Objects
Recall: Data types

- **Data type**: A category of data values.
  - Example: integer, real number, string

Java data types are divided into two sets:

- **Primitive types**: Java's 8 built-in *simple* data types for numbers, text characters, and logic.
  - boolean, char, byte, short, int, long, float, double

- **Object types**: All other types!
  - e.g., Scanner, System, String, Math
Object types

- So far, we have seen:
  - **variables**, which represent data (categorized by **types**)
  - **methods**, which represent behavior

- **object**: An variable that contains *data and behavior*.
  - There are variables inside the object, representing its data.
  - There are methods inside the object, representing its behavior.

- **class**:
  - Basic building block of Java programs (what we have seen so far)
  - AND
  - Data types for objects
Example object types

- Theoretical examples:
  - A class *Person* could represent objects that store a name, height, weight, hair color, IQ, etc…
  - A class *Laptop* could represent objects that store speed, screen size, color, dimensions, brand, etc…

- Examples from Java:
  - The class *String* represents objects that store text characters.
  - The class *Scanner* represents objects that can *tokenize* streams of characters.
Point objects
**Point object**

- **Data stored in each Point object:**

<table>
<thead>
<tr>
<th>Field name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>the point's x-coordinate</td>
</tr>
<tr>
<td>y</td>
<td>the point's y-coordinate</td>
</tr>
</tbody>
</table>

- **Useful methods in each Point object:**

<table>
<thead>
<tr>
<th>Method name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>distance((p))</td>
<td>how far away the point is from point p</td>
</tr>
<tr>
<td>setLocation((x, y))</td>
<td>sets the point's x and y to the given values</td>
</tr>
<tr>
<td>translate((dx, dy))</td>
<td>adjusts the point's x and y by the given amounts</td>
</tr>
</tbody>
</table>
Constructing objects

- **construct**: To create a new object.
  - Objects are *constructed* with the `new` keyword.

- Constructing objects, general syntax:
  \[ <\text{class}> \ <\text{name}> = \text{new} \ <\text{class}> ( <\text{arguments}> ) ; \]

- Examples:
  
  ```java
  Point p = new Point(7, -4);
  Scanner console = new Scanner(System.in);
  ```

- Q: Wait a minute! Why don’t we construct strings with `new`?
  - A1: Strings are one of the most commonly used objects, so they have special syntax (quotation marks) to simplify their construction.
  - A2: Also, you can if you want to: `String s = new String(“hi”);`
Point object: Construction

- Constructing a Point object, general syntax:
  
  ```java
  Point <name> = new Point(<x>, <y>);
  Point <name> = new Point();  // the origin, (0, 0)
  ```

- Examples:
  ```java
  Point p1 = new Point(5, -2);
  Point p2 = new Point();
  ```
Representing objects

- So far, I have drawn primitive types like this:

```java
int x = 7;
```

```
x: 7
```
Representing objects

- I will represent object types like this:

```java
Point p = new Point(7, 3);
```

![Diagram](image)
Object variables are **references** to the location in memory where their data resides.

We draw references as pointers.

Actually, $p$ stores the **address** of the location in memory where its data is.

---

**Diagram:**

- **Variable**, with slot in memory
- **Reference**, a pointer to the object’s data
- **Data**, in another part of memory
- **Fields**
What happens when the following call is made?

Point p1 = new Point(7, 2);

\[ p1: \begin{array}{c}
\text{ } \\
\text{x: 7} \\
\text{y: 2} \\
\end{array} \]
null objects

- **null** is a value for all object types that says, “this object is a reference to nothing at all”
- We draw null objects with a slash
- Note that null objects have no memory reserved for their data!

Point \( p_1 = \text{null}; \)

\( p_1: \)
Calling methods on objects

- Since the methods are bundled in the objects, calling these methods requires specifying which object we are talking to.

- Calling a method of an object, general syntax:
  
  `<variable>.<method name>(<parameters>)`

  - The results may vary from one object to another.

- Examples:
  
  ```java
  String s1 = "Homey da Clown";
  String s2 = "Bubbles the clown";
  System.out.println(s1.length()); // prints 14
  System.out.println(s2.length()); // prints 17
  ```
Calling methods on objects

- Since the methods are bundled in the objects, calling these methods requires specifying which object we are talking to.

- Calling a method of an object, general syntax:
  
  `<variable> . `<method name>` ( `<parameters>` )`

  - The results may vary from one object to another.

- Examples:

  ```java
  Point p0 = new Point(0, 0);
  Point p1 = new Point(7, 3);
  Point p2 = new Point(2, -2);
  System.out.println(p1.distance(p0));  // 7.62
  System.out.println(p2.distance(p0));  // 2.83
  ```
Dereferencing

- When we use the “.” operator on an object, we access the stuff (methods and/or data) that the object references (or points to).

- This is called **dereferencing**.

