(dog.toString());
```

- <u>**clone()**</u>

- Creates a clone or **copy an object and returns the copy**.
- Important: using clone **will create a copy of the reference variable, not the object**.
- By default: **shallow copy**. Any changes made in referenced objects will be reflected in other objects.
- If edited properly: **deep copy**. A deep copy copies all fields and makes copies of dynamically allocated memory pointed to by the fields.
- We do not need to know the details, yet..

```
//Override
public Object clone(){
}
// shallow copy
Dog myDog = new Dog("Nobby", 3, "Norfolk Long Pig");
Dog yourDog = myDog;
Dog yourdog2 = (Dog) myDog.clone()
```

- **equals()**

- Determines **whether the specified object is equal to the current object**. If you do not override the equals() method, each instance of the class is equal only to itself.
- Overriding the equals method sounds easy, but can also go wrong very easy.
- **Primitive types support "==".** Reference types need equals().
- We will not go into the details of the hashCode() method here, but the basic rule of the contract states that: if two objects are equal to each other based on equals() method, then their hash codes must be the same. However, if the hash code is the same, then equals() can return false.

```
//Override
public boolean equals(Object obj) {
}
int x = 3;
int y = 4;
if (x == y) {
        System.out.println("Yes, x is equal to y");
}
else {
        System.out.println("No, x is not equal to y");
```

For reference types, we write if (s1.equals(s2)) ... instead of if (s1 == s2) ....

This is how an equals method might look like:

```
public boolean equals(Object obj) {
    if (this == obj) {
      return true;
    } else if (obj == null || getClass() != obj.getClass()) {
      return false;
    }
    Dog d = (Dog) obj;
    return (name.equals(d.name) && age == d.age && breed.equals(d.breed));
}
```

Note: Python does not have this distinction between primitive types and reference types. Everything is an Object in Python.

PARAMETERS: ACTUAL PARAMETERS VS FORMAL PARAMETERS

- **Actual parameter (or argument):** the ***data you supply to a method***
- **Formal parameter (or argument):** the **object needs some variables to store the data** while running the method. Formal parameters are used to **temporarily store the actual parameters** within the scope of the function.

They sound similar to local variables - but they look different (declaration and initialization).

- Declared within a method (or constructor) signature,
- Initialized when the method is called by using the actual variables,
- It has its scope as the body of the method.

```
person.updateAge(10); // 10 is an actual parameter

...

public void updateAge(int newAge){

        age = newAge; // newAge is a formal parameter

}
```

PARAMETER PASSING

- Java uses **call-by-value**: the *values of actual parameters are copied into the formal parameters*.

- If the parameters are of **primitive** types:
  - the ***formal parameter simply contains a duplicate of the value of the actual parameter***
  - so, any ***updates to the formal parameter within the method do not affect the actual parameter***

- If the parameters are of **reference** types:
  - the formal parameter **contains a duplicate of what was in the actual parameter**, i.e. both reference ('point to') the same object
  - so, ***updates to the object referenced by the formal parameter do affect the original object***

EXERCISE 1

```
public class BankAccount {

public int balance;

public int numTransactions;

public BankAccount(int bal, int trans) {

balance = bal;

numTransactions = trans;

}

public void transfer(int amount, BankAccount dest) {

balance = balance - amount;

numTransactions = numTransactions + 1;

dest.balance = dest.balance + amount;

dest.numTransactions = dest.numTransactions + 1;

}

@Override

public String toString() {

return "Balance=" + balance + ", number of transactions="

+ numTransactions;

}
```

```
public static void main(String[] args) {
BankAccount ac1 = new BankAccount(100, 10);
BankAccount ac2 = new BankAccount(120, 0);
int transferAmount = 20;
ac2.transfer(transferAmount, ac1);
System.out.println(transferAmount);
System.out.println(ac1);
System.out.println(ac2);
}
```

- Output: 20
- Balance=120, number of transactions=11
- Balance=100, number of transactions=1

EXERCISE 2

```
class Swapper {
public void methodA(int x, int y) {
int temp = x;
x = y;
y = temp;
}
public void methodB(int[] a) {
int temp = a[0];
a[0] = a[1];
a[1] = temp;
}
}
```

```
class SwapTester {
public static void main(String[] args) {
Swapper sw = new Swapper();
int[] numbers = {25, 14};
System.out.println(numbers[0] + " " + numbers[1]);
sw.methodA(numbers[0], numbers[1]);
System.out.println(numbers[0] + " " + numbers[1]);
sw.methodB(numbers);
System.out.println(numbers[0] + " " + numbers[1]);
}
}
```

- Output: 25 14
- 25 14
- 14 25

Any Questions

Thank You!
shutterstock.com · 1153070891