- The “.” operator is the **dereferencing operator**.
Three kinds of method calls

- We have seen three ways to call methods:

<table>
<thead>
<tr>
<th>Type</th>
<th>No ‘.’ used</th>
<th>&lt;class&gt;.&lt;method&gt;</th>
<th>&lt;variable&gt;.&lt;method&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>When it’s used:</td>
<td>For methods defined in the same class as they’re called</td>
<td>For static methods defined in another class</td>
<td>For non-static or instance methods defined in another class</td>
</tr>
<tr>
<td>Examples:</td>
<td>myMethod();</td>
<td>Math.max(a,b)</td>
<td>myString.length()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Integer.parseInt(“6”)</td>
<td>console.nextInt()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Server.gcd(15,12)</td>
<td>myPoint.translate(2,2)</td>
</tr>
</tbody>
</table>
**Fields**

- Fields are variables that contain data for an object.
- Since the fields are bundled in the objects, referring to fields requires specifying an object.
- Referring to the field of an object, general syntax: `<variable> . <field name>`
- Examples:

  ```java
  Point p = new Point(7, 3); // p = (7,3)
p.x = 2; // p = (2,3)
p.y = p.y + 10; // p = (2,13)

  // displays "(2, 13)"
  System.out.println("(" + p.x + ", " + p.y + ")");
  ```
Using Point objects: Example

```java
import java.awt.*;

public class PointMain {
    public static void main(String[] args) {
        // construct two Point objects
        Point p1 = new Point(7, 2);
        Point p2 = new Point(4, 3);

        // print each point and their distance apart
        System.out.println("p1 is " + p1);
        System.out.println("p2: (" + p2.x + ", " + p2.y + ")");
        System.out.println("distance = " + p1.distance(p2));

        // translate the point to a new location
        p2.translate(1, 7);
        System.out.println("p2: (" + p2.x + ", " + p2.y + ")");
        System.out.println("distance = " + p1.distance(p2));
    }
}
```

To use the `Point` class, you have to `import` it from the `java.awt` `package` in Java.
Using **Point objects**: Exercise

- **Write a program** `ComputePerimeter` **that computes a right triangle's** perimeter **given two integer side lengths** \((a\text{ and } b)\).
  - The perimeter is the sum of the triangle's side lengths \(a+b+c\).

- **Example**: **Given side lengths of 12 and 5**, the program should display a perimeter of **30**.
Using Point objects: Solution

```java
import java.util.Scanner;
public class ComputePerimeter {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.println("enter coordinates for point 1: ");
        int x1 = input.nextInt(), y1 = input.nextInt();
        System.out.println("enter coordinates for point 2: ");
        int x2 = input.nextInt(), y2 = input.nextInt();

        Point p1 = new Point(x1, y1);
        Point p2 = new Point(x2, y2);
        double c = p1.distance(p2);
        System.out.println("perimeter = " + (a + b + c));
    }
}
```
Common Programming Error

- The dreaded `NullPointerException`

```java
Point p = null;
p.x = 7;  // Error!
p.setLocation(0,0);  // Error!
```

- If you try to dereference a `null` object, it will cause a `NullPointerException`. **Why?**

- This is a very common error, but one nice thing about Java is that it is often fairly easy to fix this kind of error (in my experience).
Classes and objects
Big idea: Abstraction

- **abstraction**: A distancing between ideas and details.
  - How do objects provide a level of abstraction?

- You use abstraction every day!
  - Do YOU know how your iPod works?
Classes are like blueprints

**Music player blueprint**

- **state:**
  - station/song,
  - volume, battery life
- **behavior:**
  - power on/off
  - change station/song
  - change volume
  - choose random song

**Music player #1**

- **state:**
  - station/song,
  - volume, battery life
- **behavior:**
  - power on/off
  - change station/song
  - change volume
  - choose random song

**Music player #2**

- **state:**
  - station/song,
  - volume, battery life
- **behavior:**
  - power on/off
  - change station/song
  - change volume
  - choose random song

**Music player #3**

- **state:**
  - station/song,
  - volume, battery life
- **behavior:**
  - power on/off
  - change station/song
  - change volume
  - choose random song
Recall: `Point` class

- **Constructing a Point object, general syntax:**
  
  ```java
  Point <name> = new Point(<x>, <y>);
  Point <name> = new Point();  // the origin, (0, 0)
  ```

- **Examples:**
  
  ```java
  Point p1 = new Point(5, -2);
  Point p2 = new Point();
  ```
## Recall: Point class

- **Data stored in each Point object:**

<table>
<thead>
<tr>
<th>Field name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>the point's x-coordinate</td>
</tr>
<tr>
<td>y</td>
<td>the point's y-coordinate</td>
</tr>
</tbody>
</table>

- **Useful methods in each Point object:**

<table>
<thead>
<tr>
<th>Method name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>distance($p$)</td>
<td>how far away the point is from point $p$</td>
</tr>
<tr>
<td>setLocation($x$, $y$)</td>
<td>sets the point's x and y to the given values</td>
</tr>
<tr>
<td>translate($dx$, $dy$)</td>
<td>adjusts the point's x and y by the given amounts</td>
</tr>
</tbody>
</table>
Point class

state:
int x, y

behavior:
distance(Point p)
equals(Point p)
setLocation(int x, int y)
toString()
translate(int dx, int dy)

Point object #1
state:
int x, y
behavior:
distance(Point p)
equals(Point p)
setLocation(int x, int y)
toString()
translate(int dx, int dy)

Point object #2
state:
int x, y
behavior:
distance(Point p)
equals(Point p)
setLocation(int x, int y)
toString()
translate(int dx, int dy)

Point object #3
state:
int x, y
behavior:
distance(Point p)
equals(Point p)
setLocation(int x, int y)
toString()
translate(int dx, int dy)
Definition: An object is an *instance* of its class.

Each instance has its own local copy of the state and behavior defined in the class template.

Example: each Point object has its own x,y coordinates, and its own setLocation method.
Value vs. reference semantics
Recall: Value semantics

- **value semantics**: Behavior where variables are copied when assigned to each other or passed as parameters.
  - Primitive types in Java use value semantics.
  - Modifying the value of one variable does not affect the other.

- **Example:**
  ```java
  int x = 5;
  int y = x;       // x = 5, y = 5
  y = 17;          // x = 5, y = 17
  x = 8;           // x = 8, y = 17
  ```
**Reference semantics**

- **reference semantics**: Behavior where variables refer to a common value when assigned to each other or passed as parameters.
  - Object types in Java use reference semantics.
  - Object variables do not store an object; they store the *address* of an object's location in the computer memory. We graphically represent addresses as arrows.

**Example:**

```java
Point p1 = new Point(3, 8);
```

![Diagram of variable p1 and its coordinates](image)
Reference semantics

- If two object variables are assigned the same object, the object is **NOT** copied; instead, the object’s address is copied.
  - As a result, both variables will *point* to the same object.
  - Calling a method on either variable will modify the same object.

**Example:**

```java
Point p1 = new Point(3, 8);
Point p2 = p1;
p2.setLocation(1, 2);
```

![Diagram showing object reference semantics]
Objects have reference semantics for several reasons:

- **efficiency**: Objects can be large and bulky. Having to copy them every time they are passed as parameters would slow down the program.

- **sharing**: Since objects hold important state, it is often more desirable for them to be shared by parts of the program when they're passed as parameters. Often we want the changes to occur to the same object.
Reference semantics: Example

Point p1 = new Point(3, 8);
Point p2 = new Point(2, -4);
Point p3 = p2;

- How many unique objects are there? How do you know that?
  - Two, because objects are only created with the `new` keyword.

- If we change `p3`, will `p2` be affected and vice versa?
  - Yes.

![Diagram showing the relationship between p1, p2, and p3 with their coordinates]

35
Reference semantics: Example

- If two variables refer to the same object, modifying one of them will also make a change in the other:

```java
p3.translate(5, 1);
System.out.println("(" + p2.x + " " + p2.y + ")");
```

Output:

```
(7, -3)
```
Operators and object types
How not test equality of objects

- **DO NOT DO THIS:**

```java
Point p1 = ...;
Point p2 = ...;
if(p1==p2) {
  ...
}
```

BAD
How not test equality of objects

- Objects store references, or addresses.
- Comparing two objects with `==` will test if they have the same *address*.
- This is NOT what you want.
- **DO NOT DO THIS**

```java
Point p1 = ...;
Point p2 = ...;
if(p1==p2) {
    ...
}
```

**BAD**
Why so bad?

Point p1 = new Point(7, 2);
Point p2 = new Point(7, 2);

Is p1==p2 true or false?
It’s false: they point to different spots in memory. So they store different addresses.
But we want it to be true, obviously!
The `equals` method

- All classes in Java have a built-in `equals` method
- For the `Point` class:
  
  ```java
  p1.equals(p2)
  ```

  - This returns true if `p1` and `p2` have the same data
  - It doesn’t matter if they reference different locations in memory
  
  - Likewise, use `!p1.equals(p2)` instead of `p1!=p2`
**Object and primitive types: a comparison (so far)**

<table>
<thead>
<tr>
<th>Object types</th>
<th>Primitive types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constructed with the new keyword</td>
<td>Values don’t need to be constructed</td>
</tr>
<tr>
<td>References to memory location that stores their data</td>
<td>Store data directly in memory slot</td>
</tr>
<tr>
<td>Can be null (have no data)</td>
<td>Cannot be null</td>
</tr>
<tr>
<td>Can cause NullPointerExceptions</td>
<td>Cannot cause NullPointerExceptions</td>
</tr>
<tr>
<td>Contain state and behavior</td>
<td>Contain state only</td>
</tr>
<tr>
<td>Use reference semantics</td>
<td>Use value semantics</td>
</tr>
<tr>
<td>Use equals() method</td>
<td>Use ==, !=</td>
</tr>
</tbody>
</table>